Lush 2: Reference Manual

Léon Bottou, Yann LeCun

February 5, 2011

Contents

1 Introduction **5**
- 1.1 Lush Revealed . . . 5
- 1.2 Features . . . 7
- 1.3 Libraries . . . 9
- 1.4 Application Areas . . . 9
- 1.5 Implementation . . . 10
- 1.6 History . . . 11
- 1.7 What does LUSH stand for? . . . 13

2 Getting Started: A Quick Tutorial **15**
- 2.1 Installation on Linux/UNIX . . . 15
- 2.2 Starting Lush . . . 16
- 2.3 Configuration . . . 16
- 2.4 Reporting Bugs and Problems . . . 17
- 2.5 Getting Help and Documentation Browsing . . . 17
- 2.6 Basic Syntax . . . 18
  - 2.6.1 Symbol Names and Special Characters . . . 19
- 2.7 Basic Types . . . 20
- 2.8 Defining functions . . . 20
- 2.9 Loops, Conditionals, Local Variables, and Such . . . 21
  - 2.9.1 Loops over structured objects . . . 23
- 2.10 Compiling Functions . . . 24
- 2.11 Program Files . . . 26
- 2.12 On-Line Documentation . . . 27
- 2.13 Multiple File Programs and Search Path . . . 27
- 2.14 Calling C Functions from Lush . . . 28
- 2.15 Mixing Lisp and In-Line C Code . . . 29
- 2.16 Lists . . . 30
- 2.17 Strings and Regular Expressions . . . 32
- 2.18 Scalars, Vectors, Matrices, Tensors, and Storages . . . 32
  - 2.18.1 Working with Storages and Indexes. . . . 33
  - 2.18.2 Index Manipulations . . . 35
  - 2.18.3 Simple Array Operations . . . 36
- 2.19 Matrices and Vectors . . . 37

2.19.1 Special Inner and Outer Products . . . 37
2.19.2 Simple Linear Algebra . . . 38
2.20 Object-Oriented Lush Programming . . . 38
2.20.1 Defining New Classes . . . 38
2.20.2 Inheritance . . . 40
2.21 Input and Output . . . 41
2.21.1 Simple String Input/Output . . . 41
2.21.2 Complete File Reading . . . 42
2.21.3 Lush Object Reading/Writing . . . 42
2.21.4 C-like stdio Functions . . . 43
2.21.5 Socket Communication . . . 43
2.22 Graphics . . . 43
2.22.1 Multiple Window Management . . . 44
2.22.2 PostScript File Graphics . . . 45
2.23 Curve Plotting . . . 45
2.23.1 Gnuplot . . . 46
2.24 Ogre GUI Toolkit . . . 46
2.25 Lush Executable Shell Scripts . . . 47
2.25.1 Executable GUI Applications . . . 48
2.26 Lush Standalone Binary Executables . . . 49

3 Lush Basics **51**
3.1 Lush Interpreter Basics . . . 51
3.2 Lush Startup . . . 52
3.3 The Lush Reader . . . 53
3.4 The Lush Evaluator . . . 54
3.5 Errors in Lush . . . 54
3.6 Interruptions in Lush . . . 55
3.7 Leaving Lush . . . 56
3.8 Lush Memory Management . . . 57
3.9 Lush Libraries . . . 57

4 Language Compatibility **59**
4.1 Lush 2 vs Lush 1 . . . 59
4.2 Compilable Lush . . . 59
4.3 Differences to Common Lisp . . . 60
4.3.1 NIL is not equivalent to () . . . 60

5 Core Interpreter and Startup Library **61**
5.1 Lists . . . 61
5.1.1 List Manipulation Functions . . . 62
5.1.2 Physical List Manipulation Functions . . . 69
5.2 Numbers . . . 71
5.2.1 Numerical Constants . . . 71
5.2.2 Elementary Numerical Functions . . . 71
5.2.3 Integer Arithmetic Functions and Integer Predicates . . . 74

5.2.4 Bit Functions . . . 75
5.2.5 Mathematical Numerical Functions . . . 75
5.2.6 Random Numbers . . . 81
5.2.7 Hashing . . . 82
5.3 Booleans Operators . . . 82
5.3.1 Boolean Arithmetic . . . 82
5.3.2 Predicates . . . 86
5.4 Symbols . . . 89
5.4.1 (defvar `name` [`val`]) . . . 90
5.4.2 (defparameter `name val`) . . . 90
5.4.3 (defconstant `name val`) . . . 90
5.4.4 (defalias `name existing-name`) . . . 91
5.4.5 #.`expr` . . . 91
5.4.6 (setq `s1 v1` ... [`sn vn`]) . . . 91
5.4.7 (set `v a`) . . . 91
5.4.8 (incr `v` [`n`]) . . . 92
5.4.9 (decr `v` [`n`]) . . . 92
5.4.10 (named `s`) . . . 92
5.4.11 (nameof `s`) . . . 92
5.4.12 (namedclean `s`) . . . 93
5.4.13 (lock-symbol `s1` ... `sn`) . . . 93
5.4.14 (unlock-symbol `s1` ... `sn`) . . . 93
5.4.15 (symbols) . . . 93
5.4.16 (macrochp `s`) . . . 94
5.4.17 (putp `anything name value`) . . . 94
5.4.18 (getp `anything name`) . . . 94
5.4.19 (gensym [`x`]) . . . 94
5.4.20 (gensyms `n`) . . . 94
5.4.21 (rotatef `...symbols...`) . . . 94
5.4.22 (symbol-concat `...symbols...`) . . . 95
5.5 Namespaces . . . 95
5.5.1 Class namespaces . . . 96
5.5.2 (in-namespace `ns l1 l2` ...) . . . 97
5.5.3 (in-namespace* `ns l1 l2` ...) . . . 98
5.5.4 (namespace `ns-name`) . . . 98
5.5.5 (with-namespace `ns l1 l2` ...) . . . 98
5.5.6 (with-namespaces (`ns1 ns2` ...) `l1 l2` ...) . . . 98
5.5.7 (delete-namespace `ns`) . . . 98
5.5.8 (join-namespaces `ns1 ns2` ...) . . . 98
5.5.9 (names `ns`) . . . 98
5.5.10 (qualified-names `ns`) . . . 99
5.5.11 (import `names` from `ns` [as `alt-names`]) . . . 99
5.5.12 (import all from `ns`) . . . 99
5.5.13 (defnamespace `name` '((`n1 qn1`) (`n2 qn2`) ...)) . . . 99
5.5.14 (defnamespace `name ns`) . . . 99
5.6 Control Structures . . . 99

5.6.1 (eval l1...ln) . . . 99
5.6.2 (apply f . args) . . . 100
5.6.3 (quote a) . . . 100
5.6.4 (dquote a) . . . 101
5.6.5 'a . . . 101
5.6.6 `expr ,a ,@a . . . 101
5.6.7 (progn l1...ln) . . . 102
5.6.8 (prog1 l1...ln) . . . 102
5.6.9 (let ((s1 v1) ... (sn vn)) l1 ... ln) . . . 102
5.6.10 (lete ((s1 v1) ... (sn vn)) l1 ... ln) . . . 103
5.6.11 (let* ((s1 v1) ... (sn vn)) l1 ... ln) . . . 103
5.6.12 (let-filter ((filter data)) l1...ln) . . . 103
5.6.13 (if cond yes [no1...non]) . . . 104
5.6.14 (when cond yes1...yesn) . . . 104
5.6.15 (while cond l1...ln) . . . 104
5.6.16 (do-while cond l1...ln) . . . 105
5.6.17 (repeat n l1...ln) . . . 105
5.6.18 (mapwhile cond l1...ln) . . . 105
5.6.19 (for (symb start end [step]) l1...ln) . . . 106
5.6.20 (for* (symb start end [step]) l1...ln) . . . 106
5.6.21 (mapfor (symb start end [step]) l1...ln) . . . 107
5.6.22 (cond l1...ln) . . . 107
5.6.23 (selectq s l1...ln) . . . 107
5.6.24 (mapc f l1...ln) . . . 108
5.6.25 (mapcar f l1...ln) . . . 108
5.6.26 (mapcan f l1...ln) . . . 108
5.6.27 (tree-mapcar f l1...ln) . . . 109
5.6.28 (reduce> f carry s1...sn) . . . 109
5.6.29 (reduce< f carry s1...sn) . . . 110
5.6.30 (dolist (s l) . body) . . . 110
5.6.31 (domapc ((s1 l1) ... (sn ln)) . body) . . . 110
5.6.32 (domapcar ((s1 l1) ... (sn ln)) . body) . . . 110
5.6.33 (domapcan ((s1 l1) ... (sn ln)) . body) . . . 111
5.6.34 (each ((s1 v1) ... (sn vn)) l1 ... ln) . . . 111
5.6.35 (all ((s1 v1) ... (sn vn)) l1 ... ln) . . . 111
5.7 Iterables and Iterators . . . 111
5.7.1 Writing Generic Iterator Functions . . . 112
5.7.2 Generic Iterator Functions . . . 112
5.8 Functions . . . 114
5.8.1 Argument List . . . 115
5.8.2 Compact Lambda Expressions . . . 116
5.8.3 Defining a Function . . . 117
5.8.4 Processing optional keywords arguments . . . 124
5.9 Strings . . . 126
5.9.1 String Indexing . . . 126
5.9.2 Basic String Functions . . . 127

5.9.3 Regular Expressions (regex) . . . 134
5.9.4 International Strings . . . 137
5.10 Storages . . . 139
5.10.1 Predefined element types, storage classes . . . 139
5.10.2 Storage Creation and Allocation . . . 139
5.10.3 Storage Access . . . 141
5.10.4 Miscellaneous Atorage Functions . . . 141
5.11 Arrays and Indexes . . . 142
5.11.1 Array API conventions . . . 142
5.11.2 The Index structure, Shape and Subscripts . . . 143
5.11.3 Array and Index Creation and Allocation . . . 144
5.11.4 Array Literals . . . 146
5.11.5 Index Properties and Predicates . . . 147
5.11.6 Arithmetic with Arrays . . . 149
5.11.7 General Index and Array Functions . . . 151
5.11.8 Copying of Indexes and Arrays . . . 159
5.11.9 Index Iterators . . . 161
5.11.10 Arrays of Arrays . . . 163
5.11.11 Matrix Functions . . . 169
5.11.12 Index Argument Checking . . . 171
5.11.13 Direct IDX Manipulations . . . 172
5.11.14 Loading and Saving . . . 172
5.11.15 Component-wise Unary Operations . . . 176
5.11.16 Component-wise Dyadic Operations . . . 178
5.11.17 Contracting Operations . . . 178
5.11.18 Contracting Operations with Scalar Result . . . 178
5.11.19 Operations between Tensors and Scalars . . . 180
5.11.20 Matrix/Vector and 4-Tensor/Matrix Products . . . 180
5.11.21 Outer Products . . . 180
5.12 Objects . . . 181
5.12.1 Object Terminology . . . 181
5.12.2 Inheritance . . . 181
5.12.3 Predefined Classes . . . 182
5.12.4 Defining a Class . . . 182
5.12.5 Creating Objects . . . 184
5.12.6 Accessing Slots . . . 186
5.12.7 Defining Methods . . . 188
5.12.8 Sending Messages . . . 189
5.12.9 Other Special Methods . . . 191
5.12.10 Template classes . . . 193
5.13 Hash Tables . . . 195
5.13.1 Representing Associations with Hash Tables . . . 196
5.13.2 Representing Sets with Hash Tables . . . 200
5.14 Dates . . . 201
5.14.1 (date-to-day `date`) . . . 201
5.14.2 (day-to-date `daynumber`) . . . 202

5.14.3 (today) . . . 202
5.14.4 (now) . . . 202
5.14.5 (split-date `date`) . . . 202
5.14.6 (date-type `date`) . . . 203
5.14.7 (date-extend `date from to`) . . . 203
5.14.8 (date-to-string `date` [`format`]) . . . 204
5.14.9 (string-to-date `string` [`from to`]) . . . 205
5.14.10 (date-add-second `date count`) . . . 206
5.14.11 (date-add-minute `date count`) . . . 206
5.14.12 (date-add-hour `date count`) . . . 206
5.14.13 (date-add-day `date count`) . . . 206
5.14.14 (date-add-month `date count` [`opt`]) . . . 206
5.14.15 (date-add-year `date count` [`opt`]) . . . 207
5.14.16 (date-code `date flags`) . . . 207
5.15 Loading and Executing Lush Program Files . . . 208
5.15.1 (load `in` [`out` [`prompt`]]) . . . 208
5.15.2 ^L `filename` . . . 209
5.15.3 file-being-loaded . . . 209
5.15.4 (libload `libname` [`opt`]) . . . 209
5.15.5 (reload) . . . 210
5.15.6 (libload-dependencies [`fname`]) . . . 210
5.15.7 (libload-add-dependency `fname`) . . . 210
5.15.8 (find-file `dirlist filename extlist`) . . . 210
5.15.9 (save `f s1 ... sn`) . . . 210
5.15.10 (autoload `f s1 ... sn`) . . . 210
5.16 Lisp-Style Input/Output . . . 211
5.16.1 Input . . . 211
5.16.2 Output . . . 212
5.16.3 File Names . . . 214
5.16.4 Searched Path . . . 215
5.16.5 Files and Directories Management . . . 217
5.16.6 Files and Descriptors . . . 218
5.16.7 Binary Input/Output . . . 221
5.16.8 Pipes and Sockets . . . 222
5.17 Miscellaneous . . . 224
5.17.1 Debug Toplevel . . . 224
5.17.2 Error handling functions . . . 226
5.17.3 Memory Management . . . 228
5.17.4 Querying the runtime environment . . . 230
5.17.5 System . . . 232
5.17.6 Copying Lush Objects . . . 244
5.17.7 Special Numerical Values (IEEE754) . . . 244
5.17.8 Floating Point Environment . . . 246
5.18 Graphics . . . 248
5.18.1 Creating a Window . . . 248
5.18.2 Drawing . . . 250

5.18.3 Grabbing Images . . . 253
5.18.4 Drawing with Colors . . . 253
5.18.5 Drawing Text . . . 257
5.18.6 The Drawing Context . . . 259
5.18.7 Double Buffering and Synchronization . . . 260
5.18.8 Plotting Functions . . . 260
5.18.9 Events . . . 267
5.18.10 System Specific Graphic Functions . . . 270
5.18.11 Producing Encapsulated PostScript Files . . . 274
5.18.12 Producing SVG Graphics . . . 275
5.18.13 Graphical Utilities . . . 278
5.19 Organizing multiple windows with tilings . . . 278
5.19.1 (new-tiling `n` [`m`]) . . . 279
5.19.2 (new-tiling* `w` `h`) . . . 279
5.19.3 (new-tiling* `win`) . . . 279
5.19.4 Namespace tiling- . . . 279
5.20 Events and Timers . . . 281
5.20.1 Event Queue . . . 281
5.20.2 Timers . . . 283
5.20.3 Pseudo Threads . . . 284
5.21 Documentation and Help System . . . 285
5.21.1 Using the Online Manual . . . 285
5.21.2 Writing Documentation . . . 288
5.22 Word Processor . . . 294
5.22.1 (render-brace-text `left-margin` `right-margin` `brace`) . 295
5.22.2 (render-brace-html `left-margin` `right-margin` `brace`) . 295
5.22.3 (render-brace-latex `section` `brace`) . . . 295
5.22.4 (render-brace-graphics `x` `y` `w` `brace`) . . . 296

6 Ogre: Object Oriented GUI Toolkit 297
6.1 Introduction to Ogre . . . 297
6.2 Ogre Tutorial . . . 298
6.2.1 Calling Ogre in a Lush Script . . . 300
6.3 The Ogre Class Library . . . 301
6.3.1 Ogre Methods . . . 301
6.3.2 Ogre Class Hierarchy . . . 302
6.4 Ogre Utility Functions . . . 304
6.4.1 Initializing the Ogre library . . . 304
6.4.2 Error Handling in the Ogre Library . . . 305
6.4.3 Ogre Color Palette . . . 306
6.4.4 Ogre Fonts . . . 308
6.5 Visual Objects . . . 309
6.5.1 (new VisualObject `w` `h`) . . . 310
6.5.2 VisualObject Request Messages . . . 310
6.5.3 VisualObject Implementation Methods . . . 312
6.5.4 Event Methods . . . 314

6.6 Control Objects . . . 317
6.6.1 (new Control `w h f`) . . . 317
6.6.2 Enabling or Disabling a Control Object . . . 318
6.6.3 Activation of a Control Object . . . 319
6.6.4 Appearance of a Control Object . . . 320
6.6.5 State of a Control Object . . . 320
6.6.6 Ogre Callbacks . . . 321
6.7 Container Objects . . . 322
6.7.1 (new Container `x y w h ...contents...`) . . . 323
6.7.2 Repainting . . . 323
6.7.3 Inserting an Removing Objects . . . 323
6.7.4 Geometry Management . . . 324
6.7.5 Control Management . . . 326
6.8 Container Flavors . . . 327
6.8.1 Forms . . . 328
6.8.2 Window Objects . . . 328
6.8.3 Structuring Containers . . . 331
6.9 Buttons . . . 334
6.9.1 Push Buttons . . . 334
6.9.2 Check Boxes . . . 335
6.9.3 Exclusive Buttons . . . 337
6.9.4 File Requester Button . . . 338
6.10 Character String Objects . . . 338
6.10.1 Fixed Strings . . . 339
6.10.2 Editable Strings . . . 339
6.10.3 Editable Numbers . . . 341
6.10.4 EditSecretString . . . 342
6.10.5 Multiline Text Editor . . . 342
6.11 Icons . . . 345
6.11.1 (new icon `mat` &optional [`sx` [`sy` [`map`]]]) . . . 345
6.12 OgrImage . . . 345
6.12.1 (new OgrImage `m` [`autoscale`]) . . . 345
6.12.2 (==> `OgrImage` toggle-autoscale) . . . 345
6.12.3 (==> `OgrImage` incr-zoom) . . . 346
6.12.4 (==> `OgrImage` decr-zoom) . . . 346
6.12.5 (==> `OgrImage` get-selected) . . . 346
6.13 ImageViewer . . . 346
6.13.1 (new ImageViewer `w h m` [`scroll`]) . . . 346
6.13.2 (==> `ImageViewer` get-selected) . . . 346
6.14 Menus . . . 346
6.14.1 Standard Menus . . . 347
6.14.2 Choice Menus . . . 348
6.15 Popup Requesters . . . 349
6.15.1 Requester . . . 350
6.15.2 Warning Requester . . . 352
6.15.3 Error Requester . . . 352

6.15.4 Yes/No Requester . . . 353
6.15.5 Print Requester . . . 355
6.15.6 File Requester . . . 356
6.16 Movable and Resizable Objects . . . 357
6.16.1 (new DragArea `w h`) . . . 358
6.16.2 (new SizeArea `w h`) . . . 358
6.16.3 (new Frame `x y w h ...contents...`) . . . 358
6.16.4 (new FrameSize `x y w h ...contents...`) . . . 359
6.17 Sliders and Scrollbars . . . 359
6.17.1 Sliders . . . 360
6.17.2 Scrollbars . . . 361
6.18 Composite Objects . . . 361
6.18.1 Viewers . . . 361
6.18.2 Selectors . . . 362
6.19 A Complete Example . . . 364
6.19.1 The Class Browser "classtool" . . . 365
6.19.2 The Program "classtool" . . . 366

7 Dynamic Loader/Linker 373
7.1 (mod-load `filename`) . . . 374
7.2 (mod-unload `filename`) . . . 375
7.3 (find-shared-library `name` [`extlist`]) . . . 375
7.4 (find-static-library `name` [`extlist`]) . . . 376
7.5 (find-shared-or-static-library `name` [`extlist`]) . . . 376
7.6 (mod-list) . . . 376
7.7 (mod-undefined) . . . 376
7.8 (mod-status) . . . 376
7.9 (mod-inquire `filename`) . . . 377
7.10 (mod-create-reference `string1` ... `stringN`) . . . 377
7.11 (mod-compatibility-flag `boolean`) . . . 377
7.12 Low-Level Module Functions . . . 377
7.12.1 (module-list) . . . 377
7.12.2 (module-filename `m`) . . . 377
7.12.3 (module-executable-p `m`) . . . 377
7.12.4 (module-unloadable-p `m`) . . . 378
7.12.5 (module-initname `m`) . . . 378
7.12.6 (module-depends `m`) . . . 378
7.12.7 (module-never-unload `m`) . . . 378
7.12.8 (module-defs `m`) . . . 378
7.12.9 (module-load `filename` [`hookfunction`]) . . . 378
7.12.10 (module-unload `m`) . . . 378
7.13 Extending the Interpreter . . . 379
7.14 Debugging Modules with gdb . . . 380
7.15 Compiling and Loading C Code . . . 382
7.15.1 (new LushMake [`srcdir` [`objdir`]]) . . . 382
7.15.2 (==> `lushmake` setdirs `srcdir` [`objdir`]) . . . 383

7.15.3 (==> lushmake setflags flags) . . . 383
7.15.4 (==> lushmake rule target deps [command]) . . . 383
7.15.5 (==> lushmake show [...target...]) . . . 384
7.15.6 (==> lushmake make [...target...]) . . . 384
7.15.7 (==> lushmake load [...targets...]) . . . 384

8 CLush: Compiled Lush 385
8.1 A Simple Example . . . 385
8.2 Using libload and dhc-make . . . 385
8.3 Compilation Functions . . . 386
8.3.1 High-Level Compilation Functions . . . 386
8.3.2 Customizing the Behavior of dhc-make . . . 388
8.3.3 Querying File Dependencies . . . 390
8.3.4 Low-Level Compilation Functions . . . 390
8.4 What is Compilable, and What is Not . . . 391
8.4.1 (compilablep function) . . . 392
8.5 CLush Specific Functions and Constructs . . . 392
8.5.1 Numerical Type Declarations . . . 392
8.5.2 Class Declaration and Compilation . . . 394
8.5.3 Type Conversions . . . 394
8.5.4 Inline C Code . . . 396
8.5.5 Compiler Directives . . . 398
8.5.6 Dynamic Allocation in Compiled Code . . . 399
8.6 Interfacing Existing C/C++/FORTRAN Libraries to Lush . . . 399
8.6.1 Interfacing C/C++ Code . . . 399
8.6.2 Interfacing FORTRAN Code . . . 402
8.7 Making Standalone Executables . . . 402
8.7.1 (make-standalone lushfile cdir executable main-func) 402
8.7.2 low-level support functions for make-standalone . . . 403
8.8 More on the Lisp-C Interface . . . 403
8.8.1 DH: the Compiled Function Class . . . 403
8.8.2 Classinfo . . . 404
8.8.3 Controling the Lisp/C Interface . . . 404
8.9 Compiler Internals . . . 405
8.9.1 (dhc-generate-c filename '([func1 [funcn]])) . . . 405
8.9.2 (compilablep function) . . . 405
8.9.3 (dhc-substitute-env str [htable]) . . . 406
8.9.4 (dhc-generate-include-flags [includepath]) . . . 406
8.9.5 varlushdir . . . 406
8.9.6 (dhc-make-cdir cdir [create]) . . . 406
8.9.7 (dhc-make-c-filename src) . . . 406
8.9.8 (dhc-make-o-filename src) . . . 406
8.9.9 (dhc-make-get-dependencies [snname]) . . . 406
8.9.10 dhc-make-lushflags . . . 407
8.9.11 dhc-make-command . . . 407
8.9.12 dhc-make-overrides . . . 407

8.9.13 dhc-make-force . . . 407
8.9.14 dhc-make-essential-libs . . . 407
8.9.15 (dhc-make-o `src-file` [`obj-file` [`lushflags`]]) . . . 407
8.9.16 (dhc-make-rebuild-p `target dependencies`) . . . 407
8.9.17 (dhc-make-o-maybe `src-file` [`obj-file` [`cflags`]]) . . . 408
8.9.18 (dhc-make-c `fname fsymblist`) . . . 408
8.9.19 (dhc-make-c-maybe `snname fname fsymblist`) . . . 408
8.9.20 (dhc-make-test `fname`) . . . 408
8.9.21 (dhc-make-with-c++ `fname library-list f1` [`f2` ...[`fn`]]) 408
8.9.22 (dhc-nolast `l`) . . . 408
8.9.23 (dhc-remove-dup `l`) . . . 408
8.9.24 (dhc-remove-eqdup `l`) . . . 408
8.9.25 (dhc-remove-nth `n l`) . . . 409
8.9.26 (dhc-postincr `symb`) . . . 409
8.9.27 dhc-debug-flag . . . 409
8.9.28 —#@— . . . 409
8.9.29 (dhc-add-c-declarations `str` [`str2` ...]) . . . 409
8.9.30 (dhc-add-c-statements `str` [`str2` ...]) . . . 409
8.9.31 (dhc-add-c-epilog `str`) . . . 409
8.9.32 (dhc-add-c-externs `str`) . . . 409
8.9.33 (dhc-add-c-metaexterns `str`) . . . 409
8.9.34 (dhc-add-c-header `str`) . . . 409
8.9.35 (dhc-add-c-pheader `str`) . . . 410
8.9.36 (dhc-class-to-struct-decl `type`) . . . 410
8.9.37 (dhc-class-to-vtable-decl `type`) . . . 410
8.9.38 (dhc-type-to-c-decl `type`) . . . 410
8.9.39 (dhc-declare-temp-var `type` [`clue`]) . . . 410
8.9.40 dhc-debug-stack . . . 410
8.9.41 (dhc-error `string` [`arg`]) . . . 410
8.9.42 (dhc-check-symbol `source`) . . . 410
8.9.43 (dhc-internal-error `str`) . . . 410
8.9.44 dhc-type . . . 410
8.9.45 (==> `dhc-type` print) . . . 411
8.9.46 (==> `dhc-type` hashcode) . . . 411
8.9.47 (==> `dhc-type` access [`v`]) . . . 411
8.9.48 (new dhc-type `class` [`a1`] [`a2`]) . . . 411
8.9.49 (dhc-desc-to-type `desc`) . . . 411
8.9.50 (new dhc-symbol `name lex` [`fmt`]) . . . 411
8.9.51 (dhc-search-symtable `symbolname table`) . . . 411
8.9.52 (dhc-add-to-symtable `table symbolobject`) . . . 412
8.9.53 (dhc-add-symbol-table `symb lex`) . . . 412
8.9.54 (dhc-add-global-table `symb`) . . . 412
8.9.55 (new t-node `t-node-list type` [`source`] [`symbol`]) . . . 412
8.9.56 (dhc-make-t-node `expr`) . . . 412
8.9.57 (dhc-copy-source-tree `source`) . . . 412
8.9.58 (dhc-get-treetype `source`) . . . 413

8.9.59 (dhc-get-type `source`) . . . 413
8.9.60 (dhm-t `func` (source) . `body`) . . . 413
8.9.61 (dhm-t-declare `model f1` ... `fn`) . . . 413
8.9.62 (dhm-c `symb` (source treetype retplace) . `body`) . . . 413
8.9.63 (dhm-c-declare `model f1` ... `fn`) . . . 413
8.9.64 (get-dhm-t `symb`) . . . 413
8.9.65 (get-dhm-c `symb`) . . . 414
8.9.66 (dhm-p `func` (source) . `body`) . . . 414
8.9.67 (get-dhm-p `symb`) . . . 414
8.9.68 (dhc-pp `body`) . . . 414
8.9.69 (dhc-parse-replacement-source-t `source newsource`) . . . 414
8.9.70 (process-numerical-args-t `arglist rettype`) . . . 414
8.9.71 u-nodes . . . 414

9 Standard Libraries 415
9.1 C-Style Input/Output . . . 415
9.1.1 (fprintf `file-pointer args...`) . . . 415
9.1.2 (fwrite-str `file-pointer s`) . . . 415
9.1.3 (stdout) . . . 416
9.1.4 (stdin) . . . 416
9.1.5 (fopen `filename type`) . . . 416
9.1.6 (fclose `file-pointer`) . . . 416
9.1.7 (popen `filename type`) . . . 417
9.1.8 (pclose `file-pointer`) . . . 417
9.1.9 (ftell `file-pointer`) . . . 417
9.1.10 (fseek `file-pointer pos`) . . . 417
9.1.11 (fseek-from-end `file-pointer pos`) . . . 418
9.1.12 (fseek-from-current `file-pointer pos`) . . . 418
9.1.13 (fgetc `file-pointer`) . . . 418
9.1.14 (fputc `file-pointer val`) . . . 418
9.1.15 (fread-ubyte `file-pointer`) . . . 419
9.1.16 (fwrite-ubyte `file-pointer val`) . . . 419
9.1.17 (fread-byte `file-pointer`) . . . 419
9.1.18 (fwrite-byte `file-pointer val`) . . . 419
9.1.19 (fread-short `file-pointer`) . . . 420
9.1.20 (fwrite-short `file-pointer val`) . . . 420
9.1.21 (fread-int `file-pointer`) . . . 420
9.1.22 (fwrite-int `file-pointer val`) . . . 420
9.1.23 (fread-flt `file-pointer`) . . . 421
9.1.24 (fwrite-flt `file-pointer val`) . . . 421
9.1.25 (fread-real `file-pointer`) . . . 421
9.1.26 (fwrite-real `file-pointer val`) . . . 421
9.1.27 (reverse_n `ptr sz n`) . . . 422
9.1.28 (fscan-int `file-pointer`) . . . 422
9.1.29 (fscan-flt `file-pointer`) . . . 422
9.1.30 (fscan-str `file-pointer`) . . . 422

9.1.31 (fgets `file-pointer max-size`) . . . 422
9.1.32 (file-size `file-name`) . . . 423
9.1.33 (rewind `f`) . . . 423
9.1.34 (skip-comments `start f`) . . . 423
9.2 Abstract datatypes . . . 423
9.2.1 Integer pairs . . . 423
9.2.2 Queue (FIFO) . . . 424
9.2.3 Stack (LIFO) . . . 425
9.2.4 Double-ended Queue . . . 426
9.2.5 Partition . . . 427
9.2.6 Heap / Priority Queue . . . 428
9.2.7 Sets . . . 429
9.2.8 Small Integer Sets . . . 431
9.2.9 IntGraph . . . 432
9.3 Image Processing Libraries . . . 435
9.3.1 Displaying images . . . 435
9.3.2 Reading/Writing Image Files to/from IDX . . . 436
9.3.3 Reading/Writing PPM/PGM/PBM Image Files . . . 439
9.3.4 Reading/Writing PBM Image Files . . . 442
9.3.5 Reading/Writing Run-Length-Encoded Image Files . . . 442
9.3.6 RGBA Images of ubytes . . . 443
9.3.7 RGBA Images of floats . . . 448
9.3.8 Greyscale Images of ubytes . . . 452
9.3.9 Greyscale Images of shorts . . . 456
9.3.10 Greyscale Images of floats . . . 459
9.3.11 Computing Image Warping Maps . . . 461
9.3.12 Connected Component Analysis . . . 461
9.3.13 Morphological Operations . . . 465
9.3.14 Morphological Operation on Images of shorts . . . 466
9.3.15 Miscellaneous Image Macros and Conversions . . . 468
9.3.16 Constants Used in Run-Length Encoded Images . . . 468
9.3.17 Converting RLE Image to Greyscale Pixelmap . . . 469
9.3.18 Color Clustering and Quantization . . . 470
9.3.19 Tools for Image Segmentation . . . 475
9.4 Plotting Library . . . 476
9.4.1 Plotter . . . 477
9.4.2 Low-Level Utility Functions . . . 484
9.5 Computational Geometry in the Plane . . . 485
9.5.1 Planar Meshes . . . 485
9.5.2 Simple Polygons . . . 488
9.5.3 Polygons . . . 489
9.5.4 Planar Triangulations . . . 490
9.5.5 Axis-based shape descriptors . . . 492
9.5.6 (delaunay `arg`) . . . 492
9.5.7 (convex-hull `points`) . . . 493
9.5.8 Line simplification . . . 493

9.6 Shell Commands . . . 493
9.6.1 String List Utilities . . . 494
9.6.2 File Information . . . 494
9.6.3 Filename Manipulations . . . 495
9.6.4 Variables . . . 498
9.6.5 Directories . . . 498
9.6.6 Shell Commands . . . 498
9.7 Stopwatch: Timer with Microsecond Accuracy . . . 499
9.7.1 (new stopwatch) . . . 499
9.7.2 (==> `stopwatch` get) . . . 499
9.7.3 (==> `stopwatch` reset) . . . 500
9.8 Profiling for Lush . . . 500
9.8.1 (profile-stats `fn-name`) . . . 500
9.8.2 (profile-clear-stats `fn-name`) . . . 500
9.8.3 (profile-stats-all) . . . 500
9.8.4 (profile-clear-stats-all) . . . 500
9.8.5 (profile `fn-def`) . . . 500
9.9 Tensor/Matrix/Vector/Scalar Libraries . . . 501
9.9.1 Generic IDX Macros and Functions . . . 501
9.9.2 IDX of Single Precision Floating Point Numbers . . . 504
9.9.3 IDX of Double Precision Floating Point Numbers . . . 509
9.9.4 IDX of Integers . . . 515
9.9.5 IDX of unsigned bytes . . . 517
9.9.6 IDX reading and writing . . . 519
9.9.7 IDX mapping from files . . . 537
9.9.8 Convolution, Subsampling, Oversampling. . . . 539
9.9.9 IDX sorting and binary search functions . . . 544
9.9.10 IDX operations on squares of elements . . . 545
9.9.11 Array binning . . . 546

10 Packages 547
10.1 Plotting with gnuplot . . . 547
10.1.1 The High-Level gnuplot Interface . . . 548
10.1.2 The Low-Level gnuplot Interface . . . 553
10.2 GBLearn2: Machine Learning Library . . . 555
10.2.1 GBLearn2: Basic Concepts . . . 555
10.2.2 Examples and Demos . . . 558
10.2.3 Learning Algorithms . . . 558
10.2.4 Data Sources . . . 561
10.2.5 Meters . . . 565
10.2.6 gb-states . . . 566
10.2.7 gb-param . . . 568
10.2.8 Module Libraries . . . 570
10.3 SDL: Simple DirectMedia Layer . . . 585
10.3.1 Installing SDL . . . 586
10.3.2 Introduction to Lush's "SDL screen classes" . . . 586

10.3.3 SDL Demos . . . 589
10.3.4 LibSDL: High-level Interface to SDL . . . 589
10.3.5 Low-Level Interface to SDL . . . 597
10.4 Video4Linux: video grabbing . . . 638
10.4.1 Requirements and Installation video4linux and appropriate device . . . 639
10.4.2 Video4Linux-v2 API (v4l2) . . . 639
10.4.3 Video4Linux API (v4l) . . . 641
10.4.4 Demos . . . 645
10.5 BLAS: Basic Linear Algebra Subroutine . . . 647
10.6 LAPACK . . . 648
10.7 ALSA: Advanced Linux Sound Architecture . . . 648

11 Applications 649
11.1 Graphic Tools . . . 649

12 Tutorials 651
12.1 Tutorial: Writing Games with Lush and SDL. . . . 651
12.1.1 A Quick Reminder on Basic Lush Programming . . . 651
12.1.2 A Simple Lunar Lander . . . 652
12.1.3 SpaceWar: missiles and collision detection . . . 664

13 FAQ (Frequently Asked Questions) 665
13.1 What to do when I get this error? . . . 665
13.1.1 Module has undefined references . . . 665
13.1.2 compiler : Unknown Type in: () . . . 665
13.2 How Do I ... ? . . . 666
13.2.1 Read lines from a file into a list . . . 666
13.2.2 Apply a function to all elements of a vector . . . 666
13.2.3 Get a pointer to the raw data of an idx . . . 666
13.2.4 Get a pointer to a function written in Lisp . . . 666
13.2.5 Know if a function can be used in compiled code . . . 667

Chapter 1

Introduction

An introduction to Lush and its capabilities and features is presented in this section.

1.1 Lush Revealed

Lush is an object-oriented programming language with features designed to please researchers, experimenters, and engineers interested in large-scale numerical and graphical applications. It is designed to be used in situations where one wants to combine the flexibility of a high-level, loosely-typed interpreted language, with the efficiency of a strongly-typed, natively-compiled language, and easy integration with code written in C, C++, or other languages.

The advantages of Lush are especially significant on projects where a combination of an interpreted language (e.g.Python, Perl, Matlab, S+, or even [gasp!] BASIC) and a compiled language (e.g. C) would otherwise be used. With Lush the best of both the interpreted and compiled programming worlds are obtained by wrapping three languages into one:

- 1. a loosely-typed, garbage-collected, dynamically scoped, interpreted language with a simple Lisp-like syntax;
- 2. a strongly-typed, lexically-scoped compiled language with the same Lisp-like syntax;
- 3. the C language, which can be freely mixed with Lush code within a single program or within a single function.

It sounds complicated, but it is not. In fact, Lush is designed to be very simple to learn and to use.

The main features of Lush include:

- a very clean, simple, and easy to learn Lisp-like syntax;

- a very efficient native compiler (through C);
- an easy way to interface C functions and libraries, combined with a powerful dynamic loader for the object files of libraries (`.o` , `.a` , and `.so` files) written in other compiled languages;
- the ability to freely mix Lisp and C in a single function;
- a powerful set of vector, matrix, and tensor operations;
- a huge library of numerical routines, including GSL, LAPACK, and BLAS;
- an extensive set of graphics routines (that includes an object-oriented GUI toolkit and interfaces to OpenGL and SDL);
- sound and video grabbing (via ALSA and Video4Linux);
- a library of image and signal processing routines;
- several libraries for machine learning, neural networks, and statistical estimation.

This combination of flexibility, efficiency, and extensive libraries makes Lush an ideal platform for research and development in artificial intelligence, bioinformatics, computer vision, data mining, image processing, machine learning, signal processing, and statistics, Its speed and extensive libraries allow it to be applied to areas such as real-time audio, image, and video processing. Many users put Lush to work as a general purpose scripting language or as their main language for application development. Some users are known to have used Lush to develop 2D and 3D games. A few have even used Lush to develop commercial software for embedded processors.

In comparison to Matlab, Lush is a "real" object-oriented programming language with data structures, typed matrices, simple syntax, and a native compiler. Compared to Python, Lush has an efficient native compiler (compiled numerical code is as fast as C) and provides a functional programming paradigm. The ability in Lush to easily call C libraries and freely mix C and Lisp together is unique.

If at any time you have:

- wished you had a simple interpreted language with which you could quickly try out ideas, implement efficient numerical algorithms, or prototype GUI-based applications;
- written a piece of software in C and wished you could control it from a simple, interpreted script language;
- written a script language interpreter yourself and wished it were a full-blown programming language;

- used an interpreted script-like language, such as Perl, MatLab, Mathematica, Tcl, Python, or BASIC, and wished the compiler generated efficient code and that it were easier to call C routines from it;

- written a program with scripting languages, like the ones previously mentioned, and wished you could generate a portable standalone application from it;

- or wished you could combine two languages for your programming projects: an efficient, compiled, no-frills language such as C or C++ for the low-level implementation, and an interpreted, weakly typed language with smart memory management, such as Lisp, for the high level implementation and user interface.

then Lush is for you!

Many software projects, particularly research projects, require two languages: an efficient compiled language, such as C or C++, for implementing the low-level or computationally expensive functions, and a flexible (possibly interpreted) language for high-level control, scripting, experimentation, and tinkering. Popular research-oriented interpreters like Matlab are somewhat inefficient, have little or no support for complex data structures, do not provide the power of a full-fledged object-oriented programming language, and lack simple interfacing functionality to C and other compiled languages.

The syntax of Lush is a simple form of Lisp. If the word "Lisp" sends shivers down your spine, be advised that the dialect of Lisp that Lush implements is extremely simple to learn, with one of the simplest syntaxes possibly available. Most scientists, engineers, and software developers who learn Lush become proficient with it in just a few short days, even if they have had no prior exposure to Lisp. It's very simple. Really. In fact, Lush has been used to teach programming to kids!

The Lush compiler has several interesting properties (and a few limitations). Its main advantage is that it generates very efficient C code that is compiled with the best available C compiler for the machine under consideration.

Lush currently runs on various Unix platforms including: Linux/x86, Solaris, and SGI/Irix. It can also be run on Mac OS X and Windows (under Cygwin).

1.2 Features

Lush is built around a compact, portable, and intentionally simple Lisp interpreter written in C. It features all the usual functionality and constructs found in every decent object-oriented programming language such as conditional statements, loops, local variables, functions, macros, objects, classes, methods, and inheritance. In addition it also provides a large number of functions for manipulating lists, strings (including regular expression matching and substitutions), vectors, matrices, and tensors.

The Lush interpreter is quite similar to some lisps of the mid 1980's like Le_Lisp and UCI-Lisp. It is significantly simpler than Common Lisp and very different than Scheme.

Some unusual features of Lush are its compiler to C, dynamic loader, and the ease with which interfacing to existing C functions and libraries can be accomplished. A particularily unusual feature is the language facility it provides to intermix Lisp and C source code within a single function.

The vector and matrix manipulation engine is quite powerful and efficient. This makes Lush ideal for computationally demanding numerical applications and signal and image processing. Functions are included to create, resize, and convert vectors, matrices, and tensors with up to eight dimensions. Basic matrix operations such as scalar operations (on all elements of a matrix), dot products, outer products, transpositions, highly optimized 1D and 2D convolutions are, of course, included. A set of iterators is also provided to access any matrix element without requiring costly bounds checking.

As an object-oriented language, Lush provides the ability to define and compile classes with slots and methods, and for derived classes to inherit the slots and methods of their parent class (similar to C++ semantics).

All the functions familiar to Lisp enthusiasts are included in Lush. These include: list functions, list iterators, physical list manipulators, macros, splicing macros, symbol manipulators, and so on. Most casual users will probably prefer to stay away from some of the more complicated of these.

A set of simple-to-use graphics functions are supplied to draw lines, polygons, rectangles, pictures, and text in color, with automatic refresh and double-buffering capability (for simple animations). Graphics can be drawn in an X Windows system window (on Unix) or sent to a PostScript file. The low-level portable graphics functions provided are used to build high-level functionalities, such as function plotting.

There is also a very compact and easy-to-use object-oriented graphical user interface (GUI) generator called Ogre that comes with Lush. Ogre is entirely written in Lisp on top of the low-level graphics functions mentioned above. It contains predefined classes for buttons, sliders, radio buttons, menus, string editors, and so forth. Ogre includes an automatic mechanism for placing objects in a window, thereby greatly simplifying the design and implementation of GUIs. Simple GUIs can be written in extremely short times and are very compact.

Lush provides two models for input and output. One is a set of Lisp-oriented functions that allows easy input and output of ASCII data, lisp expressions, lisp objects, and matrices. It includes goodies such as pipes and sockets. The other model essentially provides access to the standard C I/O library, including fopen, popen, fprintf, fscanf, fgetc, fputc, fgets, and various functions for reading and writing matrices. With these functions large matrices can be mapped into the virtual addressing space, instead of being explicitely loaded into memory. This allows efficient access of very large datasets.

1.3 Libraries

A huge collection of libraries and utilities are available to the user in Lush. Some of these are written in Lisp, some in C and interfaced to Lush, while others are pre-existing libraries that have been interfaced to Lush.

For our numerically enclined friends, Lush has a full interface to the GNU Scientific Library (GSL), and LAPACK and BLAS Linear Algebra libraries. This gives access to an extensive set of numerical and statistical functions (several thousand in fact).

A full interface to the industry-standard OpenGL library that enables the creation of 3D graphics and animations is provided. This interface itself includes an interface to GLUT, OpenGLU, and OpenRM (scene graph rendering engine). This feature makes Lush an excellent platform to write interactive VR applications and computer games in.

Another popular library interfaced to Lush is the Simple Directmedia Layer (SDL) video game API. This is enhanced by a high-level library that allows easy manipulation of sprites and movable screen objects with pixel-accurate collision detection. The library, combined with Lush's simple syntax, is ideal for developing simple video games and teaching programming to children.

An image processing library with functions to load, save, resize and resample, warp, filter, and analyze images is also at the users disposal. Mathematical morphology operations such as connected component analysis, distance transform, erosion, and dilation, are also available for bitonal images. Classes and functions for easily grabbing video using the Video4Linux API is included also.

Another included library provides graph functionality that allows grammar and finite state machine construction, graph transduction and composition, and viterbi search algorithm.

Lush includes an extensive library for gradient-based machine learning, including neural networks, radial basis functions, support vector machines, and many others. This library is based on an innovative object-oriented design that facilitates the construction of large learning machines from multiple learning modules and cooperative learning machine training. Commercially used optical recognition systems have been built with this library.

In addition, Lush provides various interfaces to multimedia libraries, including Video4Linux (video grabbing) and ALSA (audio recording/playing).

1.4 Application Areas

Lush is a good tool for a variety of applications. It was originally developed as an environment for experimentation and development of machine learning, neural nets, and pattern recognition applications, but over the years it has grown into a full-fledged language and rapid development environment.

Here are examples of situations in which Lush (or its predecessor SN) have been used:

- a simulation environment for neural network and machine learning experiments
- a Matlab-like prototyping tool for numerical computation, signal processing, image processing, statistical estimation, and so forth
- a control/script language for software projects
- a super debugger and diagnostic tool for large applications
- a quick prototyping tool for GUI-based applications
- a script language for quick hacks

Here is a small subset of the research projects that have been carried out with Lush:

- numerous handrwiting recognition projects
- many projects in neural networks, machine learning, and statistical estimation
- datamining, fault detection, and database marketing projects
- image processing research

A few full-fledged commercial applications have also been built with Lush, including:

- a complete check amount reader, now integrated in NCR's automatic teller machines and large back-office check reading machines (Lush-generated code runs on DSP boards). This is a huge piece of complicated code (60,000 lines of Lush Lisp code automatically converted to C)
- an early version of the foreground/background segmentation module for the DjVu image compression system
- a neural network simulator and neural network training tool
- a pen-based data entry system with handwriting recognition

1.5 Implementation

The Lush programming language has been implemented as a variation of Lisp.

In some segments of the software industry Lisp is sometimes perceived as an oddity of essentially academic interest. Its theoretical computer science heritage also causes common misconceptions about Lisp, such as it being inefficient and difficult to learn. So why choose Lisp? We chose Lisp because, contrary to the common prejudice, it is extremely easy to learn, in addition to being flexible, efficient, and compact. Lisp has such a simple, clean, and flexible syntax that it

is probably the easiest language to learn (in contrast, Perl is found at the other extreme). Our experience with teaching C developers to use Lush is that they become proficient with it in a few days. Script language designers often make the mistake of not only designing their own language functionalities, but also of designing their own syntax. Examples of this include MatLab, Mathematica, S+, and many others. Why invent a new syntax when a good one such as Lisp already exists? Lush is just such a language, it is an object-oriented dialect of Lisp that puts the emphasis on ease of use, efficiency for numerical operations (unlike many traditional Lisp implementations), and close to effortless interfacing with existing code written in C.

Another unique advantage of Lisp is that it is a programable programing language. In Lisp, a program is just a data structure that can be created and manipulated just like any other data structure. In other words, Lisp programs can create other Lisp programs. This allows users (and not just language designers) to extend the language themselves without limit. This feature also makes it easy to write such things as syntax transformers, self-optimizing programs, compilers, automatic differentiators, etcetera. Most users will probably stay away from writing such things, but they will profit from the work of others in these areas (e.g. the CLush compiler).

1.6 History

Lush is the direct descendent of the SN system, originally developed by Leon Bottou and Yann LeCun, that was the front-end of a neural network simulator. Various incarnations of SN have been developed continuously since 1987, some of which were sold commercially by Neuristique S.A. in France, and eventually grew into a full-fledged prototyping and development environment.

Versions developed at AT&T Bell Labs, and then at AT&T Labs and at the NEC Research Institute were used to build many succesful technologies and products. The most notable ones are:

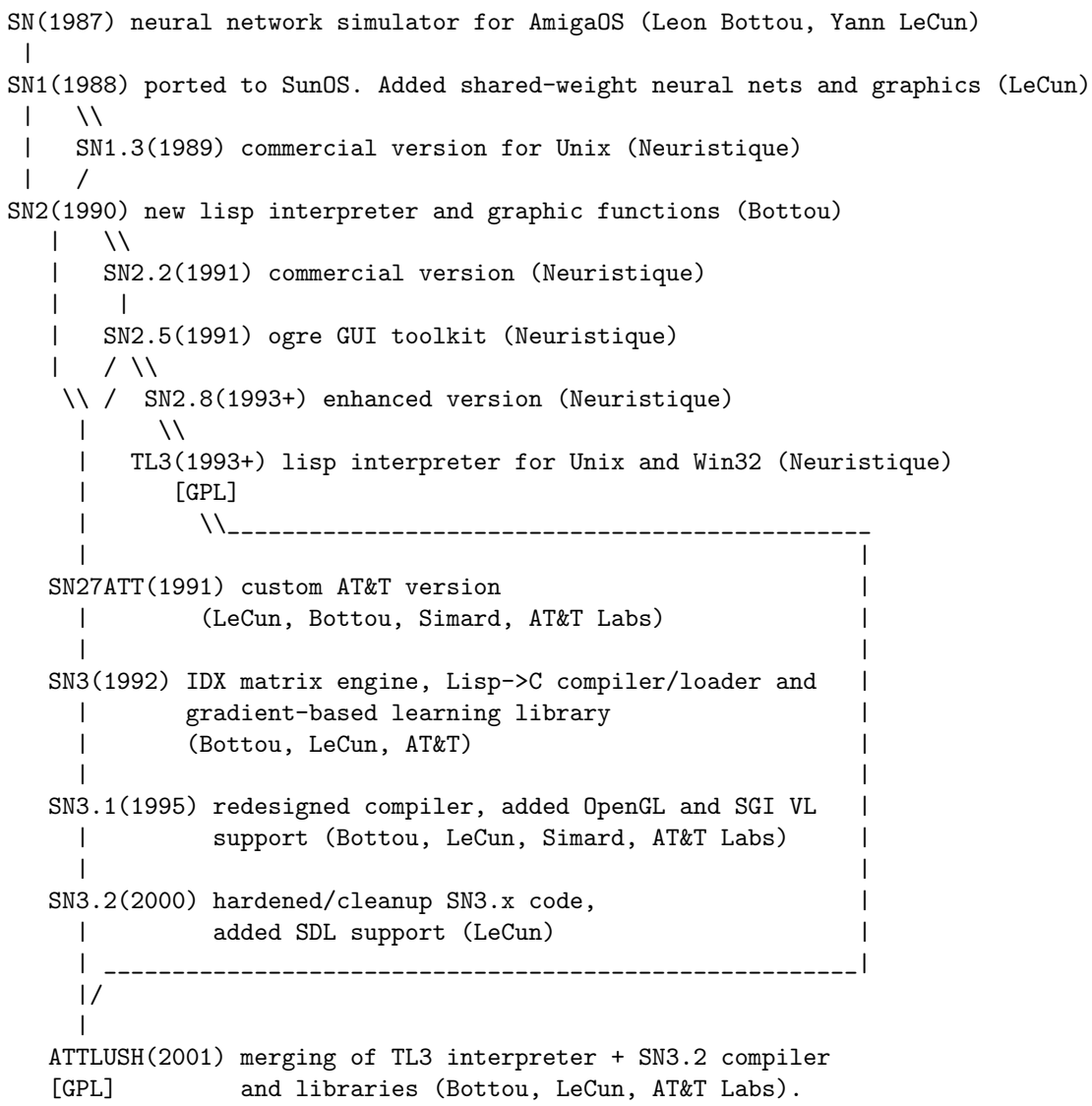
- a handwriting recognition system used by many banks across the world to automatically read checks. In fact, some ATM machines made by NCR (that can read checks) run compiled SN code on embedded DSP boards
- the prototype of the DjVu image compression system
- numerous machine learning algorithms developed at AT&T since 1988, including the LeNet family of convolutional neural networks and some early implementations of the Support Vector Machine algorithm

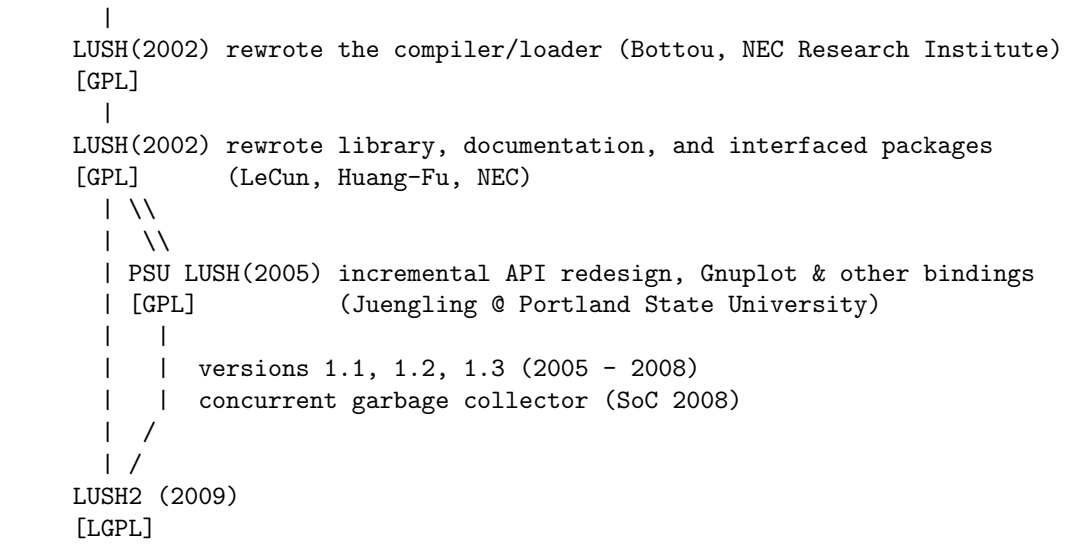
SN was primarily used internally at AT&T Bell Labs for many research projects in machine learning, pattern recognition, and image processing. But its various incarnations were used at AT&T Labs, Lucent, the Salk Institute, the University of Toronto, Universite de Montreal, UC Berkeley, and many other research institutions. The commercial versions of SN were used in several large companies as a prototyping tool: Thomson-CSF, ONERA,....

Contributors include: Leon Bottou, Yann LeCun, Jie Huang Fu, Patrice Simard, Yoshua Bengio, Jean Bourrelly, Patrick Haffner, Pascal Vincent, Sergey Ioffe, and many others.

In 2001, AT&T and Neuristique released their respective versions under the GPL, allowing the development and distribution of Lush. Turning SN into Lush was done by Yann LeCun, Leon Bottou and Jie Huang-Fu at the NEC Research Institute.

Here is a family tree of the various incarnations of SN and Lush:





1.7 What does LUSH stand for?

LUSH is supposed to stand for "Lisp Universal SHell" or something like that. But English dictionaries tell another tale. According to the 1914 edition of the Century Dictionary (http://century-dictionary.com), Lush has 4 main meanings in English:

```
LUSH (1)
 - Adjective
    1. Slack; limp; flexible.
    2. Mellow; easily turned, as ground.
    3. Fresh, luxuriant, and juicy; succulent, as grass or other vegetation.
 - Noun
    1. A twig for thatching. [Prov. Eng.]
LUSH (2)
 - Verb Intransitive
    1. To rush violently.
    2. To splash in water.
LUSH (3)
 - Noun
    1. Beer; intoxicating drink. [Slang.]
 - Verb
    1. To drink; tipple on. [Slang.]
    2. To drink intoxicating liquor. [Slang.]
LUSH (4)
 - Noun
    1. The Burbot (fresh water cod fish).
```

To some of us, Lush certainly appears fresh, luxuriant, juicy, succulent, flexible, mellow, and intoxicating.

To a few others, it might appear slack, limp and fishy.

You be the judge.

Chapter 2

Getting Started: A Quick Tutorial

Author(s): Yann LeCun

By following through the subsections of this section, and trying out the code examples and commands given, the users will be well on their way towards using Lush.

2.1 Installation on Linux/UNIX

Compilation and installation of Lush requires certain packages and libraries be available on your system. In particular, you need the xlib-development package, which is not necessarily installed by default.

To enjoy the full capabilities of Lush, you may also want to install the Gnu Scientific Library (GSL or libGSL). Most Linux distros have it in two packages libgsl or libgsl0 and libgsl-devel or libgsl0-devel. LibGSL can also be installed from source, which can be obtained at [http://sources.redhat.com/gsl]

Another nice optional package to install is libSDL and its development libraries. It is also preferrable to set up OpenGL correctly on your machine (include the development packages when installing OpenGL).

- The LAPACK and BLAS numerical libraries (which many Linux distros have pre-packaged) [http://www.netlib.org/lapack] [http://www.netlib.org/blas]

- The Intel Computer Vision Library available at [http://www.intel.com/software/products/opensource/libraries] [http://opencv.sourceforge.net]

2.2 Starting Lush

If the lush executable was installed in your path, you can start Lush by simply typing `lush2` at the Unix prompt. Otherwise you can make a symbolic link from anywhere in your path to where the lush executable resides (most likely `/usr/local/lush/bin/lush2` or `/wherever-you-installed-lush/bin/lush2`)

```
 LUSH Lisp Universal Shell 2.0 (built Oct 18 2009)
   Copyright (C) 2009 Leon Bottou, Yann LeCun, Ralf Juengling.
   Copyright (C) 2002 Leon Bottou, Yann LeCun, AT&T Corp, NECI.
 Includes parts of TL3:
   Copyright (C) 1987-1999 Leon Bottou and Neuristique.
 Includes selected parts of SN3.2:
   Copyright (C) 1991-2001 AT&T Corp.

This program is free software distributed under the terms of
the Lesser GNU Public Licence (LGPL) with ABSOLUTELY NO WARRANTY.

Type '(helptool)' to get started.
?
```

To exit Lush, press Control-D or enter `(exit)` at the Lush prompt.

If you have `emacs` installed on your system, it is highly recommended to run Lush from within Emacs so that you may enjoy such niceties as command-line editing, history, and IDE-like features. See the next section "Configuration" for details on how to do this.

2.3 Configuration

Lush can be run from the shell prompt. It provides command line editing, history, and symbol and file name completion.

Many will prefer to run Lush under Emacs. To make this convenient, the following line should be added to the .emacs file in your home directory:

```
(load "your-lush-dir/etc/lush.el")
```

This file provides two functionalities in Emacs:

- The ability to start Lush within Emacs by typing `Meta-X lush` (META-X is emacs-speak for ESC-X or ALT-X). The lush executable must be in your shell path for this to work. Lush will run in Emacs's so-called "inferior-lisp" mode. The current directory in which Lush is started is the directory of the file in the buffer you were editing right before starting Lush. It is convenient to split the Emacs window into two. The bottom

window will be the Lush execution buffer (called "*inferior-lisp*") while the top window will contain the Lush source file being worked on.

- A Lush mode for Emacs that is automatically activated when a `.lsh` file is being edited, which provides keyword highlighting, automatic indentation, matching paren flashing, etc.

Emacs's Lisp/Lush mode provides a set of nice commands to facilitate Lisp programming. Among them are C-M-e (Ctrl-Meta-e) to jump to the end of a function, C-M-a to jump to the beginning, C-M-q to indent/format a function, and C-M-x to send the function or expression in which the cursor is to the interpreter (assuming it is running in the *inferior-lisp* buffer).

2.4 Reporting Bugs and Problems

If you encounter a bug or want to signal a problem, go to: [https://sourceforge.net/apps/trac/lush/newticket]fill out the form and click "Create Ticket".

If you are not sure if something is a bug or a feature, you might want to ask the friendly folks in lush-users@lists.sourceforge.net. To sign up on the list go to [https://lists.sourceforge.net/lists/listinfo/lush-users].

2.5 Getting Help and Documentation Browsing

Lush has an on-line documentation system. It can be invoked by typing `(helptool)` at the Lush prompt, which will popup a graphic browser window with the manual. A non-graphic version of the on-line help can be invoked by typing `^Atopic` for help on functions or concepts whose name include `topic` (the ^A can be typed either as Caret and A or as Control-A, but the latter form is inconvenient when running Lush within Emacs). Here is an example:

```
? ^Aregex
Search Results for: regex
  1. Regular Expressions (regex).
  2. (regex-match <r> <s>)
  3. (regex-extract <r> <s>)
  4. (regex-seek <r> <s> [<start>])
  5. (regex-subst <r> <s> <str>)
  6. (regex-rseek <r> <s> [<n> [<gr>]])
  7. (regex-split  <r> <s> [<n> [<gr> [<neg>]]])
  8. (regex-skip  <r> <s> [<n> [<gr> [<neg>]]])
  9. (regex-count <r> <s>)
 10. (regex-tail <r> <s> [<n> [<gr> [<neg>]]])
 11. (regex-member <rl> <s>)
 12. (glob <regex> <l>)
choice? 2
```

```
------------------------------------------------------------------------
(regex-match <r> <s>)                                  [DX]
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

 Returns <t> if regular expression <r> exactly matches the entire string
<s>.  Returns the empty list otherwise.
 Example:
? (regex_match "(+|-)?[0-9]+(\\\\.[0-9]*)?" "-56")
= t

= ()
?
```

The list of symbols whose name contains a particular string can be listed using "ˆSsymbolstring".

```
? ^Sregex
cleanup-regex      regex              regex-count        regex-member
regex-rseek        regex-skip         regex-split        regex-tail
regex-extract      regex-match        regex-seek         regex-subst

= ()
```

2.6 Basic Syntax

Lush (or any Lisp dialect) might be one of the easiest-to-learn language you will ever encounter. The big idea is that every expression is a list enclosed in parentheses which, when "evaluated", returns a result. The first element of the list is the name of function, and the remaining elements are its arguments. So instead of typing `sqrt(4)` to call the square root function like in C, you type `(sqrt 4)` .

Here is an example of how an expression typed at the Lush prompt is evaluated and its result printed:

```
? (sqrt 4)
= 2
```

The list notation is also used for what we generally think of as infix operators, which in Lush (and other Lisps) are just functions like any other. Instead of writing `3+4` , we write `(+ 3 4)` :

```
? (+ 3 4)
= 7
```

Arguments of most functions are evaluated before the function is called, which allows nested expression:

```
? (sqrt (+ 3 4))
= 2.6458
```

Words like `sqrt` or `+` in the previous example are called symbols. Symbols can be assigned a value. The value of the symbol `sqrt` in the above example is a function that returns the square root of its argument. A symbol can be used to refer to any lisp object, and therefore it is the basic mechanism for variables. Binding an object to a symbol is generally performed with the function `setq` :

```
? (setq x 5)
= 5
? (* x x)
= 25
? (setq x "cou")
= "cou"
? (setq x (concat x x))
= "coucou"
```

When not bound to anything, a symbol evaluates to " `()` ", the empty list. `()` is a special but ubituous object. For instance, it is used to represent the value "false" returned by a boolean expression like

```
(> (- 2 4) 1)
= ()
```

Variables can be set to any type of Lush object. In interpreted mode, the types of the variables need not be declared, but they must be declared if the code is to be compiled (more on that later). Examples of Lush objects are numbers, matrices and vectors, strings, lists, functions, macros, classes, and hash tables.

2.6.1 Symbol Names and Special Characters

Symbols in Lush are case-sensitive (i.e. `SQRT` and `sqrt` are different names). Symbol names cannot contain certain characters: parentheses, spaces, forward and backward quotes, commas, curly braces, square brackets, hash, caret, tilde, and control characters. Those characters play special roles as we will see later.

By convention, sub-words of a symbol name are often delimited by hyphens and occasionally by dots:

```
gsl-my-new-pointer   brace.read
```

2.7 Basic Types

Lush has a small number of basic types, a number of not-so-basic built-in types, and a mechanism for creating new types (classes). The basic types include numbers, generic pointers, and list cells ("cons"). The not-so-basic types include strings, symbols, seven major types of functions (called DE, DF, DM, DX, DY, DH, and DZ), nine types of vector storage structures (for storing bytes, unsigned bytes, shorts, ints, floats, doubles, generic pointers, and lisp objects), one type of vector/matrix/tensor access structure, two types of file descriptors, hash tables, graphic windows, classes, and dates. New types with slots and methods (a la C++) can also be defined by deriving from the base class `object` .

Some of the types can be entered as literals:

- numbers: `(setq x -3.4e6)`
- strings: `(setq x "coucou")`
- symbols: `(setq x 'coucou)`
- lists: `(setq x '(1 2 3))`

Numbers and strings have the interesting property that they evaluate to themselves, that is, they may be written literally as arguments in expressions. On the other hand, symbols evaluate to their assigned value, and lists evaluate to the result of applying their first element (interpreted as a function) to the rest of the elements (the arguments). That's why the symbol and the list in the above example are preceded by a quote. The quote prevents what follows from being evaluated. Without the quote `coucou` would return its value (or `()` if it has no value), while `(1 2 3)` would produce an error because `1` is not a function.

2.8 Defining functions

Functions can be defined in many ways, but the most common one is to use the `defun` construct. The general form is:

```
(defun <name> (<arg1>....<argn>) <body1>....<bodym>)
```

Here is a specific example:

```
? (defun square (x) (* x x))
= square
? (square 4)
= 16
```

`Defun` means "define function" and is followed by the function name `square` , a list of formal arguments `(x)` , and the body of the function `(* x x)` , which means multiply `x` by itself and return the result. The return value of the

function is the value of the last expression in the function body (in the example abovert, the body consists of only one list expression).

The definition of a particular function can be displayed by typing `^Pfunctionname` :

```
? ^Psquare
(lambda (x)
  (* x x) )
= t
```

Note: A `lambda` expression is a function, it's the value bound to symbol `square` in the above example. Using `defun` as above is just short hand for

```
(setq square (lambda (x) (* x x)))
```

The directive "^P", like "^A", is a so-called macro-character. Macro-characters are special combinations of characters that are translated into function calls by the interpreter. Lush has several predefined macro-characters to reduce typing for several commonly used interactive functionalities. A particularly useful one is "^Lfilename" which loads a Lisp source file (more on this later).

2.9 Loops, Conditionals, Local Variables, and Such

Lush provides traditional loop constructs such as `for` , `while` , `repeat` , an so on. Here is an example of how to compute the sum of the inverses of the first 10 integers (the 10th term of the harmonic series).

```
? (setq z 0)
= 0
? (setq i 0)
= 0
? (while (< i 10) (incr i) (incr z (/ i)))
= 2.929
```

The function `incr` increments its first argument by the value of its second argument, or by 1 if no second argument is given, and returns the result. The function `/` called with one argument returns its inverse.

The above example is quite awful because it leaves two global variables `i` and `z` laying around for no reason. A cleaner way of doing things consists in using temporary variables that are automatically destroyed when they are no longer needed. This is done with the `let` construct:

```
(let ((z 0) (i 0))
  (while (< i 10) (incr i) (incr z (/ i))))
```

In its most general form, the `let` construct is used as follows:

```
(let ((<var1> <value1>) ... (<varN> <valueN>))
      <body1> ... <bodyP>)
```

It creates temporary local variables `var1` ... `varN` , sets them to the values `value1` ... `valueN` respectively, evaluates `body1` ... `bodyP` and returns the result of the evaluation of `bodyP` . The values of the variables are destroyed upon exiting the `let` . In fact, if a temporary variable created in a `let` has the same name as a preexisting variable, the preexisting value will be temporary inaccessible during the life of the temporary variable. The preexisting value will be restored upon exit:

```
(setq z 345)
= 345
?  (let ((z 0) (i 0)) (while (< i 10) (incr i) (incr z (/ i))))
= 2.929
? z
= 345
```

The code above can be encapsulated in a function using the `defun` construct:

```
? ;; compute the n-th term of the harmonic series
? (defun harmonic (n)
    (let ((z 0) (i 0)) (while (< i n) (incr i) (incr z (/ i)))))
= harmonic
? (harmonic 10)
= 2.929
? (printf "\\%18.14f\\n" (harmonic 10))
  2.92896825396825
= ()
```

In Lush, the text that follows one or several semi-colons on a line is taken as comments and ignored by the interpreter.

Conditional constructs include `if` , `when` , `cond` , and `selectq` . The basic form of the `if` construct is as follows:

```
(if <conditional-expression>
    <evaluate-if-non-nil>
  <evaluates-if-nil-1>
  <evaluates-if-nil-2>
  .....
  <evaluates-if-nil-n>)
```

If the conditional expression evaluates to something other than `()` , expression `evaluate-if-non-nil` is evaluated and its result returned. If it evaluates to `()` , the remaining expressions are evaluated, and the result of the last one is the returned. Multiple expressions can be grouped in the `evaluate-if-non-nil` part by grouping them in a `progn` construct: `(progn expr1 ... exprN )` . To illustrate the use of `if` , here is an inefficient but easy to grasp implementation of a function that compute the n-th Fibonacci number:

```
(defun fibo (n)
  (if (= n 0)            ; test if n equals 0
     0                   ; if yes, return 0
    (if (= n 1)          ; else test if n equals 1
       1                 ; if yes return 1
      (+ (fibo (- n 1)) (fibo (- n 2))))))
```

2.9.1 Loops over structured objects

Lush provides a generic loop construct, `do` , to cycle through all elements of a list, array, hash table etc. The general form is follows

```
(do ((e1 o1) [ (e2 o2) [...(en on) ]]) body)
```

where `e1` is subsequently bound to each element in object `o1` , `e2` is bound to each element in `o2` , etc, and the loop body is evaluated for each such "simultaneous binding".

Here is an example:

```
? (setq names (list "one" "two" "three"))
= ("one" "two" "three")
? (setq vals (list 1 2 3 4 5))
= (1 2 3 4 5)
? (do ((n names) (v vals)) (printf "the value of \\%s is \\%d\\n" n v))
the value of one is 1
the value of two is 2
the value of three is 3
= ()
```

Note that the loop terminates with the last element in any of the objects `o1` ,... , `on` .

What does it mean to loop over an array? When `do` is given an array, it takes it apart along the first dimension and binds the subarrays to the loop variable. A variation of the above example is:

```
? (setq vals [1 2 3 4 5])
```

```
= [d   1.0000   2.0000   3.0000   4.0000   5.0000]
? (do ((n names) (v vals)) (printf "the value of \\%s is \\%d\\n" n (v)))
the value of one is 1
the value of two is 2
the value of three is 3
= ()
?
```

For arrays there are other, more specialized loop constructs. One is `idx-bloop` , which behaves similar to `do` but may be used with arrays only. A similar function `idx-eloop` iterates on the last dimension of an array instead of the first dimension. More on arrays below.

2.10 Compiling Functions

To improve execution speed, Lush functions may be compiled. Some restrictions apply, however. For functions to be compilable, the function definition needs to contain type annotations for variables and use subset of the Lush language that the compiler understands.

For starters, to avoid inefficient run-time type-checking in compiled code, one must define the type of all the variables before a function can be compiled. A compilable version of `harmonic` reads:

```
? (defun harmonic (n)
     (declare (-double-) n)
     (let ((z 0) (i 0))
       (declare (-double-) z i)
        (while (< i n) (incr i) (incr z (/ i))) ))
= harmonic
```

The lines `(declare (-double-) n)` and `(declare (-double-) z i)` declare the type of the argument `n` and the local variables `z` , and `i` as double precision floating point numbers. Before we compile the function, let's measure the number of CPU seconds required to compute the sum of the million-th harmonic sum in interpreted mode (this is on a 800MHz Pentium running Linux):

```
? (cputime (harmonic 1000000))
= 2.05
```

We can compile the function by simply typing `(dhc-make "harm" harmonic)` . Lush will translate the Lisp code into C and write the C source into the file "C/harm.c". The C compiler will then compile the code and generate the object file "C/ `architecture` /file.o", where `architecture` is the name of the architecture on which Lush is currently running. Then, Lush's dynamic loader will automagically load the object code back into the interpreter.

```
(dhc-make "harm" harmonic)
Preprocessing and parsing harmonic ...
Generating C for harmonic ...
gcc -DHAVE_CONFIG_H -I/home/yann/lush/include -DNO_DEBUG -O3 -mcpu=i686 -pthread -c /tmp/C/junk.c
= "/tmp/C/i686-pc-linux-gnu/harm.o"
```

Now let's see how many CPU seconds it takes to execute the compiled version of the code. We need to repeat the code a few times to get a meaningful figure:

```
  ? (cputime (repeat 50 (harmonic 1000000)))
  = 2.37
```

The speedup over the interpreted version is around 45. An important point to remember is that the interpreter always treats numbers as double precision floating point, while numbers in compiled code are treated as specified by their type declaration. Lush currently understands the following number types: `-uchar-`, `-char-`, `-short-`, `-int-`, `-float-`, `-double-` . They correspond, respectively, to the C types `unsigned char`, `signed char`, `short`, `int`, `float`, and `double` .

Out of curiosity, let's look at the C code generated by Lush's compiler in `current-dir/C/harm.c` :

```
/*
 * FUNCTION harmonic
 */
extern_c real
C_harmonic (real L1_n)
{
  TRACE_PUSH ("C_harmonic");
  {
    double L_Tmp0;
    {
      double L2_1_z;
      double L2_2_i;
      L2_1_z = 0;
      L2_2_i = 0;
      L_Tmp0 = 0;
/* While loop */
      goto L_1;
    L_0:
      {
       L2_2_i = (L2_2_i + 1);
       L2_1_z = (L2_1_z + (1 / (double) L2_2_i));
       L_Tmp0 = L2_1_z;
```

```
      }
/* While loop test*/
    L_1:
      {
       if ((L2_2_i < L1_n))
         goto L_0;
      }
    }
    TRACE_POP ("C_harmonic");
    return L_Tmp0;
  }
}
```

2.11 Program Files

The normal way to write Lush code is to open a file, say "toto.lsh", in your favorite editor, and to write all the necessary function definitions and compilation instructions into it. The file can then be loaded into the interpreter with `(libload "toto")` . For example, a Lush file for the above function would contain:

```
;; a function that computes the sum of the inverses of the first n integers.
(defun harmonic (n)
   (declare (-double-) n)
   (let ((z 0) (i 0))
     (declare (-double-) z i)
      (while (< i n) (incr i) (incr z (/ i)))))

;; compile the function to toto.c and toto.o
;; if the first arg is nil, the .c file has the same
;; base name as the file being loaded.
(dhc-make () harmonic)
```

When loading this file, Lush processes its content as if it were typed at the prompt. Therefore, loading this file will define harmonic, and then compile it. If you subsequently reload the same file, the compilation will only occur if the source file `toto.lsh` was modified since the last compilatione.

The use of Emacs to edit Lush files is strongly recommended as it has a special Lisp mode that flashes matching parentheses, highlights keywords and comments, and automatically indents Lisp expressions (using ALT-CTRL-Q). More details on how to setup Emacs are given in the "setting things up" section of this tutorial.

2.12 On-Line Documentation

As we said earlier, Lush possesses an on-line documentation system. This system can be used by Lush users to document their own programs. Its use is strongly recommended. Documenting functions or other objects can be done with the "#?" macro-character. Here is what `toto.lsh` should have looked like in the first place:

```
#? (harmonic <n>)
;; Sum of the inverses of the first <n> integers.
? (defun harmonic (n)
     (declare (-double-) n)
     (let ((z 0) (i 0))
       (declare (-double-) z i)
        (while (< i n) (incr i) (incr z (/ i)))))

;; compile the function to toto.c and toto.o
(dhc-make () harmonic)
```

The string after the "#?" is what will be searched when search feature of (helptool) is invoked on this file. Contiguous comment lines immediately following the "#?" line constitute the text of the help. After this file is loaded, the on-line help for `harmonic` and all the functions in the same file can be displayed with:

```
(helptool "toto.lsh")
```

2.13 Multiple File Programs and Search Path

Sometimes, a Lisp program requires other Lisp files to be loaded in order to run. For example imagine that we have written a set of signal processing functions in the file `"/home/yann/lsh/dsp.lsh"` , and now we are writing a program `"tutu.lsh"` that uses some of those functions. The beginning of `"tutu.lsh"` would look like the following

```
#? *** My Cool Signal Processing Application
;; this is my really cool dsp application.
(libload "/home/yann/lsh/dsp")
.....
```

(the `.lsh` extension is automatically added if required).

The function `libload` has an interesting feature: it remembers which files were already libloaded and loads them only once into the interpreter (subsequent calls to libload with a previously libloaded file will have no effect). There is

an exception to this behavior: `libload` keeps track of the tree of dependencies between Lush program files, and also keeps track of the times of last modification of each file. When a file is loaded with `libload` , all the files on which this file depends are also loaded, but only if they have been modified since last time they were libloaded.

Here is a possible scenario: file A libloads files B and C. File A is libloaded, as a result file B and C are libloaded. File B is modified. File A is libloaded again, as a result B is re-libloaded, and A is re-libloaded. C is not re-libloaded since it was previously libloaded and was not modified. This behavior is recursive (it applies to all files directly or inderectly libloaded by A).

In certain cases, the lower-level function `load` may be preferable. `load` simply loads a file:

```
? (load "/home/yann/lsh/dsp")
```

Putting absolute path names in a program is generally a bad idea since programs have a pernicious tendency to move from time to time. The solution is to rely on Lush's search path mechanism. When any loading directive is executed, Lush looks for the file in the current directory, and then in the search path. The default search path includes `lush2/local` , `lush2/packages` , `lush2/lsh` , `lush2/sys` and several subdirectories of `lush2/lsh` . The search path can be user-extended with the `addpath` function:

```
(addpath "/home/yann/lsh")
```

now, loading `"dsp.lsh"` only requires:

```
(libload "dsp")
```

2.14 Calling C Functions from Lush

The ability to freely mix Lisp and C code is one of the most interesting features of Lush. Let's say you have written a C file called `titi.c` with the following content:

```
float sq(float x)
{
  return x*x;
}
```

You have compiled the file and produced the object file `titi.o` . Calling `sq` from the Lush interpreter is as simple as the following. First, dynamically load the object file into Lush

```
? (mod-load "titi.o")
```

then, write a lisp function whose only purpose is to call `sq` .

```
? (defun square (x)
     (declare (-float-) x)
     (cpheader "extern float sq(float);")
     (to-float #{ sq( $x ) #} ))
```

Here is what the above means: `(defun square (x) ...)` means define a new Lisp function called `square` with a single argument `x` . `(declare (-float-) x)` simply declares `x` as float. `(to-float ....)` converts its argument to float (like a cast in C). The sequence `# .... #` allows to insert C code within Lisp code. Therefore `# sq( $x ) #` simply calls the C function `sq` and returns the result. Lisp variables can be inserted in in-line C code by prepending a dollar sign, thus `$x` refers to the Lisp float variable `x` . The `cpheader` directive allows one to include a string in the "header" section of the C files generated by the Lisp to C compiler. We use it here to specify the type of the argument and the return value of `sq` .

We can now compile the above function using:

```
? (dhc-make () square)
```

Two file `C/square.c` and `C/i686-pc-linux-gnu/square.o` will be generated with the C source and the object code for `square` . The object code will be automatically loaded into Lush. Now `square` can be called from the Lisp interpreter:

```
? (square 5)
= 25
```

In the above example, the `to-float` casting function was used to tell the compiler what type the returned value has. The following "cast" functions are available: `to-double` , `to-float` , `to-int` , `to-bool` , `to-gptr` , `to-mptr` , `to-char` , `to-uchar` , `to-str` .

2.15 Mixing Lisp and In-Line C Code

If we do not have a ready-made C file for our function, we can use the inline C capability to write code in C right inside the Lisp function. Here is how we could write the familiar `harmonic` function this way:

```
(defun harmonic (n)
```

```
    (declare (-double-) n)
    (let ((r 0))
      (declare (-double-) r)
      #{ double i=0;
         while (i<$n){ i+=1.0; $r+=1/i; };
       #} r))
(dhc-make () harmonic)
Preprocessing and parsing harmonic ...
Generating C for harmonic ...
gcc -DHAVE_CONFIG_H -I/home/yann/lush/include   -DNO_DEBUG -Wall -O3 -funroll-loops -m
= "/tmp/C/i686-pc-linux-gnu/junk.o"
? (time (repeat 50 (harmonic 1000000)))
= 2.36
```

That's about the same time as the version written in compiled Lisp. In this case, writing directly in inline C was not particularly advantageous, but there are cases where writing in Lisp is inefficient, awkward, or even outright impractical. In these cases one can resort to inline C implementations.

2.16 Lists

Lush belongs to the Lisp family, and therefore has lots of functions to manipulate lists. In fact, Lush programs themselves are lists. Lists are sequences of Lush objects enclosed in parentheses. Things like `(+ 1 2)` is a list. However, when the list `(+ 1 2)` is entered at the Lush prompt, it is immediately evaluated and discarded right away:

```
? (+ 1 2)
= 3
```

To create a list, we must prevent the evaluation from happening. This is done with the `quote` function or the "'" (quote) macro-character:

```
(quote (+ 1 2))
= (+ 1 2)
? '(+ 1 2)
= (+ 1 2)
```

Lists can be put into variables like any other Lisp object:

```
(setq x '(+ 1 2))
```

List elements can be any lisp object, including other lists. Here is a list with a number, a string, a list of numbers, a symbol, and a list, and a number:

```
(setq x '(1 "coucou" (4 5 6) asd (7 (8 9) 10) 20))
```

Here is another example where the same list as above is assembled from its members using the `list` function:

```
(setq x (list 1 "coucou" '(4 5 6) 'asd '(7 (8 9) 10) 20))
```

The first element (the head) of a list can be obtained with the `car` function, and the rest of the list (tail) with the `cdr` function (pronounced kudder). The `cons` function can be used to put a head and a tail together:

```
? (setq x '(+ 1 2))
= (+ 1 2)
? (car x)
= +
? (cdr x)
= (1 2)
? (setq z (cons '+ '(1 2)))
= (+ 1 2)
```

A list can be evaluated:

```
(setq x '(+ 1 2))
= (+ 1 2)
? (eval x)
= 3
```

Operations can be performed on all the elements of a list using the `dolist` construct:

```
? (setq l '(1 3 5 7 9))
= (1 3 5 7 9)
? (dolist (v l) (print v))
1
3
5
7
9
= ()
```

The `domapcar` construct is useful to accumulate results (the funny name comes from `mapcar` , a Lisp function):

```
? (setq l '(1 3 5 7 9))
= (1 3 5 7 9)
? (domapcar ((v l)) (- v 1))
= (0 2 4 6 8)
```

This says, make a temporary variable `v` , and evaluate the body of `domapcar` with `v` set to each element of `l` in sequence. Finally, return a list of all the results.

This section only scratched the surface of what can be done with lists. The reader is referred to the List section in the "Core Interpreter" chapter of the Lush manual.

2.17 Strings and Regular Expressions

[under construction]

2.18 Scalars, Vectors, Matrices, Tensors, and Storages

Lush's piece de resistance is its array engine. Lush can operate on scalars, vectors, matrices, or high-dimensional arrays from 0 to 8 dimensions. Creating an array of `float` is done simply with:

```
? (setq m (float-array 10 8 4))   ; create 3D array
= ::INDEX3:<10x8x4>
? (m 3 4 2 45.6)       ; set value of element (3,4,2) to 45.6
= ::INDEX3:<10x8x4>
? (m 3 4 2)            ; get value of element (3,4,2).
= 45.6
```

Arrays of various basic types (mostly numerical C types) can be created with the functions listed below. The array elements are initialized to zero for arrays of numerical type, and to `nil` for atom arrays.

- `double-array` : C double
- `float-array` : C float
- `int-array` : C int (signed)
- `short-array` : C int (signed)
- `char-array` : C char
- `uchar-array` , C unsigned char

- `gptr-array` : generic pointer (C void *)
- `mptr-array` : managed C pointer

All these functions take 0 to 8 integer arguments that are the sizes in each dimension. An array with 0 dimensions is called a scalar. It is different from a regular number in that it behaves like a pointer to a number (that is, the value of a scalar passed as argument to a function can be modified by that function).

Arrays of doubles can be entered literally using square brackets:

```
(setq m [[0 2 3 4][5 6 7 8]])
```

Arrays of other types can be specified by adjoining an apropriate character after the open bracket. Here is how to create a vector of floats:

```
(setq m [f 0 2 3 4])
```

Characters `f, i, s, c, u, a` specify float, int, short, char, uchar, and atom respectively. Here is an example with atoms:

```
(setq m [a "choucroute" "garnie"])
```

Arrays are really composed of two separate entities:

- a `storage` which contains the following fields: a pointer to the actual data, an element type identifier (and the size thereof), and flags that indicate if the data is writable or not, if it is in RAM or memory-mapped from a file.
- an `index` that points to a `storage` and contains the following fields: the number of dimensions of the array, the size in each dimension, the address increment from one element to the next in each dimension (called `modulo`), and the offset of the first array element in the `storage` .

This organization of arrays allows different `index` structures to point to the same `storage` thereby allowing the data to be accessed in multiple ways without copying.

2.18.1 Working with Storages and Indexes.

Storages and indexes can be created independently of each other. So for example `(setq s (double-storage 8))` creates a new storage of doubles with 8 elements. The call `(double-storage)` returns an empty storage object. Creating an index on a particular storage `s` can be done with `(new-index s '(3 4))` . When creating an array this way, the storage must either be empty or have at least as many entries as the index provides or an error is raised.

When directly accessing storage elements, storages behave like 1D indexes (vectors). Storages can be created, allocated in memory, or memory-mapped to a file. Storage creation function are provided for the following types: double, float, int, short, char, unsigned char, generic pointer, and lisp object.

One may copy an index via `copy-index` and the copy will point to the same storage. Function `copy-array` on the other hand creates a copy of both, the index and the storage. More precisely, `copy-array` creates a new storage just big enough to fit all array elements as seen through the index. Because `copy-array` is needed so often, there is a special macro character `##` for it:

```
? (setq v [1 2 3])
= [d   1.0000   2.0000   3.0000]
? ##v
= [d   1.0000   2.0000   3.0000]
```

Here are sample uses of a few more common array-related functions, their meaning should be obvious:

```
? (setq m (arange 12))
= [d   1.0000   2.0000   3.0000   4.0000   5.0000   6.0000   7.0000   8.0000   9.0000

? ($ m)
= [i    12]

? (setq m ($$ m [3 4]))
= [d[d   1.0000   2.0000   3.0000   4.0000]
    [d   5.0000   6.0000   7.0000   8.0000]
    [d   9.0000  10.0000  11.0000  12.0000]]

? ($ m)
= [i     3     4]

$? (rank m)
= 2

? ($t m)
= [d[d   1.0000   5.0000   9.0000]
    [d   2.0000   6.0000  10.0000]
    [d   3.0000   7.0000  11.0000]
    [d   4.0000   8.0000  12.0000]]

? (idx-storage m)
= ::DoubleStorage:managed@0x826a768:<12>
```

2.18.2 Index Manipulations

Several functions are provided to conveniently manipulate the index structure itself. These functions do not manipulate the actual data (and therefore are efficient), they merely derive new index from an existing index to access the data in new ways.

`(idx-trim m n 4 12)` creates an index from `m` where the size in the `n` -th dimension is reduced to 12 elements, starting with element 4 in m. This can be used to select certain subsets of array elements:

```
? (setq m [[0 2 3 4][5 6 7 8]])
= [[ 0.00  2.00  3.00  4.00 ]
   [ 5.00  6.00  7.00  8.00 ]]
? (idx-trim m 1 1 2)
= [[ 2.00  3.00 ]
   [ 6.00  7.00 ]]
```

The select function `$*0` creates a new index from `m` with the first dimension removed, and which is the `s` -th "slice" of `m` . For example, the 3rd row of matrix `m` in the above example (which is a vector) can be obtained with

```
? ($*0 m 2)
= [ 3.00  7.00 ]
```

Select functions for the dimensions are named accordingly (`$*1` , `$*2` , etc).

Another often-needed manipulation is to flatten an array to a vector. This is always possible for a newly created array but not necessarily for an index accessing only parts of a storage. The flatten function `$-` flattens an array and the order of the elements in the resulting vector is the order they are found in memory (row-major order):

```
? (setq m [[1 2 3][4 5 6]])
= [d[d   1.0000   2.0000   3.0000]
    [d   4.0000   5.0000   6.0000]]
? ($- m)
= [d   1.0000   2.0000   3.0000   4.0000   5.0000   6.0000]
?
```

An interesting feature of the index structure is that an index may have more entries than a storage. This means several index entries reference the same memory cell in the storage. There are two simple functions that create such an index,

```
? (setq v [1 2 3])
= [d   1.0000   2.0000   3.0000]
```

```
? ($> v)
= [d[d   1.0000]
    [d   2.0000]
    [d   3.0000]]

? ($< v)
= [d[d   1.0000   2.0000   3.0000]]

? ($ ($> v))
= [i     3     1]

? ($ ($< v))
= [i     1     3]

? ($> v 3)
= [d[d   1.0000   1.0000   1.0000]
    [d   2.0000   2.0000   2.0000]
    [d   3.0000   3.0000   3.0000]]

? ($> v 3 3)
= [d[d[d   1.0000   1.0000   1.0000]
      [d   1.0000   1.0000   1.0000]
      [d   1.0000   1.0000   1.0000]]
    [d[d   2.0000   2.0000   2.0000]
      [d   2.0000   2.0000   2.0000]
      [d   2.0000   2.0000   2.0000]]
    [d[d   3.0000   3.0000   3.0000]
      [d   3.0000   3.0000   3.0000]
      [d   3.0000   3.0000   3.0000]]]
```

2.18.3 Simple Array Operations

Lush has lots of functions that operate on arrays of all types. These include component-wise unary and binary operations (applying a function to all the elements of an array or two pairs of corresponding elements in two arrays), such as `abs, atan, cos, sin, exp, log, + , -, *, ....` .

```
? (setq m [[1 2 3][4 5 6]])
= [d[d  1.00  2.00  3.00]
    [d  4.00  5.00  6.00]]
? (+ m (* m m))
= [d[d  2.00  6.00 12.00]
    [d 20.00 30.00 42.00]]
?
```

Currently, expressions such as `(+ m (* m m))` cannot be compiled when `m` is an array. For many such expressions, compilable versions can be constructed with special array functions. A compilable expression equivalent to the above is `(idx-add m (idx-mul m m))` . There are also some special dyadic operations where the second argument is a scalar, such as `(idx-addm0 m s [r])` which adds scalar `s` to each element of tensor `m` , or `(idx-addm0acc m s r)` which does the same thing but accumulates the result in `r` , or `(idx-dotm0 m1 s [r])` which multiplies each element of `m` by scalar `s` .

2.19 Matrices and Vectors

The notion of matrix and vector is slightly different in matrix calculus texts, where everything is a matrix and a distinction between "row vectors" and "column vectors" is being made. Lush comes with a set of functions to facilitate matrix calculus operations. For example:

```
? (setq rv (mat-row 1 0 -1))
= [d[d  1.00  0.00 -1.00]]
? (setq cv (mat-col 1 2 3))
= [d[d  1.00]
    [d  2.00]
    [d  3.00]]
? (mat-.* rv cv) ; inner product
= [d[d -2.00]]
? (mat-.* cv rv) ; outer product
= [d[d  1.00  0.00 -1.00]
    [d  2.00  0.00 -2.00]
    [d  3.00  0.00 -3.00]]
```

See the section on "Matrix Functions" for more information.

2.19.1 Special Inner and Outer Products

There are several functions for performing matrix-vector products, outer products, and such. They include: `(idx-m1extm1 m1 m2 [r])` (outer product of vectors), `(idx-m2dotm1 m v [r])` (matrix-vector multiply), `(idx-m2extm2 m1 m2 [r])` (outer product of matrices which gives a 4D tensor), `(idx-m4dotm2 m v [r])` (product of a 4D-tensor by a matrix where the last 2 dimension of the tensor are contract with the first two dimensions of the matrix). `idx-m4dotm2` can be conveniently combined with appropriate calls to `unfold` to perform 2D convolutions, as demonstrated in the libary file `lush/lsh/libidx/idx-convol.lsh` .

2.19.2 Simple Linear Algebra

Lush provides a interfaces to the Gnu Scientific Library as well as to BLAS and LAPACK, which contain all the linear algebra anyone could wish for. Using those functions directly may be a bit complicated, so lush provides a library with simple forms of the most common functions. This library can be loaded with `(libload "libnum/linalgebra")` . It contains the following functions:

- `(eigen-symm m)` : eigenvalues of real symmetric matrix
- `(eigen-symmv m)` : eigenvalues and eigenvectors of real symmetric matrix
- `(eigen-herm m)` : eigenvalues of complex hermitian matrix
- `(eigen-hermv m)` : eigenvalues and eigenvectors of complex hermitian matrix
- `(svd a v s)` : singular value decomposition of real matrix
- `(solve-hh a b)` : linear system solver (householder method)
- `(solve-lu a b)` : linear system solver (LU decomposition method)
- `(solve-lu-complex a b)` : complex linear system solver (LU decomposition method)
- `(solve-sv a b)` : linear system solver/least square solution (SVD method)
- `(inverse-lu a i)` : matrix inverse (LU decomposition method)
- `(determinant a)` : determinant
- `(determinant-log a)` : log determinant
- `(determinant-sign a)` : sign of determinant

2.20 Object-Oriented Lush Programming

Lush is an object-oriented language that allows users to define hierarchies of classes with inheritance, slots, and methods. As mentioned above, Lush comes with a number of built-in immutable classes such as numbers, list cells, strings, symbols, etc, for which users defined methods, but which may not be used as "superclasses" of user-defined classes. This section is about user-defined classes.

2.20.1 Defining New Classes

User defined classes are derived from the root class `object` with constructs like this:

```
(defclass Rectangle object width leng)
```

This can be interpreted as: define the class `rectangle` as a subclass of `object` with two slots `width` and `leng` . Subclasses inherit the slots of their superclass.

A new instance of the class `Rectangle` can be created with the function `new` :

```
(setq r (new Rectangle))
```

By default, the slots are left unfilled (set to the empty list). Accessing slots of an object can be done using the special "scope syntax":

```
? (setq :r:width 10)
= 10
? (setq :r:leng 3)
= 3
? ^Pr
;;*** ::Rectangle:819fa08, INSTANCE OF ::class:Rectangle
;; FROM CLASS: Rectangle
;;      width=10
;;      leng=3
;; FROM CLASS: object
```

Classes can be given methods (like virtual member functions in C++) defined as in the following example:

```
(defmethod Rectangle area () (* leng width))
```

This defines the method `area` for the class `Rectangle` which takes no argument, and returns the product of the slots `leng` and `width` . This is essentially similar to defining a function with `defun` , except that within the body of a method, the slots of the object are directly accessible by name as if they were local variables. Calling the above method can be done as follows:

```
? (==> r area)
= 30
```

which can be read as "send to `r` the message `area` ".

Methods may take arguments. Arguments may be passed to a method along with a message. A method may also refer to the current object through the special variable `this` . Here is an example of how to compute the volume of a box whose base is the current object and height is passed as argument:

```
? (defmethod Rectangle volume (height)
    (* height (==> this area)) )
= volume
? (==> r volume 10)
= 300
```

A helpful coding convention is to use CamelCase for class names. Lush's builtin classes and classes defined in the libraries adhere to this convention. The only exceptions are names of abstract classes that are never instantiated directly (for example `object` , `iterator` , `function`).

2.20.2 Inheritance

Lush allows to create subclasses of existing classes. Subclasses inherit the slots and methods of their parent class. For example, we can define a `Box` class, derived from the `Rectangle` class, and containing a `height` slot in addition the inherited `width` and `leng` :

```
? (defclass Box Rectangle height)
= Box
```

Setting the slots of an object directly as we did in the `rectangle` example is a little awkward. A better alternative is to fill the slots when the object is created. This can be done by defining a constructor method. The constructor is a method that has the same name as the class. It is called whenever the `new` function is called to create an object of that class. Here is a constructor for `box` :

```
? (defmethod Box Box (l w h)
    (setq leng l  width w  height h) )
= box
?  (setq monolith (new Box 9 4 1))
= ::box:8138958
? ^Pmonolith
;;*** ::box:8138958, INSTANCE OF ::class:Box
;; FROM CLASS: Rox
;;      height=1
;; FROM CLASS: Rectangle
;;      width=4
;;      leng=9
;; FROM CLASS: object
= ()
? (defmethod Box volume ()
    (* width leng height) )
= volume
? (==> monolith volume)
= 36
```

An alternative way to define the `volume` method would have been to multiply the `height` by whatever result the `area` method inherited from `rectangle` returns:

```
(defmethod Box volume ()
  (* height (==> this area)) )
```

Unlike C++, Lush neither support multiple inheritance, nor private slots and methods.

2.21 Input and Output

Lush offers several ways to exchange data with files, sockets, and other Lush processes.

2.21.1 Simple String Input/Output

The simplest way to print things is to use the `printf` function, which works pretty much like its C equivalent. The simplest way to read data into strings is done with `read-string` and `skip-char` , which are very flexible.

Normal output can be redirected to a file with the `writing` construct. For example, the code below will write 10 random integers to the file "blah":

```
(writing "blah" (for (i 0 9) (printf "\\%d\\n" (int (rand 0 100)))))
```

Similarly, input can be redirected from a file with the `reading` construct. For example, the code below will read the first line from text file "blah":

```
(reading "blah" (read-string))
```

The first argument of `reading` and `writing` can be one of the following:

- a string, interpreted as a file name.
- the string `"$stdin"` , which means Lush's standard input.
- the string `"$stdout"` , which means Lush's standard output.
- the string `"$stderr"` , which means Lush's standard error.
- the file descriptor of a file previously opened with `open-read` , `open-write` or `open-append` .
- a string whose first non-white character is "—", in which case the rest of the string will be interpreted as a shell command to or from which data will be piped `/ul`

Here is an example of calling a pipe:

```
(writing "| sort -n" (for (i 0 9) (printf "\\%d\\n" (int (rand 0 100)))))
```

Filename Processing

Several functions allow manipulation of strings containing filenames and directories, such as `(dirname "/asd/qwe.xxx")` which returns the directory `"/asd"` ; and `(basename "/asd/qwe.xxx")` which returns the basename (without the directory) `"qwe.xxx"` ; `(filename-get-suffixes "/asd/qwe.xxx")` , which returns the filename extensions "xxx"; and `(filename-chop-suffixes "/asd/qwe.xxx")` , which removes the extensions "/asd/qwe".

The existence of a plain file `"/asd/qwe.xxx"` can be determined with `(filep "/asd/qwe.xxx")` , and the size of the file can be obtained with `(file-size "/asd/qwe.xxx")` (which returns `()` if the file does not exist).

More sophisticated ways to examine the properties of a file are provided through the function `fileinfo` .

2.21.2 Complete File Reading

A number of functions are provided that make it easy to manipulate and process text files as a single entity (e.g. when writing shell scripts in Lush). Function `(read-lines file)` will return a list of strings, each of which contains one line of the file. Conversely, `(write-lines f)` will write a list of strings to a text file, with one line per string.

2.21.3 Lush Object Reading/Writing

Text File Reading

Lush text can be read from files using `(read)` . `(read)` will parse and read any Lush expression or atom (as is it were typed at the prompt) and return it. This allows to read lists, numbers and strings easily.

Conversely, printing Lush objects to a text file or on the terminal can be done with `(print some-lush-object)` .

Binary Serialization

One of the most powerful and convenient input/output capabilities of Lush is the ability to "serialize" (turn into a byte stream) any Lush object. This can be used to save and restore any Lush data struture to and from a file without having to worry about the file format. It can also be used by two Lush processes to communicate data (e.g., through a socket).

Saving a Lush object (say contained in variable `some-lush-object`) to file `"myfile.bin"` can be done with:

```
(writing "myfile.bin" (bwrite some-lush-object))
```

Restoring the object can be done with:

```
(reading "myfile.bin" (setq some-lush-object (bread)))
```

Functions `bread` and `bwrite` stand for "binary read" and "binary write".

2.21.4 C-like stdio Functions

Another way to perform input/output is to call function from the standard C stdio library. Interfaces to a number of common functions are provided in `libc/stdio.lsh` . Functions include `fopen` , `fclose` , `popen` , `pclose` , `ftell` , `fseek` , `fgetc` , `fputc` , `fgets` , and various functions to read and write binary numbers into Lush variables.

2.21.5 Socket Communication

Lush provides a way to read from and write data to sockets using functions `socketopen` , `socketaccept` , and `socketselect` .

There is also a simple way to start another Lush process (on the same machine as the current process or on another one), and to send expressions to that remote process to be evaluated. This is implemented through the `RemoteLush` class. An instance of Lush must be running in "slave" mode on the remote machine; this is done by starting the script "lushslave". See the documentation for "Remote Lush Execution" for more details.

2.22 Graphics

Lush offers several ways to output graphics. The most common one is the Lush graphics driver, which provides a small set of graphic primitives to display data in a window or write it to a PostScript file. Lush also provides interfaces to the OpenGL API and to the SDL API, but these will not be described in this section.

Opening a graphic window on the screen is simply done with:

```
(new-window [<x> <y> [<w> <h> [<name>]]])
```

where the arguments are respectively the position, size and name of the new window.

Drawing in a window can be done with a collection of simple primitives:

- drawing a line: `(draw-line 10 10 100 200)`
- setting the drawing color: `(color-rgb 0.4 0.1 0.8)`
- drawing a rectangle: `(draw-rect 10 10 50 80)`
- drawing a filled rectangle: `(fill-rect 10 10 50 80)`
- clearing the screen: `(cls)`

- displaying an image: `(rgb-draw-matrix 10 10 im)` where `im` must be an `NxM` , `NxMx1` , `NxMx3` , or `NxMx4` matrix that will be intepreted as an `NxM` image with with greyscale or RGB pixels (whose individual values must be between 0 and 255).
- changing the font and drawing a text:

```
(font "Helvetica-Bold-20") (gprintf 100 100 "salut les potes")
```

Double-buffered and flicker-free redrawing can be done with the `graphics-batch` function. Used in the form `(graphics-batch exp1 exp2 ... exp3)` , it will evaluate all its argument, but any graphic operation performed in that time will be performed into an off-screen graphic buffer. This graphic buffer will be slapped on the window as the call to graphics-batch terminates. This allows smooth double-buffered animation:

```
(while not-finished
  (compute-stuff)
  (graphics-batch
     (draw-stuff-1)
     (draw-stuff-2)
     (draw-stuff-3)))
```

Here is an example:

```
(libload "libimage/image-io")
(setq m
  (image-read-rgb
    (concat-fname lushdir "packages/video4linux/demos/jabba-320x240.jpg")))
(new-window)
(for (i 0 99) (graphics-batch (cls) (rgb-draw-matrix i i m)))
```

2.22.1 Multiple Window Management

The function `new-window` provides a simply way to quickly open a graphic window on whatever graphic device is available (most likely your machine's X screen). Lush supports multiple windows under X, as well as "windows" (or graphic outputs) on other graphic devices. Opening two windows under X is done as follows:

```
(setq win1 (x11-window 0 0 200 200 "Window 1"))
(setq win2 (x11-window 0 220 200 200 "Window 2"))
```

Graphic commands occur in whatever window object is bound to the `window` symbol. So drawing a line into `win2` can be done with:

```
(let ((window win2)) (draw-line 10 10 100 100))
```

Closing a window is simply done by setting its handle to nil:

```
(setq win1 ())
```

In addition, a set of "tiling" functions helps with placing and managing multiple windows on the screen (see namespace `tiling-`).

2.22.2 PostScript File Graphics

PostScript output is performed by opening a `ps-window` as follows:

```
(setq window (ps-window width height filename))
```

`filename` is the name of the PostScript file that will be generated.

2.23 Curve Plotting

Plotting curves can be done with the `plotter` class defined in the `libplot/plotter` library. Plotting can be performed in any type of window, including X11, and PS. Creating a plotter is done with:

```
(setq p (new plotter x y width height))
```

The plotter will attach itself to the current window, place its upper left hand corner at location `x,y` , and set its dimension in the window to `width,height` . A window will automatically open if there is no current window. Using a plotter consists in adding "curves" to it and then calling the method "redisplay" to actually render it in the window to which it is attached.

Here is how to plot the log function in blue on the 0, 10 interval:

```
(==> p PlotFunc "log" log 0 10 0.1 (alloccolor 0 0 1))
(==> p redisplay)
```

Here is how to plot a segment of ellipse in red:

```
(==> p PlotXY "lips" sin cos 0 5 0.1 (alloccolor 1 0 0))
(==> p redisplay)
```

Here is how to plot a sparsely sampled green sinusoid (and indicating each sample by a filled circle):

```
(setq x (range 0 10 0.5))
(setq y (mapcar sin x))
(==> p plotlists "sine" x y (alloccolor 0 1 0) closed-circle)
(==> p Redisplay)
```

A plotter can be moved or resize using the methods `move` and `setsize` :

```
(setq p move 20 20)
(setq p setsize 100 100)
(==> p Redisplay)
```

2.23.1 Gnuplot

Another alternative is to use gnuplot for plotting. Lush provides an interface to the gnuplot plotting program, [htpp://www.gnuplot.info] , which is installed on most Unix sytems. The gnuplot interface allows easy passing of text (plotting commands) and array data to the gnuplot program. See the manual section "Plotting with gnuplot" for details.

2.24 Ogre GUI Toolkit

Lush contains a fairly complete Graphical User Interface toolkit built on top of the standard graphics library. Ogre allows very rapid development of GUI applications. An introduction and a tutorial are available in the Standard Library/Ogre section of this manual.

See: Introduction to Ogre.

See: Ogre Tutorial.

See: The Ogre Class Library.

Here we will just show a couple of very short examples of how simple GUIs can be created in just a few lines. First, here is the famous one-liner button "call a function when clicked":

```
(ogre)
(de coucou (c) (printf "coucou\\n"))
(new AutoWindowObject 10 10 () () "ouch" (new StdButton "click me" coucou))
```

The `AutoWindowObject` is an ogre object that opens a new window and adjusts itself to fit the size of its content. In this case, the content is a single standard button (a.k.a. `StdButton`). The `StdButton` constructor takes a string and a function with one argument which is called each time the button is pressed. Now, here is a slightly more sophisticated example:

```
(ogre)
(libload "libimage/image-io")
(libload "libimage/rgbaimage")
;; load image
(defvar image
        (image-read-rgba
          (concat-fname lushdir "packages/video4linux/demos/jabba-320x240.jpg")))
;; make an imageviewer object
(setq image-viewer (new ImageViewer 240 220 image t))
;; define a few functions to tell the imageviewer to display rotated images
(de rot90-right (c) (==> image-viewer settext (rgbaim-rot90-right image)))
(de rot00 (c) (==> image-viewer settext image))
(de rot90-left (c) (==> image-viewer settext (rgbaim-rot90-left image)))
;; create the GUI window
(new AutoWindowObject 10 10 () () "ouaf"
  ;; insert a column
  (new Column
    ;; make a row of buttons
    (setq button-row
      (new Row
        (new StdButton "Left" rot90-left)        ; rotate left when clicked
        (new StdButton "Straight" rot00)         ; straighten up when clicked
        (new StdButton "Right" rot90-right)))    ; rotate right when clicked
    ;; insert imageviewer
    image-viewer))
```

2.25 Lush Executable Shell Scripts

Lush can be used to write executable scripts that can be invoked from the Unix prompt just like shell scripts, Perl scripts, or executable programs.

Rather than a long explanation, here is an example of a Lush script file:

```
#!/bin/sh
exec lush "$0" "$@"
!#
(printf "Capitalizing the arguments:\\n")
(each ((arg (cdr argv))) (printf "\\%s \\%s\\n" arg (upcase arg)))
```

This works because Lush recognizes everything between `#!` and `!#` as a multi-line comment. This assumes that your lush executable is in your shell command search path.

In that mode, Lush does not display the usual startup banner. It loads the standard environment (`sys/stdenv.dump` or `sys/stdenv.lsh`) and sets the variable `argv` to a list of strings, where the strings are the command line

arguments (including the command itself). Finally it reads and evaluates the content of the script file. The script terminated when the evaluation of the last expression in the script returns or when the function `exit` is called.

The library `libc/shell.lsh` contains several functions to facilitate the writing of shell scripts. Among other things, it contains Lisp versions of common shell functions such as `ls` , `cp` , `mv` , etc.

Here is another script example: a photo slideshow program in 12 lines of Lush. Put this in a file (say `lush-slideshow`), make it executable with `chmod a+x lush-slideshow` . Then, at the shell prompt, type: `lush-slideshow photo1.jpg photo2.jpg photo3.jpg` :

```
#!/bin/sh
exec lush "$0" "$@"
!#
(libload "libimage/image-io")
(libload "libimage/rgbaimage")
(setq window (x11-window 1 1 640 480 "Slide Show"))
(while t
  (do ((f (cdr argv)))
    (rgb-draw-matrix
     0 0 (rgbaim-resize (image-read-rgba f) (xsize) (ysize) 0))
    (sleep 4)))
```

2.25.1 Executable GUI Applications

Writing a Lush scripts containing Ogre applications is a very good way to write executable GUI applications in Lush. Here is an example:

```
#!/bin/sh
exec $0
!#
;; A simple Ogre GUI demo
(ogre)
(wait (new AutoWindowObject 10 10 100 100 "Simple Lush GUI Demo"
        (new Column
          (new StdButton "    hit me    " (lambda (c) (printf "OUCH\\n")))
          (new StdButton "    feed me   " (lambda (c) (printf "CRUNCH\\n"))))))
```

The `wait` function causes Lush to sit around and keep processing events until the argument of `wait` becomes nil (presumably as the result of processing an event). If we did not call `wait` in the above example, the script would terminate immediately after opening the Ogre window. The argument to `wait` is the `WindowObject` in which the Ogre application is set up. It is destroyed as soon as the user closes the window, hence terminating the script.

2.26 Lush Standalone Binary Executables

There is a somewhat crude facility that allows the generation of standalone executables from compiled Lush code. This can be useful for distributing Lush programs to people who may not have Lush installed on their machine, or for integrating Lush-generated code into other C or C++ programs (or call Lush programs from any other program).

At this time, this is only possible for Lush functions that are fully compiled (no interpreted part is allowed).

This functionality is performed by the `make-standalone` function. the `make-standalone` function can generate a complete C source directory with all the supporting functions and macros to make a particular Lush function run.

The generated code can be redistributed with no restriction (it is not subject to the obligations of the LGPL).

See the `make-standalone` documentation for more details. (Note: This facility is currently broken)

Chapter 3

Lush Basics

3.1 Lush Interpreter Basics

The Lush language works on data structures referred to as Lush objects. The main data structure is the "list". A list is an ordered sequence of Lush objects of any type including other lists.

All other Lush objects are collectively referred to as "atoms". There are in fact many kind of atoms used for implementing various Lush features. For instance:

- A number represents a numerical value.
- A symbol is just a name used for naming other lisp objects.
- A string is a sequence of characters useful for storing text.

A Lush program is stored as a collection of "functions" which are just another kind of Lush object. When a function is called, a predefined sequence of operations is applied to a couple of lisp objects (the function arguments) and a new lisp object is returned (the function result).

Once Lush is started, a loop is entered which "reads" a Lush object, "evaluates" it and displays the resulting Lush object. The evaluation of a list consists in calling the function named after the head of the list with the rest of the list as arguments.

Examples:

- Typing `(+ 3 4)` prints `7` which is the result of calling a function named `+` on the two numbers `3` and `4` .
- Typing `(de sqr(x) (* x x))` defines a function named `sqr` with one argument referred to as symbol `x` . This functions calls the multiplication function `*` with two arguments both equal to `x` . The result of this call is then returned.

- Typing `(sqr 4)` then calls function `sqr` on the number `4` . The number `16` is then returned.

3.2 Lush Startup

See: (dump `fname`)
See: (startup `...argv...`)
See: (toplevel)

At startup time, Lush locates the executable file and goes up in the directory hierarchy until it finds a suitable initialization file.

- If there is a file named `"sys/stdenv.dump"` , Lush loads this binary dump file (created with function `dump`) and proceed with the startup procedure.
- If there is a file named `"sys/stdenv.lshc"` or `"sys/stdenv.lsh"` , Lush loads this text or binary lisp file and proceed with the startup procedure. Lush refuses to start without one of the above files.

Lush then calls the lisp function `startup` with the Lush command line argument as arguments. The startup function provided by the standard `"stdenv"` file first searches and loads a file named `".lush2/lushrc.lsh"` in your home directory.

- If no arguments were passed to the Lush interpreter, the function `startup` exits. Lush then calls function `toplevel` . This function sets up some debugging code, and interactively waits for user commands.

  The toplevel function continuously prints a question mark prompt, reads a lisp expresion, evaluates this object and prints the result on the standard output. This process is repeated until you enter an end of file character. The global variable `result` always contains the last result printed by `toplevel` .

  If the outermost list has not been completed when you enter a carriage return, toplevel prompts you for rest of the list, by displaying a few spaces instead of displaying a question mark prompt.

  Example: Starting Lush

```
\\% lush2
LUSH Lisp Universal Shell (compiled on Aug 14 2002)
   Copyright (C) 2002 Leon Bottou, Yann Le Cun, AT&T Corp, NECI.
 Includes parts of TL3
   Copyright (C) 1987-1999 Leon Bottou and Neuristique.
 Includes selected parts of SN3.2:
   Copyright (C) 1991-2001 AT&T Corp.
This program is free software; it is distributed under the terms
of the GNU Public Licence (GPL) with ABSOLUTELY NO WARRANTY.
```

```
Type `(helptool)' for details.
+[/home/yann/lush/sys/stdenv.dump]
 [.lushrc]
?
```

- If one or more arguments are passed to the Lush interpreter, the `startup` function defines a variable `argv` containing the specified arguments, runs the lush file specified by the first argument, and exits. This is suitable for writing Lush scripts:

```
#!/bin/sh
exec lush "$0" "$@"
!#
(printf "Capitalizing the arguments:\\n")
(dolist (arg argv) (printf "\\%s \\%s\\n" arg (upcase arg)))
```

  The shell `/bin/sh` runs the script and starts Lush. Lush then processes the script and recognize everything between `#!` and `!#` as a multi-line comment.

The startup process can be customized by specifying a magic first argument on the command line. The magic first argument starts with character `"@"` followed by the name of an alternate initialization file which will be loaded instead of the default `"sys/stdenv.dump"` or `"sys/stdenv.lsh"` file.

3.3 The Lush Reader

At this point you can type a textual representation of certain lisp object. The reader is a program which converts this textual representation into an actual lisp object.

Lush input is composed of words separated by blank delimiters (i.e. `space` , `tab` or `end-of-line`) by parentheses, or macro-characters. Any character between a semicolon `;` and an end-of-line is considered as a comment and is ignored by the interpreter. Lush also recognizes multi-line comments surrounded by `#!` and `!#` .

- A numerical word is interpreted as a number.
- A word surrounded by double quotes is considered as a string.
- Any other words is interpreted as a symbol name.
- A list is read as a sequence of words enclosed within parenthesis. The empty list is written `()` .

The exact syntax for lisp objects is defined later in this manual. Moreover, the behavior of the reader can be modified by defining "macro-characters". Whenever the Lush reader reaches a macro-character, it executes immediately an attached function which often calls the Lisp reader recursively. For instance, the character quote (') is a macro-character.

3.4 The Lush Evaluator

The evaluator is a program which computes a new lisp object from another lisp object using a very simple rule. Every kind of object is in fact defined by a class (yet another Lush object). The class defines what happens when this object is evaluated and what happens when a list starting with this object name is evaluated.

The Lisp evaluator thus does not need to know anything about the evaluation behavior of all the different kinds of objects. In addition, it does not even need to test the nature of the objects. An indirect call is considerably more efficient than series of tests. Here is the evaluation algorithm for evaluating a Lisp object `p` :

```
eval(p) :
 if (p is the empty list)
    return the empty list
 else if (p is a list)
    q = eval(car p)
    if (q is an atom)
         call the class function 'listeval' for q with args (q,p)
    else
         call the default 'listeval' function with args (q,p)
 else if (p is an atom)
    call the class function 'selfeval' for p with arg (p)
```

3.5 Errors in Lush

See: Interruptions in Lush.

See: (on-error `p l1 ... ln`)

See: (errname)

See: (debug-hook)

The evaluation of the lists sometimes leads to an error. When an error occurs, Lush then executes the `debug-hook` function, which has been set on startup by function `toplevel` .

This function prints an error message, followed by the top elements of the evaluation stack (i.e. the list whose evaluation has caused the error, the list whose evaluation has caused the evaluation of the list that causes the error, etc...).

You are then prompted for a "Debug toplevel".

- If you answer `n` , or simply type a carriage return, Lush aborts any current evaluation, resets the stack, and restarts the toplevel function.
- If you answer `y` , a prompt `"[Debug] ?"` is displayed, and you are able to type Lush commands for examining the current variables, display the current evaluation stack or perform any useful debugging tasks. You can leave this debugging toplevel by typing `Ctrl-D` or using the function `exit` .

3.6 Interruptions in Lush

Lush can be interrupted at any time by typing `Ctrl-C` or `Ctrl-Break` . Interruption handling is essentially similar to error handling. First, Lush runs the function `break-hook` function which typically has been set up on startup by function `toplevel` .

This function prints an error message. You are then prompted for a "Break toplevel" which is basically a special kind of debug toplevel. When you leave the break toplevel by typing `Ctrl-D` or by using function `exit` , you get a chance to resume the execution at the interruption point.

Example:

```
? (for (i 1 100)
     (for (j 1 100)
        (if (= i j) (print (* i j))))))
1
4
^C9

*** Break
** in:   (if (= i j) (print (* i j)))
** from: (for (j 1 100) (if (= i j) (print (* i j))))
** from: (for (i 1 100) (for (j 1 100) (if (= i j) (print (* i j))))) ...
** from: (load "$stdin" "$stdout")
** from: (let ((break-hook (lambda () (beep) (if (not (ask "Break top ...
Break toplevel [y/n] ?y
[Break] ? i
= 3
[Break] ? j
= 4
[Break] ? (* i j)
= 12
[Break] ? (btrace)
Current eval stack:
```

```
** in:   (btrace)
** from: (load "$stdin" "$stdout" "[Break] ? ")
** from: (if (not (ask "Break toplevel")) (error "Stopped") (load "$s ...
** from: (if (= i j) (print (* i j)))
** from: (for (j 1 100) (if (= i j) (print (* i j))))
** from: (for (i 1 100) (for (j 1 100) (if (= i j) (print (* i j))))) ...
** from: (load "$stdin" "$stdout")
** from: (let ((break-hook (lambda () (beep) (if (not (ask "Break top ...
= ()
[Break] ? ^D
Resume execution [y/n] ?y
Resuming execution
16
25
36
49
64....
```

Break toplevels and debug toplevels cannot be nested. If an error or an interruption occurs when you are working in a break or debug toplevel, Lush will clean up the evaluation stack, and restart the toplevel function.

On Unix systems, Lush is inconditionnally interrupted by typing `Ctrl-\` . This method however may leave Lush with inconsistent internal states. Use it only when `Ctrl-C` does not work for some reason.

See: Errors in Lush.

See: (on-break `p l1 ... ln`)

See: (errname)

See: (break-hook)

3.7 Leaving Lush

For leaving Lush, just type `Ctrl-D` or use the `exit` function. Type `y` and a carriage return when the confirmation request is displayed.

Example:

```
? (exit)
= ()

Really quit [y/n] ?y
Bye
\\%
```

3.8 Lush Memory Management

Under normal use, Lush's memory management is based on a fast, concurrent garbage collector. [More to be said here]

3.9 Lush Libraries

Lush program files and libraries customarily end with the `".lsh"` extension.

Lush reads library files from three directories:

- `sys` : contains the system libraries without which Lush cannot run
- `lsh` : contains libraries that are part of the standard Lush package
- `packages` : contains libraries that are contributed by users, which may not (necessarily) be supported to the same extent as the ones in lsh, and may not (necessarily) be available to all installation/versions of Lush.
- `local` : is meant to contain libraries that are specific to you local installation of Lush.

Loading a program or a library is done with the `load` or the `libload` functions, as in `(libload "cmacro")` . This will search the above three directories for a file named `"cmacro.lsh"` . In other words, the above three directories constitute the default search path for finding programs and libraries. The search order is `local, packages, lsh, sys` .

The core libraries (in the sys directory) contains `"sysenv.lsh"` , which includes functions, macros, and macrocharacters that define the core language.

Some files in the library directories have the extension ".lshc" instead of ".lsh". These are "tokenized" version of the corresponding ".lsh" file. Tokenized files are binary files that load faster than the equivalent ".lsh".

The standard library directory tree (lsh directory and its subdirectories) contains many libraries and utilities without which Lush would not be nearly as much fun. These libraries can be relied on to be present with all distributions of Lush on all platforms.

When Lush is started, it looks for a file named `"stdenv.dump,lshc,lsh"` in the following directories: current directory, `"local"` , `"packages"` , and `"lsh"` . If it is found, it is loaded. There is a default `"stdenv.lsh"` in the `"lsh"` directory. This default `"stdenv.lsh"` sets the search path to, defines a few autoload functions, and reads the `".lush2/lushrc.lsh"` file in the home directory of the user.

The standard library contains many niceties including numerous numerical and graphic functions and objects.

A notable example is `"ogre"` , an object-oriented GUI toolkit which makes it very easy to build mouse driven graphical user interfaces. A couple lines of code are enough to setup a window with a button that calls a lush function when clicked upon:

```
;; start ogre
(ogre)
;; define function that will be called when button is clicked
(de print-coucou (c) (printf "coucou\\n"))
;; open ogre window and insert a button in it
(new windowobject 0 0 100 40 "hello"
     (new stdbutton "coucou" print-coucou))
```

Chapter 4

Language Compatibility

4.1 Lush 2 vs Lush 1

Lush 2.x is not fully backward compatible to Lush 1.x. Here are the main differences

- Lush 2 has a more logical and more consistently named set of array functions. See the help on "Arrays and Indexes".
- Type declarations require the `declare` keyword.
- Strings in compiled code are plain C strings (not byte storages).
- Lush 2 has a garbage collector, the `LOCK` and `UNLOCK` macros for reference counting are obsolete.

To make porting Lush 1.x legacy code easy there is a compatibility namespace `lush1-` containing obsolete function definitions. See for example the code in `packages/libnum/linalgebra.lsh` .

4.2 Compilable Lush

The design philosophy of the Lush compiler is somewhat unusual, and, admittedly, in stark violation of many commonly accepted principles of good programming language design. First, the Lush compiler is not designed to replace the interpreter, but to complement it. Lush applications are often a combination of compiled code for things where performance takes precedence over flexibility (e.g., "expensive" and heavily numerical functions), and interpreted code for things where flexibility takes precedence over performance (high-level logic, user interface).

Therefore, the Lush compiler is designed to generate very efficient code for a subset of the Lush language as understood by the interpreter. In addition, Lush functions may contain embedded C code if they are going to be compiled.

This ability makes Lush the "superglue of the scripting languages" as it allows calling C or C++ library code directly, without writing an interface first. The compilable subset of Lush will be called CLush, while the interpreted Lush dialect will be called simply Lush.

The main differences between Lush and CLush are as follows:

- CLush is statically typed (the type of formal function arguments must be declared), while Lush is dynamically typed (a single variable can be assigned values of different types at different times in the same program).
- CLush uses lexical scoping while Lush uses dynamic variable binding (if you don't know the difference, don't worry).
- Dynamic lists are not compilable, that is, the number and type of list elements must be known at compile time.
- When dealing with dynamically allocated objects in inline C, one may either use the C-style manual memory management (`malloc` and `free`) or the memory manager `cmm` , which is part of the Lush runtime. The two memory management styles may coexist but may not be mixed. (For more information on `cmm` see the documentation at [http://libcmm.sf.net] . Read also about Lush's `mptr` type).

4.3 Differences to Common Lisp

Lush is more similar to Scheme than to Common Lisp in many ways. This section lists a couple of not so obvious differences between Lush and Common Lisp.

4.3.1 NIL is not equivalent to ()

While both Lush and Lisp use `()` to represent falsity and the empty list, Lush has no synonym for it as Common Lisp has with `nil` . However, since unbound symbols evaluate to `()` in Lush, writing `nil` instead of `()` will work as expected most of the time, as long as the symbol `nil` is not bound. But there are exceptions, the following example illustrates one:

```
? nil
= ()

? (member nil '(1 nil 2))
= ()
```

The second occurrence of `nil` in the above example is not evaluated and thus is not equivalent to `()` . Since `nil` is just a symbol like any other, it is recommended to always write `()` where in one might use `nil` in Common Lisp.

Chapter 5

Core Interpreter and Startup Library

This section includes all the function available at startup with the default environment `lsh/lushenv.lsh` . Included are the core interpreter functions and the library functions defined in `sys/sysenv.lsh` , all the files in `lsh/libstd` (that are loaded at startup).

5.1 Lists

List are the elementary data structure in Lush. Several functions are designed for handling lists. Lists are stored as pairs (car . cdr). The first element of the pair (car) is a pointer to the first element of the list. The second element of the pair (cdr) is a pointer to the list of the remaining elements.

The textual representation of a list is composed of an opening parenthesis, a sequence of Lush objects separated by blanks and closing parenthesis. The empty list is thus written as `()` . (Note that `nil` is not another name for `()` as in some Lisp implementations).

The second element (cdr) of a pair is not necessarily a list. A special dotted pair notation is used for representing such a pair. This notation is composed of an opening parenthesis, the first Lush objects, a blank separated dot, the second Lush object and a closing parenthesis.

Examples:

```
(1 2 3)              ; car=2 and cdr=(2 3)
((q w) 2)       ; car=(q w) and cdr=2
(1 2 (e r))     ; car=1 and cdr=(2 (e r))
(a . b)              ; car=a and cdr=b
```

5.1.1 List Manipulation Functions

The following functions are useful for accessing or building lists.

(car `l`**)**

Return first element of a list or dotted pair `l` . The function name 'car' stands for contents of address register.

Example:

```
? (car '(a b c d))
= a
```

(cdr `l`**)**

Return rest of list `l` (or second element of dotted pair `l`). The function name 'cdr' stands for contents of destination register.

Example:

```
? (cdr '(a b c d))
= (b c d)
```

(c ... r `l`**)**

This a variation on `car` and `cdr` to access specific, possibly nested, elements of a list. The ellipsis (...) stands for any two or three letter combination of the letters `a` or `d` (i.e. the address and destination). All the combinations of two car or cdr functions are written in C. The combination of three are written in Lisp in `"sysenv.lsh"` .

Example:

```
? (cadr '(a b c d))
= b
```

(cons `a1` `a2`**)**

Builds a list whose car is `a1` and cdr `a2` .

Example:

```
? (cons '+ '(2 3))
= (+ 2 3)
```

```
? (cons 'a 'b)
= (a . b)
```

(list `a1` **...** `an`**)**

Return a list with elements `a1` to `an` .

```
? (list 'a '(b c) 'd)
= (a (b c) d)
```

(make-list `n` `v`**)**

Return a list containing `n` times element `v` .

```
? (make-list 4 'e)
= (e e e e)
```

(copy-list `l`**)**

See: copy-tree

Return a new proper list with the same elements as `l` .

(copy-tree `tr`**)**

See: copy-list

Return a copy of `tr` in wich all conses are new but all atoms are the same as in `tr` .

(circular-list `a1` **...** `an`**)**

Return a circular list with elements `a1` to `an` repeating indefinitely.

```
(circular-list 0 1)
= (0 1 0 ...)
```

(range [`n1`] `n2` [`delta`]**)**

Returns a list of all the numbers between `n1` and `n2` , stepping by `delta` . The default value for both arguments `n1` and `delta` is `1` .

See: range*

Example:

```
? (range 2 5)
= (2 3 4 5)
```

(range* [n1] n2 [delta])

Returns a list of all the numbers between `n1` and `n2` , excluding `n2` , stepping by `delta` . The default value for `n1` is 0, for `delta` it is `1` .

See: range

Example:

```
? (range* 5)
= (0 1 2 3 4)
```

(length s)

Returns the length of the sequence `s` .

Example:

```
? (length '(a b c d e))
= 5
```

```
? (length [d[d   1.0000   2.0000]
           [d   3.0000   4.0000]])
= 2
```

This function is able to detect that list `l` is circular. The value `-1` is returned when such a condition occurs.

(append l1 ... ln)

Concatenates lists `l1` to `ln` . The idiom `(append l ())` may be used to get a fresh copy of the list `l` .

Example:

```
? (append '(1 2 3) '(4 5 6))
= (1 2 3 4 5 6)
```

(reverse l)

Returns a list of `l` 's toplevel elements in reverse order.

Example:

```
? (reverse '(1 2 3))
= (3 2 1)
```

? (reverse '(1 2 3)) = (3 2 1)

(nth `n l`**)**

Returns the `n` th elements of the list `l` . The first element is numbered `0` . Example:

```
? (nth 3 '(a b c d e))
= d
```

(nthcdr `n l`**)**

Returns the `n` th pair of the list `l` . The first pair is numbered 0. This function is equivalent to `n` calls of the `cdr` functions.

Example:

```
? (nthcdr 3 '(a b c 3 e f))
= (3 e f)
```

(lasta `l`**)**

Returns the last element ("last atom") of the list `l` . Example:

```
? (lasta '(a b c d))
= d
```

```
? (lasta '(a b c . d))
= d
```

(last `l` [`n`]**)**

Return the last `n` conses of the `l` (default `n` =1).

Example:

```
? (last '(a b c))
= (c)
```

(nfirst `n l`**)**

When `n` is 0, the empty list is returned.

When `n` is greater than 0, this function returns the `n` first elements of `l` . When the length of the list is lower than `n` , a copy of the list is returned.

When `n` is lower than 0, this function returns all elements but the `n` last. When the length of the list is lower than the absolute value of `n` , the empty list is returned. If `l` is a circular list, an error occurs.

(member `e l`**)**

Searches list `l` for element `e` . If element `e` is not found, function `member` returns the empty list `()` . Otherwise function `member` returns the first sublist of `l` whose first element is equal to `e` .

There is a single equality test in Lush that is able to recursively compare lists and strings.

Example:

```
? (member 'e (range 1 4))
= ()


? (member 'e '(a b c d e f g h i j))
= (e f g h i j)
```

(flatten `x`**)**

Returns all the non nil atoms in `x` , linked in a single list.

Example:

```
? (flatten '2)
= (2)


? (flatten '(1 2 (3 (6 7)) 4))
= (1 2 3 6 7 4)
```

(flatten* `x`**)**

Return all atoms in `x` , including `nil` s, linked in a single list.

Example:

```
? (flatten* '2)
= (2)


? (flatten* '(1 2 (3 (6 7)) 4))
= (1 2 3 6 7 () () 4 ())
```

(filter `p l`**)**

See: filter*

Return list of those elements of `l` for which `p` evaluates to anything but `()` .

Example:

```
? (filter evenp (range 10))
= (2 4 6 8 10)
```

(filter* p l)

See: filter

Return two lists, the first containing all elements of `l` for which `p` evaluates to anything but `()` , the second containing all other elements.

Example:

```
? (filter* evenp (range 10))
= ((2 4 6 8 10) (1 3 5 7 9))
```

(assoc key alist)

See: (alist `...atoms...`)

See: (alist-add `key value alist`)

See: (alist-get `key alist`)

See: Hash Tables.

Function `assoc` is useful for searching an "alist". An alist is a list of pairs `(key . value)` representing associations between a search key `key` and a value `value` .

Function `assoc` returns the first pair of `alist` whose first element is equal to `key` . The empty list is returned when no matching pair is found. Once the pair is located, you can acess the value associated with the key using function `cdr` . You can also change the value using function `rplacd` .

Remark: Alists should be only used for small numbers of key-value pairs. We suggest using hash tables as soon as the association involves more that twelve key-value pairs. Hash tables indeed require more memory but are much faster and much more convenient.

(alist ...atoms...)

See: (assoc `key alist`)

Create an alist by pairwise `cons` -ing the arguments.

```
? (alist "one" 1 "two" 2 "three" 3)
= (("one" . 1) ("two" . 2) ("three" . 3))
```

(copy-alist al)

See: (alist `...`)

Create a new alist containing copies of all conses in `al` and return it. Each cons in `al` has the same car and cdr as its corresponding cons in the returned alist.

(alist-add `key value alist`**)**

See: (assoc `key alist`)

Add pair (`key . value`) to `alist` or update `alist` with new value and return the modified alist. If a pair with `car key` already exists in `alist` , then `alist-add` performs an update by setting this pair's `cdr` to `value` (that is, when updating `alist-add` modifies the input `alist`).

```
? (alist-add "three" 3 (alist "one" 1 "two" 2))
= (("three" . 3) ("one" . 1) ("two" . 2))
```

(alist-get `key alist`**)**

See: (assoc `key alist`)

Returns the value associated to `key` in `alist` .

```
? (alist-get "two" (alist "one" 1 "two" 2 "three" 3))
= 2
```

(alist-rm `key alist`**)**

See: (assoc `key alist`)

Return new `alist` with pair (`key . whatever`) removed.

```
? (alist-rm "two" (alist "one" 1 "two" 2 "three" 3))
= (("one" . 1) ("three" . 3))
```

(alist-update `alist alist2`**)**

See: (assoc `key alist`)

For each key in `alist2` , if it is present in in `alist` , update the associated value, otherwise add the (key . value) pair in `alist2` to `alist` .

```
? (let ((al (alist "one" () "two" 2))
        (al2 (alist "one" 1 "three" 3)) )
    (alist-update al al2) )
= (("three" . 3) ("one" . 1) ("two" . 2))
```

(sort-list `l cmp`**)**

Return a sorted copy of list `l` according the order relation achieved by the diadic function `cmp` .

```
? (sort-list '(12 3 1 2 3 4 5 6) >)
= (1 2 3 3 4 5 6 12)
```

(bsearch `l v`**)**

Return the largest index of the element in `l` whose value is less than or equal to `v` using a binary search. `bsearch` assumes the elements in `l` are sorted in ascending order.

```
? (let ((l '(2 4 6 8 10)))
    (nth (bsearch l 5) l) )
= 4
```

If `l` is a numeric vector, `bsearch` dispatches to the appropriate bsearch function defined in "libidx/idx-sort.lsh".

5.1.2 Physical List Manipulation Functions

Instead of building new pairs, a few fast functions directly change the pointers stored in the pairs given as argument. This is safe if the programmer knows that the function's arguments are not used elsewhere.

Side effects should be expected when another lisp object points to the argument of such a physical list modification function. The modification then appears in both places.

Physical list manipulation functions can be used to construct lists that reference themselves. Some Lush functions are able to test this condition (e.g. `length`) and avoid an infinite loop. Some Lush functions however (e.g. `print` , `flatten`) will loop forever when processing such a list. You can interrupt these loops by typing `Ctrl-C` or `Break` .

See: (length `l`)

See: (== `n1 n2`)

(rplaca `l e`**)**

Physically replaces the `"car"` of list `l` by `e` .

Example:

```
? (setq a '(1 2 3))
= (1 2 3)
? (setq b (cdr a))
= (2 3)
? (rplaca b 'e)    ; this is fast
= (e 3)
? a                ; but causes side effects
= (1 e 3)
```

(rplacd `l e`**)**

Physically replaces the cdr of list `l` by `e` .

(displace `l1` `l2`)

Replaces both the car and cdr of `l1` by the car and cdr of `l2` . The main purpose of this function is the implementation of DMD functions.

(nconc `l1` ... `ln`)

Function `nconc` returns the concatenation of the lists `l1` to `ln` . It uses physical replacement and therefore causes side effects.

(nconc1 `l` `e`)

Function `nconc` returns a list formed by appending element `e` to the end of list `l` . It uses physical replacement and therefore causes side effects.

Example:

```
? (setq a '(a b c d e))
= (a b c d e)
? (nconc1 a 'f)
= (a b c d e f)
```

(list-insert l pos x)

Function `list-insert` returns a list formed by inserting element `x` at position `pos` of list `l` . It uses physical replacement and therefore causes side effects.

(list-delete l pos)

Function `list-insert` returns a list formed by deleting the element located at position `pos` from list `l` . It uses physical replacement and therefore causes side effects.

(list-merge `l` `l2`)

Function `list-merge` returns a list formed by appending to list `l` the elements of `l2` that are not already in `l` . This function uses physical replacement and therefore causes side effects.

(list-split! `l` `n`)

Split a list after `n` elements and return the two parts consed.

Example: (let (((left . right) (list-split! (range 10) 5))) left)

5.2 Numbers

See: Special Numerical Values (IEEE754).

Numbers are the simplest kind of atoms. They are stored as double precision floating point numbers.

Numerical computations in Lush are usually performed in double precision. Single precision however is used by some number crunching functions, like the matrix calculus operations.

You can type in decimal number using the traditional exponential notation or using hexadimal numbers preceded by characters `"0x"` . Here are a few examples of legal numbers:

```
23
3.14
12e6              ;; equivalent to 12000000
1.2E6              ;; equivalent to  1200000
0x1A              ;; equivalent to 26
```

5.2.1 Numerical Constants

+2pi+

Perimeter of the unit circle.

+pi+

Half the perimeter of the unit circle.

+e+

Euler's constant.

+int-min+

Minimum representable integer value (machine-dependent).

+int-max+

Maximum representable integer value (machine-dependent).

+macheps+

Machine epsilon (machine-dependent).

5.2.2 Elementary Numerical Functions

(integerp `x`)

True when `x` is an integer. Raises an error when `x` is not a number.

(oddp x)

True when `x` is an odd number. Raises an error when `x` is not an integer.

(evenp x)

True when `x` is an even number. Raises an error when `x` is not an integer.

The following elementary functions operate on numbers:

(+ n1 ... nm)

Computes the sum of numbers `n1` to `nm` .

Example:

```
? (+ 1 2 3 4)
= 10
```

(1+ n)

Adds 1 to `n` .

Example:

```
? (1+ 4)
= 5
```

(- [n1] n2)

Subtracts `n2` from `n1` . If one argument only is provided, returns the opposite of `n2` .

Example:

```
? (- 2 5)
= -3
```

(1- n)

Subtracts 1 from `n` .

Example:

```
? (1- (+ 4 5))
= 8
```

(* n1 ... nm)

Computes the product of `n1` to `nm` .

Here is a memory intensive way of computing a factorial:

```
? (apply * (range 1 5))
= 120
```

(2* n)
Multiplies `n` by 2.
Example:

```
? (2* (+ 4 5))
= 18
```

(/ [n1] n2)
Divides `n1` by `n2` . If one argument only is provided, Function `/` returns the inverse of `n2` .
Example:

```
? (/ 3)
= 0.3333
```

(2/ n)
Divides `n` by 2.
Example:

```
? (2/ 5)
= 2.5
```

(n m**)**
Returns `n` raised to the `m` th power.
Example:

```
? (** 2 5)
= 32
```

(max l1 **...** ln**)**
Return the maximum element within `l1` ... `ln` . Arguments `li` may be numbers or strings.
Example:

```
? (max 2 3 (* 3 3) (+ 4 3))
= 9
```

(min l1 **...** ln**)**
Return the minimum element within `l1` ... `ln` . Arguments `li` may be numbers or strings.
Example:

```
? (min 2 3 (* 3 3) (+ 4 3))
= 2
```

5.2.3 Integer Arithmetic Functions and Integer Predicates

Besides the ordinary number operation functions, a few functions deal with integer numbers only. For that purpose we consider only the first 32 bits of the integer part of the number.

(div `n1` `n2`**)**

Returns the quotient of the Euclidian division of `n1` by `n2` .

Example:

```
? (div 5 2)
= 2
```

(modi `n1` `n2`**)**

Returns the remainder of the Euclidian division of `n1` by `n2` .

Example:

```
? (modi 5 2)
= 1
```

(mod `x` `y`**)**

See: rem

Return the result of `(- x (* (floor (/ x y)) y))` . The result will always have the same sign as `y` .

Example:

```
? (mod 5 2)
= 1
```

```
? (mod -1 2)
= 1
```

(rem `x` `y`**)**

See: mod

Return the result of `(- x (* (trunc (/ x y)) y))` .

Example:

```
? (rem 5 2)
= 1
```

```
? (rem -1 2)
= -1
```

(oddp n)

True if `n` is an odd integer.

(evenp n)

True if `n` is an even integer.

5.2.4 Bit Functions

(bitand n1 n2)

Returns the bitwise AND of integer numbers `n1` and `n2` .

(bitor n1 n2)

Returns the bitwise OR of integer numbers `n1` and `n2` .

(bitxor n1 n2)

Returns the bitwise exclusive or (XOR) of integer numbers `n1` and `n2` .

(bitshl n1 n2)

Shifts the bits of integer `n1` by `n2` positions to the left and returns the result. This is similar to the C operator >> .

(bitshr n1 n2)

Shifts the bits of integer `n1` by `n2` positions to the right and returns the result. This is similar to the C operator >> .

5.2.5 Mathematical Numerical Functions

Most mathematical functions are implemented as DZ functions. DZs are a remnant of a now obsolete compiler for numerical expressions. For more details about DZ functions the corresponding section.

(sgn n)

Return +1 if `n` is greater than 0, -1 if `n` is smaller than zero, and zero otherwise.

```
? (sgn -2)
= -1
```

```
? (sgn 1.2)
= 1
```

(abs `n`**)**

Return the absolute value of `n` .

```
? (abs 123.3)
= 123.3
```

```
? (abs -23.3)
= 23.3
```

(ceil `n`**)**

Return smallest integer not less than `n` .

```
? (ceil -4.5)
= -4
```

```
? (ceil 4.5)
= 5
```

(floor `n`**)**

Return largest integer not greater than `n` .

```
? (floor -4.5)
= -5
```

```
? (floor 4.5)
= 4
```

(round `n`**)**

Return nearest integer to `n` .

```
? (round -4.4)
= -4
```

```
? (round -4.5)
= -5


? (round 4.5)
= 5
```

(trunc `n`**)**

Return nearest integer to `n` in direction of zero.

```
? (trunc -4.5)
= -4


? (trunc 4.5)
= 4
```

(sqrt `n`**)**

Returns the square root of `n`

```
? (sqrt 15)
= 3.873
```

(cbrt `n`**)**

Returns the cube root of `n`

```
? (sqrt 15)
= 3.873
```

(0-x-1 `n`**)**

This function implements a piecewise saturated linear function. It returns 0 if `n` is smaller than -1. It returns 1 if `n` is larger than +1. It returns `n` if `n` is in the -1 to +1 range.

```
? (0-x-1 -2)
= 0


? (0-x-1 0.7)
= 0.7
```

```
? (0-x-1 1.3)
= 1
```

(0-1-0 n)

This function implements the indicator function of the -1 to +1 range. It returns 1 if `n` is in the -1 to +1 range. It returns 0 otherwise.

```
? (0-1-0 -2)
= 0


? (0-1-0 0.7)
= 1
```

(sin n)

Return the sine of `n` radians.

```
? (sin (/ 3.1415 3))
= 0.866
```

(cos n)

Return the cosine of `n` radians.

```
? (cos (/ 3.1415 3))
= 0.5
```

(tan n)

Return the tangent of `n` radians.

```
? (tan (/ 3.1415 3))
= 1.7319
```

(asin n)

Return the arc sine of `n` radians.

```
? (asin 1)
= 1.5708
```

(acos `n`**)**

Return the arc cosine of `n` radians.

```
? (cos 1)
= 0.5403
```

(atan `n`**)**

Return the arc tangent of `n` , in radians.

```
? (* 4 (atan 1))
= 3.1416
```

(exp `n`**)**

Return the exponential of `n` .

```
? (exp 1)
= 2.7183
```

(exp-1 `n`**)**

Return the exponential of `n` minus 1. This function gives an accurate value of `exp(n)-1` even for small values of `n` .

```
? (exp-1 0.5)
= 0.6487
```

(log `n`**)**

Return the natural logarithm of `n` .

```
? (log 2)
= 0.6931
```

(log10 `n`**)**

Return the natural logarithm to base 10 of `n` .

```
? (log10 1000)
= 3
```

(log2 `n`**)**

Return the natural logarithm to base 2 of `n` .

```
? (log2 256)
= 8
```

(log1+ `n`**)**

Return the natural logarithm of `n` plus 1. This function gives an accurate value of `log(1+n)` even for small values of `n` .

```
? (log1+ 1)
= 0.6931
```

(sinh `n`**)**

Return the hyperbolic sine of `n` .

```
? (sinh 1)
= 1.1752
```

(cosh `n`**)**

Return the hyperbolic cosine of `n` .

```
? (cosh 1)
= 1.5431
```

(tanh `n`**)**

Return the hyperbolic tangent of `n` .

```
? (tanh 1)
= 0.7616
```

(asinh `n`**)**

Return the arc hyperbolic sine of `n` .

```
? (asinh 0.2)
= 0.1987
```

(acosh `n`**)**

Return the arc hyperbolic cosine of `n` .

```
? (acosh 2)
= 1.317
```

(atanh `n`**)**

Return the arc hyperbolic tangent of `n` .

```
? (atanh 0.2)
= 0.2027
```

(findroot `min max f`**)**

Simple dichotomical root finder. `f` is a function with one argument only. `Findroot` will return a numerical approximation of the solution of `f(x)=0` between `min` and `max` . To ensure the existence of this solution, `f(min)` and `f(max)` must have different signs.

Example:

```
? (findroot 1 2 (lambda (x)
                  (- 2 (* x x)) ))
= 1.4142
```

5.2.6 Random Numbers

Statistical functions are provided for computing random numbers, computing various statistics on list of numbers or performing simple linear regressions.

(seed `n`**)**

Set the random number generator seed. The random generator is replicable. The same seed will produce the same sequence of random numbers. Argument `n` must be an integer.

(rand [[`a`] `b`]**)**

Return a uniform random number.

With no arguments `rand` returns a random number in the range [0, 1), that is, including 0 but excluding 1. With one argument `rand` returns a random number in the range [`-b` , `b`). With two arguments `rand` returns a random number in the range [`a` , `b`). If any argument is an array, the result is an array of random numbers (the usual broadcasting rules apply).

Example:

```
? (rand)
= 0.3339


? (rand [d   1.0000  10.0000 100.0000])
= [d  -0.4218   1.5670  15.4083]
```

(gauss [[m] s])

Return a Gaussian number with mean `m` and standard deviation `s` .

The default value for argument `m` is 0. The default value for argument `s` is 1. If any argument is an array, the result is an array of random numbers (the usual broadcasting rules apply).

Example:

```
? (gauss 2)
= 0.3397


? (gauss [d  10.0000  20.0000  30.0000] 5)
= [d   6.1070   7.7124  33.8397]
```

5.2.7 Hashing

(hashcode `expr`)

Return a string containing a hash code for the lisp expression `expr` . Hash codes are guaranteed to be equal when two expressions are logically equal, that is to say, equal as defined by function `=` .

5.3 Booleans Operators

Expressions in Lush are considered "false" when their evaluation returns the empty list. All other expressions are considered "true". The symbol `t` however is preferred for this meaning.

5.3.1 Boolean Arithmetic

A number of functions are available for programming tests and to return boolean values

(= n1 n2)

Test if n1 is equal to n2 . Two objects are equal if they have the same type and if they convey the same information. The criterion for deciding equality therefore depends on the type of the object.

This function supports any Lush object.

- It is able to compare the values of numbers, strings, dates, lists, matrices, arrays, hash tables and high level objects (those created by new and belonging to classes created by defclass).
- The comparison of other low-level objects (i.e. from user defined classes written in C language following the Open protocol) usually relies upon physical equality. However, it is possible to put other comparison functions into the C structures which define the classes.

Example:

```
? (= '(1 2 (3)) (cons 1 '(2 (3))))
= t


? (= [d   1.0000   2.0000] [d   1.0000   2.0000])
= t
```

Testing the equality between special numeric values may be tricky. For example, the IEEE754 specification(supported by most of the industry) defines special bit patterns named NaN(Not a Number). Comparing two NaNs should always return false. Major operating systems and compilers however do not respect this.

Lush expressly specifies that the result returned by the equality test is *always* true when objects n1 and n2 share the same memory location. In other words, if the function == returns t then the function = returns t as well.

See: Special Numerical Values (IEEE754).

See: (== n1 n2)

See: Comparison of User Defined Objects.

(<> n1 n2)

Tests if n1 is different from n2 . This function is equivalent to

```
(de <> (n1 n2) (not (= n1 n2)))
```

Example:

```
? (<> 2 "abcd")
= t
```

```
? (<> 2 2)
= ()
```

See: (= n1 n2)

(== n1 n2)

See: (= n1 n2)

Test if n1 and n2 are physically equal.

This function does not even look at the information conveyed by the lisp objects n1 or n2 . It just tests that the pointers returned by the expression n1 and n2 are equals.

Pointer equality not only means that objects referred to by expression n1 and n2 are equal, but also means that they are located at the same memory addresses.

This information is meaningful as soon as you use functions that modify objects (as opposed to functions returning a modified copy of the object and leave the initial object unchanged). These functions include the following:

- Physically changing list component using rplaca or rplacd .
- Setting the contents of an array or a matrix.
- Setting the value of a slot of an object

Function == may be used in conjunction with these functions. Modifying indeed the object referred to as n1 will therefore also modify the object referred to as n2 . Function == may also be used as a fast way to compare objects and matrix when you know that the only possibility of equality is physical equality.

Example:

```
? (setq a 3)
= 3
? (== a 3)
= ()
? (= a 3)
= t
? (== a a)
= t
? (setq a [1 2])
= [1 2]
? (setq b [1 2])
= [1 2]
? (= a b)
= t
? (== a b)
```

```
= ()
? (a 0)
= [0 2]
? (= a b)
= ()
? b
= [1 2]
```

(0= n)

Test if `n` is equal to 0.
See: Special Numerical Values (IEEE754).

(0<> n)

Test if `n` is different from 0.
See: Special Numerical Values (IEEE754).

(<= n1 n2)

Test if `n1` is less or equal than `n2` . Arguments `n1` or `n2` may be dates, strings or numbers. User defined objects may be supported as well.
See: Special Numerical Values (IEEE754).
See: Comparison of User Defined Objects.

(< n1 n2)

Test if `n1` is less than `n2` . Arguments `n1` or `n2` may be dates, strings or numbers. User defined objects may be supported as well.
See: Special Numerical Values (IEEE754).
See: Comparison of User Defined Objects.

(>= n1 n2)

Test if `n1` is greater or equal than `n2` . Arguments `n1` or `n2` may be dates, strings or numbers. User defined objects may be supported as well.
See: Special Numerical Values (IEEE754).
See: Comparison of User Defined Objects.

(> n1 n2)

Test if `n1` is greater equal than `n2` . Arguments `n1` or `n2` may be dates, strings or numbers. User defined objects may be supported as well.
See: Special Numerical Values (IEEE754).
See: Comparison of User Defined Objects.

(and `l1` **...** `ln`**)**

Evaluates sequentially `l1` .. `ln` . If a result is the empty list, function `and` immediately returns `()` . The result of the evaluation of `ln` is returned otherwise.
Example:

```
? (and (= 2 2) (= 2 3) (print (= 2 2)))
= ()
```

(or `l1` **...** `ln`**)**

Evaluates sequentially `l1` .. `ln` . If a result is not the empty list, this result is immediately returned. The empty list is returned otherwise.
Example:

```
? (or (= 2 2) (= 2 3))
= t
```

(not `l`**)**

Alias for `null` .

```
l
```

This macro-character expands to `(null l)` .

5.3.2 Predicates

Predicates test the type of any lisp object.

(listp `obj`**)**

True when `obj` is a list.
Example:

```
? (listp ())
= t
```

```
? (listp "abc")
= ()
```

```
? (listp '(2 3))
= (2 3)
```

```
? (listp (cons 2 3))
= (2 . 3)
```

(null `obj`**)**

True when `obj` is the empty list.
Example:

```
? (null t)
= ()
```

(consp `obj`**)**

True when `obj` is a non-empty list.
Example:

```
? (consp ())
= ()
```

```
? (consp "abc")
= ()
```

```
? (consp '(2 3))
= (2 3)
```

(proper-list-p `obj`**)**

True when `obj` is a proper list.
Example:

```
? (proper-list-p '(a b . c))
= ()
```

```
? (proper-list-p '(a b c))
= t
```

'EX (proper-list-p "abc")

(atom `obj`**)**

True when `obj` is an atom.
Example:

```
? (atom ())
= t
```

(numberp `obj`**)**

True when `obj` is a number.
Example:

```
? (numberp 3.14)
= 3.14
```

(symbolp `obj`**)**

True when `obj` is a symbol.
Example:

```
? (symbolp 'a)
= a
```

(classp `obj`**)**

True when `obj` is a class.
Example:

```
? (classp object)
= t
```

(functionp `obj`**)**

True when `obj` is a function.
Example:

```
? (functionp "abc")
= ()
```

```
? (functionp functionp)
= ::DX:functionp
```

(stringp `obj`**)**

True when `obj` is a string.
Example:

```
? (stringp "abc")
= "abc"
```

(emptyp `obj`**)**

True when `obj` is empty.

```
? (emptyp '(1 2 3))
= ()


? (emptyp ())
= t


? (emptyp "123")
= ()


? (emptyp "")
= t
```

5.4 Symbols

Symbols are the only named objects. Symbol names may be up to 40 characters long.

The evaluation of a symbol returns the "value" of the symbol. Function `setq` changes the value of a symbol. The value of a new symbol is always the empty list.

During a call to a Lush function, or during the execution of certain special functions (e.g. function `let`) predefined symbols take a temporary value. The previous values are then restored when the function exits.

For instance, assume that we define a function `(de sqr(x) (* x x))` . Evaluating list `(sqr 4)` will perform the following actions:

- save the old values of symbol `x` which is the name of a function argument,
- set the value of symbol `x` to `4` ,
- execute the function body `(* x x)` and save the result,
- restore the previous value of symbol `x` ,

- and return the saved result.

This dynamical binding makes the interpreter getting faster, but somewhat precludes the development of efficient compilers.

The textual representation of a symbol is composed by the characters of its name. The reader usually converts symbol names to lowercase and replaces underscores by dashes. This can be prevented by quoting the symbol name with vertical bars. Such vertical bars are useful for defining symbols whose name contains any character usually forbidden: parenthesis, spaces, semi-colon, macro-characters, upper-case characters etc....

Examples :

```
gasp GasP          ; are the same symbol named "gasp"
|GasP|             ; is different symbol named "GasP"
|);( '|            ; is a symbol named ");( '"
"gasp"             ; is a string
12                 ; is a number
|12|               ; is a symbol named "12"
```

Unlike several dialect of lisp, Lush does not provide other fields in a symbol for storing a specific functional value or a property list. The value of a symbol is the only storage provided by symbols.

5.4.1 (defvar name [val])

If global variable `name` is undefined, bind it to `val` . Return symbol ' `name` .

If there was already an existing global variable `name` , function `defvar` does nothing. Argument `val` is not even evaluated. If there was no existing global variable `name` , function `defvar` defines such a global variable, and initializes it with the result of the evaluation of argument `val` .

Traditionally function `setq` was used for that purpose in SN/TLisp because there was no notion of undefined global variables. All undefined global variables were assumed to contain an empty list. This behavior was the source of many bugs. The lush kernel now prints a warning when using `setq` to define a new global variable.

5.4.2 (defparameter name val)

See: defvar

Bind global variable `name` to `val` . Return symbol ' `name` .

Unlike `defvar` , this function unconditionally evaluates `val` .

5.4.3 (defconstant name val)

See: defparameter, #.

Bind global variable `name` to `val` , lock the binding and return ' `name` .

5.4.4 (defalias name existing-name)

See: defvar, defparameter, defconstant

Bind `name` to globally bound value of `existing-name` and return ' `name` .

Symbol ' `existing-name` must be globally bound.

5.4.5 #.expr

Evaluate `expr` at read time.

A common use case for the read-time eval macro is to simulate symbolic constants in compiled code (see help for `find-c-include` for an example).

5.4.6 (setq s1 v1 ... [sn vn])

Set the value of symbols `s1` ... `sn` to `v1` ... `vn` and return the last value `vn` . It is good practice to create the symbol being setq'ed beforehand using `defvar` , `let` , `let*` or other constructs that create global or local variables.

The `setq` function has a special behavior when the `vi` are not symbols. This behavior is documented later, with the scope function.

Example:

```
? (setq a 3)
= 3
? (setq b 6)
= 6
? a b
= 3
= 6
? (setq a b b a)
= 3
? a b
= 6
= 6
```

5.4.7 (set v a)

Sets the value of the symbol `v` to `a` . set is different from setq because `v` is evaluated first.

Example:

```
? (setq s 'a)
= a
? (set s 3)
= 3
? s
```

```
= a
? a
= 3
```

5.4.8 (incr v [n])

Increment `v` by `n` (default 1). `V` may be variable or an indexed array. Examples:

```
? (let ((s 15))
    (incr s)
     s)
= 16
?
? (let ((m (int-array 3 3)))
    (incr (m 1 1) 3) )
= [i[i     0     0     0]
    [i     0     3     0]
    [i     0     0     0]]
```

5.4.9 (decr v [n])

Decrement `v` by `n` (default 1). Same as "(setq `v` (- `v` `n`))"

5.4.10 (named s)

Returns a new symbol whose name is the string `s` .

Example:

```
? (named "a")
= a


? (named "A")
= |A|
```

5.4.11 (nameof s)

Returns a string containing the name of symbol `s` . This is the converse of `named` .

Example:

```
? (nameof 'a)
= "a"
```

```
? (nameof '|A|)
= "A"
```

5.4.12 (namedclean s)

Returns a new symbol whose name is computed by normalizing the string `s` using the same algorithm as the Lisp reader. This is the converse of function `pname` .

Example:

```
? (namedclean "a")
= a


? (namedclean "A")
= a


? (namedclean "|A|")
= |A|
```

5.4.13 (lock-symbol s1 ... sn)

Locks symbols `s1` to `sn` . Symbols may be locked. You can no longer change the value of locked symbols, but you may still modify them temporarily by using the let function. Some Lush functions and all C functions are stored in locked symbols. This avoids the accidental loss of a C function.

5.4.14 (unlock-symbol s1 ... sn)

Unlocks symbols `s1` to `sn` .

5.4.15 (symbols)

Return all symbols in the global namespace.

Example:

```
? (length (symbols))
= 3761
```

5.4.16 (macrochp s)

Returns `t` if `s` symbol defines a macro-character.

Example:

```
? (macrochp '|'|)
= t
```

5.4.17 (putp anything name value)

Every Lush object (atoms, cons, etc.) may be enriched by defining properties identified by a symbolic name. Function `putp` is used to define such properties. Its first argument is an arbitrary Lush object `anything` . Function `putp` sets the property named by symbol `name` to value `value` . Properties can be later retrieved using `getp` .

5.4.18 (getp anything name)

Retrieves the property `name` for the Lush object `anything` .

5.4.19 (gensym [x])

Create and return a new symbol. The name of the new symbol is the concatenation of a prefix and a suffix, the prefix being any string, the suffix being the decimal representation of a number. The default for the prefix string is "G", the default for the suffix number is the value of *gensym-counter*. If `x` is a string, it is taken as the prefix, if `x` is a number, it is taken for the suffix for the new symbol's name.

See: gensyms

```
? (list (gensym) (gensym "eXtra") (gensym 99))
= (|G449| |eXtra450| |G99|)
```

5.4.20 (gensyms n)

Return a list of `n` new symbols created by `gensym` .

5.4.21 (rotatef ...symbols...)

Change the values the `symbols` are bound to by rotating them one position to the left.

```
? (let (((a b c) '(5 6 7)))
    (rotatef a b c)
    (list a b c) )
= (6 7 5)
```

5.4.22 (symbol-concat ...symbols...)

Create a new symbol by concatenating the given symbols.

```
? (let ((prfx 'gnu))
    (symbol-concat prfx '-hurd) )
= gnu-hurd
```

5.5 Namespaces

Forms like `defvar` , `defun` , and `defclass` create a new object and bind a symbol in the global namespace to it. Technically, Lush has only this one global namespace (exceptions are slot names and method names). The namespace facility helps with avoiding name clashes. It offers a mechanisms to systematically modify the names (symbols) new objects are bound to and to manage sets of names.

The first mechanism, the `in-namespace` form, is to give a number of names a common prefix. All names occurring in definitions enclosed by `in-namespace` are prefixed by the namespace name. For example, evaluating the form

```
(in-namespace aaa-

(defun f (i)
  (+ i 1) )

(defun g (i)
  (- i 1) )

(print (f (g 5)))
)
```

results in the symbols `aaa-f` and `aaa-g` being bound to functions `(lambda (i) (+ i 1))` , and `(lambda (i) (- i 1))` , respectively.

The above example defines a new namespace `aaa-` including the names `f` and `g` . Expressions may be evaluated in namespace `aaa-` by use of the `with-namespace` form. Example:

```
? (with-namespace aaa-
    (f (f (f 3)))
  )
= 6
```

An alternative to enclosing code in `with-namespace` is to use `import` to globally bind selected unqualified names (a good place is at the beginning of a source file, just after a `libload` statement). Examples:

```
? (import (f) from aaa-)
= (f)
? (f (f (f 3)))
= 6


? (import (col row) from mat-)
= (row col)
? (row 1 2 3)
= [d[d  1.00  2.00  3.00]]
```

You may also "load definitions into a namespace" by placing `libload` statemens in a `in-namespace` form:

```
(in-namespace tools-
(libload "project/a-tools.lsh" t)
(libload "project/b-tools.lsh" t)
)
```

It is easy to create new namespaces from existing ones. This may be useful for creating a customized namespace that contains everything for a particular application. Examples:

```
(in-namespace my-custom-ns
(import all from mat-)
(import (x y z) from the-other-ns-)
)


(defnamespace my-custom-ns (join-namespaces mat- mut- meat-))
```

You may also use `defnamespace` to declare any mapping from symbols to symbols a namespace. Example:

```
(defnamespace abc- '((a apples) (b beets) (c chicory)))
```

Finally, `delete-namespace` deletes all objects bound to the qualified names of a namespace, then undefines the namespace. One use case is to put in a namespace all auxiliary definitions that are required for compilation of some code, then delete the namespace after compilation is completed.

5.5.1 Class namespaces

Every class defined with `defclass` has an associated namespace. All definitions placed in this namespace are only visible to method definitions for this class and derived classes. Class namespaces are referred to by namespace names of the form `(class X)` , where `X` is a class. Example:

```
(defclass AClass object
  pos)

(in-namespace (class AClass)

(defun -> (x y)
  ;; vector from x to y
  (- y x) )

(defun nrm2 (v)
  ;; euclidean length of vector v
  (sqrt (+ (* (v 0) (v 0)) (* (v 1) (v 1)))) )

) ; in-namespace

(defmethod AClass AClass (-pos)
  (declare (-idx1- (-double-)) -pos)
  (assert (= 2 (length -pos)))
  (setq pos -pos))

(defmethod AClass distance-to (other)
  (declare (-obj- (AClass)) other)
  (nrm2 (-> :this:pos :other:pos)) )
```

Definitions may be imported from a class namespace, just like from a regular namespace.

```
? ->
= ()
? (import (->) from (class AClass))
= (->)
? (-> [1 1] [2 2])
= [d  1.00  1.00]
```

5.5.2 (in-namespace `ns l1 l2` ...)

For every definition found in the forms `l1 l2` ... map the name `obj-name` the object is being bound to to the qualified name. The qualified name is the result of `(symbol-concat 'ns obj-name)` . Return the names of all newly defined objects.

When an object being defined references any name in the namespace, this name will be mapped to the corresponding qualified name, too. In other words, the code of the object being defined is modifed at read time. To protect a

symbol `name` from being mapped to its qualified name write `#:name` instead of `name` .

5.5.3 (in-namespace* ns l1 l2 ...)

Map references to names in namespace `ns` and return the names of all newly defined objects.

This is similar to `in-namespace` in that unqualified names of objects in namespace `ns` are mapped to qualified names. Unlike `in-namespace` , no new names are added to namespace `ns` .

5.5.4 (namespace ns-name)

Yield namespace named `ns-name` or `()` when there is no namespace called `ns-name` . Example:

```
? (pretty (namespace 'tiling-))
()
= t
```

5.5.5 (with-namespace ns l1 l2 ...)

Evaluate forms `l1 l2` ... in an environment where the objects bound to the qualified names of namespace `ns` are bound to their corresponding unqualified names. Example:

```
(with-namespace mat-
  (setq m (rot90 m))
)
```

5.5.6 (with-namespaces (ns1 ns2 ...) l1 l2 ...)

Shortcut for nested `with-namespace` forms.

5.5.7 (delete-namespace ns)

Delete all objects bound to the qualified names of `ns` , then undefine `ns` itself.

5.5.8 (join-namespaces ns1 ns2 ...)

Create a new namespace including all the names of the given namespaces `ns1` , `ns2` , etc.

5.5.9 (names ns)

Return list of names in namespace `ns` .

5.5.10 (qualified-names ns)

Return list of qualified names in namespace `ns` .

5.5.11 (import names from ns [as alt-names])

For each name in `names` defined in namespace `ns` create an alias in the current namespace. When alternative names `alt-names` are given, define the alias with the name in `alt-names` otherwise use the name in `names` . Examples:

```
(import (flipud rot90) from mat-)
```

```
(import (flipud rot90) from mat- as (mirror-vertical rotate90))
```

5.5.12 (import all from ns)

For every pair (`name qualified-name`) in namespace `ns` create an alias `name` for `qualified-name` and return the list of unqualified names.

5.5.13 (defnamespace name '((n1 qn1) (n2 qn2) ...))

Create a namespace mapping names `n1` , `n2` , ... to qualified names `qn1` , `qn2` , ..., bind global variable `name` to it, and return symbol `'name` .

5.5.14 (defnamespace name ns)

Bind global variable `name` to namespace `ns` and return symbol `'name` . Example:

```
(defnamespace blas- (join-namespaces blas1- blas2- blas3-))
```

5.6 Control Structures

Control structure are lisp functions specially designed for controlling how a program is executed. There are control structures for starting the execution (e.g. `eval`) and control structures for programming loops (e.g. `while`).

5.6.1 (eval l1...ln)

See: The Tlisp Evaluator.

The function `eval` calls the evaluator on objects `l1` to `ln` , and returns the result of the last evaluation, i.e. `(eval ln)` .

Example:

```
? (eval (cons '+ '(3 4)))
= 7
```

5.6.2 (apply f . args)

Function `apply` applies function `f` to all following arguments `args` . Evaluating `(apply f l)` is quite similar to `(eval (cons f l))` . It blocks however the evaluation of the arguments stored in list l. Example:

```
? (apply + 1 2 '(3 4 5))
= 15


? (apply append '((list 2 3) (list 4 (+ 1 2))))
= (list 2 3 list 4 (+ 1 2))


? (eval (cons append '((list 2 3) (list 4 (+ 1 2)))))
= (2 3 4 3)
```

5.6.3 (quote a)

See: ' `a`

See: ‘ `expr` , `a` ,@ `a`

Return `a` unevaluated.

Functions evaluate their arguments and operate on the results of these evaluations. In `(car (range 2 5))` for instance, function `car` evaluates the arguments `(range 2 5)` giving `(2 3 4 5)` and returns the first element of this list.

```
? (car (range 2 5))
= 2
```

The purpose of the special form `quote` is to easily pass S-expressions to a function. For instance, to pass to `car` the list consisting of the symbol `range` , the number `2` and the number `5` , you could write

```
? (car (list (make-symbol "range") 2 5))
= range
```

Here is how you would do it using `quote` :

```
? (car '(range 2 5))
= range
```

5.6.4 (dquote a)

See: quote

Return a quoted.

In some situations you need a "double quote".

```
? (mapcar dquote (list 'abc 123 "hi"))
= ('abc '123 '"hi")
```

5.6.5 'a

See: (quote a)

See: ‘ expr , a ,@ a

The function quote usually is called by using the quote ' macro-character. Writing 'a is a shorthand for (quote a) .

Example:

```
? (car '(range 2 5))
= range
```

```
? '(range 2 5)
= (quote (range 2 5))
```

5.6.6 ‘expr ,a ,@a

The backquote ‘ macro provides a handy syntax for producing complex lists depending of external conditions. The backquote macro works like the quote macro with the following additions:

- Within a backquoted expression, a comma , indicates that the next Lush object must be evaluated.
- If the comma is followed by an AT sign (,@), the list returned by the evaluation of the next Lush object will be merged within the current list.

Example:

```
? ‘(1 2 ,(+ 4 5) (+ 4 5 ,(range 1 4))))
= (1 2 9 (+ 4 5 (1 2 3 4)))
? ‘(1 2 ,(+ 4 5) (+ 4 5 ,@(range 1 4))))
= (1 2 9 (+ 4 5 1 2 3 4))
```

Implementation: Combinations of backquote and comma macros are expanded into lisp expressions composed of calls to functions list , append and cons , which builds the result.

Example:

```
;; note the quote <'> followed by a backquote <`>
? ' `(1 2 ,(+ 4 5) (+ 4 5 ,@(range 1 4)))
= (list '1 '2 (+ 4 5) (append '(+ 4 5) (range 1 4)))
```

5.6.7 (progn l1...ln)

Evaluates `l1` to `ln` and returns the result of the evaluation of `ln` .

Example:

```
? (progn
    (print (+ 2 3))
    (* 2 3) )
5
= 6
```

5.6.8 (prog1 l1...ln)

Evaluates `l1` to `ln` and returns the result of the evaluation of `l1` .

Example:

```
? (prog1
      (print (+ 2 3))
    (* 2 3) )
5
= 5
```

5.6.9 (let ((s1 v1) ... (sn vn)) l1 ... ln)

The function `let` is used to define local variables.

Function `let` performs the following operations:

- Expressions `v1` ... `vn` are evaluated.
- The values of symbols `s1` ... `sn` are saved.
- The results of the evaluations of `v1` ... `vn` are stored into symbols `s1` ... `sn` .
- Expressions `l1` ... `ln` are evaluated. If these expressions refer to symbols `s1` ... `sn` , they will see the values returned by expressions `v1` ... `vn` (unless you change these values using `setq` !)
- The saved values of `s1` ... `sn` are restored.
- The result of the evaluation of `ln` is returned.

Example:

```
? (let ((s 0))
    (for (i 1 10)
      (setq s (+ s i)) ) )
= 55
```

5.6.10 (lete ((s1 v1) ... (sn vn)) l1 ... ln)

See: `let` , `delete`

Same as `let` but explicitly `delete` all local variables after evaluating `ln` .

5.6.11 (let* ((s1 v1) ... (sn vn)) l1 ... ln)

Function `let*` is also used for defining local variables. Unlike function `let` however, the evaluation of the variable initial values are performed after setting the previous variables. You can therefore refer to the previously defined variabled.

Function `let*` performs the following operations:

- For each pair `(si vi)` , the value of symbol `si` is saved, the expression `vi` is evaluated and the result is stored into symbol `si` .
- Expressions `l1` ... `ln` are evaluated. If these expressions refer to symbols `s1` ... `sn` , they will see the values returned by expressions `v1` ... `vn` (unless you change these values using `setq` !)
- The saved values of `s1` ... `sn` are restored.
- The result of the evaluation of `ln` is returned.

Example:

```
? (let* ((i 1)
         (j (1+ i)) )
    (print i j)
    (* 2 j) )
1 2
= 4
```

5.6.12 (let-filter ((filter data)) l1...ln)

The function `let-filter` attempts to match the data expression `data` with the filter expression `filter` . A filter expression is a lisp expression composed uniquely with lists and symbols. For instance, argument `filter` can be a single symbol, a list of symbols, or something more complex such as `(a ((b c)) d . e)` .

- A symbol in the filter expresssion matches anything in the data expression. When such a match occurs, the previous value of the symbol is saved and the symbol is assigned the matching part of the data expression.

- A list in the filter expression matches a list in the data expression if all its elements match the correspondong elements in the data expression.

If there is a match, function `let-filter` evaluates the lists `l1` ... `ln` with the new values of the symbols listed in the match expression, restores the initial value of the symbols, and return the result of the last evaluation. Otherwise, function `let-filter` simply restores the initial values of the symbol and returns the empty list.

5.6.13 (if cond yes [no1...non])

First, expression `cond` is evaluated. If the result is not the empty list, expression `yes` is evaluated, otherwise the remaining expressions `no1` ... `non` are evaluated. Function `if` returns the result of the last evaluation.

Example:

```
? (if (> 3 4)
      3
    4 )
= 4
```

5.6.14 (when cond yes1...yesn)

First, expression `cond` is evaluated. If the result is not the empty list, expressions `yes1` to `yesn` are evaluated and the result of `yesn` is returned.

Example:

```
? (when (> 3 4)
    (print "error") )
= ()
```

5.6.15 (while cond l1...ln)

While the evaluation of `cond` gives a result different from the empty list, expressions `l1` to `ln` are evaluated in a loop.

If the result of the first evaluation of `cond` is the empty list, expressions `l1` to `ln` are never evaluated and function `while` returns the empty list. Otherwise function `while` returns the result of the last evaluation of `ln` .

Example:

```
? (let ((l (range 2 5)))
    (while l
      (print (car l) (sqrt (car l)))
      (setq l (cdr l)) ) )
2 1.4142
3 1.7321
4 2
5 2.2361
= ()
```

5.6.16 (do-while cond l1...ln)

Function `do-while` evaluates lists `l1` to `ln` and loops until the evaluation of condition `cond` returns the empty list. The result of the last evaluation of `ln` is then returned.

Expressions `l1` to `ln` are evaluated before testing for the loop condition. If the result of the first evaluation of `cond` is the empty list, expression `l1` ... `ln` are evaluated exactly once.

Example:

```
(de input(prompt regex)
   (let ((answer ()))
      (do-while (not (regex-match regex answer))
         (printf "\\%s" prompt)
         (flush)
         (setq answer (read-string)) )
      answer) )
```

5.6.17 (repeat n l1...ln)

Repeats `n` times the evaluation of `l1` to `ln` . Function `repeat` returns the result of the last evaluation of `ln` .

Example:

```
? (repeat 4
    (prin 1) )
1111= 1
```

5.6.18 (mapwhile cond l1...ln)

See: (while `cond l1` ... `ln`)

Function `mapwhile` first evaluates expression `expr` . If this evaluation returns a non nil result, it evaluates lists `l1` to `ln` . This process is repeated until the evaluation of `cond` returns the empty list.

Unlike function `while` however, function `mapwhile` returns a list containing all the results of the successive evaluation of `ln` .

```
? (let ((i 0))
    (mapwhile (< i 5)
      (incr i)
      (* i i) ) )
= (1 4 9 16 25)
```

5.6.19 (for (symb start end [step]) l1...ln)

See: for*

Function `for` implements a classical "for" loop. The value of symbol `symb` is first saved. Then `symb` takes numeric values from `start` to `end` stepping by `step` (default 1). For each value, expression `l1` to `ln` are evaluated. The value of `symb` is then restored.

Function `for` returns the result of last evaluation of `ln` .

Example:

```
? (for (i 2 5)
    (print i (sqrt i)) )
2 1.4142
3 1.7321
4 2
5 2.2361
= 2.2361
```

5.6.20 (for* (symb start end [step]) l1...ln)

See: for

Function `for*` implements a classical "for" loop. The value of symbol `symb` is first saved. Then `symb` takes numeric values from `start` to `end` , not including `end` , stepping by `step` (default 1). For each value, expression `l1` to `ln` are evaluated. The value of `symb` is then restored.

Function `for*` returns the result of last evaluation of `ln` .

Example:

```
? (for* (i 2 5) (print i (sqrt i)))
2 1.4142
3 1.7321
4 2
= 2
```

5.6.21 (mapfor (symb start end [step]) l1...ln)

See: (for (symb start end [step]) l1 ... ln)

Function mapfor implements a loop like function for . It returns however a list containing the results of the evaluation of ln for all values of the loop index.

Example:

```
? (mapfor (i 2 5)
    (sqrt i) )
= (1.4142 1.7321 2 2.2361)
```

5.6.22 (cond l1...ln)

Function cond implements the standard lisp "cond" form.

Arguments l1 ... ln are lists of the form (cond . body) . The conditions cond are evaluated sequentially until one returns a result different from the empty list. The function progn is applied to the corresponding body and the result is returned.

Example:

```
;; a function for returning the sign of a number
(de sign(x)
  (cond
     ((< x 0) -1)
     ((> x 0) +1)
     (t        0) ) )
```

5.6.23 (selectq s l1...ln)

The arguments l1 to ln are lists of the form (case . body) .

The argument s is first evaluated. The lists l1 to ln are then checked until s is equal to case , or s is a member of the list case , or case is equal to t . The function progn is applied to the corresponding body and the result is returned.

Example:

```
(selectq x
   (0             (print "zero"))
   ((2 4 6 8)     (print "even"))
   ((1 2 3 5 7)   (print "prime"))
   (t             (print "nothing interesting")) )
```

5.6.24 (mapc f l1...ln)

Apply `f` to successive tuples of arguments, where one argument is taken from each list `l1` ... `ln` . Return the first list `l1` . The iteration terminates with the last atom in the shortest list, excess elements in other lists are ignored.

Any argument `l1` ... `ln` that is not a list will automagically be turned into an infinite list.

Example:

```
? (mapc print '(1 2 3))
1
2
3
= (1 2 3)


? (mapc (lambda (x y)
          (print (+ x y)) )
    '(1 2 3)
    (range 10) )
2
4
6
= (1 2 3)
```

5.6.25 (mapcar f l1...ln)

Apply `f` to successive tuples of arguments, where one argument is taken from each list `l1` ... `ln` . Accumulate the results of the successive function applications in a list and return it. The iteration terminates with the last atom in the shortest argument list, excess elements in other lists are ignored.

Any argument `l1` ... `ln` that is not a list will automagically be turned into an infinite list.

Example:

```
? (mapcar + '(1 2 3) '(4 5 6 7 8 9))
= (5 7 9)


? (mapcar + '(1 2 3) 5)
= (6 7 8)
```

5.6.26 (mapcan f l1...ln)

Apply `f` to successive tuples of arguments, where one argument is taken from each list `l1` ... `ln` . Concatenate the results of the successive function applica-

tions in a list and return it. The iteration terminates with the last atom in the shortest argument list, excess elements in other lists are ignored.

Any argument `l1` ... `ln` that is not a list will automagically be turned into an infinite list.

Example:

```
? (mapcan (lambda (n e)
            (make-list n e) )
    '(1 2 3)
    '(a b c d e) )
= (a b b c c c)
```

5.6.27 (tree-mapcar `f l1...ln`)

See: (mapcar `f l1` ... `ln`)

`mapcar` for trees.

Example:

```
? (tree-mapcar + '(1 2 (3 (4)) 5) '(1 2 (3 (4)) 5))
= (2 4 (6 (8)) 10)
```

5.6.28 (reduce> `f carry s1...sn`)

Apply a function of n+1 arguments recursively to `carry` and all `car` s of `s1` ... `sn` , from first to last. Return the final carry. The lists `s1` ... `sn` must have the same length. Examples:

```
? (reduce> cons 's '(1 2 3 4))
= ((((s . 1) . 2) . 3) . 4)


? (reduce> + 0 '(1 2 3 4))
= 10


? (reduce> idx-trim (double-array 5 5) '(0 1) '(1 3))
= [d[d   0.0000   0.0000]
    [d   0.0000   0.0000]
    [d   0.0000   0.0000]
    [d   0.0000   0.0000]]
```

5.6.29 (reduce< f carry s1...sn)

Apply a function of n+1 arguments recursively to `carry` and all `car` s of `s1` ... `sn` , from last to first. Return the final carry. The lists `s1` ... `sn` must have the same length. Example:

See: reduce>

```
? (reduce< cons 's '(1 2 3 4))
= ((((s . 4) . 3) . 2) . 1)
```

5.6.30 (dolist (s l) . body)

See: domapc domapcar

For each element `e` in list `l` , evaluate `body` with `s` set to `e` . Return `nil` . Example:

```
? (dolist (i '(1 2 3 5 7)) (print i (sqrt i)))
1 1
2 1.4142
3 1.7321
5 2.2361
7 2.6458
= ()
```

5.6.31 (domapc ((s1 l1) ... (sn ln)) . body)

See: dolist

Iteratively evaluate `body` with variables `s1` ... `sn` set to the element of `l1` ... `ln` with index i, respectively, with i=0, 1, ... The iteration terminates when the shortes list in `l1` ... `ln` is exhausted. Return the first list `l1` . Example:

```
? (domapc ((i '(1 2 3)) (j '(3 2 1))) (print (+ i j)))
4
4
4
= (1 2 3)
```

5.6.32 (domapcar ((s1 l1) ... (sn ln)) . body)

See: domapc

Evaluate `body` like `domapc` but return all evaluation results in a list. Example:

```
? (domapcar ((i '(1 2 3 5 7))) (sqrt i))
= (1 1.4142 1.7321 2.2361 2.6458)
```

5.6.33 (domapcan ((s1 l1) ... (sn ln)) . body)

See: domapcar

Evaluate `body` like `domapcar` but the result of each evaluation must be a list, and combine all result lists into one list. Example:

```
? (domapcan ((i '(1 2 3 5 7))) (list i (sqrt i)))
= (1 1 2 1.4142 3 1.7321 5 2.2361 7 2.6458)
```

5.6.34 (each ((s1 v1) ... (sn vn)) l1 ... ln)

Alias for `domapc` .

5.6.35 (all ((s1 v1) ... (sn vn)) l1 ... ln)

Alias for `domapcar` .

5.7 Iterables and Iterators

Iterators are an abstraction for sequential data access and deferred computation (akin to "lazy evaluation"). A stock of generic iterator functions brings the convenience of generic list processing functions to a wider class of objects, that is, objects that are "iterable".

An object `obj` is "iterable" if the expression `(iterate obj)` evaluates to an iterator.

An iterator `it` is an object that supports the "iterator protocol", which means that the four expressions `(emptyp it)` , `(iterate it)` , `(next it)` , and `(peeknext it)` are legal and adhere to the following semantics:

- `(iterate it)` evaluates to `it`
- `(emptyp it)` evaluates to true (`t`) or false (`()`). When `(emptyp it)` evaluates to false then a subsequent evaluation of `(next it)` or `(peeknext it)` is guaranteed to succeed.
- `(next it)` evaluates to an item or causes a runtime error.
- `(peeknext it)` evaluates to an item or causes a runtime error. When `(peeknext it)` evaluates to an item, then `(next it)` will evaluate to the same item.

In a typical use case, an iterator object gives access to "items" of some iterable object and yields these items in a particular order. Function `next` prompts the iterator to yield the next item and "crosses it off", so that in general, subsequent calls to `next` return different items. Function `peeknext` returns the next item without "crossing it off", so subsequent calls to `peeknext` always return the same item. Function `emptyp` finally tests whether there are any more items.

5.7.1 Writing Generic Iterator Functions

A function that uses only the four functions `iterate` , `emptyp` , `next` , and `peeknext` to access items of an iterable object is "generic" in that it works with all iterable objects. Code that uses only generic functions is also generic.

(iterate `obj` [`arg`])

Return an iterator for object `obj` or raise an error when `obj` cannot be iterated.

There are two behaviors for iteration of functions: First, when no argument `arg` is provided, the iterator calls `obj` without arguments in each iteration step. Second, if an argument `arg` is provided, the iterator calls `obj` with argument `arg` in the first step and with the last result in every subsequent step. Examples:

```
? (take 8 (iterate rand))
= (0.4579 0.2628 0.1898 0.3359 0.6329 0.0636 0.5097 0.4751)

? (take 10 (iterate (lambda (x) (+ x x)) 1))
= (2 4 8 16 32 64 128 256 512 1024)
```

(next `it`**)**

Extract next item from iterator `it` and return it.

(peeknext `it`**)**

Return next item in iterator `it` without removing it.

5.7.2 Generic Iterator Functions

(do ((`i1 obj1`**) ... (**`in objn`**)) [while** `cond`**] .** `body`**)**

Iterate objects `obj1` to `objn` in parallel until any object is empty. If an optional condition `cond` is given, then also terminate when `cond` is not true.

Iteration terminates prematurely when `continue` is set to false in `body` . Examples:

```
(do ((ln (range 1 1000)) (line (lines "readme.txt")))
  (printf "\\%3d :  \\%s\\n" ln line) )

(do ((i hp)) while (<> i -1)
  (print i))
```

(take `n obj`**)**

Take up to `n` items from `obj` and return in a list.

(atake `n obj [element-type]`**)**

Take up to `n` items from `obj` and return in a vector. Example:

```
? (atake 10 1s)
= [a 1 1 1 1 1 1 1 1 1]
? (atake 10 1s 'double)
= [d  1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00]
```

(drop `n obj`**)**

Drop up to `n` items and return the iterator.

(take-while `p obj`**)**

Take items from `obj` as long as (`p item`) is true; return the items in a list.

(drop-while `p obj`**)**

Drop items from `obj` as long as (`p item`) is true; return the iterator.

(take-until `x obj`**)**

Take items from `obj` until item `x` ; return the items in a list (including `x`).

(take-until* `x obj`**)**

Take items from `obj` until item `x` ; return the items in a list (not including `x`).

(drop-until `x obj`**)**

Drop items from `obj` until item `x` ; return the iterator. The last item returned by the iterator was `x` .

(drop-until* `x obj`**)**

Drop items from `obj` until item `x` ; return the iterator. The next item delivered by the iterator will be `x` .

Builtin Iterables and Iterators

0s
Iterable generating `0` .
1s
Iterable generating `1` .
nils
Iterable generating `()` .

ts
Iterable generating `t` .
(ints)
Iterator generating the numbers 0, 1, 2, ...
(irange [`from`] `to` [`step`]**)**
Like `range` but returns an iterator, not a list.
(irange* [`from`] `to` [`step`]**)**
Like `range*` but returns an iterator, not a list.
(irange*/s [`from`] `to`**)**
Generate integers from `from` to `to` but shuffled. Example:

```
(take 10 (irange*/s 10)))
= (8 3 6 1 9 4 7 2 5 0)
```

(lines `filename`**)**
Iterator generating the lines of file `filename` .

5.8 Functions

A Lush function is an unnamed object which defines how to evaluate a list whose first element evaluates to this function. Functions are usually stored in the value field of a symbol usually referred to as the function name.

There are thus two equivalent ways to define with global scope a function named `square` which computes the square of its argument:

```
(defvar square (lambda(x) (* x x)))
or

(de square(x) (* x x))
```

The evaluation of the list `(lambda...)` returns a function that `setq` stores into symbol `square` in the global environment. The function `de` is a shorthand for this manipulation.

A function behaves as any atom does. Its class however defines how to evaluate a list whose first element evaluation returns that function. The evaluation of a list whose first element is a symbol, the value of which is a function, thus simply calls that function. The evaluation of the following list then returns 25

```
(square 5)
```

But the evaluation of the following list also returns 25

```
((lambda(x) (* x x)) 5)
```

There are several kinds of functions in Lush; briefly described hereafter:

- DX functions are written in C, using the standard Lisp to C interface package. Most of the C functions are written as DX.
- DY functions are directly written in C. These functions are usually control structures, like `cond` , `progn` , etc...
- DE functions are written in Lisp. These functions evaluate their arguments before calling their body. Most Lisp functions are DE functions.
- DF functions are also written in Lisp. They do not however evaluate their arguments before giving control to the function body.
- DM functions are also known as "macro functions". They are written in Lisp. Their argument list includes the function name itself, and its elements are not evaluated. The body of the function is supposed to return an evaluable list, known as the macro expansion. This list is then evaluated in the calling context, and gives the result of the function call.
- DMD functions are a refinement of DM functions. Unlike the DM functions, their argument list does not include the function name. The result of the body evaluation replaces the calling form itself in the calling context. The evaluation process is then restarted. This replacement process happens just once, because the code is physically replaced by the result of the macro expansion during the first call.
- DMC are known as "macro characters". These are not real function. They are bound to a single character name and take no arguments. Once the Lush reader reaches such a character, it calls the function body and substitutes the result to the macro-character.

5.8.1 Argument List

A formal argument list is associated to each each Lush function (i.e. DE, DF, DM or DMD) This argument list defines which symbols will be bound to the actual argument of the function, while evaluating its body.

Valid formal argument lists may be a single symbol, a list of symbols, or something more complex like `(a b (c d) . f)` , or `(a b &optional c d)` .

If the formal argument list is a single symbol, it will be bound to the list of arguments when the function is called.

If the formal argument list is a list, its "car" will be matched against the "car" of the actual arguments, and its "cdr" will be matched against the "cdr" of the actual arguments.

Example:

```
? (de surface((x1 y1) (x2 y2) )
       (* (- x1 x1) (- y2 y2)) )
```

```
= surface
? (surface '(4 5) '(8 10))
= 40
```

Moreover, two symbols `&optional` and `&rest` in argument lists have a special meaning, similar to the Common Lisp conventions.

Only symbols and lists composed of a symbol and a default value may follow the `&optional` symbol in the formal argument list. These symbols are optional arguments. If they are omitted, they are bound to their default value, or the empty list.

Example:

```
? (de printline(s &optional (indent 0) (terminate "."))
       (tab indent)
       (printf "\\%s\\%s\\n" s terminate) )
= printline
? (printline "hello")
hello.
= ()
? (printline "hello" 6)
      hello.
= ()
? (printline "hello" 6 "@")
      hello@
= ()
```

Finally, the formal argument list may be terminated by `&rest` followed by another symbol. The list of all remaining actual arguments will be bound to that symbol.

5.8.2 Compact Lambda Expressions

Anonymous functions (lambda expressions) are often used in combination with higher-order functions like `mapcar` or `filter` .

```
(filter (lambda (f) (str-endswith f "jpg")) (ls "data/images"))
```

Lush allows to define lambda expression more compactly using the macro-character `'#\'` . The macro character must be followed by a lisp form, the body of the lambda expression. Positional arguments to the lambda have the canonical names `$1` , `$2` , and so on. The highest-numbered symbol of the form `$i` determines the number of arguments of the lambda expression (i less than 10).

```
(filter #\\(str-endswith $1 "jpg") (ls "data/images"))

(#\\(- $5 $3) 10 11 12 13 14)
```

The positional arguments may be followed by an arbitrary number of optional arguments. The optional arguments are referred to with the ellipsis, `..`.

```
(#\\(+ $3 ..) 1 2 3 4 5)  ; ignore first two

(#\\(/ (+ ..) (length (list ..))) 1 2 3 4 3)  ; average
```

5.8.3 Defining a Function

(de `symb args . body`**)**

See: Argument List.

Creates a new function which evaluates its arguments (DE) and binds it to the symbol `symb` in the global environment.

A valid argument list `args` may be a single symbol, a list of symbols or something more complex like `(a b (c d) . f)` . A valid function body `body` is a list of Lush expressions which are evaluated whenever the function is called.

Whenever a DE function is called,

- The previous values of the argument names are saved,
- The actual arguments are evaluated and stored in the corresponding argument names,
- Each expression in the function body is evaluated and the last result is saved,
- The previous values of the argument names are restored,
- The saved result is returned.

Example:

```
? (de square (x) (* x x))
= square
? (square 9)
= 81
? (de first-arg l
        (car l) )
= first-arg
? (first-arg 'a 'b)
= a
? (first-arg 'prems 'deuse 'troise 'quatrz)
= prems
```

(df `symb args . body`**)**

See: Argument List.

See: (dm `symb args . body`)

Creates a new function which does not evaluates its arguments (DF) and binds it to the symbol `symb` in the global environment.

A valid argument list `args` may be a single symbol, a list of symbols or something more complex like `(a b (c d) . f)` . A valid function body `body` is a list of Lush expressions which are evaluated whenever the function is called.

Whenever a DF function is called,

- The previous values of the argument names are saved,
- The actual arguments are not evaluated but directly stored in the corresponding argument names,
- Each expression in the function body is evaluated and the last result is saved,
- The previous values of the argument names are restored,
- The saved result is returned.

Example: Defining a control structure

```
? (df ifn(cond no . yes)
      (if (not (eval cond))
          (eval no)
        (apply progn yes)) )
= ifn
? (ifn (= 2 3) 1 2))
= 2
```

Such a definition causes problems if some subexpression depend on a the value of a symbol named like a function argument.

```
? (ifn (= 2 3) cond 2)
= (= 2 3)
```

The previous expression should indeed return the value of symbol `cond` in the current context rather than the value of symbol `cond` in the context of function `ifn` .

This problem is alleviated by using “macro functions”.

(dm `symb args . body`**)**

See: Argument List.

Creates a macro-function (DM) and stores binds it to symbol `symb` in the global environment. The `body` of a DM function actually computes the expression which will be evaluated in the calling context. This expression can be found with the function `macroexpand` .

A valid argument list `args` may be a single symbol, a list of symbols or something more complex like `(a b (c d) . f)` . A valid function body `body` is a list of Lush expressions which are evaluated whenever the function is called.

Whenever a DM function is called,

- The previous values of the argument names are saved,
- The elements of the entire calling expression, including the function name, are stored in the corresponding argument names,
- Each expression in the function body is evaluated and the last result (known as the macro expansion) is saved,
- The previous values of the argument names are restored,
- The saved macro-expansion is evaluated,
- The result is returned.

Example:

```
? (dm ifn(fname cond . rest)
      (list 'if (list 'not cond) . rest) )
= ifn
? (ifn (= 2 3) "yes" "no")
= "yes"
? (setq cond "yes")
= "yes"
? (ifn (= 2 3) cond)
= "yes"
```

(macroexpand `macrocall`**)**

See: (dm `symb args . body`)

Returns the "expansion" of a call to a DM function.

Example:

```
? (dm ifn(fname cond . rest)
      (list 'if (list 'not cond) . rest) )
= ifn
? (macroexpand (ifn (= 2 3) "yes" "no"))
= (if (not (= 2 3)) "yes" "no")
```

(dmd `symb args . body`**)**

See: Argument List.

See: (dm `symb args . body`)

Creates a DMD function and binds it to symbol `symb` in the global environment. DMD functions display two differences with DM functions:

- Unlike the DM functions, their argument list does not include the function name.
- The result of the body evaluation replaces the calling form itself in the calling context.

A valid argument list `args` may be a single symbol, a list of symbols or something more complex like `(a b (c d) . f)` . A valid function body `body` is a list of Lush expressions which are evaluated whenever the function is called.

Whenever a DMD function is called,

- The previous values of the argument names are saved,
- The arguments are stored in the corresponding argument names,
- Each expression in the function body is evaluated and the last result (known as the macro expansion) is saved,
- The previous values of the argument names are restored,
- The macro expansion is physically installed in the calling form using function `displace` , replacing forever the call to the DMD function in the calling code,
- The saved macro-expansion is evaluated,
- The result is returned.

Example:

```
? (dmd ifn(cond . rest)
       (list 'if (list 'not cond) . rest) )
= ifn
? (de if-test(n)
       (ifn (= n 2) "yes" "no") )
= if-test
? (if-test 2)
= "no"
? (pretty if-test)
(de if-test(n)
   (if (not (= n 2)) "yes" "no") )
= (if-test)
```

(dmc `symb . body`**)**

Defines a macro-character `symb` . A macro-character is not really functions, since it interacts only with the Lisp reader.

There are three kinds of macro-characters:

- The first ones are single character macro-characters. When the Lush reader encounters one of them, it immediately calls the associated DMC function and returns its result.
- Caret macro-characters have a two character name, whose first character is a caret `^` . A caret DMC behaves exactly like a single characters DMC: When the reader encounters a caret followed by another character, it immediately calls the associated DMC function and returns its result.

  Caret macro-character may also be abbreviated by typing the corresponding control character (if this control character exists and is not intercepted by the operating system). If you are running WinLush, the fastest method consists in typing `Ctrl+Shift+Letter` . It is always possible however to type the caret followed by the character.
- Hash macro-characters are also two character macro-characters, whose first character is a hash sign `#` . If their body returns (), nothing is read. If it returns a one element list, this element is read. If it returns anything else, an error is signaled.

  Hash macro characters are useful for conditionally reading certain pieces of code.

Since the reader performs special actions whenever it encounters a macro-character, it is advisable to surround the name of symbol `symb` with vertical bars `|` .

Examples:

The macro character `'` which expands into a call to function `quote` is defined as :

```
(dmc |'| (list 'quote (read)))
```

The macro character `^P` which expands into a call to function `pretty` is defined as :

```
(dmc |^P| (list 'pretty (read)))
```

The following macro-character `#!` could be used to signal debug instructions that will be not read if variable `ndebug` is set.

```
(dmc |#!| (let ((inst (read))) (when (not ndebug) inst)))
```

(lambda `args . body`**)**

See: Argument List.

See: (de `symb args . body`)

Returns a function which evaluates its arguments (DE). This function operates like function `de` but does not store the function in a particuler symbol.

Example:

```
? ((lambda (x)
     (* x x) ) (+ 4 5))
= 81
```

(flambda `args . body`**)**

See: Argument List.

See: (df `symb args . body`)

Returns a function which does not evaluates its arguments (DF). This function operates like function `df` but does not store the function in a particuler symbol.

Example:

```
? ((flambda (x)
     (print x) ) (+ 4 5))
(+ 4 5)
= (+ 4 5)
```

(mlambda `args . body`**)**

See: Argument List.

See: (dm `symb args . body`)

Returns a macro-function (DM). This function operates like function `dm` but does not store the function in a particuler symbol.

(funcdef `n`**)**

Returns a list which defines the function `n` . This works for DEs, DFs, DMs and other interpreted functions, but returns nil for DH, DX, DY, and other compiled or intrinsic functions.

Example:

```
? (funcdef caddr)
= (lambda (l)
    (car (cddr l)) )
```

(pretty f)

Display a nicely indented definition of function `f` .

Actually, `pretty` sends a pretty message to `f` . Each class defines how an instance of that class will be displayed. In particular, functions are displayed by printing an indented version of the definition of the functions f1 to fn.

Example

```
? (pretty caddr)
(lambda (l)
  (car (cddr l)) )
= t
```

^P `function`

See: (pretty `f`)

This macro character expands into a call of function `pretty` .

Example:

```
? ^Paddpath
(de addpath (dir)
  (setq dir (concat_fname dir))
  (let ((oldpath (path))
        (newpath (list dir)) )
    (while oldpath
      (when (<> dir (car oldpath))
        (setq newpath (nconc1 newpath (car oldpath))) )
      (setq oldpath (cdr oldpath)) )
    (apply path newpath) ) )
= ()
```

(defun `name args` **...**`body`**...)**

See: (de `symb args . body`)

See: (defmacro `name args` ... `body` ...)

See: (defvar `name` [`val`])

Function `defun` defines a global DE function named `name` . Whereas function `de` defines symbol `name` in the current scope, function `defun` defines symbol `name` in the global scope.

(defmacro `name args` **...**`body`**...)**

See: (de `symb args . body`)

See: (defun `name args` ... `body` ...)

See: (defvar `name` [`val`])

Function `defun` defines a global DM function named `name` . Whereas function `dm` defines symbol `name` in the current scope, function `defun` defines symbol `name` in the global scope. The argument list `args` only matches the arguments of the macro. It should not contain a symbol for matching the function name itself.

5.8.4 Processing optional keywords arguments

Unlike Common Lisp, Lush does not have special syntax for keyword arguments. But using the function `parse-kwdargs` it is easy to process optional keyword arguments that are given as an alternating list of keywords (symbols) and values. For example, a function `make-rect` may take four regular, positional arguments `x` , `y` , `w` , and `h` , and two optional keyword arguments `color` , and `filled` . Typical incovations of `make-rect` would look like this:

```
(make-rect 10 10 100 200)
(make-rect 10 10 100 200 'color 'green)
(make-rect 0 0 500 500 'filled t 'color 'blue)
```

Keyword arguments may appear in any order and combination after the positional arguments. Since they are optional, there must be a default value specified for each of them. Let's assume the default values for `color` and `filled` in our example are `black` , and `t` , respectively. The definition of `make-rect` then might look like this:

```
(de draw-rect (x y w h &rest args)
  (let* ((kwddefs  (alist 'filled t  'color 'black))
         (arglist  (apply alist args))
         (option   (lambda (kwd) (alist-get kwd arglist))) )
    (when (< (length kwddefs) (length arglist))
      (error "unknown keyword argument") )
    (set-color (option 'color))
    (draw-rectangle x y w h)
    (when (option 'filled)
      (fill-region (1+ x) (1+ y)) )))
```

Function `parse-kwdargs` implements this common keyword processing pattern and creates a hash table of key-value pairs. It also raises an error with a specific error message when an unkown keyword is passed to a function. Using `parse-kwdargs` the above function would be written like this:

```
(de draw-rect (x y w h &rest args)
  (let ((option (parse-kwdargs args  'filled t  'color 'black)))
    (set-color (option 'color))
```

```
(draw-rectangle x y w h)
(when (option 'filled)
  (fill-region (1+ x) (1+ y)) )))
```

(parse-kwdargs `args kwd1 val1 ... kwdn valn`**)**

See: Processing optional keyword arguments

See: parse-kwdargs*

Parse keyword argument list. For each atom with odd index in list `args` raise an error if it is not in the set `kwd1 ... kwdn` . Create a hash table with keys `kwd1 ... kwdn` and return it. For each key `kwdi` the associated value is either `vali` , or, if `kwdi` is present in `args` , is the item following `kwdi` in `args` . Example:

```
? (let ((args '(filled t  color black)))
    (parse-kwdargs args 'filled () 'color 'white) )
= ::HTable:938a158
```

(parse-kwdargs* `args kwd1 val1 ... kwdn valn`**)**

See: Processing optional keyword arguments

See: parse-kwdargs

Parse keyword argument list. Like `parse-kwdargs` but does not raise an error when an unkown keyword is encountered.

(remove-kwdargs `args kwd1..kwdn`**)**

See: Processing optional keyword arguments

See: parse-kwdargs

Remove arguments from keyword argument list. With `args` a list of alternating keywords and values, and `kwd1 ... kwdn` being keyword symbols, remove symbols `kwd1 ... kwdn` and their subsequent values, respectively, and return the shortened argument list. Example:

```
? (let ((args '(filled t  color black)))
    (remove-kwdargs args 'color) )
= (filled t)
```

(update-kwdargs `args kwd val` [`kwd2 val2 ...`]**)**

See: Processing optional keyword arguments

See: parse-kwdargs

Set or update arguments in keyword argument list. With `args` a list of alternating keywords and values, `update-args` replaces the value of keyword `kwd` by `val` when `kwd` is in `args` , or adds the pair `kwd val` to the list when `kwd` is not in `args` . Example:

```
? (let ((args '(filled t  color black)))
    (update-kwdargs args 'color 'yellow 'area 24) )
= (area 24 color yellow filled t)
```

5.9 Strings

Strings are nul-terminated sequences of characters. `String` objects are immutable.

The usual ASCII characters are represented with a single bytes. Some characters are represented with multiple bytes. Most Lush functions deal with strings as sequences of bytes without regard to their character interpretation. Exceptions to this rule are indicated when appropriate.

The textual representation of a string is composed of the characters enclosed between double-quotes. A string may contain macro-characters, parentheses, semi-colons, as well as any other character. A line terminating backslash indicates a multi-line string.

The following "C style" escape sequences are recognized inside a string:

- \\ for a single backslash,
- \" for a double quote,
- \n , \r , \t , \b , \f respectively for a linefeed character (ASCII LF), a carriage return (ASCII CR), a tab character (ASCII TAB), a backspace character (ASCII BS), and a formfeed character (ASCII FF),
- \e for a `end-of-file` character (Stdio's EOF),
- \^? for a control character `control-?` ,
- \ooo for a byte whose octal representation is `ooo` .
- \xhh for a byte whose hexadecimal representation is `hh` .
- \uhhhh or \Uhhhhhh for the representation of unicode character `hhhh` or `hhhhhh` in the current locale. If no such representation exists, the UTF-8 representation is used.

5.9.1 String Indexing

Strings may be indexed like vectors. An indexed string expression evaluates to the code point of the character at the indexed position.

Example:

```
? ("halleluja" 1)
= 97
```

```
? (chr ("halleluja" 1))
= "a"
```

5.9.2 Basic String Functions

Like most Lush functions, the basic functions operating on strings do not modify their arguments. They create instead a new string on the basis of their arguments.

(make-string `n` `s`**)**

Create string by `n` times concatenating string `s` .
Example:

```
? (make-string 5 ">-<")
= ">-<>-<>-<>-<>-<"
```

(vector-to-string `v`**)**

See: string-to-vector
Create a string from `int` vector `v` .
Example:

```
? (vector-to-string [i    65    66    67])
= "ABC"
```

(string-to-vector `s`**)**

See: vector-to-string
Create an `int` vector from string `s` .
The values in the resulting vector are the character codes of the characters in `s` .
Example:

```
? (string-to-vector "offhand")
= [i   111   102   102   104    97   110   100]
```

(str-cat `s1` **...** `sn`**)**

Concatenate strings `s1` to `sn` and return the new string.
Example:

```
? (str-cat "hello" " my friends")
= "hello my friends"
```

(str-mid `s n` [`l`]**)**

See: substring

Returns a substring of `s` composed of `l` characters starting at position `n` (a number between 0 and length of `s` minus 1). With two arguments `str-mid` returns characters until the end of string `s` .

Example:

```
? (str-mid "alphabet" 3 2)
= "ha"


? (str-mid "alphabet" 3)
= "habet"
```

(substring `s l r`**)**

See: str-mid

Return a substring of `s` starting with the `l` -th character and ending before the `r` -th character.

Examples:

```
? (substring "alphabet" 0 5)
= "alpha"


? (substring "alphabet" 1 100)
= "lphabet"


? (substring "alphabet" 1 -1)
= "lphabe"
```

(str-right `s n`**)**

See: str-left

Return the `n` rightmost characters of `s` as a new string. When `n` is negative, drop the leftmost `-n` characters of `s` and return a new string.

Examples:

```
? (str-right "alphabet" 3)
= "bet"


? (str-right "alphabet" -3)
= "habet"
```

(str-left `s n`**)**

See: str-right

Return the `n` leftmost characters of `s` as a new string. When `n` is negative, drop the rightmost `-n` characters of `s` and return a new string.

Examples:

```
? (str-left "alphabet" 3)
= "alp"


? (str-left "alphabet" -3)
= "alpha"
```

(str-insert `s n s1`**)**

Insert string `s1` before character `n` of string `s1` and return the new string. When `n` is equal to 0, `str-insert` actually concatenates `s2` and `s1` .

Example:

```
? (str-insert "alphabet" 3 "***")
= "alp***habet"
```

(str-subst `s s1 s2`**)**

Substitute all occurrences of `s1` in `s` by `s2` and return new string.

Example:

```
? (str-subst "moon in the afternoon" "oo" "++")
= "m++n in the aftern++n"
```

(str-del `s n l`**)**

Remove `l` characters from string `s` starting with character `n` .

Example:

```
? (str-del "alphabet" 3 2)
= "alpbet"
```

(str-join `unsep ss`**)**

Concatenate all strings in list `ss` with string `unsep` in between. Return the string. `Str-join` is the inverse of `str-split` .

See: str-split

Example:

```
? (str-join "+" (mapcar str (range 10)))
= "1+2+3+4+5+6+7+8+9+10"
```

(str-split `s sep`**)**

Break string `s` into pieces at all occurrences of substring `sep` . Return the list of pieces (if `sep` does not occur in `s` , return a list with `s` as its single element). `Str-split` is the inverse of `str-join` .

See: str-join

Example:

```
? (str-split "voelker" "e")
= ("vo" "lk" "r")
```

(str-startswith `s prefix`**)**

True if string `s` starts with `prefix` .

(str-endswith `s suffix`**)**

True if string `s` ends with `suffix` .

(str-find `s r` [`n`]**)**

Searches the first occurrence of the string `s` in the string `r` , starting at byte position `n` . `Str-find` returns the position of the first match. If such an occurrence cannot be found, it returns the empty list.

Example:

```
? (str-find "pha" "alpha alphabet alphabetical" 4)
= 8
```

(upcase `s`**)**

Returns string `s` with all characters converted to uppercase according to the current locale.

Example:

```
? (upcase "alphabet")
= "ALPHABET"
```

(upcase1 s)

Returns string `s` with first character converted to uppercase according to the current locale.

Example:

```
? (upcase1 "alphabet")
= "Alphabet"
```

(downcase s)

Returns string `s` with all characters converted to lowercase according to the current locale.

Example:

```
? (downcase "alPHABet")
= "alphabet"
```

(str-val s)

Returns the numerical value of `s` considered as a number. Returns the empty list if `s` does not represent a decimal or hexadecimal number.

Example:

```
? (str-val "3.14")
= 3.14


? (str-val "abcd")
= ()


? (str-val "0xABCD")
= 43981
```

(str n)

Returns the decimal string representation of the number `n` .

Example:

```
? (str (2* 3.14))
= "6.28"
```

(strhex n)

Returns the hexadecimal string representation of integer number `n` .

Example:

```
? (strhex 18)
= "0x12"
```

(asc s)

Return the code point of the first character in string `s` . This function causes an error if `s` is an empty string.

Example

```
? (asc "abcd")
= 97
```

(chr n)

Return string containing a single character whose encoding is `n` . (Current limitation: `n` must be in the range 0 to 255.)

Example

```
? (chr 48)
= "0"
```

(isprint s)

Returns `t` if string `s` contains only printable characters according to the current locale.

Example:

```
? (isprint "alpha bet")
= t


? (isprint "alpha\\^Cbet")
= ()
```

(pname l)

Returns a string representation for the lisp object `l` . `pname` is able to give a string representation for numbers, strings, symbols, lists, etc...

Example:

```
? (pname (cons 'a '(b c)))
= "(a b c)"
```

(sprintf `format` ... `args` ...)

Like the C language function `sprintf` , this function returns a string similar to a format string `format` . The following escape sequences, however are replaced by a representation of the corresponding arguments of `sprintf` :

- `"%%"` is replaced by a single \%.
- `"%l"` is replaced by a representation of a lisp object.
- `"%[-][n]s"` is replaced by a string, right justified in a field of length `n` if `n` is specified. When the optional minus sign is present, the string is left justified.
- `"%[-][n]d"` is replaced by an integer, right justified in a field of `n` characters, if `n` is specified. When the optional minus sign is present, the string is left justified.
- `"%[-][n[.m]]c"` where `c` is one of the characters `e` , `f` or `g` , is replaced by a floating point number in a `n` character field, with `m` digits after the decimal point. `e` specifies a format with an exponent, `f` specifies a format without an exponent, and `g` uses whichever format is more compact. When the optional minus sign is present, the string is left justified.

Example:

```
? (sprintf "\\%5s(\\%3d) is equal to \\%6.3f\\n" "sqrt" 2 (sqrt 2))
= " sqrt(  2) is equal to  1.414\\n"
```

(str-strip `s`)

This function deletes the leftmost and rightmost spaces in string `s` .

```
(str-strip "  lots of space   ")
```

(str-stripl `s`)

This function deletes the leftmost spaces in string `s` .

```
(str-stripl "  lots of space   ")
```

(str-stripr s)

This function deletes the rightmost spaces in string `s` .

```
(str-stripr "   plenty of spaces   ")
```

5.9.3 Regular Expressions (regex)

A regular expression describes a family of strings built according to the same pattern. A regular expression is represented by a string which "matches" (using certain conventions) any string in the family.

The conventions for describing regular expressions in Lush are quite similar to those used by the `egrep` unix utility:

- An ordinary character matches itself. Some characters, `( ) \ [ ] | . ? * ` and `\` have a special meaning, and should be quoted by prepending a backslash `\` . The string `"\\\\"` actually is composed of two backslashes (because backslashes in strings should be escaped!), and thus matches a single backslash.

- A dot `.` matches any byte.

- A caret `^` matches the beginning of the string.

- A dollar sign `$` matches the end of the string.

- A range specification matches any specified byte. For example, regular expression `[YyNn]` matches `Y y N` or `n` , regular expression `[0-9]` matches any digit, regular expression `[^0-9]` matches any byte that is not a digit, regular expression `[]A-Za-z]` matches a closing bracket, or any uppercase or lowercase letter.

- The concatenation of two regular expressions matches the concatenation of two strings matches regular expression. Regular expressions can be grouped with parenthesis, and modified by the `? + ` and `*` characters.

- A regular expression followed by a question mark `?` matches 0 or 1 instance of the single regular expression.

- A regular expression followed by a plus sign `+` matches 1 or more instances of the single regular expression.

- A regular expression followed by a star `*` matches 0 or more instances of the single regular expression.

- Finally, two regular expressions separated by a bar — match any string matching the first or the second regular expression.

Parenthesis can be used to group regular expressions. For instance, the regular expression `"(+|-)?[0-9]+(\.[0-9]*)?"` matches a signed number with an optional fractional part. Furthermore, there is a "register" associated with each parenthesized part of a regular expression. The matching routines use these registers to keep track of the characters matched by the corresponding part of the regular expression. This is useful with functions `regex-extract` and `regex-subst` .

(regex-match `r` `s`)

Returns `t` if regular expression `r` exactly matches the entire string `s` . Returns the empty list otherwise.

Example:

```
? (regex-match "(+|-)?[0-9]+(\\\\.[0-9]*)?" "-56")
= t
```

(regex-extract `r` `s`)

If regular expression `r` matches the entire string `s` , this function returns a list of strings representing the contents of each register, that is to say the characters matched by each section of the regular expression `r` delimited by parenthesis. This is useful for extracting specific segments of a string.

If the regular expression `r` does not match the string `s` , function `regex-extract` returns the empty list. If the regular expression `r` matches the string but does not contain parenthesis, this function returns a list containing the initial string `s` .

Example:

```
? (regex-extract "(+|-)?([0-9]+)(\\\\.[0-9]*)?" "-56.23")
= ("-" "56" ".23")
```

(regex-seek `r` `s` [`start`])

Searchs the first substring in `s` that matches the regular expression `r` , starting at position `start` in `s` . If the argument `start` is not provided, string `s` is searched from the beginning.

If such a substring is found, `regex-seek` returns a list `(begin length)` , where `begin` is the index of the first character of the substring, and `length` is the length of the subscript. The instruction `(mid s begin length)` may be used to extract this substring.

If no such substring exists, `regex-seek` returns the empty list.

Example:

```
? (regex-seek "(+|-)?[0-9]+(\\\\.[0-9]*)?," "a=56.2, b=57,")
= (2 5)
```

(regex-subst `r s str`**)**

Replaces all substring matching regular expression `r` in string `str` by string `s` .

A "register" is associated to each piece of the regular expression `r` enclosed within parenthesis. Registers are numbered from `%0` to `%9` . During each match, the substring of `str` matching each piece of the regular expression is stored into the corresponding register.

During the replacement process, characters `%0` to `%9` in the replacement string `s` are substited the content of the corresponding register. (A single `%` is denoted as `%%`).

Example:

```
? (regex-subst "([a-h])([1-8])" "\\%1\\%0" "e2-e4, d7-d5, d2-d4, d5xd4?")
= "2e-4e, 7d-5d, 2d-4d, 5dx4d?"
```

(regex-rseek `r s` [`n` [`gr`]]**)**

This function seeks recursively the first occurence of `r` in `s` . and returns the list made of the locations.

When argument `n` is provided, it seeks and returns the locations of the `n` first occurences and it returns `()` on failure.

Optional regex `gr` defines the allowed garbage stuff before and between occurences. When `n` is not provided, this function checks the garbage stuff after the occurences too. If unallowed garbage stuff is found, the function returns `()` . By default, any garbage stuff is allowed.

Since even void garbage is checked, a caret "^" is often added to `gr` .

(regex-split `r s` [`n` [`gr` [`neg`]]]**)**

This function splits a string `s` into occurences of `r` .

When integer `n` is provided, this function provides only the `n` first occurences.

When regex `gr` is provided, garbage is checked (see function `regex-rseek`).

When `neg` is provided and non nil, this function returns the garbage stuff instead. When both `n` and `neg` are provided and non nil, the `n` garbages before and between the `n` first occurences are returned.

(regex-skip `r s [n [gr [neg]]]`**)**

This function skips the `n` first occurences of regex `r` in a string `s` .

When `n` is equal to 0, it returns `s` . When `n` is lower than 0, it generates an error. When `n` is either nil or undefined, it is set to 1.

When `neg` is either nil or undefined, it returns the right residual of `s` just following the `n` th occurence.

When `neg` is not nil, it returns the right residual of `s` begining with the `n` th occurence.

When regex `gr` is provided, garbage is checked (see function `regex-rseek`).

(regex-count `r s`**)**

This function recursively seeks the occurences of regex `r` in string `s` and returns the number of occurences found.

(regex-tail `r s [n [gr [neg]]]`**)**

This function seeks recursively the occurences of regex `r` in string `s` .

When `neg` is either nil or undefined, it returns the right residual of `s` begining before the `n` th last occurence.

When `neg` is non nil, it returns the right residual of `s` begining after the `n` th last occurence (and thus begining before the `n` th garbage.

When `n` is either nil or undefined, it is set to 1.

When regex `gr` is provided, garbage is checked (see function `regex-rseek`).

(regex-member `rl s`**)**

This function returns the first member of list `rl` which is a matching regex for string `s` .

5.9.4 International Strings

Lush contains partial support for multibyte strings using an encoding specified by the locale. This is work in progress.

(locale-to-utf8 `s`**)**

Converts a string from locale encoding to UTF-8 encoding. This is a best effort function: The unmodified string is returned if the conversion is impossible, either because the string `s` is incorrect, or because the system does not provide suitable conversion facilities.

(utf8-to-locale-to `s`)

Converts a string from UTF-8 encoding to locale encoding. This is a best effort function: The unmodified string is returned if the conversion is impossible, either because the string `s` is incorrect, or because the system does not provide suitable conversion facilities.

(explode-chars `s`)

Returns a list of integers with the wide character codes of all characters in the string. This function interprets multibyte sequences according to the encoding specified by the current locale.

Example (under a UTF8 locale):

```
? (explode-chars "\\xe2\\x82\\xac")
= (8364)
```

(implode-chars `l`)

Returns a string composed of the characters whose wide character code are specified by the list of integers `l` . Multibyte characters are generated according to the current locale. For instance, under a UTF8 locale,

Example

```
? (implode-chars '(8364 50 51 46 53 32 61 32 32 162 50 51 53 48))
= "23.5 =  2350"
```

(explode-bytes `s`)

Returns a list of integers representing the sequence of bytes in string `s` , regardless of their character interpretation.

Example

```
? (explode-bytes "")
= (226 130 172)
```

(implode-bytes `l`)

Assemble a string composed of the bytes whose value is specified by the list of integers `l` , regardless of their multibyte representation.

Example

```
? (implode-bytes '(226 130 172 50 51))
= "23"
```

5.10 Storages

A storage is a "chunk of memory" which holds objects of the same type. Storages use less main memory and provide faster access than lists; they are mainly used as data storage areas for numerical objects like scalars, vectors, matrices, tensors. There are a few predefined element types for which storages can be created (e.g., float, int).

Storage objects allow only simple, "flat" access, that is, objects are accessed by one-dimensional indices. Therefore, data in a storage is usually accessed by means of an IDX object, which provides customized access by multi-dimensional indices. Several IDXs can point to (parts of) the same storage, allowing access of the data in multiple ways simultaneously. Storages can be resized/reallocated without adverse effects on the IDX that point to them.

The data in a storage can reside in memory (a memory storage) or on disk (a memory-mapped storage). Disk files can be memory mapped into a storage and accessed element by element or byte by byte through storage or IDX access functions.

5.10.1 Predefined element types, storage classes

The hash table `storage-class` provides a mapping from element-type symbols to storage class objects:

- `'atom` -> `IntStorage` : Elements are Lisp objects.
- `'float` -> `FloatStorage` : Elements are single precision floats (4 bytes).
- `'double` -> `DoubleStorage` : Elements are double precision floats (8 bytes).
- `'int` -> `IntStorage` : Elements are 4 byte signed integers.
- `'short` -> `ShortStorage` : Elements are 2 byte signed integers.
- `'char` -> `CharStorage` : Elements are single-byte signed integers.
- `'uchar` -> `UcharStorage` : Elements are single-byte unsigned integers.
- `'gptr` -> `GptrStorage` : Elements are pointers to unmanaged memory.
- `'mptr` -> `MptrStorage` : Elements are pointers to managed memory.

5.10.2 Storage Creation and Allocation

Depending on the element type a storage is created for, its class is one of a few possible `storage-classes` (see below). The storage creation functions `new-storage` and `new-storage/managed` take a symbol indicating the element-type.

(new-storage `et`**)**

Create an empty storage object for element-type `et` .

Memory may be allocated for this storage object with `storage-alloc` . See also `new-storage/managed` . Examples:

```
? (new-storage 'int)
= ::IntStorage:unallocated@(nil):<0>

? (new-storage 'short)
= ::ShortStorage:unallocated@(nil):<0>

?
```

(new-storage/managed `et n` [`init`]**)**

Create a storage object for element-type `et` and allocate memory for `n` elements.

The memory is managed by the Lush runtime. If `init` is given and is not `()` , all elements of the storage will be initialized with `init` . Examples:

```
? (new-storage/managed 'float 100)
= ::FloatStorage:managed@0x827b2d8:<100>
```

(new-storage/mmap `et file` [`offs` [`readonly`]]**)**

See: storage-readonlyp

Map `file` into memory and associate that memory with a storage with element-type `et` ; return the storage.

`File` may be a file descriptor or a string containing a filename. Each element of the file starting at byte `offs` is made accessible through the corresponding element of the storage `srg` . Storages of storage class `AtomStorage` or `MptrStorage` cannot be memory mapped. Mapped storages are readonly by default (pass `()` as fourth argument to create a writeable mmapped storage).

(new-storage/foreign `et n p` [`readonly`]**)**

Create a storage object for element-type `et` and `n` and use the memory at address `p` .

The memory at `p` is not managed by the lush runtime; the user, providing the address `p` , is responsible for managing it. The storage object may be tagged for readonly accesses via the fourth argument `readonly` .

Warning: Lush will segfault if the storage object is used to access invalid memory. This may happen if `p` is not a valid address (not and address to

readable/writable memory), or if `n` was chosen too large to prevent out-of-bounds memory accesses.

(storage-alloc `srg n init`**)**

Allocate memory for `n` elements, initialize them to `init`, return `()`. If `init` is `()`, don't initialize; the storage content is undefined in this case. Storage `srg` must be unallocated (as returned by `new-storage`).

(storage-realloc `srg n [init]`**)**

Enlarge the storage `srg` to `n` elements, set new elements to `init`, and return `()`. Don't initialize when `init` is `()` (default). IDXs that point to `srg` are not affected, that is, they point to the right place and their content data is unchanged. The newly allocated data segment is initialized to `init`.

5.10.3 Storage Access

Storage elements can be read and set as if the storage were an IDX with one dimension (a vector). In other words `(s 3)` returns the value of element 3 of storage `s`. `(s 3 5)` sets it to 5. The function `storage-clear` sets all elements of a storage to the same value.

5.10.4 Miscellaneous Atorage Functions

(storage-clear `srg init`**)**

Set all elements in storage `srg` to `init`; return `nil`.

(storage-save `srg file`**)**

Save the content of storage `srg` into file `file`; return `nil`. `file` may be a string containing a filename or a file descriptor. Data is written raw, without a header, and in the native format of the machine (multibyte elements will not be portable between big and small-endian machines).

(storage-load `srg file`**)**

Load content of file `file` into storage `srg`; return `nil`. If `srg` is an unsized storage, alloc a storage that fits the contents of the file. If `srg` is a sized storage, attempt to read (storage-nelems `srg`) elements from `file`; raise an error if `file` is too small. `file` may be a filename or a file descriptor.

(storage-set-readonly `srg`**)**

Mark storage `srg` readonly; return `nil`. No Lisp function can write into `srg` after this.

(storage-readonlyp `srg`**)**

Return `t` if `arg` is a storage and is readonly, return `nil` if `arg` is a storage and is writable. Raise an error if `arg` is not a storage.

(storagep `srg`**)**

Return `t` if `arg` is a storage and `nil` otherwise.

(storage-nelems `srg`**)**

Return the number of elements in storage `srg` . Return 0 if the storage is unsized.

(storage-nbytes `srg`**)**

Return the number of bytes occupied by storage `srg` . Return 0 if the storage is unsized.

5.11 Arrays and Indexes

Lush has a powerful mechanism for manipulating tabular data such as scalars, vectors, matrices, and higher dimensional arrays.

The basic Lush object for accessing tabular data is called an index, and is of type `Index` . An index is merely an access structure, the actual data is stored in a storage referenced by the index. The data contained in a storage and accessed with an index can be of numeric type like `double` , `float` , `int` , etc. The data may also be pointers, (type `gptr`), or arbitrary Lush objects (type `atom`). See variable `storage-classes` for a list of available storage element types.

5.11.1 Array API conventions

We distinguish arrays from indexes. An array is an index plus an associated storage. Lush's array API uses this naming convention to distinguish two types of functions, index functions and array functions. Index functions create or manipulate indexes but leave the storage data unchanged. Array functions create or manipulate storage contents (and possibly the index, too). For example, `copy-index` creates a new index from an existing one but does not copy the associated storage. The function `copy-array` on the other hand creates a new index and a new storage; function `array-copy` copies data from one storage into another. Index functions are prefixed by `idx` , (e.g., `idx-reshape`), array functions are prefixed by `array` (e.g., `array-extend`).

Many index manipulation functions have a destructive counterpart, that is, the counterpart alters the index argument rather than creating a new index and leaving the argument unchanged. The convention is that if a destructive counterpart exists, its name is derived from the non-destructive function by adding

a trailing '!'. For example, `idx-reverse!` is the destructive counterpart to `idx-reverse` .

More specialized index functions assume the index argument is a vector (a one-dimensional index) or a matrix (a two-dimensional index). These index functions are prefixed wit `vec` , or `mat` , respectively (e.g., `mat-fliplr` .)

Lush comes with a long list of index functions. Many are described in this section, many more are described in the corresponding Standard Libraries section.

5.11.2 The Index structure, Shape and Subscripts

A storage is a container for homogeneous data. Its only properties are the number and type of elements. All other information necessary for the interpretation of storage data as an array are summarized in the index structure. The index structure contains:

- a base pointer (a pointer to the associated storage data)
- an offset to the first element in the array (the element with subscript (0 0 ... 0)
- the number of dimensions (0 to 8)
- the array size in each dimension
- a "modulo" for each dimension, that is, the data pointer increment associated with a subscript increment of 1 in that dimension

The list of array sizes, or "extents", is called the shape of the index. Some functions like `idx-reshape` take a shape as an argument. Arrays with zero dimensions have shape "()" and are called scalars. The length of the shape is sometimes called the rank of the index or array. Thus, scalars, vectors, and matrices have rank 0, 1, and 2, respectively.

Any particular element accessible through an index has a unique subscript, that is, a list of indices like `'(1 2 3)` . In the above example the index object was "applied" to the subscript in order to access an array element, in other words,

```
(apply m '(3 4 2))
```

evaluates to the element (3, 4, 2) of array m.

Many index functions taking subscripts as arguments and allow them to be negative. This corresponds to a "backwards" enumeration of the array elements. The subscript index "-1" refers to the last element of a vector, "-2" to last but one, and so on.

```
? ((array-range 1 5) -1)
= 5
```

The modulo data is used to map subscripts to pointers. As a Lush user you do not need to know about modulos until you need to implement your own functions in C to manipulate Lush arrays.

5.11.3 Array and Index Creation and Allocation

Arrays of any type and shape can be created with `make-array` .

(make-array `storage-class shp init`**)**

See: clone-array

Create a storage of class `storage-class` and allocate memory for an array of shape `shp` . If `init` is not `nil` , all elements of the array are set to `init` . The array is not initialized if init is `nil` and the array element values are undefined.

Examples:

```
? (make-array FloatStorage '(100 100) 0) ;; initialize with 3.1415 instead
= ::Index:<100x100>
? (make-array FloatStorage '(100 100) 3.1415)
= ::Index:<100x100>
?
```

More convenient array creation functions of the form " `element-type` -array" exist.

(double-array `e1...en`**)**

Create an `n` -dimensional double array of zeros. The arguments `e1` ... `en` are the extents of the new array.

(double-array* `e1...en`**)**

Create an `n` -dimensional double array. The arguments `e1` ... `en` are the extents of the new array. The array content is undefined.

(atom-array `e1...en`**)**

Create an `n` -dimensional atom array of `nil` s. The arguments `e1` ... `en` are the extents of the new array.

(arange [`n1`] `n2` [`delta`]**)**

See: range, arange*

Similiar to `range` but the result is a one-dimensional double array instead of a list. Example:

```
? (arange 2 5)
= [d   2.0000   3.0000   4.0000   5.0000]
```

(arange* [n1] n2 [delta])

See: range*, array-range

Similiar to `range*` but the result is a one-dimensional double array instead of a list. Example:

```
? (arange* 2 5)
= [d   2.0000   3.0000   4.0000]
```

(scalar arg)

Create a scalar of type double with value `arg` .

(vector ... args **...)**

Create a vector of type double from arguments. Example:

```
? (vector 1 2 (+ 1 2))
= [d   1.0000   2.0000   3.0000]
```

(clone-array prototype)

See: copy-array, array-clear

Make a new array of same shape and element type as `prototype` . Unlike an array created with `copy-array` , the contents of a cloned array are undefined.

```
? (clone-array (vector 1 2 3))
= [d   0.0000   0.0000   0.0000]
```

(array-clear m value)

Set all elements of array `m` to `value` .

```
? (let ((m (int-array 2 2)))
    (array-clear m 4) )
 = [i[i     4     4]
    [i     4     4]]
```

An index to an existing storage can be created with `new-index` .

(new-index `st` [`shp`]**)**

Create an index pointing to storage st. If no shape `shp` is specified, the new index is one dimensional. If a shape `shp` is specified, the number of elements must match the number of elements in the storage. See also `make-array` . Example:

```
? (let ((st (new-storage/managed 'short 12 1)))
     (new-index st '(3 4)) )
= [s[s     1     1     1     1]
    [s     1     1     1     1]
    [s     1     1     1     1]]
```

+MAXDIMS+

Maximum number of array dimensions. (This is a builtin limit on array dimensions; it is defined in file `header.h`).

5.11.4 Array Literals

Small arrays can be created easily using special syntax. Array literals consist of nested square brackets enclosing the array elements. Some examples:

```
(setq m [d 3 4 5])   ; a vector of doubles
(setq m [3 4 5])     ; a vector of doubles
(setq m [[1 2 3] [3 4 5]])  ; a matrix of doubles
```

Vectors and matrices of other types than doubles can be created by specifying the type with a single character right after the first opening bracket. Examples:

```
(setq m [d 1 2 3])   ; doubles
(setq m [f 1 2 3])   ; floats
(setq m [i 1 2 3])   ; ints
(setq m [s 1 2 3])   ; shorts
(setq m [c 1 2 3])   ; char
(setq m [u 1 2 3])   ; uchars
(setq m [a "choucroute" [1 2 3]])  ; lisp objects
```

Scalar literals

Scalar literals can be entered by placing an @ sign after the type indicator:

```
(idx-dotm0 vector [f@ 34] output)
```

Here are examples for each storage element type:

```
(setq s [d@ 42])
(setq s [f@ 42])
(setq s [i@ 42])
(setq s [s@ 42])
(setq s [c@ 42])
(setq s [u@ 42])
(setq s [a@ "choucroute"])
```

5.11.5 Index Properties and Predicates

(indexp `arg`**)**

Return `t` if `arg` is an index and `nil` otherwise.

(idx-numericp `arg`**)**

Return `t` if `arg` is an index whose storage is of a numerical type. Return `nil` if the storage class is `at-storage` or `gptr-storage` .

(idx-contiguousp `m`**)**

True if the elements of index `m` are contiguous in memory. Indexes resulting from `idx-transpose` or `idx-trim` usually are non-contiguous.

(idx-emptyp `m`**)**

True if array `m` has zero elements.

(idx-shape `m` [`d`]**)**

With one argument, return the shape of `m` as a list. With two arguments, return the extent of `m` in its `d` -th dimension (starting at zero). `shape` is an alias for `idx-shape` .

```
? (shape [d[d   2.0000   3.0000   4.0000]
           [d   5.0000   6.0000   7.0000]])
= (2 3)
```

($ `a` [`d`]**)**

Shape of array `a` as a vector, or extend in dimension `d` .

(idx-rank `m`**)**

Return rank (the number of dimensions) of index `m` .

```
? (length (idx-shape [d[d   2.0000   3.0000   4.0000]
                       [d   5.0000   6.0000   7.0000]]))
= 2


? (idx-rank [d[d   2.0000   3.0000   4.0000]
              [d   5.0000   6.0000   7.0000]])
= 2
```

(idx-nelems `idx`**)**

Return the number of elements accessible through index `idx` .

(idx-offset `m`**)**

Return the offset of the first element of index `m` in its storage. The offset is in number of elements, not bytes.

(idx-mod `m` [`d`]**)**

With one argument, return the list modulos. With two arguments return the modulo for dimension `d` .

(idx-storage `idx`**)**

Return the storage accessed through index `idx` .

(idx-base `m`**)**

Return a `gptr` that points to the first element of `m` . This must be used with caution because the data pointed to by the pointer may be garbage collected (when `m` is destroyed), or even displaced (when `m` is resized). `idx-ptr` is primarily used when calling C function that require a pointer to numerical data.

(idx-element-type `idx`**)**

Return the symbol indicating the storage element type.

(idx-dc `a`**)**

Return element with subscript (0, 0, ...) of `a` as a scalar. This is useful for manipulating the DC component of a discrete signal, or for checking the element-type of arrays.

```
? (let ((psp (arange 10 1 -1)))
    ((idx-dc psp) 0)
    psp )
= [d   0.0000   9.0000   8.0000   7.0000   6.0000   5.0000   4.0000   3.0000   2.0000   1.0000]
```

(array-dc `a` [`val`])

See: idx-dc

Return element with subscript (0, 0, ...) of `a` as a new scalar. When `val` is present, set the value of the scalar to `val` .

5.11.6 Arithmetic with Arrays

The usual arithmetic functions `+` , `-` , `*` , `/` , as well as many other mathematical functions from Section "Numbers" (`abs` , `sin` , `exp` , etc.) may be applied to arrays as well. If applied to an array, the result is a double array of the same shape as the input array.

```
? (let ((m ($$ (arange 20) '(4 5))))
    (+ (** (cos m) 2) (** (sin m) 2)) )
= [d[d   1.0000   1.0000   1.0000   1.0000   1.0000]
    [d   1.0000   1.0000   1.0000   1.0000   1.0000]
    [d   1.0000   1.0000   1.0000   1.0000   1.0000]
    [d   1.0000   1.0000   1.0000   1.0000   1.0000]]
```

Index Broadcasting

When more than one array is passed to an arithmetic function, then not all arrays need to have the same shape. The indexes must be `broadcastable` to the same shape, however. The result then has same shape as the broadcasted input indexes. In this example, both indexes are of equal rank:

```
? (let ((v (vector 1 2 3)))
      (+ ($< v) ($> v)) )
= [d[d  2.00  3.00  4.00]
    [d  3.00  4.00  5.00]
    [d  4.00  5.00  6.00]]
```

But equal rank is not a necessary condition as these examples show:

```
? (let ((v (vector 1 2 3)))
      (+ v ($> v)) )
= [d[d  2.00  3.00  4.00]
```

```
    [d  3.00  4.00  5.00]
    [d  4.00  5.00  6.00]]
?
? (let ((m (double-array 3 4)))
      (+ m 5) )
= [d[d  5.00  5.00  5.00  5.00]
    [d  5.00  5.00  5.00  5.00]
    [d  5.00  5.00  5.00  5.00]]
?
? (let ((m (double-array 3 4)))
      (+ m ($> [1 2 3])) )
= [d[d  1.00  1.00  1.00  1.00]
    [d  2.00  2.00  2.00  2.00]
    [d  3.00  3.00  3.00  3.00]]
?
? (let ((m (double-array 3 4)))
      (+ m ($< [1 2 3 4])) )
= [d[d  1.00  2.00  3.00  4.00]
    [d  1.00  2.00  3.00  4.00]
    [d  1.00  2.00  3.00  4.00]]
```

The term "broadcasting" as well as the algorithm are adopted from the Numarray project [http://www.stsci.edu/resources/software_hardware/numarray]. The broadcasting algorithm is:

```
(defun idx-broadcast2 (a b)
  (let ((a (copy-index a)) (b (copy-index b)))
    (while (< (rank a) (rank b))
      (idx-lift! a 1) )
    (while (< (rank b) (rank a))
      (idx-lift! b 1) )
  (domapc ((d (range* (rank a))) (ea (shape a)) (eb (shape b)))
    (cond
     ((= ea eb) ())
     ((= ea 1) (idx-expand! a d eb))
     ((= eb 1) (idx-expand! b d ea))
     (t (error "indexes not broadcastable")) ))
  b))
```

(idx-broadcast1 `a b`**)**
See: Index Broadcasting
Broadcast the array `a` to the shape of array `b` and return it. Raise an error if `a` may not be broadcasted to `b` .
(idx-broadcast2 `a1 a2`**)**
See: Index Broadcasting

Broadcast the two indexs `a1` and `a2` and return them, or raise an error if `a1` and `a2` are not broadcastable. Numbers are treated as scalars.

(idx-broadcastable-p `a1` `a2`**)**

See: Index Broadcasting

True if indexes `a1` and `a2` are broadcastable.

```
? (let ((m ($$ (arange 20) '(5 4))))
    (idx-broadcastable-p m (vector 4 3 2 1)) )
= t


? (let ((m ($$ (arange 20) '(5 4))))
    (idx-broadcastable-p m (vector 5 4 3 2 1)) )
= ()
```

(idx-broadcast `m0` **...** `mn`**)**

See: Index Broadcasting

Return list of broadcasted indexes `m0` ... `mn` . Raise an error if the indexes are not broadcastable. Numbers are treated as scalars.

```
? (let ((m ($$ (arange 12) '(4 3))))
    (idx-broadcast m (vector 3 2 1) [d@ 5]) )
= ([d[d   1.0000   2.0000   3.0000]
     [d   4.0000   5.0000   6.0000]
     [d   7.0000   8.0000   9.0000]
     [d  10.0000  11.0000  12.0000]] [d[d   3.0000   2.0000   1.0000]
                                       [d   3.0000   2.0000   1.0000]
                                       [d   3.0000   2.0000   1.0000]
                                       [d   3.0000   2.0000   1.0000]]
    [d[d   5.0000   5.0000   5.0000]
      [d   5.0000   5.0000   5.0000]
      [d   5.0000   5.0000   5.0000]
      [d   5.0000   5.0000   5.0000]] )
```

5.11.7 General Index and Array Functions

Most functions in this section create a new index from a given index so that the new index restricts access to certain parts of the input array. E.g., `idx-trim` reduces the extent of a selected dimension of the input index. When you are studying the index mapping functions in this section for the first time, it is helpful to not only ask what index properties a certain function alters, but also, what index properties are invariant. For example, `idx-reshape` changes the shape of an index, possibly its rank, too, but not the number of elements accessible through the index. `Idx-trim` changes the shape and the number of elements, but it does not change the the rank of an index. Finally, `idx-select`

reduces the rank and the number of elements, `array-lift` on the other hand increases the rank and the number of elements.

(idx-reshape `m shp`**)**

See: idx-reshape!

Return a new index with shape `shp` , referring to the same storage as index `m` . The number of elements of `m` must match the number of elements with the new shape.

```
? (idx-reshape (array-range 20) '(4 5))
= [d[d   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   6.0000   7.0000   8.0000   9.0000  10.0000]
    [d  11.0000  12.0000  13.0000  14.0000  15.0000]
    [d  16.0000  17.0000  18.0000  19.0000  20.0000]]
```

($$ `a shp`**)**

Reshape array `a` to have shape `shp` .

(idx-nick `m d`**)**

See: idx-nick!

Add a new singleton dimension `d` and return the new index. This is a special reshape operation, it increases the rank by one.

```
? (shape (idx-nick (double-array 4 5) 1))
= (4 1 5)
```

(idx-squeeze-shape `m`**)**

See: idx-reshape

Return a new index referring to the same storage as index `m` , with a shape derived from `(shape m)` by removing all singleton dimensions.

```
? (shape (idx-squeeze-shape (double-array 1 2 1 3 1)))
= (2 3)
```

(idx-flatten `m` [`n`]**)**

Reshape index `m` by collapsing the first `n` dimensions (default is `(rank m) - 1`), or raise an error when `m` is not contiguous. When `n` is negative, the last `n` dimensions of `m` are collapsed.

($- a)

Flatten array `a` .

(idx-select `m d s`)

See: idx-select*

Return a new index with the `d` -th dimension of `m` removed, and which is the `s` -th "slice" of `m` , in the `d` -th dimension.

```
? (setq m [[0 1 2 3 4][10 11 12 13 14]])
= [[ 0.00  1.00  2.00  3.00  4.00 ]
   [10.00 11.00 12.00 13.00 14.00 ]]
? (idx-select m 1 2)
= [ 2.00 12.00 ]
```

($*0 `a s`)

Select slice `s` of `a` in dimension 0.

($*1 `a s`)

Select slice `s` of `a` in dimension 0.

($*2 `a s`)

Select slice `s` of `a` in dimension 0.

($*3 `a s`)

Select slice `s` of `a` in dimension 0.

(idx-select* `m s0 s1` ...)

See: idx-select

Recursively select index with argument `si` for dimension `i` of `m` .

```
? (setq m [[0 1 2 3 4][10 11 12 13 14]])
= [[ 0.00  1.00  2.00  3.00  4.00 ]
   [10.00 11.00 12.00 13.00 14.00 ]]
? (idx-select* m 1 2)
= [@d 12.00]
```

(idx-lift `m e1...en`**)**

See: array-lift, idx-sink

Create a new index with `n` + `(rank m)` dimensions, where the first `n` dimensions have extents `e1` ... `en` . Applying an `n` -element subscript to the resulting array yields an index equivalent to `m` .

```
? (let* ((m (reshape (arange 9) '(3 3)))
         (m* (idx-lift m 2)) )
    (list (shape m) (shape m*) m*) )
= ((3 3) (2 3 3) [d[d[d   1.0000   2.0000   3.0000]
                     [d   4.0000   5.0000   6.0000]
                     [d   7.0000   8.0000   9.0000]]
                   [d[d   1.0000   2.0000   3.0000]
                     [d   4.0000   5.0000   6.0000]
                     [d   7.0000   8.0000   9.0000]]])
```

($< a e1...en)

See: idx-lift

Lift index `a` .

(array-lift `m e1...en`**)**

See: idx-lift, array-sink

Create a new array with `n` + `(rank m)` dimensions, where the first `n` dimensions have extents `e1` ... `en` . Applying an `n` -element subscript to the resulting array yields a copy of `m` .

```
? (let* ((m (reshape (arange 9) '(3 3)))
         (m* (array-lift m 2)) )
    (list (shape m) (shape m*) m*) )
= ((3 3) (2 3 3) [d[d[d   1.0000   2.0000   3.0000]
                     [d   4.0000   5.0000   6.0000]
                     [d   7.0000   8.0000   9.0000]]
                   [d[d   1.0000   2.0000   3.0000]
                     [d   4.0000   5.0000   6.0000]
                     [d   7.0000   8.0000   9.0000]]])
```

(idx-sink `m e1...en`**)**

See: array-sink, idx-lift

Create a new index with `n` + `(rank m)` dimensions, where the last `n` dimensions have extents `e1` ... `en` . Similar to `idx-lift` but the new dimensions are added to the end of `m` 's shape.

```
? (let* ((m (reshape (arange 9) '(3 3)))
         (m* (idx-sink m 2)) )
    (list (shape m) (shape m*) m*) )
= ((3 3) (3 3 2) [d[d[d   1.0000   1.0000]
                     [d   2.0000   2.0000]
                     [d   3.0000   3.0000]]
                   [d[d   4.0000   4.0000]
                     [d   5.0000   5.0000]
                     [d   6.0000   6.0000]]
                   [d[d   7.0000   7.0000]
                     [d   8.0000   8.0000]
                     [d   9.0000   9.0000]]])
```

$? ($> a e1 ... en)
See: idx-sink
Sink index a .

(array-sink m e1...en)

See: array-lift

Create a new array with n + (rank m) dimensions, where the last n dimensions have extents e1 ... en . Similar to array-lift but the new dimensions are added to the end of m 's shape.

```
? (let* ((m (reshape (arange 9) '(3 3)))
         (m* (array-sink m 2)) )
    (list (shape m) (shape m*) m*) )
= ((3 3) (3 3 2) [d[d[d   1.0000   1.0000]
                     [d   2.0000   2.0000]
                     [d   3.0000   3.0000]]
                   [d[d   4.0000   4.0000]
                     [d   5.0000   5.0000]
                     [d   6.0000   6.0000]]
                   [d[d   7.0000   7.0000]
                     [d   8.0000   8.0000]
                     [d   9.0000   9.0000]]])
```

(idx-trim idx d new-zero [new-extent])

See: idx-trim*, idx-trim!, idx-strim

Make a copy of index idx and reduce the extent in the d -th dimension to new-extent elements, starting with element new-zero . If new-extent is omitted, then the old extent minus new-zero is taken as default. If new-zero is negative, trimming at the higher-index end of that dimension is performed.

```
? (setq m [[0 1 2 3 4][10 11 12 13 14]])
= [d[d  0.00  1.00  2.00  3.00  4.00]
    [d 10.00 11.00 12.00 13.00 14.00]]
? (idx-trim m 0 1)
= [d[d 10.00 11.00 12.00 13.00 14.00]]
? (idx-trim m 0 -1)
= [d[d  0.00  1.00  2.00  3.00  4.00]]
? (idx-trim m 1 -1)
= [d[d  0.00  1.00  2.00  3.00]
    [d 10.00 11.00 12.00 13.00]]
? (idx-trim m 1 -1 3)
= [d[d  1.00  2.00  3.00]
    [d 11.00 12.00 13.00]]
```

(idx-trim! `idx d new-zero` [`new-extent`]**)**

See: idx-trim, idx-strim!

Trim index `idx` in `d` -th dimension in-place (see help on `idx-trim`).

(idx-trim* `m e0 e1` **...)**

See: idx-trim

Trim `m` to extent `e0` in first dimension, to extent `e1` in second dimension, and so on (see help on `idx-trim` for details).

```
? (idx-trim* (double-array 5 5) 2 3)
= [d[d   0.0000   0.0000   0.0000]
    [d   0.0000   0.0000   0.0000]]
```

(idx-strim `idx d delta`**)**

See: idx-strim!, idx-trim

Make a copy of `idx` and trim it symmetrically by `delta` along dimension `d` .

(idx-strim! `idx d delta`**)**

See: idx-strim, idx-trim!

Symmetrically trim index `idx` by `delta` along dimension `d` .

(idx-expand `m d ne`**)**

See: idx-expand!

Expand singleton dimension `d` of `m` to new extend `ne` . This manipulation increases the logical number of elements of `m` but not the physical number of

elements. That is, all elements with subscripts that differ only in the new dimension refer to the same storage location.

```
? (idx-expand (reshape (array-range 5) '(1 5)) 0 4)
= [d[d   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   1.0000   2.0000   3.0000   4.0000   5.0000]]
```

(idx-extend `m d n`**)**

See: idx-extend!

Extend dimension `d` of `m` by `n` ; `n` may be negative. This operation is only successful if there are enough elements in the storage, so that every possible subscript of the resulting index refers to a storage location.

```
? (idx-extend (idx-trim* (array-range 10) 4) 0 2)
= [d   1.0000   2.0000   3.0000   4.0000   5.0000   6.0000]
```

(idx-extend* `m de0 de1` **...)**

See: idx-extend

Extend `m` by `de0` in first dimension, by `de1` in second dimension and so on (see help on `idx-extend` for details).

```
? (idx-extend* (idx-trim* (array-range 10) 4) 2)
= [d   1.0000   2.0000   3.0000   4.0000   5.0000   6.0000]
```

(array-extend `m d n` [`init`]**)**

Create a new array which has the same shape as `m` save for dimension `d` , which is extended by `n` (`n` may be negative). The new array has the same contents as `m` for all subscripts valid for `m` , and is set to `init` for all other subscripts (default is zero). If `init` is `()` , don't init. This operation is always successful (except when memory is exhausted), and the resulting array is contiguous.

```
? (array-extend (array-range 5) 0 5)
= [d   1.0000   2.0000   3.0000   4.0000   5.0000   0.0000   0.0000   0.0000   0.0000   0.0000]
```

(array-extend! `m d n` [`init`]**)**

Extend array `m` in-place. (Fixme: Currently this operation is only implemented for the case that `m` is a contiguous, numeric array and `d` =0.)

(idx-shift `m d n`**)**

See: idx-shift*, idx-shift!

Create a copy of index `m` and alter its offset so that the new index is "shifted" by `n` elements along dimension `d` with respect to `m` . This operation can be useful in combination with `idx-trim` or `idx-extend` .

```
? (idx-shift (idx-extend (array-range 10) 0 -5) 0 2)
= [d   3.0000   4.0000   5.0000   6.0000   7.0000]
```

(idx-shift! `m d n`**)**

See: idx-shift, idx-shift*

Shift `m` by `n` along dimension `d` in-place and return `m` (see help on `idx-shift` for more info).

(idx-shift* `m n0 n1 ...`**)**

See: idx-shift

Shift `m` by `n0` first dimension, by `n1` in second dimension, and so on (see help on `idx-shift` for details).

```
? (idx-shift* (idx-trim* (array-range 10) 5) 5)
= [d   6.0000   7.0000   8.0000   9.0000  10.0000]
```

(idx-transpose `m dimlist`**)**

Create an index where the dimensions of `m` have been permuted according to the list of dimension indices `dimlist` . For example,

```
(idx-transpose m '(0 2 1))
```

permutes the second and third dimensions of m.

(array-transpose `m dimlist`**)**

Similiar to `idx-transpose` but the result is a new contiguous array.

(idx-reverse `m d`**)**

See: idx-reverse!

Reverse the order of elements of `m` along dimension `d` . This transformation leaves the shape of `m` unchanged but turns a contiguous array into a non-contiguous. Use `array-reverse` if the result ought to be contiguous.

(array-reverse `m d`**)**

Reverse the order of elements of `m` along dimension `d` . Similiar to `idx-reverse` but the result is a new contiguous array.

5.11.8 Copying of Indexes and Arrays

Besides the usual copy functions `copy-idx` and `copy-array` there are also some functions that copy arrays conditionally (e.g., `as-int-array`). This is helpful when processing array arguments to functions that need the input arrays to have certain properties.

(copy-index `idx`**)**

Return a copy of index `idx` .

(copy-array `m`**)**

Create a copy of array `m` by creating copy of the storage referenced by `m` and creating a new index to that storage. The resulting index is always contiguous.

(array-copy `m1 m2`**)**

Copy contents of `m1` into `m2` and return `m2` . `M1` and `m2` must have the same shape. If `m1` and `m2` are of different numerical types, appropriate conversion will be performed.

(array-swap `m1 m2`**)**

Swap contents of `m1` and `m2` and return `nil` . `M1` and `m2` must have identical shape and element-type.

(array-take `m [d] im`**)**

See: array-take*, idx-select

With two arguments, take elements from `m` according to subscripts in `im` . With three arguments, take slices of `m` in dimension `d` (like `idx-select`) for all elements in `im` , and combine them into a new array.

With two arguments, the rank of the result is `(rank im) - 1` , with three arguments the rank is `(rank m) + (rank im) - 1` .

Examples:

```
(let ((m (reshape (arange 20) '(4 5))))
   (array-take m [i 1 1]) )

(let ((m (reshape (arange 20) '(4 5))))
     (array-take m [i [1 1] [2 2]]) )
```

```
  (let ((m (reshape (arange 20) '(4 5))))
    (array-take m 0 [3 2 1 1 2 3]) )
```

(array-take* `m` `map`)

See: array-take

Take slices of `m` in leading dimensions and combine them into a new array. Take the slices at indices where `map` is nonzero. `(shape map)` must be a prefix of `(shape m)` . The result has rank `(rank m) - (rank map) + 1` .

Examples:

```
? (let ((m (reshape (arange 20) '(4 5))))
    (array-take* m [i     0     1     1     0]) )
= [d[d[d   6.0000   7.0000   8.0000   9.0000  10.0000]]
    [d[d  11.0000  12.0000  13.0000  14.0000  15.0000]]]
```

(array-where-nonzero `m`)

Return indices of nonzero elements of `m` . The result is an NxD array, where N is the number of nonzero elements in `m` , and D is `(rank m)` .

```
? (let ((m (reshape (arange 16) '(4 4))))
    (array-where-nonzero (c< m 8)) )
= [i[i     0     0]
    [i     0     1]
    [i     0     2]
    [i     0     3]
    [i     1     0]
    [i     1     1]
    [i     1     2]]
```

(where `m`)

Alias for `array-where-nonzero` .

(as-double-array `m`)

See: as-int-array

Turn `m` into a double array. If `m` is a number, create a double scalar (a zero-dimensional array) with value `m` . If `m` is a list of numbers, create a double vector of length `(length m)` and fill it with the elements of `m` . If `m` is an array and the element type of `m` is double, return `m` . If `m` is a numeric array but not double, create a new double array of same shape and copy contents of `m` into it using `array-copy` . In all other cases, raise an error.

```
? (as-double-array (list 3 2 1))
= [d   3.0000   2.0000   1.0000]
```

(as-int-array `m`**)**

See: as-double-array

Same as `as-double-array` except that the resulting array has element type int. `as-int-array` does not check for over- or underflow during conversion.

(as-uchar-array `m`**)**

See: as-double-array

Same as `as-double-array` except that the resulting array has element type uchar (unsigned char). `as-uchar-array` does not check for over- or underflow during conversion.

(as-contiguous-array `m`**)**

If `m` is a contiguous array, return `m` . If `m` is a non-contiguous array, return a contiguous copy of `m` . Raise an error if `m` is not an index.

5.11.9 Index Iterators

(idx-bloop ((`symb1 idx1`**) [(**`symb2 idx2`**) [...(**`symbn idxn`**)]]) **`body`**)**

make each `symbi` be an idx that loops over the first dimension of its corresponding `idxi` . Execute `body` for each value. More precisely, each `si` will be an idx with one less dimension than the corresponding `idxi` , and will simulataneously loop over the successive "slices" of `idxi` for each possible value of the first index. In other words, applying function `myfunc` to each element of a vector `v1` and putting the result in the corresponding element in `v2` can be done with:

```
(idx-bloop ((x1 v1) (x2 v2)) (x2 (myfunc (x1))))
```

`x1` and `x2` are scalars (i.e. zero-dimensional tensors). The above function work just as well is `v1` and `v2` are `d` -dimensional tensors and `myfunc` accepts `n-1` -th dimensional tensors as arguments.

(idx-eloop ((`symb1 idx1`**) [(symb2**> `idx2`**) [...(**`symbn idxn`**)]]) **`body`**)**

Make each `symbi` be an idx that loops over the last dimension of its corresponding `idxi` . Execute `body` for each value. This is like idx-bloop, but it loops on the last dimension, instead of the first. For example, the matrix product operation `C = A*B` can be written as follows:

```
(de idx-m2timesm2 (A B C)
   (idx-eloop ((Bj B)(Cj C)) (idx-m2dotm1 A Bj Cj)))
```

where `idx-m2dotm1` is the usual matrix-vector product. The `idx-eloop` construct simultaneously iterates over all columns of `B` and `C` .

(cidx-bloop (`i_1` [`i_2...i_n`] (`c_1 l_1`) [(`c_1 l_1`)...(`c_m l_m`)) `p_1` [`p_2...`])

This iterator is desinged to facilitate the implementation of inner loops of tensor functions in C, while leaving all the bookkeeping to the Lisp. A call to `cidx-bloop` as shown in the synopsis is somewhat equivalent to `n` nested `idx-bloop` s, which will loop over the first `n` dimensions of idxs `l_1` to `l_m` simultaneously. The arguments `i_1` to `i_n` are strings containing names of C local variables that will be created and set to the loop index in each of the `n` dimensions. At each iteration, the C variables provided in strings `c_1` to `c_m` will point to the appropriate values in the idxs `l_1` to `l_m` . For example, the following function will fill matrix `a` with cos(i+j).

```
(de foo (a)
   ((-idx2- (-flt-)) a)
   (cidx-bloop ("i" "j" ("a" a)) #{ *a = cos(i+j); #}) a)
```

The return value is (like in idx-bloop) the last IDX specified in the declaration (in the example above, the return value is superfluous).

(idx-gloop (`p1...`[`pn`]) `body`)

Enhanced version of `bloop` which allows to "bloop" through a pointer table, and to have access to the current index value of the loop. Each `pi` is a list with 1, 2, or 3 elements. If it has 2 elements the meaning is like a in regular bloop. If it has 1 element, which must be a symbol, it will be used as a local variable which contains the current index of the loop. With 3 elements, it must be of the form (`symbol p m`), where `p` is a 1D index and `m` an index of at least one dimension. `Symbol` will take the values (idx-select m 0 (p i)) for all possible values of i.

(array-reduce `op m` [`d`])

See: array-reduce*

Apply operator `op` along dimension `d` of `m` (default for `d` is -1). The result has rank `(- (rank m) 1)` , hence the name "reduce". Function `op` must accept two array arguments of equal shape and must return a result of the same shape.

Examples:

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (list (array-reduce + m) (array-reduce min m 0)) )
```

```
= ([d  15.0000  40.0000  65.0000  90.0000] [d   1.0000   2.0000   3.0000   4.0000   5.0000])
```

(array-reduce* `op m [d]`**)**

See: array-reduce

Apply operator `op` along dimension `d` of `m` (default for `d` is -1). The result has rank `(- (rank m) 1)` . Function `op` must accept three array arguments of equal shape, the third being the explicit output.

Examples:

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (array-reduce* idx-add m) )
= [d  15.0000  40.0000  65.0000  90.0000]
```

5.11.10 Arrays of Arrays

Sometimes one needs to manipulate arrays block-wise. An obvious way to represent an array partitioned into blocks is by creating indexes addressing the blocks of an array and then arranging these indexes into an atom array. For example, let's assume we want to partition a 4x4 matrix into four 2x2 blocks:

```
? (setq m (reshape (arange 16) '(4 4)))
= [d[d  1.00  2.00  3.00  4.00]
    [d  5.00  6.00  7.00  8.00]
    [d  9.00 10.00 11.00 12.00]
    [d 13.00 14.00 15.00 16.00]]
? (setq block00 (idx-trim* m 2 2))
= [d[d  1.00  2.00]
    [d  5.00  6.00]]
? (setq block01 (idx-shift* block00 0 2))
= [d[d  3.00  4.00]
    [d  7.00  8.00]]
? (setq block10 (idx-shift* block00 2 0))
= [d[d  9.00 10.00]
    [d 13.00 14.00]]
? (setq block11 (idx-shift* block00 2 2))
= [d[d 11.00 12.00]
    [d 15.00 16.00]]
? (setq bm (atom-array 2 2))
= [a[a 0 0]
    [a 0 0]]
? (progn (bm 0 0 block00) (bm 0 1 block01)
         (bm 1 0 block10) (bm 1 1 block11))
= [a[a [d[d  1.00  2.00]
```

```
          [d  5.00  6.00]] [d[d  3.00  4.00]
                            [d  7.00  8.00]]]
     [a [d[d  9.00 10.00]
          [d 13.00 14.00]] [d[d 11.00 12.00]
                            [d 15.00 16.00]]]]
```

Note, that the indexes in `bm` refer to the same data as `m` :

```
? (m 0 0 123)
= [d[d 123.00  2.00  3.00  4.00]
    [d  5.00  6.00  7.00  8.00]
    [d  9.00 10.00 11.00 12.00]
    [d 13.00 14.00 15.00 16.00]]
? (bm 0 0)
= [d[d 123.00  2.00]
    [d  5.00  6.00]]
```

What took a number of steps above is much easier to accomplish by using `idx-slice` .

```
? (idx-slice* m 2 2)
 = [a[a [d[d 123.00  2.00]
          [d  5.00  6.00]] [d[d  3.00  4.00]
                            [d  7.00  8.00]]]
     [a [d[d  9.00 10.00]
          [d 13.00 14.00]] [d[d 11.00 12.00]
                            [d 15.00 16.00]]]]
```

Assume now that we want to transpose all blocks of `m` individually. Having created `bm` , this could be done by looping over all elements of `bm` , transposing each element, and finally copying the transposed data back.

```
? (idx-bloop ((block (ravel bm)))
    (let ((b (block)))
      (array-copy (copy-array b) (mat-transpose b)) ))
= ...
? m
= [d[d 123.00  5.00  3.00  7.00]
    [d  2.00  6.00  4.00  8.00]
    [d  9.00 13.00 11.00 15.00]
    [d 10.00 14.00 12.00 16.00]]
```

Alternatively one may use `array-map` and `array-splice` . This does not modify `m` in-place, however, but involves the creation of a new array:

```
? (array-splice (array-map mat-transpose bm))
= [d[d 123.00  5.00  3.00  7.00]
    [d  2.00  6.00  4.00  8.00]
    [d  9.00 13.00 11.00 15.00]
    [d 10.00 14.00 12.00 16.00]]
```

Now assume that we want to transpose `m` "on the block level". This again is easy using `idx-slice` and `array-splice` .

```
? (let* ((m (reshape (arange 16) '(4 4)))
         (bm (idx-slice* m 2 2)) )
    (array-splice (mat-transpose bm)) )
= [d[d  1.00  2.00  9.00 10.00]
    [d  5.00  6.00 13.00 14.00]
    [d  3.00  4.00 11.00 12.00]
    [d  7.00  8.00 15.00 16.00]]
```

Map for arrays

Just as `mapcar` applies a function to all elements of a list and returns a list of results, `array-map` applies a function to all elements of an array and returns an array of results.

(array-map `f a1...an`)

See: array-map-to, array-mapc

Apply function `f` to all identically subscripted elements of arrays `a1` ... `an` , store the results in a new array and return it. All arrays `a1` ... `an` must have the same shape.

Examples:

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (array-map + m (mat-flipud m)) )
= [d[d  17.0000  19.0000  21.0000  23.0000  25.0000]
    [d  17.0000  19.0000  21.0000  23.0000  25.0000]
    [d  17.0000  19.0000  21.0000  23.0000  25.0000]
    [d  17.0000  19.0000  21.0000  23.0000  25.0000]]
```

```
? (let* ((m (reshape (array-range 20) '(4 5)))
         (bm (idx-slice m 1 '(2))) )
    (array-splice (array-map mat-fliplr bm)) )
= [d[d   2.0000   1.0000   5.0000   4.0000   3.0000]
    [d   7.0000   6.0000  10.0000   9.0000   8.0000]
    [d  12.0000  11.0000  15.0000  14.0000  13.0000]
    [d  17.0000  16.0000  20.0000  19.0000  18.0000]]
```

(array-map-to `out f a1...an`**)**

Apply function `f` to all identically subscripted elements of arrays `a1` ... `an` and store result in the corresponding element of array `out` . Return `out` . All arrays must have the same shape.

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (array-map-to (clone-array m) + m (mat-flipud m)) )
= [d[d  17.0000  19.0000  21.0000  23.0000  25.0000]
    [d  17.0000  19.0000  21.0000  23.0000  25.0000]
    [d  17.0000  19.0000  21.0000  23.0000  25.0000]
    [d  17.0000  19.0000  21.0000  23.0000  25.0000]]
```

(array-mapc `f a1...an`**)**

Apply function `f` to all identically subscripted elements of arrays `a1` ... `an` , and return the first array `a1` . All arrays `a1` ... `an` must have the same shape.

See: array-map-to, array-map

```
? (let ((m (reshape (array-range 4) '(2 2))))
    (array-mapc print m) )
1
2
3
4
= [d[d   1.0000   2.0000]
    [d   3.0000   4.0000]]
```

Slicing and splicing arrays

Some functions discussed in this section operate on "sliced arrays", that is, arrays of arrays as they are returned by function `idx-slice` . Not all arrays of arrays are sliced arrays. In this example, array `b` is not a sliced array because the rank of `b` does not match the rank of its elements:

```
? (setq b (let ((v (idx-lift (arange 3) 1))) (box v v)))
= [a [d[d  1.00  2.00  3.00]] [d[d  1.00  2.00  3.00]]]
```

In this example the array is not a sliced array because the extents of the element arrays are not compatible:

```
? (let ((m (idx-lift (arange 3) 1)))
    (idx-lift (box m (mat-transpose m)) 1))
= [a[a [d[d  1.00  2.00  3.00]] [d[d  1.00]
                                  [d  2.00]
                                  [d  3.00]]]]
```

(box `i1...in`**)**
Box items. Create one-dimensional atom array containing items `i1` ... `in` .
See: box1

```
? (let ((l '(1 2 3)))
    (box l "wisdom" 42) )
= [a (1 2 3) "wisdom" 42]
```

(box1 `item`**)**
Box item. Create an atom scalar containing `item` .
See: box

```
? (let ((l '(1 2 3)))
    (box1 l) )
= [a@ (1 2 3)]
```

(idx-slice `m d zero-idx`**)**
Slice array `m` along dimension `d` at element `zero-idx` . The result is a sliced array of smaller blocks than `m` . The input array `m` may be a sliced array or a plain array. `Zero-idx` may be a slice index (a number) or a list of indices.

A plain array `m` is treated as a sliced array consisting of a single block, and the rank of the sliced array is the rank of the plain array. An array may be sliced only once per dimension.

See: box, idx-slice*, array-splice

```
? (let ((m (reshape (arange 20) '(4 5))))
    (idx-slice m 0 '(2 3)) )
= [a[a [d[d   1.0000   2.0000   3.0000   4.0000   5.0000]
         [d   6.0000   7.0000   8.0000   9.0000  10.0000]]]
    [a [d[d  11.0000  12.0000  13.0000  14.0000  15.0000]]]
    [a [d[d  16.0000  17.0000  18.0000  19.0000  20.0000]]]]
```

(idx-slice* `m z0 z1` **...)**
Recursively slice `m` with zero index (indices) `z0` for dimension 0, `z1` for dimension 1 and so on.

See: idx-slice

(array-splice `m`**)**
Splice sliced array `m` . Given a sliced array, create a new plain array that holds the contents of all element arrays as if spliced along the non-singleton dimensions of array `m` . `Array-splice` is the inverse to `idx-slice` .

See: idx-slice

```
? (let ((m (reshape (arange 16) '(4 4))))
    (array-splice (idx-slice m 0 '(2))) )
```

```
= [d[d   1.0000   2.0000   3.0000   4.0000]
    [d   5.0000   6.0000   7.0000   8.0000]
    [d   9.0000  10.0000  11.0000  12.0000]
    [d  13.0000  14.0000  15.0000  16.0000]]
```

(array-combine `m1 m2...mn`**)**

See: array-combine*, idx-broadcast, mat-catcols

Combine argument arrays by stacking them into a "column of arrays". The argument arrays `m1 ... mn` must be broadcastable, the result has length n. The rank of the result is the rank of the broadcasted arrays plus 1. Examples:

```
? (array-combine [d   1.0000   2.0000   3.0000] [d   4.0000   5.0000   6.0000])
= [d[d   1.0000   2.0000   3.0000]
    [d   4.0000   5.0000   6.0000]]


? (let ((m (+ 1 (double-array 2 3)))
        (v (vector 2 2 2))
        (s [d@ 3]) )
    (array-combine s v m) )
= [d[d[d   3.0000   3.0000   3.0000]
      [d   3.0000   3.0000   3.0000]]
    [d[d   2.0000   2.0000   2.0000]
      [d   2.0000   2.0000   2.0000]]
    [d[d   1.0000   1.0000   1.0000]
      [d   1.0000   1.0000   1.0000]]]
```

(array-combine* `m1 m2...mn`**)**

See: array-combine, idx-broadcast, mat-catrows

Combine argument arrays by stacking them into a "row of arrays". The argument arrays `m1 ... mn` must be broadcastable. The rank of the result is the rank of the broadcasted arrays plus 1. Examples:

```
? (array-combine* [d   1.0000   2.0000   3.0000] [d   4.0000   5.0000   6.0000])
= [d[d   1.0000   4.0000]
    [d   2.0000   5.0000]
    [d   3.0000   6.0000]]


? (let ((m (+ 1 (double-array 2 3)))
        (v (vector 2 2 2))
        (s [d@ 3]) )
    (array-combine* s v m) )
= [d[d[d   3.0000   2.0000   1.0000]
      [d   3.0000   2.0000   1.0000]
```

```
      [d   3.0000   2.0000   1.0000]]
    [d[d   3.0000   2.0000   1.0000]
      [d   3.0000   2.0000   1.0000]
      [d   3.0000   2.0000   1.0000]]]
```

5.11.11 Matrix Functions

This is a collection of linear algebra and matrix manipulation functions. More of these "mat- functions" are part of the LAPACK interface (`(libload "lapack/mat")`).

Namespace mat-

Creating Matrices

(mat-col `v1` **...** `vn`**)**
Create a single-column matrix of type double.

(mat-colp `m`**)**
True if argument is single-column matrix.

(mat-row `v1` **...** `vn`**)**
Create a single-row matrix of type double.

(mat-rowp `m`**)**
True if argument is a single-row matrix.

(mat-catcols `m1` `m2` **...)**
Concatenate matrices `m1` , `m2` , etc. along the columns. All input matrices must have the same number of columns.

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (mat-catcols m m) )
= [d[d   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   6.0000   7.0000   8.0000   9.0000  10.0000]
    [d  11.0000  12.0000  13.0000  14.0000  15.0000]
    [d  16.0000  17.0000  18.0000  19.0000  20.0000]
    [d   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   6.0000   7.0000   8.0000   9.0000  10.0000]
    [d  11.0000  12.0000  13.0000  14.0000  15.0000]
    [d  16.0000  17.0000  18.0000  19.0000  20.0000]]
```

(mat-catrows `m1` `m2` **...)**
Concatenate matrices `m1` , `m2` , etc. along the rows. All input matrices must have the same number of rows.

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (mat-catrows m m) )
= [d[d   1.0000   2.0000   3.0000   4.0000   5.0000   1.0000   2.0000   3.0000   4.0000   5.0000]
    [d   6.0000   7.0000   8.0000   9.0000  10.0000   6.0000   7.0000   8.0000   9.0000  10.0000]
```

```
    [d  11.0000  12.0000  13.0000  14.0000  15.0000  11.0000  12.0000  13.0000  14.0000
    [d  16.0000  17.0000  18.0000  19.0000  20.0000  16.0000  17.0000  18.0000  19.0000
```

(mat-diag `elements`**)**

Create diagonal matrix with diagonal entries `elements` . `Elements` may be a list or a vector.

(mat-id `n`**)**

Identity matrix of size `n` x `n` .

Matrix functions

(mat-shape `m`**)**

Return the extent in the first two dimensions, that is, the shape of `m` if viewed as a matrix. This is useful, for instance, to get the size of an image, not knowing whether it is rank 2 or rank 3.

(mat-squarep `m`**)**

True when matrix `m` is square.

(mat-flipud `m`**)**

Reverse dimension 0 (columns) of array `m` . Raise an error if `m` has less than two dimensions.

(mat-fliplr `m`**)**

Reverse dimension 1 (rows) of array `m` . Raise an error if `m` has less than two dimensions.

(mat-transpose `m`**)**

Transpose array `m` as a matrix. Raise an error if `m` has less than two dimensions.

(mat-rot90 `m`**)**

Rotate array `m` counterclockwise by 90 degrees. Raise an error if `m` has less than two dimensions.

(mat-magnify `m f0 f1`**)**

Replicate elements of matrix `m f0` times along dimension 0, and `f1` times along dimension 1. `f0` and `f1` must be integer values.

```
? (let ((m (reshape (array-range 9) '(3 3))))
    (mat-magnify m 2 3) )
= [d[d   1.0000   1.0000   1.0000   2.0000   2.0000   2.0000   3.0000   3.0000   3.0000
    [d   1.0000   1.0000   1.0000   2.0000   2.0000   2.0000   3.0000   3.0000   3.0000
    [d   4.0000   4.0000   4.0000   5.0000   5.0000   5.0000   6.0000   6.0000   6.0000
    [d   4.0000   4.0000   4.0000   5.0000   5.0000   5.0000   6.0000   6.0000   6.0000
    [d   7.0000   7.0000   7.0000   8.0000   8.0000   8.0000   9.0000   9.0000   9.0000
    [d   7.0000   7.0000   7.0000   8.0000   8.0000   8.0000   9.0000   9.0000   9.0000
```

(mat-sum `m`**)**

Sum of each column, return results in a vector.

See: array-sum

(mat-prod `m`**)**

Product of each column, return results in a vector.
See: array-prod

5.11.12 Index Argument Checking

Functions with prefix `chk-` check a property of an argument and raise an error if that property does not hold. They return nothing. Functions with prefix `validate-` also check a property of an argument, but in addition carry out a normalization of the argument. They return the normalized argument, and raise an error if the property in question does not hold.

(scalarp `arg`**)**

Return `t` if `arg` is a zero-dimensional index and `()` otherwise.

(vectorp `arg`**)**

Return `t` if `arg` is a one-dimensional index and `()` otherwise.

(chk-idx `arg`**)**

Raise an error if `arg` is not an index, otherwise return `arg` .

(chk-idx-numeric `arg`**)**

Raise an error if `arg` is not a numeric index, otherwise return `arg` .

(chk-idx-contiguous `arg`**)**

Raise an error if `arg` is not a contiguous index, otherwise return `arg` .

(validate-dimension `a d`**)**

Check that dimension argument `d` is valid for array `a` . Return `d` or the non-negative equivalent when `d` is negative. Example:

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (validate-dimension m -1) )
= 1
```

(validate-shape `a shp-template`**)**

Check that shape of array `a` matches `shp-template` . The shape of `a` matches `shp-template` when the dimension of `a` equals the length of `shp-template` and when the extent of `a` in dimension `d` equals the `d` th element in list `shp-template` , or when the `d` th element in `shp-template` is `()` . Example:

```
? (let ((m (reshape (array-range 20) '(4 5))))
    (validate-shape m '(() 5)) )
= (4 5)
```

(same-shape-p `a1 a2`**)**

True when arrays `a1` and `a2` have the same shape.

5.11.13 Direct IDX Manipulations

(idx-set-dim `m d v`**)**

Set the `d` -th dimension of `m` to `v` . This generates an error if the resulting IDX overflows its storage.

```
? (setq m (double-array 3 4))
= [[ 0.00  0.00  0.00  0.00 ]
   [ 0.00  0.00  0.00  0.00 ]
   [ 0.00  0.00  0.00  0.00 ]]
? (m () () (range 0 11))
= [[ 0.00  1.00  2.00  3.00 ]
   [ 4.00  5.00  6.00  7.00 ]
   [ 8.00  9.00 10.00 11.00 ]]
? (idx-set-dim m 1 3)
= ()
? m
= [[ 0.00  1.00  2.00 ]
   [ 4.00  5.00  6.00 ]
   [ 8.00  9.00 10.00 ]]
```

(idx-set-mod `m d v`**)**

Set modulo of `d` -th dimension of `m` to `v` .

(idx-set-offset `m n`**)**

Set offset of `m` in its storage to `n`

5.11.14 Loading and Saving

There are two ways to load and save matrices and tensors in Lush which deal with two incompatible formats. The functions listed below use the so-called "classic" format, but cannot be used in compiled code. The other set of functions, described in the standard library section uses a different format (so-called IDX format) and are based on the C stdio library. This discrepancy exists largely for historical reasons and will be fixed eventually.

Loading, Saving, and Mapping Matrices

There are two types of "classic" matrix files: ASCII files and binary files. Binary files are a more accurate, more compact, and more efficient way of saving matrix data. ASCII matrix files should be used only reserved to

(save-array `a` `f`)

Store array `a` in the binary file `f` and return the file.

Argument `f` may be a file name (string) or a writable file descriptor created with `open-write` or `open-append` . When `f` is a file name, a suffix `".mat"` is added when needed. [Binary Matrix File Format.]

(save-array/text `a` `f`)

Store array `a` in text file `f` and return the file.

Argument `f` may be a file name (string) or a writable file descriptor created with `open-write` or `open-append` . When `f` is a file name, a suffix `".mat"` is added when needed. [Ascii Matrix File Format.]

(load-array [`a`] `f`)

Load array from file `f` and return it.

This function senses the file format and performs the adequate actions. When then name (symbol) `a` is specified, `load-array` binds `a` to the new array.

Argument `f` may be a file name (string) or a writable file descriptor. When `f` is a file name, a suffix `".mat"` is added when needed. [Binary Matrix File Format.][Ascii Matrix File Format.]

(map-array [`a`] `f`)

Map file contents of `f` to array storage of array `a` .

This function may not be available on all platforms. Mapped arrays are always read-only.

Native Matrix File Formats

This section describes the "classic" or "native" file formats used for Lush matrices. The IDX format (used by compilable matrix I/O functions) is not described here.

Ascii Matrix File Format

ASCII matrix files are generated with function `save-array/text` . These machine independent files are handy for transferring data between different computers. They tend however to eat a lot of disk space.

The first line of the file is a matrix header. It is composed of the letters ".mat", the number of dimensions and the size of each dimension. The array elements are written then, separated by spaces and newlines characters.

The following lines display the content of a valid ascii matrix file.

```
.MAT 2 3 4
 1 -1  1  2  -1  1 -3
 1 -1
 4  0  0
```

The same matrix could have been written in the following, more natural way.

```
.MAT 2 3 4
 1 -1  1
 2 -1  1
-3  1 -1
 4  0  0
```

Binary Matrix File Format

Binary matrix files are generated with function `save-array` . Binary matrix files begin with a header which describes the type and the size of the matrix. Then comes a binary image of the matrix.

The core header is a C structure defined as follows :

```
struct header {
        int magic;
        int ndim;
        int dim[3];
};
```

It can be followed by further `int` when the matrix has more than 3 dimensions.

The first member, `magic` , of this structure is a "magic" number, which encodes the type of the matrix. This number must be:

- `0x1E3D4C51` for a single precision matrix
- `0x1E3D4C53` for a double precision matrix
- `0x1E3D4C54` for an integer matrix
- `0x1E3D4C56` for a short matrix
- `0x1E3D4C55` for a byte matrix
- `0x1E3D4C52` for a packed matrix

The second member of this structure, `ndim` , is the number of dimensions of the matrix. Then come the dimensions themselves in the array `dim` . If there are more than three dimensions, the array `dim` is extended to accomodate the extra dimensions.

When the number of dimensions (specified in `ndim`) is greater than 3, the header is completed by `ndim - 3` further integers.

This header is followed by a binary image of the matrix. Elements are stored with the last index changing faster, i.e.

```
(0,0,0) (0,0,1) (0,0,2) ...  (0,1,0) (0,1,2) etc...
```

- as float numbers for single precision matrix

- as double numbers for double precision matrix
- as int numbers for integer matrix
- as short numbers for short matrix
- as unsigned char numbers for byte matrix
- and as char for packed matrix.

In this latter case, each byte represents a fixed point number between -8 and +8 (+8 not included), the first 4 bits (most significant nybble) contain the integral part and the remaining 4 bits contain the fractional part. It uses a two's complement format. Here are two C functions which convert a "compacted fixed point" number into a floating point number and back.

```
/* Converts a packed number to a float */
float unpack(b)
int b;
{
    if (b & 0x80)
        b |= ~0x7f;
    else
        b &= 0x7f;
    return (float)b / 16.0;
}

/* Converts a float into a single byte packed number */
unsigned char pack(x)
float x;
{
    if (x > 8.0-1.0/16.0)
        return 0x7f;
    else if (x < -8.0)
        return 0x80;
    else
        return x*16;
}
```

Foreign Matrix Files

The following functions are for reading and writing information stored in files created by other programs than Lush. These functions make it easy to read binary or text files into a Lush matrix.

Foreign Binary Matrices

(import-array `a f` [`offset`])

Read the contents of file `f` into array `a` and return `a` .

The byte stream read from `f` is interpreted as numbers (float), double precision numbers (double), integers (int), short integers (short), and so on, depending on the type of `a` . Array `a` must be contiguous. When argument `offset` is specified, the first `offset` bytes of the byte stream are skipped.

After executing this function, the file descriptor `f` points to the first byte following the data read.

(export-array `a` `f`)

Store data of array `a` in a binary file `f` .

Argument `f` may be a filename or a writeable file descriptor createdwith `open-write` or `open-append` . No header is written.

Foreign Ascii Matrices

(import-array/text `a` `f`)

Read numbers from text file `f` and store them in array `a` . Return `a` .

Array `a` must be a contiguous. Argument `f` is a filename or a readable file descriptor.

After executing this function, the file descriptor `f` points to the first nonblank character following the matrix data.

(export-array/text `a` `file`)

Store the data of the array `a` in text file `f` . Argument `file` may be a filename string or a file descriptor created with `open-write` or `open-append` . No header is stored.

5.11.15 Component-wise Unary Operations

All of these functions apply a unary function to each element and either write the result in the elements of the second argument if it is present, or return a new tensor with the result if the second argument is not present (except for idx-clear). The two argument must have the same dimensions. If their numerical types are different, appropriate conversions are performed.

(idx-clear `src`)

Set elements of `src` to 0.

(idx-minus `src` [`dst`])

negate all elements of `src` .

(idx-abs `src` [`dst`])

absolute value of elements of `src` .

(idx-sqrt `src` [`dst`])

square root of elements of `src` .

(idx-inv `src` [`dst`])

inverse of elements of `src` .

(idx-sin `src` [`dst`])

apply sine to elements of `src` .

(idx-cos `src` [`dst`])

apply cosine to elements of `src` .

(idx-atan `src` [`dst`])

apply arctangent to elements of `src` .

(idx-log `src` [`dst`])

apply log to elements of `src` .

(idx-exp `src` [`dst`])

apply exp to elements of `src` .

(idx-qtanh `src` [`dst`])

apply rational approximation to hyperbolic tangent to elements of `src` .

(idx-qdtanh `src` [`dst`])

apply derivative of the rational approximation to hyperbolic tangent to elements of `src` .

(idx-stdsigmoid `src` [`dst`])

apply the "standard" neural-net sigmoid function to elements of `src` .

(idx-dstdsigmoid `src` [`dst`])

apply derivative of the "standard" neural-net sigmoid function to elements of `src` .

(idx-expmx `src` [`dst`])

apply a rational approximation of (exp -x) to elements of `src` .

(idx-dexpmx `src` [`dst`])

apply derivative of `expmx` to elements of `src` .

5.11.16 Component-wise Dyadic Operations

All of these functions apply a dyadic function to each pair of corresponding elements in the first two arguments. They either write the result in the elements of the third argument if it is present, or return a new tensor with the result if the third argument is not present All the arguments must have the same dimensions. If their numerical types are different, appropriate conversions are performed.

(idx-add `m1 m2 [r]`**)**

component-wise addition of `m1` and `m2` . Result in `r` if present, or returned if not present.

(idx-sub `m1 m2 [r]`**)**

component-wise subtraction of `m1` and `m2` . Result in `r` if present, or returned if not present.

(idx-mul `m1 m2 [r]`**)**

component-wise multiplication of `m1` and `m2` . Result in `r` if present, or returned if not present.

(idx-div `m1 m2 [r]`**)**

component-wise division of `m1` and `m2` . Result in `r` if present, or returned if not present.

5.11.17 Contracting Operations

(array-sum `m [d]`**)**

Sum `m` over dimension `d` (default -1).

(array-prod `m [d]`**)**

Multiply `m` over dimension `d` (default -1).

5.11.18 Contracting Operations with Scalar Result

The following functions include dot products, distances, sums of terms, min, max, etc and return scalars. These operations "contract" all the dimensions. The dyadic ones can be seen as generalized dot product of two tensors (e.g. the sum of all the products of corresponding terms in the two tensors). These function have a base form and an accumulating form (which accumulates the result in idx0 past as last argument). If the last (and optional) idx0 argument is present, the result is written in it. If it is not present, a number (not an idx0) is returned.

(idx-sum `m` [`r`])

sum of all terms of `m` .

(idx-sup `m` [`r`])

max of all terms of `m` .

(idx-inf `m` [`r`])

min of all terms of `m` .

(idx-sumsqr `m` [`r`])

sum of squares of all terms of `m` .

(idx-dot `m1` `m2` [`r`])

generalized dot product of `m1` and `m2` , i.e. the sum of all products of corresponding terms in `m1` and `m2` .

(idx-sqrdist `m1` `m2` [`r`])

generalized Uclidean distance between `m1` and `m2` , i.e. the sum of squares of all the differences between corresponding terms of `m1` and `m2` .

(idx-sumacc `m` `r`)

sum of terms of `m` . Result accumulated in idx0 `r` .

(idx-supacc `m` `r`)

max of terms of `m` . Result accumulated in idx0 `r` .

(idx-infacc `m` `r`)

min of terms of `m` . Result accumulated in idx0 `r` .

(idx-sumsqracc `m` `r`)

sum square of terms of `m` . Result accumulated in idx0 `r` .

(idx-dotacc `m1` `m2` `r`)

generalized dot product of `m1` and `m2` , i.e. the sum of all products of corresponding terms in `m1` and `m2` . Result accumulated in idx0 `r` .

(idx-sqrdistacc `m1 m2 r`**)**

generalized Uclidean distance between `m1` and `m2` , i.e. the sum of squares of all the differences between corresponding terms of `m1` and `m2` . Result accumulated in idx0 `r` .

5.11.19 Operations between Tensors and Scalars

(idx-dotm0 `m s [r]`**)**

multiply all terms of `m` by scalar `s` (an idx0).

(idx-addm0 `m s [r]`**)**

add scalar `s` (an idx0) to all terms of `m` .

(idx-dotm0acc `m s r`**)**

multiply all terms of `m` by scalar `s` (an idx0). Result accumulated in `r` .

(idx-addm0acc `m s r`**)**

add scalar `s` (an idx0) to all terms of `m` . Result accumulated in `r` .

5.11.20 Matrix/Vector and 4-Tensor/Matrix Products

(idx-m2dotm1 `m1 m2 [r]`**)**

matrix-vector multiply.

(idx-m4dotm2 `m1 m2 [r]`**)**

4-tensor by 2-matrix multiplication: R_ij = sum_kl M1_ijkl * M2_kl

(idx-m2dotm1acc `m1 m2 r`**)**

matrix-vector multiply. Result accumulated in `r` .

(idx-m4dotm2acc `m1 m2 r`**)**

4-tensor by 2-matrix multiplication with accumulation: R_ij += sum_kl M1_ijkl M2_kl

5.11.21 Outer Products

(idx-m1extm1 `m1 m2 [r]`**)**

outer product between vectors: R_ij = M1_i * M_j

(idx-m2extm2 `m1 m2 [r]`**)**

outer product between matrices. Gives a 4-tensor: R_ijkl = M1_ij * M2_kl

(idx-m1extm1acc `m1 m2 r`**]**

outer product between vectors with accumulation: R_ij + = M1_i * M_j

(idx-m2extm2acc `m1 m2 [r]`**)**

outer product between matrices with accumulation. Gives a 4-tensor: R_ijkl += M1_ij * M2_kl

5.12 Objects

Every Lush object has an associated type which may be queried with function `classof` . The type may either by a *builtin type* (also *builtin class*) or a *user-defined class* . This section talks about *objects* in the narrower, object-oriented sense, as a thing whose type is a user-defined class.

5.12.1 Object Terminology

A class defines two major kinds of information:

- An atom can contain storage for one or several lisp objects. Such storage areas are called *slots* , and have symbolic names. A class defines the number of slots allocated in a new atom as well as their names. This slot information is statically defined by the `defclass` function during class definition;
- Atoms can receive messages sent with the function ==> . Messages are identified by a symbol called the *message selector* . A class defines the possible message selectors and functions (i.e methods) to execute when it receives such messages. These *methods* are dynamically defined with the `putmethod` function.

In addition, predefined classes associated with standard Lush objects often have hidden properties. Such classes are special and do not inherit from the `object` class. For example, a symbol has some storage for its value; the storage is not a slot. A symbol also returns its value when it is evaluated; this behavior is not a method.

5.12.2 Inheritance

New classes defined with the `defclass` function are always subclasses of another class (i.e. its superclass). The superclass itself may have its own superclasses. In this way:

- *A class inherits the slots of its superclasses* . When a new object is created, space is allocated for the slots defined by its class and also for the slots defined by all of its superclasses;
- *A class inherits the methods of its superclasses* . When an object receives a message with a given selector, it searches the methods defined by its class for the selector. If no method is found that matches the selector it searches the methods of its superclass, the methods of the superclass of its superclass, and so on. When it reaches a matching method it executes the associated function in the context of the class that defines the matching method.

5.12.3 Predefined Classes

Here is a compact list of the main predefined classes. Most of them are named using uppercase symbols, which ought to be surrounded with bars to avoid the usual lowercase conversion:

- `object` : the root class of most user defined classes;
- `Class` : classes are actually lisp objects, instance of the `Class` class. Thus, class `Class` is an instance of itself;
- `Number` : the class of the numbers;
- `Cons` : the class of the pairs used for building lists;
- `Symbol` : the class of the symbols;
- `String` : the class of the strings;
- `DE DF DM DX DY` : the classes of functions;
- and so on.

All user defined classes must be direct or indirect subclasses of the class `object` . You cannot inherit a predefined class like `|SYMB|` , it is not a subclass of `object` . The `object` class defines a regular lisp object with no hidden properties. Instances of subclasses of the class `object` are easily created with the function `new` .

5.12.4 Defining a Class

(defclass `name superclass s1 ... sn`**)**

Define a subclass of class `superclass` with slots `s1` ... `sn` . The class is stored into symbol `name` , which is returned. Slot specifications `s1` ... `sn` can be symbols or lists `(symbol default)` which indicate initial values assigned to the slots of a new object.

Example:

```
;;; Creates a class <rect> with two slots: <width> and <height>.
? (defclass rect object
    width height )
= rect
;;; Creates a subclass of <rect>,
;;; plus an additional slot <name> with default value <"noname">.
? (defclass namedrect rect
    (name "noname") )
= namedrect
```

(makeclass `classname superclass slotnamelist slotdefaultlist`**)**

This a lower-level function to create a new class named `classname` . Most user will prefer to use `defclass` instead. The new class inherits class `superclass` . List `slotnamelist` contains the names of the additional slots defined by the new class. List `slotdefaultlist` contains the default values for these slots. Both lists are in reverse order (i.e. the first slot in the list appears last when pretty-printing the class). This function does not set the value of symbol `classname` , use the macro function `defclass` for that purpose.

(classname `cl`**)**

Return the name of class `cl` (a symbol).
Example:

```
? (classname rect)
= rect
```

(slots `cl`**)**

See: allslots
Return the list of slots defined for class `cl` .
Example:

```
? (slots rect)
= (width height)
? (slots namedrect)
= ((name "noname"))
```

(allslots `cl`)

See: slots

Return list of all slots owned by objects of class `cl` .

(methods `cl`)

See: allmethods

Return the list of method names defined for class `cl` (a list of symbols).

(allmethods `cl`)

See: methods

Return the list of names of all the methods that class `cl` can receive (a list of symbols).

(super `cl`)

Return the superclass of class `cl` . Return `nil` if `cl` has no superclass.

Example:

```
? (super rect)
= ::class:object
? (classname (super namedrect))
= rect
```

(subclasses `cl`)

Return the list of all subclasses of class `cl` . When argument `cl` is the empty list `subclasses` return the list of root classes (classes without superclass).

Example:

```
(subclasses object)
```

5.12.5 Creating Objects

(new `cl` ... `args` ...)

Create new instance of class `cl` .

If no constructor method is defined for class `cl` the constructor of its superclass is used by default. The constructor of class `object` takes no arguments and does nothing.

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Creates instances of classes <rect> and <namedrect>.
? (setq r (new rect))
= ::rect:06fa0
? (setq nr (new namedrect))
= ::namedrect:06fe8
```

This function only works when creating instances of subclasses of class `object` . It is not possible to directly create instances of subclasses of other predefined classes.

(new-empty `cl`**)**

Create new instance of class `cl` without calling the constructor. All slots are initialized to `nil` .

Use this function with care because the empty object might trump the expectations of other code working with this class.

(copy-object `src` [`dest`]**)**

Create a new instance with the same instance variables as object `src` . When object `dest` is given, don't create a new instance but copy to `dest` . Return the new object or `dest` , respectively.

The copy is an instance of the same class with all slots initialized to the same values. The contents of the slots are not copied. That is, `copy-obj` equivalent to:

```
(let* ((cl (classof obj))
       (dest (new-empty cl)) )
  (setq :src:slot1 :dest:slot1)
  (setq :src:slot2 :dest:slot2)
  ...
  (setq :src:slotN :dest:slotN)
  n )
```

Use this function with care because other code working with this class might not be able to deal with multiple copies of the same object. Calls to destructors are particularly problematic.

(delete `obj`**)**

Delete object `obj` .

The operation of this function is somewhat tricky because all other references to object `obj` must be converted to references to the empty list.

If destructor methods are defined for an object then destruction messages are sent. Object `oj` is then converted to an instance of the class `|ZOMBIE|` . During its normal operation, the interpreter recognizes these zombies and always replaces a reference to a zombie by a reference to the empty list.

(classof `obj`**)**

Return the class of an object `obj` .

Example:

```
(classof (new object))
```

(isa `obj cl`**)**

True when `obj` is of class `cl` .

5.12.6 Accessing Slots

There are various ways to read or change the values stored in the slots of an object. Changing the value stored in a slot changes the object in a way comparable to `rplaca` and `rplacd` , in the case of lists. This change will be reflected through all pointers referring to the same object.

See: (== `n1 n2`)

`:obj:slot1...:slotn`

See: (scope `obj slot1` ... `slotn`)

See: (scope `symb`)

This macro character is expanded as a call to the `scope` function.

(scope `obj slot1` **...** `slotn`**)**

See: : `obj` : `slot1` ...: `slotn`

See: (scope `symb`)

The simplest method for setting or getting the slots of an object is called the *scope macro* . The syntax `:obj:slot` refers to the slot `slot` of object `obj` . Actually, the scope macro character converts expression `:obj:slot` into a list `(scope obj slot)` .

When this list is evaluated the `scope` function returns the value of slot `slot` of object `obj` . Moreover, most functions that affect the value of a symbol recognize such a list as a reference to slot `slot` of object `obj` .

Nested scope macros are allowed: `:obj:slot_1:slot2` refers to slot `slot2` of the object contained in slot `slot1` of object `obj` .

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
? (setq :nr:name "myrectangle")
= "myrectangle"
? (setq :nr:width  8
        :nr:height 6 )
= 6
? (print :nr:width :nr:height :nr:name)
8 6 "myrectangle"
= "myrectangle"
```

(scope `symb`**)**

See: (scope `obj slot1 ... slotn`)

See: : `obj : slot1 ...: slotn`

The scope macro has another important use as the *unary scope macro* . Expression `:symb` is converted into list `(scope symb)` which refers to the global value of symbol `symb` .

Example:

```
? (defvar x 3)
= 3
? (let ((x 8))
    (print :x)
    (setq :x 6)
    (print x) )
3
8
= 8
? x
= 6
```

(with-object [`cl`] `obj l1` **...** `ln`**)**

See: (scope `obj slot1 ... slotn`)

See: this

Calls function `progn` on `l1 ... ln` , within the scope of object `obj` , and returns the last result. Within the object scope each slot of the object, either defined by its class or inherited from its superclasses, can be directly accessed as the value of its symbolic slot name.

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
? (setq width ())
= ()
? (with-object r
    (setq width 4)
    (setq height 5) )
= 5
? width
= ()
? (with-object r (* width height))
= 20
```

Within an object scope the symbol `this` always refers to object itself.

The optional argument `cl` must be the object class or one of its superclasses. The only visible slots are then defined by class `cl` and its superclasses. This can make a difference when `obj` is an instance of a subclass of `cl` .

5.12.7 Defining Methods

(defmethod `cl symb args . body`**)**

See: Argument List

See: (==> `obj symb` ... `args` ...)

Defines a method named `symb` for class `cl` . Argument `symb` must be a symbol and is not evaluated. Argument `args` must be a valid argument list.

When an instance of class `cl` receives a message whose selector is `symb` the body `body` of the method is executed with a call to function `progn` . During this execution the slots defined by `cl` and its superclasses are bound to their symbolic names, as in `with-object` , and the arguments of the message are evaluated and bound to the symbols in argumentlist `args` .

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
;;
;; This is a method for computing the surface of a rect.
? (defmethod rect surface()
    (* width height) )
= surface
;; a method for printing  n times the name of a namedrect
? (defmethod namedrect showname(n)
    (repeat n (print name)) )
= showname
```

(demethod `cl symb args . body`**)**

Identical to `defmethod` .

(dfmethod `cl symb args . body`**)**

Defines an flambda-method named `symb` for class `cl` . flambda-methods do not evaluate their arguments, unlike regular methods defined with demethod or defmethod.

(dmmethod `cl symb args . body`**)**

Defines an mlambda-method named `symb` for class `cl` . mlambda-methods are to regular methods what macros (defined with `dm` or `mlambda`) are to regular functions (defined with `de` or `lambda`).

(putmethod `cl symb func`**)**

Add method `symb` to class `cl` . `func` must be a lambda, flmabda, or mlambda.

(pretty-method `cl symb`**)**

Prints a nicely indented definition of method `symb` of class `cl` .

5.12.8 Sending Messages

(send `obj method ... args ...` **)**

Invoke `method` of object `obj` with arguments `args` .
Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
;; Method <surface> is defined as an example for <defmethod>.
;;
;; Send a surface message to rect <r>
? (send r 'surface)
= 20
;; Send a showname message to namedrect <nr>
? (send nr 'showname 2)
"myrectangle"
"myrectangle"
= "myrectangle"
;; Send a surface message to namedrect <nr>
;; The method is inherited from superclass <rect>
? (send nr 'surface)
= 48
```

When an object receives a message its method is executed in the object scope defined by the class that owns the method. If the method is defined by a superclass only the slots of that superclass and of its superclasses may be directly referred to by their names.

When method `method` is not defined by the class or any of its superclasses, a method named `-unknown` is searched for. If found, the `-unknown` method is executed with two arguments: the initial method name and the list of the evaluated arguments `args` . An error occurs if method `-unknown` is not defined.

(==> `obj name ... args ...`)

Invoke method named `name` of object `obj` with arguments `args` . In contrast to `send` , this macro does not evaluate its second argument, thus, `name` must the name of a method.

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
;; Method <surface> is defined as an example for <defmethod>.
;;
;; Send a surface message to rect <r>
? (==> r surface)
= 20
;; Send a showname message to namedrect <nr>
? (==> nr showname 2)
"myrectangle"
"myrectangle"
= "myrectangle"
;; Send a surface message to namedrect <nr>
;; The method is inherited from superclass <rect>
? (==> nr surface)
= 48
```

(==> `obj` (`cl . name`) `... args ...`)

This construction is known as a *cast-and-send* . A message with selector `name` and arguments `args` is sent to the object `obj` , considered as an instance of class `cl` . Methods are searched for in class `cl` and its superclasses, instead of the class of `obj` and its superclasses.

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
;; Method <surface> is defined as an example for <defmethod>.
```

```
;;
;; Override surface method of class rect
? (defmethod namedrect surface()
       ;; prints the name by sending a showname message
       (==> this showname 1)
       ;; returns 1+ the result of the surface method of class rect
       (1+ (==> this (rect . surface))) )
= surface
? (==> nr surface)
"myrectangle"
= 49
;; Still call the surface method of class rect
? (==> nr (rect . surface))
= 48
```

this

See: (with-object `obj l1 ... ln`)

See: (==> `obj symb ... args ...`)

While Lush is executing a method you can use symbol `this` to refer to the object receiving the message.

(getmethod `cl name`)

Searches class `cl` and its superclasses for a method called `name` (a symbol). If a matching method exists this function returns the method as a DF function.

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
;; Method <surface> is defined as an example for <defmethod>.
;;
;; There is no method yoyo in rect
? (getmethod rect 'yoyo)
= ()
;; there is a method surface in rect
? (getmethod rect 'surface)
= DF:0604c
```

5.12.9 Other Special Methods

Special methods are methods that can be called implicitly by other functions or macros. An example is the special method `-destructor` which gets called

by `delete` . Another special method is `-listeval` which lets objects behave as functions.

(==> `obj` -listeval ... `args` ...)

Method `-listeval` method is called implicitly when Lush evaluates a list whose first element is an object.

Example:

```
? (defclass A Object)
= a
? (defmethod A -listeval args
     (printf "I got \\%d arguments\\n" (length args)))
= -listeval
? ((new A) 1 2 3 4)
I got 4 arguments
= ()
```

(==> `obj` -emptyp)

Return true when `obj` is empty. This method is used by `emptyp` .

(==> `obj` pname)

See: (pname `l`)

The function `pname` returns a string that describes a lisp object. When executing this function Lush searches the object for a method `pname` before applying the hardcoded defaults. Overriding method `pname` lets you redefine how Lush prints certain objects.

Example:

```
;; Classes <rect> and <namedrect> are defined as examples for <defclass>.
;; Objects <r> and <nr> are defined as examples for <new>.
;; Method <surface> is defined as an example for <defmethod>.
;;
;; Defines a pname function for the rect class
? (defmethod rect pname()
    (sprintf "<rectangle \\%d by \\%d>" width height) )
= pname
? (new rect 8 6)
= <rectangle 8 by 6>
```

(==> `obj` print)

See: (prin `l1` ... `ln`)

When Lush prints an object with function `prin` or `print` it searches for a method `print` before applying hardcoded defaults based on function `pname` .

It is a good idea to override method `print` instead of method `pname` when large amounts of text are necessary (e.g. in the case of matrices).

(==> `obj` pprint)

See: (pprin `l1` ... `ln`)

When Lush prints an atom with function `pprin` or `pprint` it searches for a method `pprint` before applying the hardcoded defaults.

(==> `obj` pretty)

See: (pretty `f`)

Function `pretty` displays an object by sending them a message `pretty` . The default method `pretty` defined by class `object` displays the slots of the object. Specific `pretty` methods are also defined for functions, symbols, classes, and so forth.

(==> `obj` -deepcopy `clone`)

See: deepcopy

Make `clone` and independent copy of `obj` and return `()` .

(==> `obj` -destructor)

See: Constructors and Destructors

This method is invoked when the object is destroyed. Note: Never explicitly call the destructor method of the super class. Lush does this implicitly.

(==> `obj` -unknown `symb` `args`)

See: Sending Messages

This method is invoked when a message `symb` is sent to an object `obj` whose class or superclasses do not define a suitable method. Argument `args` is the list of message arguments the object is destroyed. An error message is produced when this method is not defined.

5.12.10 Template classes

Template classes are a way to define template code for classes. Good examples for the use of template classes are container datatypes. A template class differs from a regular classes in a number of ways:

- The defining form is `deftemplate` instead of `defclass` .

- A template class is not meant to be instantiated but is meant to be used as a superclass.
- The type of some slots may be declared `-any-` , which effectively leaves the slot type unspecified. These slots are called *template slots* .
- Those methods that do not make references to slots of unspecified type may be compiled, others may not be compiled. Compiled methods are inherited the regular way by subclasses, non-compiled methods are called *template methods* and are inherited *as source* . The class itself always has to be compiled.

The type of template slots has to be declared in subclasses. There are two ways of referring to the type of template slots in template methods. In type declarations, use `-typeof-` in conjunction with the template slot name. In all other places use `classof` . The following example defines a template for stack classes.

```
(deftemplate Stack object
  ((-idx1- (-any-)) st)   ; stack data
  ((-any-) el) )          ; stack element (dummy slot)

(defmethod Stack hashcode ()
  (sprintf "\\%10d" (to-int #{ $this #})) )

(defmethod Stack Stack ()
  (let ((v (double-array* 8)))
    (setq st (clone-array (to-obj (classof st) (to-gptr v)))) )
  (idx-trim! st 0 0 0)
  (setq el 0)
  ())

(defmethod Stack push (i)
  (declare (-typeof- el) i)
  (let ((n (length st)))
    (declare (-int-) n)
    (array-extend! st 0 1)
    (st n i) )
  ())

(defmethod Stack pop ()
  (when (idx-emptyp st)
    (error "stack is empty") )
  (let ((n (length st)))
    (declare (-int-) n)
    (prog1 (st (- n 1))
```

```
      (idx-extend! st 0 -1) )))

(defmethod Stack -emptyp ()
  (idx-emptyp st) )
```

In this example `st` and `el` are template slots. Note that the value of template slot `el` is never being used. It was included in the definition for its type only (which is referred to in the definition of method `push`).

Method `hashcode` does not refer to template slots, so we may compile it. All other methods are template methods.

```
  (dhc-make "stack_template" (Stack hashcode))
```

Now we may define stack classes for specific types.

```
(defclass ByteStack Stack
  ((-idx1- (-byte-)) st)
  ((-byte-) el) )

(defclass DoubleStack Stack
  ((-idx1- (-double-)) st)
  ((-double-) el) )

(defclass IntStack Stack
  ((-idx1- (-int-)) st)
  ((-int-) el) )

(dhc-make-class () DoubleStack ByteStack IntStack)
```

Note that all three stack classes share the implementation of method `hashcode` but have specific implementations for all other methods.

(deftemplate `name super s1...sn`**)**

Define a template class with superclass `super` and slots `s1` ... `sn` .

5.13 Hash Tables

Hash tables provide a powerful data structure to maintain associations between lisp objects. These data structures behave like arrays except that any lisp object is an acceptable subscript. This simple and flexible access makes hash tables extremely convenient.

They are so convenient that you will be quickly tempted to use them everywhere. However, keep in mind that a simple array requires less memory (about five times less) and less processor cycles than a hash table.

5.13.1 Representing Associations with Hash Tables

See: (= `n1 n2`)

See: (== `n1 n2`)

A hash table is an instance of class `HTable` . Like most lisp objects, hash tables can be compared using function `=` and saved using function `bwrite` .

A hash table maintains a set of associations between two lisp objects, namely a key and a value. This set is organized in a way that allows fast retrieval of the value associated to a given key.

Hash table queries search an association whose key is equal to the queried key. Since there are two ways to test equality (i.e. functions `=` and `==`), there are two kinds of hash tables.

The most useful hash tables use the logical equality, as implemented by function `=` . Two objects are logically equal if they convey the same useful information. These hash table work properly as long as you do not modify the objects used as keys by the hash table. This can happen if you used arrays or custom objects as keys in your hash table. See the description of function `htable-rehash` for more information about this point.

Hash tables can also rely on pointer equality, as implemented by function `==` . These hash tables are useful to associate a piece of information with a particular lisp object. Functions `getp` and `putp` , for instance, make use of pointer-equality hash tables. The associations of a pointer-equality hash table are automatically removed when the key object is deallocated.

(htable [`nelem` [`==-equality` [`keyerror-flag`]]])

Function `htable` returns a new empty hash table. The arguments `nelem` , `equality-flag` and `keyerror-flag` are optional, default values are 31, `nil` , and `nil` , respectively.

Argument `nelem` specifies an estimate of the number of elements in the hash table. This estimate is useful for saving a few processor cycles when filling the table.

Argument `==-equality` is a boolean flag specifying whether the hash table will use logical equality ("=-equality") or pointer equality ("==-equality"), the default is `()` . With pointer equality references to objects used as keys are weak references.

Argument `keyerror-flag` is a boolean flag specifying whether the hash table will raise an error when a query with a non-existent key is made (default is `()`).

(`htable key`)

This expression returns the value associated with key `key` in hash table `htable` . Expression `key` can be any lisp object. If no association is defined for key `key` , the empty list is returned.

(htable key value [key2 value2 ...])

This expression associates each `value` to each `key` in the hash table `htable` . The value can be subsequently retrieved by expression `(htable key)` .

This operation is performed in two logical steps: (a) any existing association matching key `key` is first deleted, and (b) a new association is created if argument `value` is not the empty list.

Example:

```
? (setq a (htable))
= ::HTable:6f3cac
? (a 'hello "hi"  '(3 4) "ho")
= ::HTable:6f3cac
? (a 'hello)
= "hi"
? (a '(3 4 5))
= ()
? (a '(3 4))
= "ho"
```

(htable-update ht1 ht2)

Update `ht1` with key-value pairs in `ht2` and return the modified htable `ht1` .

(htable-size htable)

Function `htable-size` returns the number of associations managed by the hash table.

(htable-keys htable)

Function `htable-keys` returns a list of the keys of all the associations managed by the hash table. It is guaranteed that querying these keys will eturn a non nil result.

```
? (setq a (htable))
= ::HTable:6f43c
? (for (i 0 10) (a i (sqrt i))
= ::HTable:6f43c
? (htable-keys a)
= (7 6 3 10 5 9 8 4 2 0 1)
```

(copy-htable ht)

;; Create a new htable with the same keys and values as `ht` ;; and return it.

(htable-alist `htable`**)**

See: (assoc `key alist`)

Function `htable-alist` returns an alist representing all associations managed by the hash table. This function also provide a way to iterate over all associations:

```
? (setq a (htable))
= ::HTable:6f43c
? (for (i 0 10) (a i (sqrt i))
= ::HTable:6f43c
?  (each (((key . value) (htable-alist a)))
     (printf "\\%4d \\%6.4f\\n" key value))))
   7 2.6458
   6 2.4495
   3 1.7321
  10 3.1623
   5 2.2361
   9 3.0000
   8 2.8284
   4 2.0000
   2 1.4142
   0 0.0000
   1 1.0000
= ()
```

(htable-delete `ht k`**)**

;; Remove item with key `k` from htable `ht` . Return the modified htable.

(htable-rehash `htable`**)**

Each association managed by a hash table contains a pointer to the key object and a pointer to the value object. Functions that modify the values of these objects also modify the state of the hash table.

Hash table queries search an association whose key is logically equal to the queried key (unless pointer equality was specified when creating the hash table). The hash table works by storing each association in locations depending on the information conveyed by the key objects.

Modifying the object used as a key creates a lot of problems. The corresponding association is no longer stored in a location corresponding to the new information conveyed by the key objects. The hash table state is no longer consistent.

Function `htable-rehash` triggers the relocation of all associations of the hash table `htable` to the location corresponding to the new values of the key objects. You can then reliably use the hash table again.

```
? (setq a (htable))
= ::HTable:6f32c
? (setq m [1 2])
= [1 2]
? (a m 3)
= ::HTable:6f32c
? (a [1 2])
= 3
? (m 0 0)
= [ 0.00 2.00 ]
? (a [1 2])
= ()      ;; we lost it
? (a [0 2])
= ()      ;; result is undefined
? (htable-rehash a)
= 1
? (a [0 2])
= 3       ;; entry is no longer lost
? (a [1 2])
= ()      ;; this key no longer exists
```

The relocation of the hash table entries actually happens when you first access the hash table after calling `htable-rehash` . You can call function `htable-rehash` several times in a row. The cost of relocating all the hash table entries will be only incurred once when you will access the hash table again.

(htable-info `htable`)

Function `htable-info` returns a list containing statistical information about the hash table. This list is an alist with the following form:

```
((size . 1294) (buckets . 3841) (hits 1132 145 14 3))
```

In this example, the hash table contains 1294 associations. Associations are stored into 3841 buckets. Out of the 1294 associations, 1132 canbe retrieved directly, 145 require one extra search iteration, 14 require two extra search iteration, etc.

5.13.2 Representing Sets with Hash Tables

Sets are easily represented by creating a hash table and defining associations whose keys are the set elements and whose value is non nil. Here are some of the basic hash table expressions that can be used:

- `(htable element)` returns a non nil result if element `element` belongs to the set represented by `htable` .
- `(htable element t)` adds element `element` to the set represented by `htable` .
- `(htable element ())` removes element `element` from the set represented by `htable` .
- `(htable-keys htable)` returns a list of all the elements of the set represented by `htable` .

Other functions given in this section perform additional operations on sets represented by hash tables.

(hset `eltlist` [`flag`]**)**

Author(s): Raymond Martin (count flag part)

Function `hset` returns a hash table representing the set of all elements of list `eltlist` (as a set, the elements are only entered once). The hash table associations are not returned in any particular order (e.g. when using `htable-keys` , `htable-alist` , etc.), use `sort-list` or other function to get a sorted or ordered list or other representation.

Argument `flag` is a boolean value specifying whether or not the hash table will maintain a count as the value of the same elements encountered when building the hash table (as opposed to `t` when `flag` is `nil`). By default, `flag` is `nil` .

These counts can be useful for performing statistics or other calculations.

```
? (setq a (hset '(1 2 3 4 5 4 3 2 1 5 4 2 6) t ))
= ::HTable:6f43c
? (each (((elem . value) (htable-alist a)))
     (printf "\\%4d \\%4d\\n" elem value))))
   4   3
   2   3
   1   2
   6   1
   3   2
   5   2
= ()
```

(hset-and `htable1 htable2`**)**

Function `hset-and` returns a new hash table representing the intersection of the sets represented by hash tables `htable1` and `htable2` .

```
? (htable-keys (hset-and (hset '(1 2 3 4)) (hset '(2 4 5 6))))
= (4 2)
```

(hset-or `htable1 htable2`**)**

Function `hset-or` returns a new hash table representing the union of the sets represented by hash tables `htable1` and `htable2` .

```
? (htable-keys (hset-or (hset '(1 2 3 4)) (hset '(2 4 5 6))))
= (4 2 1 6 3 5)
```

5.14 Dates

Dates are instances of a special class `|DATE|` . This facility is built on top of various system functions which are themselves based on a count of seconds since January 1st, 1970. Most systems are able to handle a date between 1910 and 2030.

Date are stored using several formats depending of their accuracy. Each format is defined as a range over various components of a date: `year` , `month` , `day` , `hour` , `minute` , `second` .

- For instance, a date " `year` to `day` " identifies any single day between 1910 and 2030.
- Similarly, a date " `year` to `second` " identifies any single second in the same interval.
- A date " `hour` to `minute` " identifies any single minute within a day.

Finally we have defined a standard way to numerically encode dates: The possibly fractional number of days (daynumber) since January 1st, 1970 at zero hour local.

5.14.1 (date-to-day `date`)

Returns the possibly fractional number of days between January 1st, 1970 and the specified `date` .

```
? (date-to-day (string-to-date "1993-12-04" 'year 'day))
= 8738
```

```
? (date-to-day (now))
= 15010.9969
```

5.14.2 (day-to-date daynumber)

Builds a date string from a possibly fractional day number.

```
? (day-to-date 8738)
= ::DATE:year-second:(1993-12-04 00:00:00)


? (day-to-date 8738.23)
= ::DATE:year-second:(1993-12-04 05:31:11)
```

5.14.3 (today)

Returns today's date at zero hour as a date " `year` to `day` "

```
? (today)
= ::DATE:year-day:(2011-02-05)
```

5.14.4 (now)

Returns today's date and now's time as a date " `year` to `second` "

```
? (now)
= ::DATE:year-second:(2011-02-05 23:55:30)
```

5.14.5 (split-date date)

Returns an alist contaning all the components of a date `date` . Each component can be accessed using function `assoc` . The following components are provided:

```
<year>:       The year (minus 1900).
<month>:       The month number (0..11).
<day>:       The day of the month (1..31).
<hour>:       The hour (0..23).
<minute>:       The number of minutes (0..59).
<second>:       The number of seconds.
<wday>:       The day of the week (0..6).
<yday>:       The day of the year (1..366).
```

These items are present if the resolution of the date object allows it. The last two items require at least a date " `year` to `day` "

```
? (split-date (now))
= ((year . 111) (month . 2) (day . 5) (hour . 23) (minute . 55) (second . 30)
    (wday . 6) (yday . 35) )


? (split-date (today))
= ((year . 111) (month . 2) (day . 5) (wday . 6) (yday . 35))
```

5.14.6 (date-type `date`)

This function returns a string describing the current resolution of a date `date` as a list of two symbols chosen in `year` , `month` , `day` , `hour` , `minute` , `second` .

```
? (date-type (today))
= (year day)


? (date-type (now))
= (year second)
```

5.14.7 (date-extend `date from to`)

This function returns a new date equivalent to `date` with the resolution specified by the two symbols `from` and `to` which may be any of `year` , `month` , `day` , `hour` , `minute` , `second` .

Reducing the resolution of a date is achieved by truncating unneeded components. Extending the resolution of a date towards more accuracy is achieved by inserting zeroes. Extending the resolution of a date towards fields of larger magnitude is achieved by inserting the current year, month, day, etc...

```
? (date-extend (now) 'year 'minute)
= ::DATE:year-minute:(2011-02-05 23:55)


? (date-extend (date-extend (now) 'year 'minute) 'hour 'second)
= ::DATE:hour-second:(23:55:00)
```

5.14.8 (date-to-string `date` [`format`])

Format a date `date` according to `format` .

Argument `format` is a format string reminiscent of `printf` . When argument `format` is omitted, a regular ANSI string is produced.

```
YYYY-MM-DD HH:MM:SS              for <year> to <second>
YYYY-MM-DD                      for <year> to <day>
MM-DD HH:MM                      for <month> to <minute>
```

The format for `date-to-string` is a character string that consists of field descriptors and text characters, reminiscent of `printf` . Each field descriptor consists of a % character followd by another character that specifies the replacement for the field descriptor. All other characters are copied from fmt into the result. The following field descriptors are supported:

- %%: Same as %.
- %a: Day of week, using locale's abbreviated weekday names.
- %A: Day of week, using locale's full weekday names.
- %b,%h: Month, using locale's abbreviated month names.
- %B: Month, using locale's full month names.
- %c: Date and time as %x %X.
- %C: Date and time, in locale's long-format date and time representation.
- %d: Day of month (01-31).
- %D: Date as %m/%d/%y.
- %e: Day of month (1-31; single digits are preceded by a blank).
- %H: Hour (00-23).
- %I: Hour (00-12).
- %j: Day number of year (001-366).
- %k: Hour (0-23; single digits are preceded by a blank).
- %l: Hour (1-12; single digits are preceded by a blank).
- %m: Month number (01-12).
- %M: Minute (00-59).
- %n: Same as \n.

- %p: Locale's equivalent of AM or PM.
- %r: Time as %I:%M:%S %p.
- %R: Time as %H:%M.
- %S: Seconds (00-59).
- %t: Same as \t.
- %T: Time as %H:%M:%S.
- %U: Week number of year (01-52), Sunday is the first day of the week.
- %w: Day of week; Sunday is day 0.
- %W: Week number of year (01-52), Monday is the first day of the week.
- %x: Date, using locale's date format.
- %X: Time, using locale's time format.
- %y: Year within century (00-99).
- %Y: Year, including century (for example, 1988).

The difference between %U and %W lies in which day is counted as the first day of the week. Week number 01 is the first week with four or more January days in it.

```
? (date-to-string (today))
= "2011-02-05"


? (date-to-string (now))
= "2011-02-05 23:55:30"


? (date-to-string (now) "\\%C \\%X")
= "20 11:55:30 PM"
```

5.14.9 (string-to-date `string` [`from` `to`])

Function `string-to-date` converts the ansi date string `string` to a date using the resolution specified by `from` and `to` .

Argument `from` and `to` are symbols specifying the resolution of the target date. They may be any of the following symbols: `year` , `month` , `day` , `hour` , `minute` , `second` . When these arguments are ommited, a date " `year` to `second` " is assumed.

Abbreviated dates are handled by filling the leftmost fields with the current date and the rightmost fields with zeroes.

```
? (string-to-date "1993-12-12 8:30" 'year 'minute)
= ::DATE:year-minute:(1993-12-12 08:30)
```

5.14.10 (date-add-second date count)

This function adds `count` seconds to date `date` .

```
? (date-add-second (now) 67)
= ::DATE:year-second:(2011-02-05 23:56:37)
```

5.14.11 (date-add-minute date count)

This function adds `count` minutes to date `date` .

```
? (date-add-minute (now) 67)
= ::DATE:year-second:(2011-02-06 01:02:30)
```

5.14.12 (date-add-hour date count)

This function adds `count` hours to date `date` .

```
? (date-add-hour (now) 67)
= ::DATE:year-second:(2011-02-08 18:55:30)
```

5.14.13 (date-add-day date count)

This function adds `count` days to date `date` .

```
? (date-add-day (now) 67)
= ::DATE:year-second:(2011-04-13 23:55:30)
```

5.14.14 (date-add-month date count [opt])

This function adds `count` months to date `date` .

This is tricky because months have different lengths. Adding an integral number of month to certain dates may produce an illegal day number for the target month. (eg. `(date-add-month "1996-05-31" 1)`). The behavior of this function is therefore further specified by the value of optional argument `opt` .

- When `opt` is nil or omitted, this function generates a Lush error whenever the answer would be an illegal date.

- When `opt` is `'no-error` , this function returns the empty list `()` whenever the answer would be an illegal date.
- When `opt` is `'end-of-month` , this function always returns the last day of the target month regardless of the specified day number.
- When `opt` takes any other value (eg. `'loosely` or `t`) this function always returns a legal date. Answers with oversized day numbers are corrected by returning the last day of the target month.

```
? (date-add-month (now) -2 t)
= ::DATE:year-second:(2010-12-05 23:55:30)
```

5.14.15 (date-add-year `date count` [opt])

This function adds `count` months to date `date` .

This is tricky because adding an integral number of years to february 29th dates may produce an illegal date. The behavior of this function is therefore further specified by the value of optional argument `opt` .

- When `opt` is nil or omitted, this function generates a Lush error whenever the answer would be an illegal date.
- When `opt` is `'no-error` , this function returns the empty list `()` whenever the answer would be an illegal date.
- When `opt` takes any other value (eg. `'loosely` or `t`) this function always returns a legal date. The day number is changed to 28 whenever the answer would be the february 29th during a non leap year.

```
? (date-add-year (now) 2 t)
= ::DATE:year-second:(2013-02-05 23:55:30)
```

5.14.16 (date-code `date flags`)

This function returns a list of numbers encoding a date `date` suitably for a statistical analysis. Argument `flag` is an integer composed by oring (using `bitor` for instance) the following constants controlling which values to generate:

- 1: One value representing the number of days since January 1st, 1970.
- 16: Two values encoding a point moving on the unit circle once a day.
- 8: Two values encoding a point moving on the unit circle once a week.
- 4: Two values encoding a point moving on the unit circle once a month.
- 2: Two values encoding a point moving on the unit circle once a year.

These components are generated only if they make sense with the resolution of the date `date` .

```
? (date-code (now) 24)
= (-0.0196 0.9998 -0.0028 1)


? (date-code (today) 31)
= (15010 NAN NAN -0.7818 0.6235 0.7818 0.6235 0.5667 0.8239)
```

5.15 Loading and Executing Lush Program Files

5.15.1 (load `in` [`out` [`prompt`]])

See: (open-write `s` [`suffix`])

See: (open-read `s` [`suffix`])

See: ^L `filename`

This function is used for loading a Lush source or binary file.

This function starts a new toplevel with the file named `in` as input file, and the file named `out` as the output file. Strings `in` and `out` represent filenames according to the general filename conventions explained with functions `open-read` and `open-write` .

Function `load` searches its input file `in` along the current search path. It also tries various suffixes usually associated with Lush program files (ie. `.lsh` `.lshc` as well as `.sn` `.tl` `.snc` `.tlc` for backward compatibility with TL3 and SN).

Function `load` then enters a loop which reads a Lush expression on the input file, evaluates this expression, and prints the result on the output file. This loop terminates when an end-of-file condition is detected or when function `exit` is called. Function `load` returns the full filename of the input file.

When the output file argument is omitted `out` , function `load` does not display the evaluation results. This is useful for silently loading a Lush program file. This file however can produce an output because it may contain calls to some printing function (eg. `print` .)

The optional string `prompt` indicates the prompts strings displayed by Lush when reading from the standard input. String prompt contains up to three prompt strings separated by the vertical bar character `"|"` . The first prompt string is used when Lush waits for a new expression. The second prompt string is used when Lush waits for the continuation of an already started expression. The last prompt string is used when the execution of the Lush expression reads information from the console (for instance using function `read`).

Example:

```
;; The toplevel loop is implemented as
(load "$stdin" "$stdout" "? |  |> ")
```

5.15.2 ^L filename

See: (load in [out [prompt]])

This macro-character expands into a call to function load . If a filename filename is not provided, the last file name used with macro-character ^L is used. This file name is saved in variable last-loaded-file .

5.15.3 file-being-loaded

See: (load in [out [prompt]])

This variable is set when you load a Lush program file using function load . It contains the full name of the file being loaded. It provides a way to know the full filename of the Lush program file in which this variable is checked.

5.15.4 (libload libname [opt])

See: file-being-loaded

See: (load in [out [prompt]])

See: (reload)

This is the primary function used to load a library or a Lush file into another Lush file or library. A typical Lush program file will begin with a bunch of libload . The difference between load and libload is that libload remembers when a particular file was previously libloaded, and only loads it once. Libload also keeps track of the dependencies between Lush program files (which file libloads which other file). This is the basis of the automatic and transparent on-demand compilation mechanism as described in detail below.

Function libload first searches the file libname . When libload is called from an existing Lush file, it first searches file libname relative to the location of the existing Lush file. If no such file exists, it uses function filepath to locate the file along the lush search path.

Function libload then updates its internal database of dependencies between Lush files. When function libload is called from an existing Lush file, it records the fact that this Lush file depends on the Lush file filename .

Function libload then tests whether it is necessary to load the file. It is necessary to load the file if

- Argument opt is non nil.
- File libname has never been loaded,
- File libname has been modified since the last time it was loaded,
- File libname has not been modified since the last time it was loaded, but depends on a Lush file that has been modified since the last time it was loaded.

When this is the case, libload calls load to load the file, and stores a timestamp for the file in its internal database. Function libload always returns the full name of libname .

5.15.5 (reload)

Load the file that was loaded last using `libload` .

5.15.6 (libload-dependencies [fname])

Function returns the names of all the files on which `fname` depends. This information is collected whenever `libload` is called. Argument `fname` defaults to the currently loaded file.

5.15.7 (libload-add-dependency fname)

This function modifies the dependency database managed by function `libload` . It records that the file currently loaded depends on file `fname` . This function is useful when it is not practical to call `libload` on function `fname` . For instance `fname` might not be a Lush file but a data file processed by the current file.

This function also claims that `fname` has been successfully loaded. Calling `libload` on `fname` will not do anything unless someone modifies file `fname` after calling `libload-add-dependency` .

5.15.8 (find-file dirlist filename extlist)

Finds the first file that exists fo the form `dir` / `file` . `ext` where `dir` is one of the directories in list `dirlist` and `ext` is one of the suffixes in `extlist` .

Example:

```
(find-file '("local" "lsh") "stdenv" '(".lshc" ".lsh"))
```

Note: This is somewhat similar to `filepath` but allows searching through any directory tree.

5.15.9 (save f s1 ... sn)

This function just saves indented definitions of the symbols `s1` to `sn` into the file named `f` . A suffix `".lsh"` is automatically added to `f` if it has no suffix.

5.15.10 (autoload f s1 ... sn)

Defines functions `s1` to `sn` with a code that first loads file `f` , and then restart the evaluation.

The first time one of these functions is called, its definition is loaded from file `f` , and the execution continues. Such autoloading functions often are defined by `"stdenv.lsh"` . They avoid loading many files while allowing the user to call those functions at any time.

5.16 Lisp-Style Input/Output

5.16.1 Input

(read)

See: The Lush Reader.

Calls the Lisp reader, and returns the next Lush object available on the current input. Function `read` is able to read both text and binary Lush files.

(read-string [spec])

Reads a string on the current input.

- When argument `spec` is a number, it indicates how many character should be read from the current input and returned as a string.
- When argument `spec` is a string, it specifies which characters are to be read. For example, `"0-9"` matches any number, `" 0-9"` matches anything but a number, `" \t\n\r\f\e"` anything but a blank character or end of file, `"yYnN"` reads only one of these characters.

The default `spec` for function `read-string` is " \n\r\f\e" and means "read anything until the end of line".

(skip-char [spec])

Skips the characters matching string `spec` and returns the next available character. The default value of argument `spec` is `"\t\n\r\f"` and means "skip any blank characters".

Example:

This function reads the input stream and builds a list with each line, until it reaches an end-of-file character

```
(de read-lines()
 (let ((ans ()))
  (while (<> (skip-char "\\n\\r\\f") "\\e")
   (setq ans (nconc1 ans (read-string))) ) ) )
```

This function reads the input stream and builds a list with each word, until it reaches an end-of-file character

```
(de read-words()
 (let ((ans ()))
  (while (<> (skip-char) "\\e")
    (setq ans (nconc1 ans (read-string "~ \\t\\n\\r\\f\\e"))) ) ) )
```

(ask `s`**)**

Prints the string `s` on the standard output and waits for a `"yes"` or `"no"` answer on the standard input. Function `ask` returns `t` or `()` according to the user's answer. Function `ask` is not affected by the current input or output file.

Example:

```
? (ask "Your answer")
Your answer [y/n] ?
```

5.16.2 Output

(prin `l1` **...** `ln`**)**

Prints the Lush objects `l1` to `ln` separated by spaces. Unlike `print` , the function `prin` does not terminates the line with a linefeed. Function `prin` returns the result of the last evaluation.

(print `l1` **...** `ln`**)**

Prints the Lush objects `l1` to `ln` separated by spaces and output a newline character. Function `print` returns the result of the last evaluation.

Example:

```
? (print "hello" (+ 3 4) '(a b c d))
"hello" 7 (a b c d)
= (a b c d)
```

(pprin `l1` **...** `ln`**)**

Prints the Lush objects `l1` to `ln` with a nice indentation. The indentation is defined according many heuristics suitable for printing nicely function definitions. Unlike function `pprint` , `pprin` does not terminates the line with a linefeed.

(pprint `l1` **...** `ln`**)**

Prints the evaluations of `l1` to `ln` with a nice indentation. The indentation is defined according many heuristics suitable for printing nicely function definitions. Function `pprint` terminates the line with a linefeed.

Although the current implementation of `pretty` is a more complicated, a simple implementation of `pretty` would be :

```
(df pretty(f) (pprint (funcdef f)))
```

(printf `format ... args ...` **)**

This function is similar to the `printf` function in C language. String `format` will be printed verbatim, except for the following escape sequences, which refer to `args` .

- `"%%"` is replaced by a single \%.
- `"%l"` is replaced by a representation of a lisp object.
- `"%[-][n]s"` is replaced by a string, right justified in a field of length `n` if `n` is specified. When the optional minus sign is present, the string is left justified.
- `"%[-][n]d"` is replaced by an integer, right justified in a field of `n` characters, if `n` is specified. When the optional minus sign is present, the string is left justified.
- `"%[-][n[.m]]c"` where `c` is one of the characters `e` , `f` or `g` , is replaced by a floating point number in a `n` character field, with `m` digits after the decimal point. `e` specifies a format with an exponent, `f` specifies a format without an exponent, and `g` uses whichever format is more compact. When the optional minus sign is present, the string is left justified.

Function `printf` is the only way to print the real contents of a character string, without adding surrounding double quotes or escaping the special characters.

Example:

```
? (printf "\\%5s\\%3d is equal to \\%6.3f\\n" "sqrt" 2 (sqrt 2))
 sqrt  2 is equal to  1.414
= ()
```

(tab [n]**)**

When a numeric argument `n` is given, `tab` outputs blanks until the cursor reaches the `n` -th column on the current line. The cursor position is unchanged if the cursor is located after the `n` -th column. Function `tab` always returns the current cursor position.

```
? (tab)
= 0
```

5.16.3 File Names

(basename `path` [`suffix`]**)**

Returns the terminal component of the pathname `path` by removing any prefix ending with `/` . If string `suffix` is provided, a possible suffix `suffix` is removed from the pathname.

This function makes no reference to the actual contents of the filesystem. It just chops the filename according to the file naming conventions.

(dirname `path`**)**

Returns the name of the directory containing the file named `path` .

This function makes no reference to the actual contents of the filesystem. It just chops the filename according to the file naming conventions.

(concat-fname [`filename1`] `filename2`**)**

Returns an absolute filename equivalent to `filename2` parsed from current directory `filename1` . Argument `filename1` defaults to the current directory.

This function makes no reference to the actual contents of the filesystem. It just concatenates the filenames according to the file naming conventions. Nevertheless, if the resulting filename refers to an actual directory, a directory separator `/` is appended to the returned filename.

Examples:

```
? (concat-fname "..")
= "/home/gemi/Projects/fedora/packages/lush/f14"


? (concat-fname (getenv "HOME") "myfile.lsh")
= "/home/gemi/myfile.lsh"
```

(relative-fname `dirname` `filename`**)**

If `filename` represents a file located below directory `dirname` , this function returns a string representing the relative file name of `filename` with respect to directory `dirname` . Function `concat-fname` can then be reused to reconstruct the absolute file name. This function returns `()` otherwise.

Examples:

```
? (relative-fname "/home" "/home/lush/zoo")
= "lush/zoo"


? (relative-fname "/usr/share" "/home/lush/zoo")
= ()
```

The following idiom can be used to relocate a filename `x` from directory `fromdir` to directory `todir` .

```
(concat-fname <todir>
   (or (relative-fname <fromdir> <x>) <x>) )
```

(tmpname [[`dir`] `suffix`]**)**

This function creates a temporary file and returns its name. When Lush exits, an automatic cleanup code removes these files.

The optional string `dir` specifies the directory where the temporary file will be created. A value of `()` is interpreted as the default system directory for temporary files.

This function generates file names using the optional suffix `suffix` . If you do not specify this argument, the generated filename will have no suffix.

Examples:

```
(tmpname)           ;; creates a temporary file.
(tmpname () "sn")   ;; creates a temporary file whose name ends with ".lsh".
(tmpname ".")       ;; creates a temporary file in the current directory.
```

5.16.4 Searched Path

See: Lush Startup.

When Lush is looking for a file, it searches for it among a list of directories stored in variable `|*PATH|` . Several functions are provided to handle this feature.

(path `s1`**...**`sn`**)**

The function `path` allows you to get and set this list of directories searched by Lush. Without arguments, function `path` returns the current path. Otherwise, the strings `s1` to `sn` become the current path.

The initial path is automatically set while Lush is starting up by retrieving the Lush executable file and scanning up the directories until it finds a suitable library directory.

```
? (progn
    (printf "--> The current path is:\\n")
    (path) )
--> The current path is:
= ("." "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/local"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/packages"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/contrib"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/lsh/libogre"
```

```
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/lsh/compiler"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/lsh/libidx"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/lsh/libstd"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/lsh"
    "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/sys" )
```

(addpath s)

See: (path `s1` ... `sn`)

The function `addpath` adds the directory `s` to the file search path. Directory `s` is added at the head of the search path. Other occurrences of this directory in the path are removed, in order to keep the search path small and non redundant.

Example:

```
? (addpath (concat (getenv "HOME") "/mysnlib"))
```

(filepath `filename` [`suffixes`])

See: (path `s1` ... `sn`)

See: (concat-fname [`filename1`] `filename2`)

See: (load `in` [`out` [`prompt`]])

The function `filepath` searches file `filename` along the current search path, using the suffixes specified by argument `suffixes` . Function `filepath` returns the absolute path name of the file. The empty list is returned if no file matches the argument along the current search path.

Argument `suffixes` is a string containing a sequence of possible suffixes (including the initial period) separated by vertical bars "—". The sequence represents the priority order. A suffix may have zero characters.

```
".dump"      Try suffix <.dump>.
".lshc|.lsh|"  Try suffixes <.lshc> and <.lsh>, then try without suffix.
"|.lsh"       Try without suffix, then try suffix <.lsh>.
".lsh|"       Try suffix <sn>, then try without suffix.
```

Argument `suffixes` may specify a single suffix without the initial period. This is a shorthand for testing first the specified suffix and then testing without a suffix.

```
"sn"         Equivalent to ".lsh|"
```

When argument `suffixes` is omitted, `filepath` uses the same suffixes than function `load` . This default provides a way to determine which file would be loaded by a given call to function `load` .

```
? (filepath "sysenv")
= "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/sys/sysenv.lsh"
```

5.16.5 Files and Directories Management

(files [`directory`])

Returns the list of the files located in directory `directory` . If argument `directory` is ommitted, the current directory is assummed.

(fileinfo `filename`)

Returns an a-list containing lots of information about the file named `filename` in the form of an association list. Information includes type, dates, permissions, size,.... This function returns nil if the named file does not exist.

```
? (fileinfo ".")
= ((type . dir) (size . 4096) (mode . 16877) (dev . 64770) (ino . 3935079)
    (nlink . 17) (uid . 500) (gid . 500)
    (atime . ::DATE:year-second:(2011-02-05 23:53:55))
    (mtime . ::DATE:year-second:(2011-02-05 23:53:18))
    (ctime . ::DATE:year-second:(2011-02-05 23:53:18)) )
```

(dirp `path`)

Returns `t` if string `path` is the filename of a directory. This function also returns `t` if string `path` is the filename of a symbolic link referring to a directory.

```
? (dirp "..")
= t
```

(filep `path`)

Returns `t` if string `path` is the filename of a regular file (not a directory). This function also returns `t` if string `path` is the filename of a symbolic link referring to a regular file.

```
? (filep "..")
= ()
```

(lockfile `path`**)**

Attemps to create file `path` .

- If this file already exists, function `lockfile` returns the empty list `()` .
- If this file does not exists already, function `lockfile` writes the current date into the file and returns `t` .

This function is useful for implementing a simple locking scheme.

(mkdir `dirname`**)**

This function creates a new directory `dirname` .
See: (mkdir-on-need [`s`])
See: (chdir [`s`])

(unlink `path`**)**

This function removes the file or directory `path` . It does not complain if the file or directory `path` does not exist.

(rename `path1` `path2`**)**

This function renames file `path1` as `path2` .

(copyfile `file1` `file2`**)**

Copy file `file1` onto file `file2` . Arguments `file1` and `file2` may be file descriptors or file names (as described in `open-read` and `open-write` .

5.16.6 Files and Descriptors

(script [s]**)**

Creates a file named `s` and records all the input/output operations on that file. The suffix `".script"` is added to the name `s` . This function is handy for keeping a trace of the Lush session. If function `script` is called without arguments, the current script file is closed.

(open-read `s` [`suffixes`]**)**

See: (sys `shellcmd`)
See: (filepath `filename` [`suffixes`])
This function returns a read file descriptor of the file named `s` . The empty list is returned when an error occurs while opening the file. The file descriptor is closed when the garbage collector detects that the file is no longer necessary. File descriptors however can be manually closed with the function `delete` .
Filenames have the following form:

- A legal operating system filename will be searched along the current path.
- `"$stdin"` stands for the Lush standard input.
- On some systems, `"| shell command"` may be used to open a pipe to another process. See the description of function `sys` to have comments about the implementation of this feature.

When string `suffixes` is specified, the suffixes specified by this string are tried while searching the file along the search path. The possible values of the string `suffix` are documented with function `filepath` .

(open-write `s` [`suffix`]**)**

See: (sys `shellcmd`)

This function returns a write file descriptor of the file named `s` . The empty list is returned when an error occurs while opening the file. The file descriptor is closed when the garbage collector detects that the file is no longer necessary. File descriptors however can be manually closed with the `delete` function.

Filenames have the following form:

- A legal operating system filename.
- `"$stdout"` stands for the Lush standard output stream.
- `"$stderr"` stands for the Lush standard error stream.
- On some systems, `"| shell command"` may be used to open a pipe to another process. See the description of function `sys` to have comments about the implementation of this feature.

  An optional suffix can be specified as argument `suffix` . This suffix is appended to the filename unless the filename already contains a suffix.

(open-append `s` [`suffix`]**)**

This function returns a "append" file descriptor of the file named `s` . All output on this file descriptor shall be appended to the file. The empty list is returned when an error occurs while opening the file. The file descriptor is closed when the garbage collector detects that the file is no longer necessary. File descriptors however can be manually closed with the `delete` function.

An optional suffix can be specified as argument `suffix` . This suffix is appended to the filename unless the filename already contains a suffix.

(writing `f` `l1` **...** `ln`**)**

See: (open-write `s` [`suffix`])

See: (open-append `s` [`suffix`])

Calls `progn` on `l1` to `ln` , while redirecting the current output to `f` . Argument `f` may be a filename or a file descriptor obtained by `open-write` or `open-append` .

(reading `f l1... ln`**)**

See: (open-read `s` [`suffix`])

Calls `progn` on `l1` to `ln` , while redirecting the current input to `f` . Argument `f` may be a filename or a read file descriptor obtained by `open-read` .

Example: This function returns the list of the files in directory s

```
(de directory(s)
       (let ((ans ()))
               (reading (concat "|/bin/ls " s)
                       (while (<> (skip-char) "\\e")
                               (setq ans (nconc1 ans (read-string))) ) )
               ans ) )
```

(reading-string `str l1... ln`**)**

Calls `progn` on `l1` to `ln` , while redirecting the current input to read the contents of string `str` .

```
? (reading-string "(1 2 3)" (read))
= (1 2 3)
```

(read8 `f`**)**

Reads a byte on file descriptor `f` . This function returns a number in the range 0 to 255.

(write8 `f b`**)**

Writes a byte `b` on file descriptor `f` . Argument `b` must be a number in the range 0 to 255. This function does not flush the file `f` . The function `flush` must be used in order to write out all remaining data.

(fsize `f`**)**

Returns the number of bytes remaining in the file. When the file is not exhaustable or not rewindable (such as pipes and terminals) it causes an error.

(flush [`f`]**)**

Flushes the file descriptor `f` .

When no arguments are provided, both the current input and current output are flushed: All characters pending on the current input are removed until a newline is encountered, all characters buffered on the current output are printed out.

Example:

```
;;; This function ask a question and returns a string.
(de input(question)
       (printf "\\%s" question)
       (flush)
       (read-string) )
```

5.16.7 Binary Input/Output

(bwrite `l1` ... `ln`)

See: (bread)

Function `bwrite` writes the lisp objects `l1` to `ln` on the current output stream as a sequence of binary tokens. This function is able to write correctly lisp objects with circular references. It cannot however write lisp objects useful for their side effects like open file descriptors or windows.

(bwrite-exact `l1` ... `ln`)

See: (bwrite `l1` ... `ln`)

See: (bread)

Function `bwrite-exact` is similar to `bwrite` with a few twists that makes it more accurate but less frequently useful.

- Function `bwrite` writes objects by first specifying the class name. This implies the class must be defined and have the same name when reading the object. Function `bwrite-exact` instead writes a complete definition of the class. This implies that `bread` will create a new unnamed class regardless of the existing classes.

- Function `bwrite` writes indexes by saving their content regardless of how the index is laid out in the corresponding storages. This implies that `bread` will create a new storage for each of them. Function `bwrite-exact` instead writes the storage and the index layout. But this is wasteful if the storage is much larger than the index.

(bread)

See: (read)

Function `bread` reads on the current input stream a binary sequence of tokens and builds the corresponding lisp object. Functions `read` and `load` are also able to recognize binary tokens in the middle of a text file. This allows for reading a file containing a mixture of lisp objects representing as binary sequence of tokens or as text.

(bread-exact)

See: (bread)

Function `bread-exact` is similar to `bwrite` with one twists that makes it more accurate but less frequently useful.

- Saving an instance of a user defined class with `bwrite` saves only the slots of the instance and a reference to the class name. Function `bread` will be able to read this binary file if the class has already been defined in the interpreter. This class must define the same slots with the same order as the class of the actual object stored in the file. Additional slots will just be skipped and left with their default value. Function `bread-exact` instead causes an error when there are additional slots.

Note that both `bread` and `bread-exact` can read files produced by either `bwrite` or `bwrite-exact` . There is no difference between `bread` and `bread-exact` when reading a file produced by `bwrite-exact` .

(tokenize `fin` `fout`)

Function `tokenize` reads all lisp expressions from file `fin` and writes their binary equivalent into file `fout` .

This function is useful for converting Lush program files (SN files) into preparsed binary file (SNC files). Loading a preparsed file is much faster than loading a text file. Preparsed copies of certain system libraries are located in directory `"lushdir/lib"` .

(dump `fname`)

See: Lush Startup.

Writes the status of the lisp interpreter into file `fname` and return size of file in bytes. When `fname` has no suffix, ".dump" is used.

Dump files are useful for lauching Lush with preloaded libraries and applications. During startup Lush can load a binary dump file specified by the command line arguments. If Lush does not find the system library `"stdenv.lsh"` , it attempts to undump a file names `"stdenv.dump"` .

5.16.8 Pipes and Sockets

(filteropen `cmd`)

Launch another program with control over its stdin and stdout streams.

This function runs the shell command `cmd` in the background and returns a list of length three, including a write file descriptor, read file descriptor and the process id of the child process. Writing to the write file descriptor causes data to be send to the processes' stdin, and the read file descriptor reads from the processes' stdout.

Example:

(socketopen `host port fin fout`**)**

This command is available under Unix.

This function connects a server process listening on port number `port` on the remote computer named `host` . It stores into symbol `fin` a file descriptor for reading from the server, and stores into symbol `fout` a file descriptor for writing to the server.

Arguments `fin` and `fout` are evaluated and must be quoted symbols.

This function causes an error if the server is not listening for connections. Make the port number negative to prevent this. Function `socketopen` will then return `()` if the port cannot be opened.

(socketaccept `port` [`fin fout`]**)**

This command is available under Unix.

This commands creates a server side socket listening on port number `port` . It then waits for a client connection (for instance with `socketopen`). Then it stores file descriptors for reading from and writing to the client into symbols `fin` and `fout` respectively. Arguments `fin` and `fout` are evaluated and must be quoted symbols.

When arguments `fin` and `fout` are not provided, it returns a boolean indicating whether it was possible to wait for connections on the specified port.

(socketselect [**...**`filedescriptors`**...**] [`timeout`]**)**

This command waits until data becomes available on the specified file descriptors. The optional argument `timeout` is a number specifying how many milliseconds the function should wait for data. Providing a timeout of zero simply checks for the availability of data. Providing no timeout potentially waits forever.

Remote Lush Execution

Lush provides facilities for controlling multiple Lush processes running on different machines.

The script `lushslave` starts Lush in slave mode. It first prints the host name and a port number and then waits for a connection. A master instance of Lush can use class `RemoteLush` to establish a connection, send commands, obtain the results, and check for errors.

(new RemoteLush `host` [`port`]**)**

Creates a remote lush connection to the slave instance of Lush running on the specified `host` and `port` . The default port is 4000. Commands can then be submitted using method `exec` .

Slot `status` indicates the outcome of the last command execution. Value `ok` indicates that the command was successful. Value `error` indicates that an error was detected. The error message is then available in slot `error` .

Slot `fin` can be used in command `socketselect` in order to determine whether the command output is available, and possibly wait for several remote processes.

(==> `remotelush` exec `command`)

Executes command `command` on the remote Lush process. The lisp object `command` is evaluated in the remote Lush process. Method `exec` returns the result of this evaluation.

When the remote evaluation causes an error, method `exec` sets the slot `status` to `error` and returns the symbol `error` . The error message is then available in slot `error` .

This method works by calling methods `send` and `receive` .

(==> `remotelush` send `command`)

Sends command `command` to the remote Lush process and returns immediatly. Slot `status` is set to the empty list until method `receive` is called.

(==> `remotelush` receive [`nowait`])

Receive the result of the remote execution of a command.

Method `receive` returns the result of the evaluation of a remote command submitted with method `send` . When the remote evaluation causes an error, method `exec` sets the slot `status` to `error` and returns the symbol `error` . The error message is then available in slot `error` .

Method `receive` usually waits for the execution of the remote command. This can be changed by setting argument `nowait` to true. Method `receive` then immediately returns the empty list if the result is not yet available. In this case, slot `status` remains set to the empty list.

(==> `remotelush` print)

Prints informative stuff.

5.17 Miscellaneous

5.17.1 Debug Toplevel

The debug toplevel is a simple facility for debugging programs. It allows recording function and method invocations in a log file ("tracing"), interrupting execution at invocations of specified functions or methods ("stopping") and step-by-step evaluation of expressions ("stepping").

Currently only non-compiled code may be debugged.

(dbg)

Start `dbg` .

When `dbg` is active, traced function and method invocations are logged in a file. Errors, the invocation of specified functions and methods, or `CTRL-C` trigger stepping mode.

(tracef [`fname` **...**]**)**

Start tracing function `fname` . With no arguments, list all functions currently being traced.

Example:

```
(tracef >= isnan)
```

(untracef `fname` **...)**

Stop tracing function `fname` .

(tracem [`mname` **...**]**)**

Start tracing method `mname` . With no arguments, list all methods currently being traced.

(untracem [`mname` **...**]**)**

Stop tracing method `mname` .

(stopf [`fname` **...**]**)**

Start interrupting calls to function `fname` . With no arguments, list all functions whose invocations are currently being interrupted.

Example:

```
(stopf >= isnan)
```

(unstopf `fname` **...)**

Stop interrupting function `fname` .

(stopm [`fname` **...**]**)**

Start interrupting method invocations to `fname` . With no arguments, list all methods whose invcations are currently being interrupted.

(unstopm `fname` **...)**

Stop interrupting method invocations to `fname` .

5.17.2 Error handling functions

(error [symb] string [l])

Causes an error whose text is given by `symb` , `string` and `l` . When `symb` is provided, `error` rewinds the evaluation stack until it reaches a call to a function named `symb` . The error message then displays the stack relative to that function call. When `symb` is omitted, the stack is not displayed.

Example:

```
? (error 'myfunc "bad number" 4)
*** myfunc : Bad number : 4
Debug toplevel [y/n] ?
```

(errorf format arg1 .. argn)

Raise an error message, where `format` is an sprintf-like format string followed by any number of arguments.

(assert test [error-form])

Raise an error when `test` evaluates to (), return () otherwise. The optional `error-form` must evaluate to a string.

(pause string)

print `string` and start a toplevel prompt. Inserting a `(pause string)` expression in a function allows to stop the execution and get a prompt in the context of the `pause` . This allows to examine and/or modify variable values etc.

(debug l1 ... ln)

See: (trace-hook `level line expr info`)

Evaluates `l1` to `ln` , in debug mode. When in debug mode, Lush calls function `trace-hook` before and after each evaluation. The default `trace-hook` function just prints an indented trace of each evaluator call, as well as each value.

Example:

```
?  (debug (* (+ 3 4) a)))
-> (* (+ 3 4) a)
 -> (+ 3 4)
  -> 3
  <- 3
  -> 4
  <- 4
 <- 7
 -> a
```

```
 <- 8
<- 56
= 56
```

(nodebug `l1` **...** `ln`**)**

Calls function `progn` on expression `l1` to `ln` without displaying a trace, even if this function is called inside a call to function `debug` . Functions like `pretty` and `save` use this function to produce clean output in debug mode.

(btrace [`n`]**)**

See: Interruptions in Lush.

See: Errors in Lush.

Print current evaluation stack. When `n` is given and non-negative, print only top `n` expressions on the stack. When `n` is negative, don't print but return the whole stack.

(on-error `p l1` **...** `ln`**)**

See: Errors in Lush.

This function evaluates lists `l1` to `ln` and returns the last result like function `progn` . If however an errors occur during these evaluations, the expression `p` is evaluated before the usual error processing.

The function `on-error` does not stop the error processing, but allows for performing cleanup tasks, or for printing a custom error message.

(on-error-macro `pm l1` **...** `ln`**)**

See: Errors in Lush.

See: (on-error `p l1` ... `ln`)

This function first evaluates list `pm` and stores the result which is usually an expression to be evaluated if an error occurs. It evaluates then lists `l1` to `ln` and returns the last result like function `progn` . If however an errors occur during these evaluations, the result of the initial evaluation of `pm` is evaluated before the usual error processing.

Actually, this function works like `on-error` , except the expression which will be executed in case of error is the result of the evaluation of `pm` in the call context of `on-error-macro` i.e. before `l1` ... `ln` are evaluated.

This allows a safer behaviour than `on-error` , since the evaluation of `pm` is made in the call context instead of being made in the error context. Indeed, the error context may have values pushed on the stacks by functions called in `l1` ... `ln` , therefore hiding the expected values.

```
(let* ((my-window (new windowobject 100 100 200 100 "Handler"
                       (new string "processing..."))))
```

```
      (errq (new errorrequester my-window)))
 (on-error-macro
  `(==> ,errq popup (errname))
  (let* ((errq ()))
    (error "this is is a programmed error\\nused as test."))))
```

(on-break `p l1 ... ln`**)**

See: Interruptions in Lush.

This function evaluates lists `l1` to `ln` and returns the last result like function `progn` . If however the user interrupts the evaluation by depressing `Ctrl-C` , the expression `p` is evaluated before the usual interruption processing.

The function `on-break` does not stop the interruption processing, but allows for performing cleanup tasks, or for printing a custom error message.

(on-break-macro `pm l1 ... ln`**)**

See: Errors in Lush.

See: on-break, on-error-macro

Same as `on-error-macro` for user breaks.

(on-interrupt `p l1 ... ln`**)**

See: on-error, on-break

Evaluate lists `l1` to `ln` and return result of `ln` . If an error or user interrupt occurs, evaluate expresssion `p` before starting the usual interrupt processing.

(on-interrupt-macro `p l1 ... ln`**)**

See: on-interrupt, on-error-macro

Same as `on-error-macro` but works for both errors and user breaks.

(errname)

This function returns the last error message as a string. This function is usually called by the `break-hook` or `debug-hook` functions, whenever an error or an interruption occurs.

5.17.3 Memory Management

The Lush interpreter is implemented atop a memory manager for C programs. For all Lush objects there is an "associated memory type" known by the memory manager. When a new Lush object is to be created, Lush requests memory of the associated memory type for the new object. It is sometimes interesting to investigate the allocation behavior of performace critical sections of code. The `memprof` -form may help with this.

(gc)

Trigger an instantaneous garbage collection and return the number of objects reclaimed.

(meminfo [`level`])

Print information about current memory consumption to console.

```
? (meminfo)

Small object heap: 256.00 MByte in 65536 blocks (4104 used)
Managed memory   : 9.26 MByte in 523069 + 1115 objects
Memory used by MM: 7.65 MByte total
= ()
```

(memprof `l1` **...** `ln`**)**

See: memprof/p

Evaluate forms `l1` ... `ln` and count all allocations during those evaluations. After evaluation of `ln` , update global variable `*memprof-stats*` with the allocation counts. Memprof may not be called from within compiled code.

```
? (memprof (concat "a" "b"))
= "ab"


? (*memprof-stats* "blob")
= 0
```

(memprof/p `l1` **...** `ln`**)**

See: memprof

Same as `memprof` , but also print profile data to console after evaluation.

```
? (memprof/p (concat "a" "b"))

*** memprof of (concat "a" "b"):
              at:         1
           blob8:         1
= "ab"
```

(with-nogc `l1 ... ln`**)**

See: memprof

Evaluate forms `l1 ... ln` without doing garbage collections.

Use this form with caution. Since memory reclamation is temporarily suspended, the memory required to evaluate `l1 ... ln` must be limited or the interpeter will crash. The form is useful for timing or debugging code.

5.17.4 Querying the runtime environment

(print-symbols `str`**)**

See: ^S `str` , symbols

Prints all the symbols whose name contains the string `str` . This is especially useful if you don't remember the exact spelling of a function or variable.

^S `str`

See: (symbols `str`)

Macro-character for function `symbols`

(sizeof `c_type`**)**

Returns the number of bytes needed to implement a C item of C type named `c_type` . Supported types are signed numerical types and pointer type `"gptr"` .

```
? (sizeof "double")
= 8


? (sizeof "gptr")
= 8


? (sizeof "short")
= 2
```

This function is mainly used for formated I/O.

(used)

Returns the number of currently used Lush cells. This function is especially useful to track underlocks and overlocks when debugging a new Lush primitive written in C.

result

The last result produced by the toplevel is always stored in the variable result.
Example:

```
? (sqrt 81)
= 9
? (* result result)
= 81
```

version

This symbol contains a string, whose contents is the full name of the Lush based program you are running.

lushdir

This variable contains the filename of the root of the Lush tree.

(exit [n])

See: Lush Startup.

This functions exits the current toplevel, as if a `Ctrl-D` had been typed. When the optional argument `n` is given, function `exit` quits the Lush process with return code `n` .

(discard `l1` **...** `ln`**)**

Evaluates `l1` to `ln` like `progn` , but prevents the entire result to be printed by the toplevel. It will print its first line instead. This function is useful for working quietly on very long lists.

(startup `...argv...`**)**

See: Lush Startup.

Lush calls this function once during the startup procedure as soon as the system library `"stdenv.lsh"` or `"stdenv.dump"` has been loaded. The behavior of the default `startup` procedure is explained in section "Lush Startup.".

The arguments `argv` are the command line arguments. Neither the executable name nor the magic first command line argument are passed to `startup` .

(toplevel)

See: Lush Startup.

This function usually is defined by `"sysenv.lsh"` and is called after function `startup` during the startup process. This function is restarted whenever an error occurs.

(debug-hook)

See: Errors in Lush.

This function is defined by `"sysenv.lsh"` and is called whenever an error occurs. Use care when modifying this function, since an incorrect `debug-hook` function may result into a deadly infinite loop!.

(break-hook)

See: Interruptions in Lush.

These functions is defined by `"sysenv.lsh"` and is called whenever the user causes an interruption by depressing `Ctrl-C` .

(trace-hook `level line expr info`**)**

See: (debug `l1` ... `ln`)

This function is called before and after each evaluation when the Lush kernel runs in debug mode (see function `debug`). Redefining this function allows interactive Lush debugger to execute programs step by step.

- When `trace-hook` is called before an evaluation, argument `level` is a positive number indicating the number of recursive calls to the evaluator since Lush entered the debug mode. Argument `line` is a string displaying the first line of the expression being evaluated. Argument `expr` is the expression being evaluated. Argument `info` is the list of the expressions whose recursive evaluation led to the evaluation of `expr` .

  If function `trace-hook` returns `()` , the evaluation will continue without calling `trace-hook` for recursive evaluations. The next call to `trace-hook` will occur after the current evaluation returns. If function `trace-hook` returns `t` , the recursive calls to the evaluator (calls performed while evaluating the current expression) will be traced.

- When `trace-hook` is called after an evaluation, argument `level` is a negative number representing the opposite of the number of recursive calls to the evaluator since Lush entered the debug mode. Argument `line` is a string representing the first line of the returned value. Argument `expr` is the expression being evaluated. Argument `info` is the value returned by the evaluation.

  If function `trace-hook` returns `()` , the remaining evaluations will continue without calling `trace-hook` until an evaluation with a smaller recursion level returns. If function `trace-hook` returns `t` , the next evaluations will be traced normally.

5.17.5 System

A few functions are strongly related to the operating system. The quality of the implementation however depends on the quality of the underlying operating system.

Some functions are not available under all operating systems:

- The function `bground` is only available on Unix system.
- The function `xdmc` is only available on Apollo Domain/OS computers.
- The functions `winlushp` , `winsys` and `winedit` are only available under Windows/95 and Windows/NT.

(beep)

Function `beep` is self explanatory.

(sleep `n`)

Function `sleep` waits `n` seconds and returns.

(wait `obj`)

This function (defined in `libogre/ogre.lsh`) is an event processing loop. It keep processing events until its argument becomes nil (presumably as the result of processing an event). This function's primary use is to allow Lush scripts that run Ogre GUI applications to run until their main window is closed. Here is an example:

```
#!/bin/sh
exec lush "$0" "$@"
!#
(ogre)
(wait (new autowindowobject 10 10 100 100 "Simple Lush GUI Demo"
          (new stdbutton "    hit me    " (lambda (c) (printf "OUCH\\n")))))
```

(sys `shellcmd`)

See: (sysf `format arg1 .. argn`)
See: #$ `shellcmd`
See: (open-write `s` [`suffix`])
See: (open-read `s` [`suffix`])
Execute shell command and return process exit code.
Example (Unix):

```
? (sys "pwd")
/home/toto/bip
= 0
```

The implementation of function `sys` and of the pipe convention in filenames (see `open-read` and `open-write`) is highly system dependent.

- This implementation is extremely reliable on Unix systems. Using pipes and forking background processes has been the tradition of Unix since the origins.

  Function `sys` will always wait until the spawned process exits and return the process exit code.

- This implementation is much less reliable under Windows 95 or NT. You can indeed run very different kinds of programs, including legacy MS-DOS and WIN16 programs. A number of programs running under Windows have been written with the assumption that they would run using a MSDOS style console. Running them as background processes or with standard handles redirected to pipes will cause problems.

  If you are running the Console based version of Lush, function `sys` replicates as well as possible the behavior of the Unix version. The spawned process inherits the Lush console and may be controlled by the user. If you are running a MSDOS or WIN32 Console application, function `sys` returns when the application exits. If you are running a Windows 3.1 or WIN32 GUI application, function `sys` returns immediately.

  If you are running the GUI based version of Lush, the spawned processed no longer inherit a MSDOS console. We have chosen to handle MSDOS and WIN32 Console application like a GUI application. Function `sys` will create a new console for the process and return immediately regardless of the application type.

  Using pipes is reasonably reliable under the Console based version of Lush. It becomes very adventurous under the GUI version of Lush because we do not want to create a console and let the user interact with the program. Bugs in Windows 95 make pipes so dangerous that we just forbid them.

  Please investigate function `winsys` to start a new process with a collection of refined options (redirecting standard handles, using the command interpreter, detaching the process, waiting until the process exits). You should nevertheless remember that these options do not work welle with all application types.

(sysf `format arg1 .. argn`**)**

See: (sys `cmd`)

Execute shell command and return process exit code. `format` is an sprintf-like format string followed by any number of arguments.

#$ `shellcmd`

See: (sys `shellcmd`)

Macro-character for function `sys` .

```
? #$pwd
/home/toto/bip
= 0
```

(chdir [s])

See: (path `s1 ... sn`)

See: (sys `shellcmd`)

If the string `s` is provided, this function sets the current working directory to directory `s` . This function returns the current working directory name.

The current directory specified with `chdir` is used by all shell commands launched with function `sys` . When accessing a file, Lush also searches the current directory before the directories specified by function `path` .

Example:

```
? (chdir)
= "/home/gemi/Projects/fedora/packages/lush/f14/lush-2.0"
```

(getpid)

Returns the process ID of this process.

Example:

```
? (getpid)
= 24887
```

(getuid)

Returns the user ID of this process.

Example:

```
? (getuid)
= 500
```

(getusername)

Returns the user name for this process.

Example:

```
? (getusername)
= "gemi"
```

(edit `file` [`createp`]**)**

See: ^E `file`

Calls the standard editor on file `file` . If the file does not exist in the current directory, it is searched for in Lush's directory structure (using `libload.search`). If no file was found, `edit` raises an error unless the optional flag `createp` is true. The editor name is stored in the symbol `edit-call` , and is initialized by looking at the environment variable `EDITOR` or `VISUAL` .

^E `file`

See: (edit `file`)

Macro-character for function `edit` .

edit-call

This variable contains the name of the editor command.

(fedit `func`**)**

See: ^F `func`

Copies an indented definition version of `func` in a temporary file and calls the function `edit` on this file. When the editor returns, the edited definition of `func` is loaded.

^F `func`

See: (fedit `func`)

Macro-character for function `fedit` .

(getconf `varname`**)**

Obtains the value of variables determined by the shell script `configure` . String argument `varname` can take the following values:

- `"SHELL"` The shell interpretor, e.g `"/bin/sh"` .
- `"OPTS"` The optimization options for the compilers, e.g. `"-DNO_DEBUG -O2"` .
- `"DEFS"` The symbol definition options for the compilers, e.g. `"-DHAVE_CONFIG_H"`
- `"LIBS"` Libraries used for linking the main executable, e.g. `"-lbfd -liberty -ldl -lm"` .
- `"host"` String describing the host computer, e.g. `"i686-pc-linux-gnu"` .
- `"CPP"` Command for running the C preprocessor, e.g. `"gcc -E"` .

- "CPPFLAGS" The preprocessing options for the compilers.
- "CC" Command for running the C compiler, e.g. "gcc" .
- "CFLAGS" The C specific compiler options besides those specified in CPPFLAGS , OPTS and DEFS .
- "CXX" Command for running the C++ compiler, e.g. "g++" .
- "CXXFLAGS" The C++ specific compiler options besides those specified in CPPFLAGS , OPTS and DEFS .
- "F77" Command for running the Fortran compiler, e.g. "g++" .
- "FFLAGS" The Fortran specific compiler options besides those specified in CPPFLAGS , OPTS and DEFS .
- "LDCC" Command for linking executables, e.g. "g++" .
- "LDFLAGS" Options for linking executables.
- "PTHREAD_FLAGS" Flags for compiling or linking programs that use the Posix thread library.
- "PTHREAD_LIBS" Libraries for linking programs that use the Posix thread library.
- "AR" Command for creating static libraries, e.g. "/usr/bin/ar" .
- "MV" Command for renaming and moving files, e.g. "/bin/mv" .
- "CP" Command for copying files, e.g. "/bin/cp" .
- "LN_S" Command for creating symbolic links, e.g. "ln -s" .
- "TOUCH" Command for updating the modification time of a file, e.g. "/bin/touch" .
- "INDENT" Command for running the GNU indent program.
- "RANLIB" Command for creating the table of contents of a static library created with AR , e.g. "ranlib" .
- "X_LIBS" Options for linking executables using the X11 library, e.g. "-L/usr/X11R6/lib -lSM -lICE -lX11" .
- "X_CFLAGS" Options for compiling programs using the X11 library, e.g. "-I/usr/X11R6/include" .
- "CC_PIC_FLAG" Compiler flag for producing position independent code appropriate for creating shared libraries, e.g. "-fPIC" .

- `"CC_EXP_FLAG"` Linker flag for linking executables that export their symbol tables to resolve undefined symbols in shared libraries, e.g. `"-Wl,-export-dynamic"` .
- `"MAKESO"` Command for creating a shared library, e.g. `"gcc -shared -o"` .
- `"EXEEXT"` Filename extension for executable programs, e.g. `""` or `"exe"` .
- `"OBJEXT"` Filename extension for object files, e.g. `"o"` or `"obj"` .
- `"SOEXT"` Filename extension for shared libraries, e.g. `"so"` or `"sl"` or `"dll"` .
- `"LUSH_MAJOR"` The major version of Lush specified in loadable modules.
- `"LUSH_MINOR"` The minor version of Lush specified in loadable modules.
- `"LUSH_DATE"` The date lush was compiled.

(getenv s)

Returns the value of the environment variable whose name is `s` .

Example:

```
? (getenv "HOME")
= "/home/gemi"
```

(getconf s)

Returns the value of the autoconf variable determined at compilation time. These variables indicate how the lisp interpreter was compiled. See file `"include/autoconf.h.in"` for a list of variables.

Example:

```
? (getconf "CC")
= "gcc"
```

```
? (getconf "CFLAGS")
= " -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buf
```

(time)

Returns real time spent (in seconds) since a system dependent date.

Note: The deprecated form

```
(time <l1...ln>)
```

is equivalent to function `cputime` but prints a warning message.

(cputime `l1....ln`)

Returns the CPU time (in seconds) scheduled by the operating system for evaluation expressions `l1` to `ln` .

Example:

```
? (cputime (repeat 40000
             (sqrt 2) ))
= 0
```

On uniprocessor systems, this is alway smaller than the real time since the processor might be also used for other tasks.

On multiprocessor systems, this function adds the times spent by each CPU. If the evaluation of `l1` ... `ln` involves several threads, the returned value might be greater than the real time.

(realtime `l1....ln`)

Returns the real time (in seconds) elapsed during the evaluation of expressions `l1` to `ln` .

Example:

```
? (realtime (repeat 40000
              (sqrt 2) ))
= 0.003
```

(ctime)

This function is identical to the Unix `ctime` function. It returns date and time as a 26 character string.

Example:

```
? (ctime)
= "Sat Feb  5 23:55:31 2011"
```

(localtime)

This function is identical to the Unix `localtime` function. It returns a list of the following form:

```
( tm_isdst <day light saving time>
  tm_yday  <day of the year (0..365)>
  tm_wdat  <day of the week (sunday=0)>
  tm_mon   <month number (0..11)>
  tm_mday  <day of the month (1..31)>
  tm_hour  <hour (0..23)>
  tm_min   <minutes (0..59)>
  tm_sec   <seconds (0..59)> )
```

Example:

```
? (localtime)
= (tm-isdst 0 tm-yday 35 tm-wday 6 tm-year 111 tm-mon 1 tm-mday 5 tm-hour 23
    tm-min 55 tm-sec 31 )
```

(isatty `filename`**)**

Returns `t` if file `filename` is an interactive terminal.

(bground `s l1 ... ln`**)**

Creates a background process which evaluates expressions `l1` to `ln` and exits. The output of this process is redirected to a file named `s` . A suffix `".job"` is concatenated to filename `s` when necessary.

(lush-is-quiet [`flag`]**)**

See: Lush Startup.

This function returns `t` if the Lush interpreter runs in script mode and `()` if the Lush interpreter runs in interactive mode. This function is useful to print less information when running in script mode.

Lush starts in script mode when arguments are passed on the command line. The initial banner is suppressed. The lush file named as first argument is loaded. Lush terminates as soon as it either reaches the end of file or encounters an error.

Lush starts in interactive mode when no arguments are passed on the command line. It displays a startup banner and generally prints more verbose messages. Then it enters the interactive toplevel loop (read-eval-print) and restarts the toplevel loop whenever an error occurs.

Switching mode changes the verbosity of the various messages. Switching to interactive mode also causes Lush to enter the interactive toplevel loop when

it would otherwise exit (i.e. when reaching an end-of-file or encountering an error).

(winlushp)

This function is only available under Windows. It returns `()` if you are running the console version of Tlisp. It returns `t` if you are running the GUI version of Tlisp.

Examples:

```
;; You can test if you are running under windows as follows.
(cond
     ((not winlushp)
            (printf "Not running under Windows\\n") )
     ((winlushp)
            (printf "Running WinLush (GUI version of Lush under WIN32)\\n") )
     (t
            (printf "Running Lush (Console version of Lush under WIN32)\\n") ) )
```

(winedit `filename` [`as-untitled`]**)**

This function is only available when you are running the GUI version of Lush for Windows (WinLush).

Function `winedit` opens a text editor window for the file named `filename` . If the name `filename` ends with suffix `".lsh"` , WinLush will allow syntax coloring and automatic indentation.

The optional flag `as-untitled` tells WinLush to open the file as an untitled window.

- If flag `as-untitled` is the empty list, WinLush first searches an already open text editor window for this file. If such a window already exists, WinLush just pops up the window. Otherwise, WinLush creates a new window and loads the file. This mode is convenient for editing well defined files.

- If flag `as-untitled` is not nil, WinLush creates a new untitled window and transfers the contents of the file. The new editor window will not remember the file name. This mode is convenient when you are using a temporary text file for transfering some text into an editor.

- If you do not specify argument `as-untitled` , WinLush makes a decision on the basis of the file location. If the file is located under the system temporary directory, WinLush creates an untitled editor window. Otherwise, WinLush creates a regular text editor window.

(winsys `stdin stdout stderr commandp detachp waitp cmdline`**)**

See: (sys `shellcmd`)

This function is available when you are running Lush under Windows 95 or NT.

Function `winsys` creates a process for the program specified by command line `cmdline` . The other arguments allow you to tune finely the process parameters.

The legal and reliable combinations depend on:

- whether you are running the console based (Lush) or GUI based (WinLush) version of Lush,
- whether you are running Windows 95 or Windows NT,
- whether you are executing a MSDOS application, WIN16 application, WIN32 Console based application, WIN32 GUI application, OS2 2.1 application (under NT) or Posix application (under NT).

Arguments `stdin` , `stdout` and `stderr` allow you to define the standard input, output and error handles of the program. These argulents may be `()` , `t` or a variable name.

- Value `()` represents an invalid handle. Therefore the process will not be able to use the corresponding standard handle. This may cause Windows to create a MSDOS console as a replacement.
- Value `t` represents the handle inherited from the Lush process. Under WinLush, this value redirects the standard output or standard error to the WinLush console.
- A variable name tells Lush to pass one end of the pipe to the process. The other end of the pipe will stored into the specified variable as a Lush file descriptor. If you specify the same variable name for the standard output and error, Lush will use a single pipe for both standard handles.

Flag `commandp` is used to determine how the command line is processed. If this flag is set, the command line is passed to the command interpreter. You can thus run the internal commands implemented by CMD.EXE (Windows NT) or COMMAND.COM (Windows 95 - an MSDOS program!!). If this flag is not set, the command line is directly interpreted by WIN32's CreateProcess function.

Flag `detachp` is used to run a process in the background. This flag should prevent Windows to create a console or inherit the console handles. It is rather buggy under Windows 95.

Flags `waitp` tells if function `winsys` must wait until termination of the process and return the process exit code. If this flag is unset, the function returns immeditely.

Examples:

```
;; Here is an alias for function SYS under WinLush
(winsys () () () t () () cmdline)
;; Here is an alias for function SYS under Console Lush
(winsys t t t t () t cmdline)
;; Here is a way to open a read pipe
(winsys () 'readpipe t () t () cmdline)
```

(win-show-workbench `flag`)

See: (wbm-window [[[`x y`] `w h`] `name`])

See: (wbm-toplevel-window [[[`x y`] `w h`] `name`])

This function is available when you are running Lush under Windows 95 or NT. This function controls the visibility of the graphical interface of the GUI version Lush for Windows (WinLush). Calling this function does nothing if you are running the console version of Lush for Windows.

When argument `flag` is the empty list, WinLush switches to "hidden" mode. The main WinLush window (containing the Lush console, the text editor windows, and possibly a few graphical windows) disappears from the computer screen.

WinLush reverts to normal mode when function `win-show-workbench` is called again with a non nil argument. There is also a safety feature that automatically reverts WinLush normal mode when some user input is required on the Lush console.

When WinLush runs in "hidden" mode, function `wbm-window` works like function `wbm-toplevel-window` . Instead of creating a child window of the invisible main WinLush window, `wbm-window` creates a toplevel window which directly appears on the Windows desktop.

It is possible to directly start WinLush in "hidden" mode by specifying option `"-hide"` as the first command line argument. The remaining argument are interpreted as usual. This command line argument allows for creating self contained WinLush applications.

You can for instance create a file `"c:/calctoolapp.lsh"` containing the following three lines:

```
(setq mainwnd (calctool))
(while mainwnd (waitevent) (process-pending-events))
(exit 0)
```

You can then start WinLush with the following command line:

```
winlush -hide stdenv c:/calctoolapp.lsh
```

This command starts WinLush in hidden mode (`"-hide"`). WinLush will first load the standard environment (`"stdenv"`) and then load our example file (`"c:/calctoolapp.lsh"`). This file will create a calculator window on the

Windows desktop, processes graphical events, and terminates execution when the user closes the calculator window.

The user will only see the calculator window. The WinLush main window never appears on the screen because WinLush remains in "hidden" mode as long as there is no need for user input on the Lush console.

5.17.6 Copying Lush Objects

There are many type-specific copy functions, like `copy-list` , `copy-array` , `copy-object` , etc. These typically create a "shallow copy" of the argument. To create a "deep copy", use the generic function `deepcopy` . If objects of user-defined defined classes have an intricate structure, it might be necessary to override the generic deepcopy behavior. This can be done by providing a special method `-deepcopy` .

(deepcopy `obj`**)**

See: -deepcopy, special methods

Create a deep copy of `obj` .

The argument `obj` may be any list object. By default, `deepcopy` recursively copies all components or data members of `obj` , assuming a tree structure.

If the object is an instance of a user-defined class, and the class provides a special method `-deepcopy` , then `(deepcopy obj)` is equivalent to:

```
(let ((clone (new-empty (classof obj))))
  (==> obj -deepcopy clone)
  clone)
```

5.17.7 Special Numerical Values (IEEE754)

This section is valid on computers supporting the IEEE754 floating point representation scheme. Most computers and compilers nowadays support this.

Numerical computations may encounter three kinds of problems:

- Problems arise when we attempt to perform a clearly forbidden operation, such as taking the logarithm of a negative operand. This attempt causes a low level floating exception which generates a Lush error.
- A computation may return a number too large or too small to be represented by the usual floating point data types. Too small number are approximated by zero. Too large numbers are represented using special bit patterns for representing infinities.
- Special bit patterns are used to represent infinities or missing data. The special bit pattern representing missing data is called "NaN" (NaN stands for "Not a Number"). Computations involving NaNs usually return NaNs.

The special bit patterns representing infinities or NaNs are still Lush numbers and `numberp` will always return `t` when applyed on a NaN. However NaNs and Infinities do not behave like regular numbers.

Special bit patterns have no litteral representation for the Lush reader. Most machines however print them as `"Nan"` or `"Inf"` or `"-Inf"` . You should not save these bit patterns into ascii format files because the reader will not be able to read them back. You must save them into binary files.

According to IEEE specifications, testing the equality of two special bit patterns should always return false. Comparing two bit patterns should cause a floating exception. Major operating systems and compilers do not abide with this specification. For instance the Microsoft Visual C++ 4.2 compiler considers that NaNs are equal to anything. It is unclear if this is a bug or a speed compromise (comparisons of actual number are much more frequent).

These problems make us unable to comply fully with the standard. Portable programs can only rely on the following garantees:

- NaNs and Infinities are different from regular numbers. The equality test will return false (we had to hand code this with Visual C++ in the Windoze version of TL3!).
- The Negative Infinity is smaller than any regular number. The Positive Infinity is greater than any regular number.
- Overflows may produce a floating point exception or return an Infinity. This is operating system and compiler dependent.
- Illegal operations (eg. comparing NaNs) are likely to cause a floating exception. They may however return NaNs without causing a floating exception on certain systems.

NaNs and Infinities however can be safely manipulated and tested by the following functions:

(nanp `x`)

True if `x` is a NaN.

(infinityp `x`)

True if `t` is an inifinite number (positive or negative).

(isfinite `x`)

Return a nonzero integer if `x` is neither a NaN nor an infinite value, and zero otherwise.

(isnan `x`)

Return a nonzero integer if `x` is a NaN and zero otherwise.

(isinf x)

Return a nonzero integer if `x` is an infinite value and zero otherwise.

(isnormal x)

Return a nonzero integer if `x` is a normal floating point value and zero otherwise.

(signbit x)

Return a nonzero integer if `x` is negative and zero otherwise.

(eps x)

Distance to next larger magnitude double precision floating point number.

```
? (eps 1)
= 2.22045e-16
```

5.17.8 Floating Point Environment

IEEE754-conforming FPUs can be configured with respect to

- The internal precision for floating point operations (e.g., compute with double precision even though the operands are single precision).
- The rounding mode for floating point operations (e.g., nearest or always towards zero).
- Which floating point exceptions to ignore and for which to raise an error (e.g., error on divide-by-zero vs. returning Inf on divide-by-zero).

The complete state of the FPU is also sometimes referred to as the "Floating Point Environment".

When starting a new toplevel, Lush configures the FPU to compute with `extended` precision, to round `to-nearest` , and to raise errors on `overflow` events. You may change this behavior with the `fpu-` functions listed below. Third-party library code may also reconfigure the FPU. Remember that you may always set the FPU back to its initial state by calling `fpu-reset` .

(fpu-reset)

Set FPU to its initial state.

(fpu-info)

Print current FPU state.

(with-fpu-env `l1..ln`**)**

Evaluate forms in dedicated floating point environment and return last value.

`With-fpu-env` copies the current floating point enviroment, evaluates the forms `l1` .. `ln` as `progn` would do, but resets to the copied environment before return the value of `ln` .

```
? (- (with-fpu-env (fpu-round 'upward) (/ 1 3))
    (with-fpu-env (fpu-round 'downward) (/ 1 3)) )
= 5.55112e-17
```

(fpu-trap `e1..en`**)**

See: fpu-untrap, fpu-clear, fpu-test

Clear status flags of named exceptions and configure FPU to trap the named exceptions.

After evaluating `fpu-trap` , the floating point exceptions corresponding to arguments `e1` .. `en` will trigger an error. Possible argument values are the symbols `invalid` , `div-by-zero` , `overflow` , `underflow` , `inexact` , or `all` for all exceptions.

(fpu-untrap `e1..en`**)**

See: fpu-trap, fpu-test

Ignore named floating point exceptions.

Possible argument values are the symbols `invalid` , `div-by-zero` , `overflow` , `underflow` , `inexact` , or `all` for all exceptions.

After evaluating `fpu-untrap` , the floating point exceptions corresponding to arguments `e1` .. `en` will not trigger an error. However, the FPU still records occurrence of exceptions by setting corresponding exception status flags. Use `fpu-test` to find out what exception status flags are currently set.

(fpu-test `e1..en`**)**

See: fpu-clear, fpu-trap

Return all of the named exceptions status flags currently flagged.

Possible argument values are the symbols `invalid` , `div-by-zero` , `overflow` , `underflow` , `inexact` , or `all` for all exceptions.

(fpu-clear `e1..en`**)**

See: fpu-test

Clear the named exception status flags.

Possible argument values are the symbols `invalid` , `div-by-zero` , `overflow` , `underflow` , `inexact` , or `all` for all exceptions.

(fpu-precision `prec`**)**

Set operating precision of FPU.

Possible arguments values are `single` , `double` and `extended` .

(fpu-round `mode`**)**

Set rounding mode of FPU.

Possible argument values are `toward-zero` , `to-nearest` , `upward` , and `downward` .

5.18 Graphics

Lush features a collection of graphics functions for creating windows, drawing figures, plotting curves, and handling graphical events.

A window descriptor is a lisp object of class `Window` . Most graphical commands implicitely operate on the current window defined by the window descriptor stored in the symbol `window` .

Each window descriptor forwards all drawing requests to a "Graphic Driver" which performs the system dependent calls required to display the drawings on the corresponding device. The most common type of window (and driver) is the `x11-window` , which displays on your screen under Xwindows.

Three drivers are very useful to produce printable graphs. Driver `ps-window` sends all graphic command to an Encapsulated PostScript file. Driver `svg-window` procudes a SVG file that can be edited with programs such as `"inkscape"` or `"sodipodi"` .

5.18.1 Creating a Window

The standard function for creating a window on your screen is `new-window` . This function determines which system your are running and calls the appropriate low level function.

Function `print-window` performs the same task for creating a window descriptor that accesses your printer.

(new-window [[`x y`] `w h`] [`name`]**)**

Create new window object and return it.

See: (new-window! [[[`x y`] `w h`] `name`])
See: (x11-window [[[`x y`] `w h`] `name`])
See: (svg-window [[`w h`] `name` [`units-per-mm`]])
See: (ps-window [[`w h`] `name`])
See: (wbm-window [[[`x y`] `w h`] `name`])
See: window

This is the device independent function for creating a window. It figures out which drivers are available, and call the appropriate function for creating a window.

Function `new-window` function creates a window named `name` . Arguments `w` and `h` specify the size of the window. Arguments `x` and `y` specify the location of the window.

Windows are closed when they are no longer referenced by the lisp interpreter, when they are deleted with the `delete` function, or when the windowing system sends a deletion request.

(new-window! [[`x y`] `w h`] [`name`])

Same as `new-window` but in addition store the new window object in global variable `window` .

(print-window `w h` [`destination`])

See: (ps-window [[`w h`] `name`])

See: (wpr-window [[`w h`] `name`])

See: (wpr-printers)

See: Print Requester.

See: Producing Encapsulated PostScript Files.

See: window

This is the device independent function for printing graphics on the system printer. This function figures out the best way to send graphics on the printer or the file specified by string `destination` . Arguments `w` and `h` specify the size of the printing area. The drawings will be scaled to fit a standard size page.

Function `print-window` actually calls a system dependent function (e.g. `wpr-window` under Windows 95 and NT, `ps-window` under Unix operating systems). Argument `destination` is passed verbatim to these functions. There are therefore two ways to write a portable program. The simplest way consists in omitting argument `destination` . The best way consists in using the `PrintRequester` class provided with the Ogre library.

The window descriptors returned by `print-window` are very similar to the window descriptors returned by `new-window` . They print pages instead of rendering graphics on your screen. A page is output when you call function `cls` or when the window descriptor is closed (because it is no longer referenced by the interpretor or because you called function `delete`).

Note: Calling function `cls` while a clip rectangle is active (see function `clip`) will not output a page. It will simply clears the clipping zone. Before using `cls` , you can use function `clip` to make sure that no clipping rectangle is active.

5.18.2 Drawing

The coordinate system of a window has its origin in the upper left hand corner. Positive x and y coordinates go to the right and downwards.

(cls)

Clears the current graphics window.

(draw-line `x1 y1 x2 y2`**)**

Draws a line from point (`x1` , `y1`) to the point (`x2` , `y2`) in the current window, with the current color.

(draw-rect `x y w h`**)**

Draws a hollow rectangle in the current window, with the current color. Its top-left corner is located at coordinates (`x` , `y`); its width is `w` , and its height is `h` .

(draw-round-rect `x y w h` [`r`]**)**

Draws a hollow rectangle whith rounded corners of radius `r` .

(draw-circle `x y r`**)**

Draws a hollow circle in the current window, with the current color. The center of the circle is located at position (`x` , `y`). The radius of the circle is radius is `r` .

(draw-arc `x y r fromangle toangle`**)**

Draws a segment of a circle in the current window, with the current color. The center of the circle is located at position (`x` , `y`). The radius of the circle is radius is `r` . The segment is delimited by angle `fromangle` and `toangle` which are values between -360 and 360 inclusive. Argument `toangle` must be larger than `fromangle` .

(fill-rect `x y w h`**)**

Fills a rectangle in the current window, with the current color. Its top-left corner is located at coordinates (`x` , `y`); its width is `w` , and its height is `h` .

(fill-round-rect `x y w h` [`r`]**)**

Fills a rectangle whith rounded corners of radius `r` .

(fill-circle `x y r`**)**

Fills a circle in the current window, with the current color. The center of the circle is located at position (`x` , `y`). The radius of the circle is radius is `r` .

(fill-arc `x y r fromangle toangle`**)**

Fills a segment of a circle in the current window, with the current color. The center of the circle is located at position (`x` , `y`). The radius of the circle is radius is `r` . The segment is delimited by angle `fromangle` and `toangle` which are values between -360 and 360 inclusive. Argument `toangle` must be larger than `fromangle` .

(fill-polygon `x1 y1 ... xn yn`**)**

Fills the polygon defined by the points (`x1` , `y1`) to (`xn` , `yn`), in the current window, with the current color.

(rgb-draw-matrix `x y mat` [`sx sy`]**)**

This is the main function to draw an RGB or greyscale image.

This function displays the pixels contained in idx `mat` at location (`x` , `y`) in the current window. The optional arguments `sx` and `sy` specify the horizontal and vertical zoom factors.

The matrix `mat` can be an idx2 or idx3. If it is an idx2, each value is interpreted as a greyscale value where 0 is black and 255 is white. If it is an idx3, the last dimension must be 1, 3, or more. If the last dimension is 1, the values are interpreted as greayscale values. If the last dimension is 3 or more, the first 3 values in the last dimension are interpreted as RGB intensities (between 0 and 255).

(draw-value `x y val maxv maxs`**)**

This function displays a real value as a white or black square.

Arguments `x` and `y` are the coordinates of the center of the square. Argument `val` is the value to be displayed. Argument `maxv` is the maximum absolute value for `val` . Argument `maxs` is the maximum size of the square. When `val` is larger than `maxv` , a square of size `maxs` is drawn.

Note: The square will not be visible if it is drawn in the same color as the background.

(draw-list `x y l ncol maxv apart maxs`**)**

High level drawing function to graphically represent a list of real numbers. A list of real numbers `l` will be represented as a series of black or white squares on a grey background (actual colors depend on the implementation).

Arguments `x` and `y` are the coordinates of the top left edge of the background area. The squares are arranged in an array with `ncol` columns, the

number of lines is then defined by the length of the list `l` . The grey background is always rectangular, even if `ncol` is not a divisor of the length of `l` .

As with `draw-value` , argument `maxv` is the maximum absolute value for the list elements. The size of the square is however bounded by `maxs` .

Argument `apart` defines the size of the space occupied by a single square, i.e, the centers of two neighboring squares will be apart pixels apart. It is generally suitable to choose a value for `apart` slightly greater than maxs so that no squares will overlap. For best results, the difference between `apart` and `maxs` should be an even number.

Example:

```
; draw 6 values on 3 columns and 2 lines, in 50 pixel squares
(draw-list 50 50 '( 4 5 -10
                    0 4  16 ) 3 16 52 50)
; draw the values in l on a single line
(draw-list 100 100 l (length l) 10 20 18)
```

(gray-draw-list `x y l ncol minv maxv apart`**)**

High level drawing function to graphically represent a list of real numbers. A list of real numbers `l` will be represented as a series of gray squares whose gray level is related to the represented value. This function uses a clever dithering algorithm on black&white displays.

Arguments `x` and `y` are the coordinates of the topleft edge of the first square (representing the first element in the list). The squares are arranged in an array with `ncol` columns, the number of lines is defined by the length of the list `l` .

Values between `minv` and `maxv` will be displayed as gray levels. A value of `minv` will be rendered as a black square. A value of `maxv` will be rendered as a white square. Argument `minv` can be defined to be greater than `maxv` in order to produce a reverse video effect. The sizes of the squares are defined by the `apart` parameter.

(gray-draw-matrix `x y mat minv maxv apartx aparty`**)**

High level drawing function to graphically represent a 2D matrix `mat` as a series of gray rectangles whose gray level is related to the represented value. This function uses a clever dithering algorithm on black and white displays.

Arguments `x` and `y` are the coordinates of the topleft edge of the first square (representing the first element in the list). The values between `minv` and `maxv` will be displayed as gray levels. A value `minv` will be rendered as a black rectangle. A value `maxv` will be rendered as a white rectangle. Value `minv` can be defined to be greater than `maxv` in order to produce a reverse video effect. The sizes of the rectangles are defined by the `apartx` and `aparty` parameter.

(mat-disp `m` [`x y`]**)**

See: gray-draw-matrix, new-window!

Display matrix contents in current window and return the window object.

This is an easy to use wrapper for `gray-draw-matrix` . It automatically re-scales the matrix values to use the full dynamic range of the display. When there is no current window, `mat-disp` creates one.

(color-draw-list `x y l ncol minv maxv apart cmap`**)**

See: (alloccolor `r g b`)

This function is essentially similar to `gray-draw-list` . Argument `cmap` however must be a one-dimensional matrix of 64 color numbers returned by `alloccolor` . Values between `minv` and `maxv` are rendered using the ramp of colors specified by this matrix.

(color-draw-matrix `x y mat minv maxv apartx aparty cmap`**)**

See: (alloccolor `r g b`)

This function is essentially similar to `gray-draw-matrix` . Argument `cmap` however must be a one-dimensional integer matrix of 64 color numbers returned by `alloccolor` . Values between `minv` and `maxv` are rendered using the ramp of colors specified by this matrix.

5.18.3 Grabbing Images

(rgb-grab-matrix `x y mat`**)**

See: (rgb-draw-matrix `x y mat` [`sx sy`])

This function grabs a rectangular image from the current window. The top left corner of the image is located at coordinates (`x` , `y`). The size of the image is determined by the first two dimensions of matrix `mat` .

The matrix `mat` can be an idx2 or idx3 as with `rgb-draw-matrix` .

This function only works on true color displays.

(save-window-as-ppm `filename`**)**

See: (rgb-grab-matrix `x y mat`)

This function uses `rgb-grab-matrix` to save a snapshot of the current window into the PPM file `filename` .

5.18.4 Drawing with Colors

Color selection is highly dependent on the capabilities of your graphics hardware and drivers.

- There are still computers equipped with black and white screens. You can check that your hardware can display enough colors using function `colorp` .
- Many graphic devices use color palettes. The color palette defines a small set of colors that can be displayed simultaneously on the screen. Lush will allocate palette entries as soon as you start using a new color. Since palette entries are rare resources, you should use colors cautiously.

Lush provides two elementary functions for dealing with colors. Function `alloccolor` takes a set of three numbers (ranging from 0 to 1) representing the red, green and blue component of the color. This function allocates a palette entry if necessary and returns a hardware dependent color number. These color numbers can be used with function `color` to set the current drawing color.

It happens sometimes that it is not possible to allocate a color because the color palette is full. Lush will first try to create its own color palette and install it whenever you activate a Lush window. Windows created by other programs can display funny colors until you activate one of these windows. If the private color palette is full, function `alloccolor` simply returns a default color.

The use of a limited number of colors is therefore highly encouraged. The function `color-stdmap` provides several convenient sets of standard colors for this purpose.

See: Ogre Color Palette.

(colorp)

The function `colorp` is a predicate which returns `t` when the display is known to support color. This function returns `()` when the display does not support color.

(color [c]**)**

See: (alloccolor `r g b`)

If `c` is specified, sets the current color to color number `c` . Function `color` always returns the current color. Color numbers are obtained with function `alloccolor` . A few predefined color numbers however are defined by the file `"graphenv.lsh"` :

- `color-fg` or -1 for the system foreground color.
- `color-bg` or -2 for the system background color.
- `color-gray` or -3 for a 50 percent dithered gray level.

(alloccolor `r g b`**)**

See: (color [`c`])

See: distinct-colors

Ask the driver for a color number defined by its three primitive values. Arguments `r` , `g` , `b` are reals between 0 and 1 that define the red, green and blue components of the desired color.

Returns a color number suitable for using with the `color` function. This function returns the constant `color-gray` if the system is not able to display the requested color.

(distinct-colors)

Yield a sequence of color numbers of distinctly looking colors.

Example:

```
? (do ((cn (distinct-colors)))
    (color cn)
    ;; draw something
    )
```

color-fg

See: (color [`c`])

System foreground color.

color-bg

See: (color [`c`])

System background color.

color-gray

See: (color [`c`])

A 50 percent dithered gray level.

(color-shade `x`)

This function calls `alloccolor` then `color` for setting a grayscale color. Argument `x` is a real between 0 and 1. Function `color-shade` provides support for grayscale monitors. In addition, drivers often use dithering for filling circles or rectangles on black and white monitors .

(color-rgb `r` `g` `b`)

This function calls `alloccolor` then `color` for setting the current color according to its three primitive values. Arguments `r` , `g` , `b` are reals between 0 and 1. Of course, this function has a poor effect without a color screen.

(color-stdmap [map]**)**

See: (show-stdmap [map])

Create a set of predefined colors. `Color-stdmap` returns an integer vector containing color identifiers that can be passed to function `color` . In addition, `(color-stdmap 'rainbow)` establishes a global `htable *color-to-colornumber*` , that map color names (symbols) to color numbers.

Using this function maximizes the chances that several parts of your Lush programs will use the same palette entries and therefore avoid palette saturation.

The argument `map` is a symbol which specifies ont of the predefined color sets:

- Symbol `rainbow` defines 64 colors located on a circle linking the pure red, green and blue hues. The first 48 colors actually represent a rainbow. The last 16 colors directly interpolate purple to red.
- Symbol `spread` defines 7 colors that are easy to discriminate.
- Symbol `shade` defines a set of 64 gray levels going from black to white.

Example:

```
? (let ((win (new-window)))
    (color-stdmap 'rainbow)
    (*color-to-colornumber* 'yellow))
= 16772352
```

(show-stdmap [map]**)**

Open a new graphics window and display all colors as defined by `(color-stdmap map)` .

See: (color-stdmap [map])

(color-std (x [map]**)**

The function `color-std` provides access to the convenient sets of colors allocated by `color-stdmap` .

- Argument `x` is number whose decimal part selects a color in the set returned by function `color-stdmap` . Value `0` selects the first color of the set. Value `0.9999` selects the last color of the set.
- Optional argument `map` is a symbol which specifies a color set for function `color-stdmap` .

See: (color-stdmap [map])

(linestyle [ls])

This function allows for drawing dashed lines. Argument `ls` can take the following values:

- `0` for solid lines.
- `1` for dotted lines.
- `2` for dashed lines.
- `3` for dotdashed lines.

Calling this function without arguments returns the last selected line style.

5.18.5 Drawing Text

Text may be printed in graphic window by using the functions `draw-text` and `gprintf` . The current font may be changed by the function `font` . The size and/or rectangle of a graphic text may be scaned by the functions `text-width` , `text-height` and `rect-text` .

(draw-text `x y s`**)**

Draws the text of the string `s` in the current window at position (`x` , `y`), using the current font and the current color.

(gprintf `x y fmt ... args ...`**)**

See: (draw-text `x y s`)

Function `gprintf` behaves like function `printf` , but prints out the text on the current graphic window, with the current font and color, starting at location (`x` , `y`).

(font [fontname])

See: (draw-text `x y s`)

When used without argument, function `font` returns the current font name.

Otherwise function `font` sets the font used for rendering characters in subsequent calls of function `draw-text` . If a matching font is found, `font` returns the name of the selected font. Otherwise it returns `()` and sets a default font.

The default font name "default" is recognized by all drivers and selects a reasonably small default font. Always using font `"default"` is the best guarantee to write fully portable programs.

All drivers however recognize PostScript(tm) style font names. These font names have the following form:

```
"<family>[-<style>][-<size>]"
```

- The recognized font family `family` usually depends heavily on the operating system configuration. It is reasonable to assume that the basic families `"Times"` , `"Helvetica"` and `"Courier"` are available (or translated) everywhere.
- The optional style is preceded by a dash. Style `"Roman"` lets you print plain text. More complex style are composed by concatenating an optional weight specification (e.g. `Light` , `DemiBold` , `Bold` , `Black` , etc...) and an optional slant specification (i.e. `Italic` or `Oblique`).
- The optional size is a number between 1 and 128.

Not all combination of families, styles and size are supported on all systems. A good compromise between fancy display and portable graphics consists in using PostScript font names, but to limit yourself to the following fonts:

```
Courier-X, Courier-Bold-X, Courier-Italic-X, Courier-BoldItalic-X
Times-Roman-X, Times-Bold-X, Times-Italic-X, Times-BoldItalic-X,
Helvetica-X, Helvetica-Bold-X, Helvetica-Oblique-X,
Helvetica-BoldOblique-X, Symbol-X
```

where size `X` is one of `8` , `10` , `11` , `12` , `14` , `18` , `24` .

Most drivers also understand window system dependent font names (when such font names are defined by the window system). For instance, the X11 driver recognizes XFLD font names (such as `"-*-times-*medium-*r-*--18-*"`) and, on some systems, also recognizes fontconfig patterns (such as `":family=Bitstream Vera Sans:pixelsize=11"` . Function `x11-fontname` can be used to translate a Postscript font name into a system specific font name.

(rect-text `x y s`**)**

Returns a list `(x y w h)` describing the rectangle that would be affected by a call to `(draw-text x y s)` .

A few driver, including the PS driver, do not implement this function. In that case, `rect-text` returns the empty list.

(text-width `s`**)**

See: (rect-text `x y s`)

Returns the width of the text in string `s` , if it is printed with the current font. This function uses `rect-text` if it is available, or uses some crude heuristics.

(text-height `s`**)**

Returns the height of the text in string `s` , if it is printed with the current font. This function uses `rect-text` if it is available, or uses some crude heuristics.

5.18.6 The Drawing Context

Several features define the drawing context. The most usefull ones are the current driver, the current font, the current color and the clipping rectangle.

(gsave `l1` ...`ln`)

Function `gsave` saves the graphics state of the current window, evaluates lists `l1` to `ln` with a call to `progn` , and restores the saved graphics state. Function `gsave` returns the result of the last evaluation.

(gdriver)

Returns the name of the graphic driver of the current window.

(xsize)

Returns the width of the current window.

(ysize)

Returns the height of the current window.

window

This variable `window` defines the current window. Most graphics functions refers implicitly to the window descriptor stored in this variable.

(clip [`x y w h`])

When a clip rectangle has been set, graphics output is restricted to the inner part of the clip rectangle. the `clip` function allows for manipulating the clip rectangle.

- `(clip x y w h)` sets a new clip rectangle whose top left corner is located at position (`x` , `y`), whose width is `w` , and whose height is `h` .
- `(clip ())` cancels clipping and unset the clipping rectangle.
- `(clip)` just returns the current clipping rectangle.

Function `(clip)` always returns the previous clip rectangle, as a list `(x y w h)` , or the empty list if no previous clipping rectangle was set.

(addclip `rect`)

Argument `rect` must be a list of the form `(x y w h)` , where `x` , `y` , `w` , and `h` are numbers.

Function `addclip` sets the current clip rectangle to the intersection of the current clipping rectangle with a rectangle whose top left corner is located at position (`x` , `y`), whose width is `w` , and whose height is `h` .

If this intersection is empty, the empty list is returned. Otherwise, `addclip` returns the new clip rectangle.

5.18.7 Double Buffering and Synchronization

(graphics-batch `l1` ... `ln`)

Most graphics drivers can work with an off-screen bitmap to update the screen after several graphics commands. The `graphics-batch` function provides a support for this double buffering abilities.

This function evaluates the lists `l1` to `ln` , but if a graphic instruction is executed, the screen update is delayed until the end of these evaluations, and is performed only once.

(graphics-sync)

Inside a `graphics-batch` construct, this function immediatly updates the screen without waiting for the completion of the `graphics-batch` instruction.

5.18.8 Plotting Functions

See: Plotting Library.

Lush offers three ways to plot curves.

- The old SN plotting functions are still available by loading file `"oldplotenv.lsh"` . We do not recomment using these functions, but we keep them around for compatibility.
- The new SN plotting functions provide a more modern plotting experience while preserving a fair level of backward compatibility with the old SN plotting functions. These function are defined in file `"plotenv.lsh"` and are loaded on demand using `autoload` .
- A completely redesigned library `"libplot/plotter.lsh"` provides a third set of plotting function. This library is documented in the Standard Libraries section.

Plotting Basics

(graph-xy `lx ly ...options...`)

See: (graph-options `...options...`)

Plots a curve composed of points whose X coordinates are specified by list `lx` and Y coordinates by list `ly` . The arguments `...options...` can be any of the options described under function `graph-options` .

This function returns an object of class `PlotContext` representing the axis system used for the plot. This object can then be passed among the `...options...` argument of a subsequent call to `graph-xy` in order to plot several curves inside the same axis.

Example: Plot sine and cosine waves in the same axes.

```
(setq x (range 0 10 .2))
(setq v (graph-xy x (mapcar sin x)
          '(color-rgb 1 0 0) ))
(setq v (graph-xy x (mapcar cos x) v
           '(color-rgb 0 0 1)
           '(linestyle 2) ))
```

The graphs themselves are displayed inside an interactive window of class `PlotWindow` . Menus provide ways to print, export, and change the plot attributes.

(graph-xyv `lx ly lsd ...options...`**)**

See: (graph-xy `lx ly ...options...`)

See: (graph-options `...options...`)

This function is similar to `graph-xy` but accepts a third list containing the sizes of standard deviation bars.

Example:

```
(setq lx (range 0 10 .2))
(graph-xyv lx
          (all ((x lx)) (sin x))
          (all ((x lx)) (abs (* .3 (cos x)))) )
```

(graph-options `...options...`**)**

This function provides for creating an empty plot or changing the appearance of a plot. The optional arguments `...options...` are used to specify which curve or plot context should be modified and which attributes should be modified.

The following arguments can be used to specify which curve or plot context should be modified. Absent such an argument, a new plot is created.

- Pass a `PlotContext` object to select the plot context of interest. Such objects are returned by function `graph-xy` for instance. If no curve is specified, curve attributes will modify the last curve plotted in the specified context.

- Pass a `PlotCurve` object to select the curve of interest. Curves can be obtained in the slot `curves` of the plot context object.

- Pass a port (see next section) to select both the plot context and the curve.

Function `graph-options` always returns the selected plot context. A new plot context is created if none is selected. Therefore, calling this function without arguments simply creates and returns a new plotting context.

The remaining options specify attributes for the selected plot context or plot curve. A warning is displayed for each unrecognized attribute.

The following plot context attributes are recognized:

- `(title string)` to select the plot title.
- `(xtitle string)` and `(ytitle string)` to select the legend of the axes.
- `(margintop x)` , `(marginleft x)` , `(marginbottom x)` , and `(marginright x)` to specify the size of the specified margin.
- `(xlog boolean)` and `(ylog boolean)` to select a logarithmic scale for an axis.
- `(xgrid boolean)` and `(ygrid boolean)` to display a grid for an axis.
- `(xbounds b)` or `(ybounds b)` to specify the bounds of an axis. Argument `b` can be the empty list (for the automatic mode) or a list `(min max)` containing the axis minimum and maximum values.
- `(xlabels b)` , `(ylabels b)` , `(xticks b)` , and `(yticks b)` to specify the position of the axis labels and ticks. Argument `b` can be the empty list (for the automatic mode), a single number (for a specified interval), or a list of values.
- `(xlabel2str f)` and `(ylabel2str f)` to specify how labels should be displayed. Argument `f` can be the empty list or a function taking a value and returning a string.

The following curve attributes are recognized:

- `(color-rgb r g b)` and `(color c)` to specify the curve color. Arguments `r` , `g` , and `b` are numbers in range 0 to 1 representing the color components. Argument `c` is a color returned by function `alloccolor` .
- `(linestyle n)` to specify the line style of the curve. Argument `n` is a small integer suitable for function `linestyle` .
- `(sd-bar-size x)` to specify the size of standard deviation bars. This is used with function `graph-xyv` .
- `(object f)` and `(object-size n)` to specify the kind and size of the small symbols used for representing the curve points. Argument `f` is a function described below. Argument `n` is a small positive integer.

Each curve point is marked with a small symbol defined by a symbol function. A few symbol functions are predefined:

- `object-nil` does not draw anything.
- `open-square` , `closed-square` , `open-circle` , and `closed-circle` respectively draw a small hollow square, a small filled square, a small hollow circle or a small filled circle.
- `open-up-triangle` , `closed-up-triangle` , `open-down-triangle` , and `closed-down-triangle` draw various kind of small triangles.
- `straight-cross` and `oblique-cross` draw small crosses.

Plotting using Ports

Functions `graph-xy` and `graph-xyv` do not provide for plotting curves in the midst of a computation, or for changing curve attributes from one point to the next. Such capabilities are available when one uses "ports".

A plotting port is a data structure which references a plotting context, a curve inside this context, and the last plotted point. This data structure is implemented as a list for maintaining backward compatibility with the old plotting functions.

plot-port

The current plot port is stored in this global variable. Most of the following plotting functions implicitly refer to the plot port stored in variable `plot-port` .

(new-plot-port)

Function `(new-plot-port)` creates a new plotting window and returns a fresh plot port for this window. It does not change the contents of variable `plot-port` .

(copy-plot-port `p` [`object`]**)**

Returns a new plot port that plots inside the same axis system as plot port `p` . This function allows the user to simultaneously plot several curves in the same axes by swapping the current plot port.

The optional argument `object` changes the initial object plotting function for this plot port. Function `graph-options` provides much more control.

(setup-axes)

This function creates a new plot port and makes it current. It is conceptually similar to

```
(setq plot-port (new-plot-port))
```

(plt-color-rgb `r g b`**)**
See: (graph-options `...options...`)
Sets the color for subsequent drawing operations in the current plot port.

(plt-color `c`**)**
See: (graph-options `...options...`)
Sets the color for subsequent drawing operations in the current plot port.

(plt-linestyle `n`**)**

See: (graph-options `...options...`)

Sets the linestyle for subsequent drawing operations in the current plot port.

(plt-object `f`)

See: (graph-options `...options...`)

Sets the symbol drawing function for subsequent drawing operations in the current plot port.

(plt-object-size `f`)

See: (graph-options `...options...`)

Sets the symbol size for subsequent drawing operations in the current plot port.

(plt-sd-bar-size `f`)

See: (graph-options `...options...`)

Sets the standard deviation width for subsequent drawing operations in the current plot port.

(plt-clear)

This function clears the current point in the current plot port. Calling `plt-draw` after this function does not draw a line but simply sets the current point.

(plt-move `x y`)

This function sets the current point in the current plot port. Calling `plt-draw` after this function draws a line starting from the point of real coordinates `x` , `y` .

(plt-draw <x y)

Draws a line in the current plot port starting from the current point to the point of real coordinates `x` and `y` . The current point is then moved to `x` , `y` . When this function is called, the plot context might recompute its axis system and redraw all the curves to account for the new axis mapping.

(plt-plot `x y`)

See: current-object

See: object-size

Moves the current point to the real coordinates `x` and `y` , and draws a small symbol like those used for showing the data points on a curve.

(plt-sd `x y v`)

See: sd-bar-size.

Draws a standard deviation bar of size `v` around point (`x` , `y`).

(plot-lists `lx ly`)

Plots a curve taking using in list `lx` as abscissas and values in `ly` as ordinates in the current plot port. This function uses uses `plt-draw` and `plt-plot` to draw a symbol at each data point.

(plot-lists-sd `lx ly lsd`)

Plots a curve taking values in list `lx` as abscissas and values in `ly` as ordinates. Argument `lsd` is a list of values that will be displayed as uncertainty bars around each data point. This function uses uses `plt-draw` , `plt-sd` and `plt-plot` to draw a symbol and an uncertainity bar at each data point.

Miscellaneous

The functions described in this section exist mostly for backward compatibility with the old plotting package.

(new-plot-port `brect rect object`**)**

DEPRECATED USAGE

Function `new-plot-port` may be called with additional arguments in order to create a plot port that draws inside a specified area of the current window instead of creating an interactive plotting window.

Argument `brect` is a list `(bxmin bymin bxmax bymax)` whose elements are the pixel coordinates of the target rectangle in the current window. Argument `rect` is a list `(xmin ymin xmax ymax)` whose elements define the axis bounds. Argument `object` is the initial symbol drawing function.

(draw-axes `brect xl yl name` [`x2l`] [`y2l`]**)**

DEPRECATED

Creating a plot port with the old plotting library did not automatically draw the axes. Calling function `draw-axes` was mandatory.

The compatible implementation of this function simply changes the plotting port attributes to match the desired axes. Argument `brect` is a list `(bxmin bymin bxmax bymax)` whose elements are the pixel coordinates of the target rectangle in the current window. Arguments `xl` and `yl` are lists of label values for the `x` and `y` axis. These labels are numbers expressed with the same scale than the axes. The optional functions `x2l` and `y2l` are used to convert these numbers into strings to display at the proper positions along the axes. Finally, argument `name` is the title of the plot.

(setup-axes `x1 y1 x2 y2` [`xst yst` [`s` [`object`]]]**)**

DEPRECATED USAGE

Function `setup-axes` may be called with additional arguments in order to create a plot port that draws inside the current window instead of creating an interactive plotting window. The returned plot port is also stored in variable `plot-port` to make it current.

Arguments `x1` and `y1` are the lower bounds of the axis coordinates; arguments `x2` and `y2` are the upper bounds of the axis coordinates. The axes will be labeled every `xst` units horizontally and every `yst` units vertically. Argument `s` is the title string. Argument `object` is the default plot object used by `plt-plot` .

current-object

DEPRECATED, DANGEROUS

The `plt-XXX` functions always check whether this variable contains a suitable symbol drawing function. If it does so, that symbol drawing function is made current in the current plot port as if `plt-object` had been called.

object-size

DEPRECATED, DANGEROUS

The `plt-XXX` functions always check whether this variable contains a suitable size for drawing symbols. If it does so, that size is made current in the current plot port as if `plt-object-size` had been called.

sd-bar-size
DEPRECATED, DANGEROUS
The `plt-XXX` functions always check whether this variable contains a suitable size for drawing standard deviation bars. If it does so, that size is made current in the current plot port as if `plt-sd-bar-size` had been called.

(in-plot-port `l1 ... ln`**)**
DEPRECATED, NOT IMPLEMENTED.
This function was part of the old plotting package and is no longer meaningful. If you need this function, please load the old plotting packate `"oldplotenv.lsh"` .

Plotting Internals

(plot-context-grid)
Defines the default style and color of grid lines.

(plot-context-axes)
Defines the default style and color of grid lines.

(plot-context-title)
Defines the default font and color of the title.

(plot-context-labels)
Defines the default font and color of the axis labels and legends.

(plot-range `lo hi step` [`inside`]**)**
Returns a list of consecutive multiples of `step` that spans the interval `lo` ... `hi` . When flag `inside` is set all multiples are stricly between `lo` and `hi` . Otherwise they range from the largest multiple smaller than `lo` to the smallest multiple higher than `hi` .

(plot-nice-range `lo hi` [`len` [`logscale`]**)**
Returns a sequence of about `len` nice numbers that can be used as labels for representing the interval `lo` ... `hi` . Flag `logscale` suggests using a logarithmic progression instead of a linear one.

FancyChoiceMenu.
Subclass of Ogre's ChoiceMenu where each item is drawn by a callback function.

PlotContext.
This class represents a system of axes, and keeps track all the curves to be displayed inside these axes.

PlotGenericCurve
All curve types are subclasses of this class. Must define two methods: `execute` to plot the curve, and `bounds` to compute its bounding rectangle.

PlotCurve
The only curve type currently implemented. Contains an arbitrary list of plt-xxx commands.

PlotWindow
The ogre class for interactive plotting windows.

PlotDataSetup
A class that implements the title setup page in the plot setup dialog.

PlotAxisSetup
A class that implements the axis attribute dialog page in the plot setup dialog.

PlotCurveSetup
A class that implements the curve attribute dialog page in the plot setup dialog.

PlotSetup
The plot setup non modal dialog.

5.18.9 Events

See: Ogre.

Several functions are provided for handling mouse and keyboard events occurring on graphic windows. Lush has a centralized event queue (see a full description in the Events and Timers section). It is possible to obtain the events occuring on a window by creating an `EventLock` object. Messages `read-event` and `check-event` allow for testing and waiting for events.

This mechanism is used by a few handy functions `get-click` , `get-vector` and `get-rect` , as demonstrated by the incredibly shrinking paint program `lushpaint` below:

```
(de lushpaint()
    (let ((window ()))
      (new-window)
      (draw-line 2 5 18 5)
      (draw-rect 22 2 18 8)
      (fill-rect 42 2 18 9)
      (gprintf 102 10 "Quit")
      (clip 0 12 1000 1000)
      (let ((ok t)
            (mode draw-line)
            (echo get-vector))
       (while ok
          (let (((x y) (get-click)))
            (cond
             ((> y 10)
              (apply mode (echo)) )
             ((point-in-rect x y '(0 0 20 10))
              (setq mode draw-line echo get-vector) )
             ((point-in-rect x y '(20 0 20 10))
              (setq mode draw-rect echo get-rect) )
             ((point-in-rect x y '(40 0 20 10))
              (setq mode fill-rect echo get-rect) )
             ((point-in-rect x y '(100 0 20 10))
              (setq ok ()) )) ) ) )
```

```
(delete window) ) )
```

(NOTE: more serious paint programs should probably be written with Ogre).

Graphic Event Functions

(new EventLock `window`**)**

Creating an `EventLock` object sets up window `window` to record graphic events in the event-queue. Events can be retrieved with messages `read-event` and `check-event` .

The initial state of the window is restored when the `EventLock` object is destroyed.

(==> `EventLock` **read-event)**

See: Event Lists.

See: (eventinfo)

This function returns the first available event on the window associated to the `EventLock` object. Each event is represented by a list whose first element broadly describes the event type. Additional information can be obtained using function `eventinfo` .

If the event queue is empty, this function blocks until an event occurs.

(==> `EventLock` **check-event)**

See: Event Lists.

See: (eventinfo)

This function returns the first available event on the window associated to the `EventLock` object. Each event is represented by a list whose first element broadly describes the event type. Additional information can be obtained using function `eventinfo` .

This function returns the empty list when no events are available.

(hilite `mode x1 y1 x2 y2`**)**

The `hilite` function is used for displaying transient drawings when the mouse button is depressed. As soon as the mouse button is released, these transient drawings are cleared, and the window is refreshed.

Four kinds of drawings are supported, whose names are defined by `"graphenv.lsh"` :

- When `mode` is `hilite-invert` , the rectangle defined by the points (`x1` , `y1`) and (`x2` , `y2`) is inverted. The result however is poor on color displays.
- When `mode` is `hilite-rect` , the rectangle defined by the points (`x1` , `y1`) and (`x2` , `y2`) is outlined with a dashed line.
- When `mode` is `hilite-line` , a dashed line is drawn from point (`x1` , `y1`) to point (`x2` , `y2`).
- When `mode` is `hilite-none` , any transient graphics are cleared, and the window is refreshed.

Transient drawings created with `hilite` are very efficient, and totally disappear when the mouse button is released. This function is useful for providing an echo to the user during a mouse drag operation. For example, function `hilite` can display a rectangular outline while the user is dragging the mouse for selecting a part of an image.

(get-click [`rect`]**)**

Waits for a mouse click on the current window. Returns a list `(x y)` giving the coordinates of the mouse during the mouse click. Mouse clisks will be ignored outside the rectangle specified by the optional argument `rect` .

(get-vector [`rect`]**)**

Waits for a mouse down on the current window. Tracks the mouse moves with a dashed line, until the mouse button is released. Returns a list `(x1 y1 x2 y2)` telling where the mouse button has been depressed and where it has been released.

The optional argument `rect` indicates a bouding rectangle which must entirely contain the resulting vector. The echoed dashed line reflects then this limitation.

(get-rect [`rect`]**)**

Waits for a mouse down on the current window. Tracks the mouse moves with a dashed rectangle, until the mouse button is released. Returns a list `(x y w h)` telling the location of the top left corner of the rectangle, its width and its height.

The optional argument `rect` indicates a bouding rectangle which must entirely contain the resulting rectangle. The echoed dashed rectangle reflects then this limitation.

Event Lists

See: (eventinfo)

Events returned by methods `read-event` and `check-event` are encoded as lists, according to the following templates:

- `(mouse-down x y)` : The mouse button has been depressed at position (`x` , `y`). You may use function `eventinfo` to query the identity of the button and the status of the shift and control keys.
- `(mouse-drag x_1 y_1 x_2 y_2)` : The mouse button has been depressed at position (`x1` , `y1`).The mouse has then been moved to position (`x2` , `y2`). The mouse button still is depressed. You may use function `eventinfo` to query the identity of the button and the status of the shift and control keys.
- `(mouse-up x_1 y_1 x_2 y_2)` : The mouse button has been depressed at position (x_1,y-1). The mouse has then been moved to position (x_2,y_2), where the button has been released. You may use function `eventinfo` to query the identity of the button and the status of the shift and control keys.

- `("c" x y)` : The key associated to the single character string `"c"` has been hit, while the mouse pointer was located at position (`x` , `y`). These events often are referred as keypress events.
- `(arrow-left x y)` , `(arrow-right x y)` , `(arrow-up x y)` or `(arrow-down x y)` : The left (respectively right, up and down) arrow has been hit, while the mouse pointer was located at position (`x` , `y`). You may use function `eventinfo` to query the exact identity of the depressed key and the status of the shift and control keys.
- `(help x y)` : The keyboard dependent help key has been hit, while the mouse pointer was located at position (`x` , `y`). You may use function `eventinfo` to query the exact name of the depressed key and the status of the shift and control keys.
- `(fkey x y)` : A function key has been depressed. You must use function `eventinfo` to get more information about the identity of the function key and of the status of the shift and control keys.
- `(resize w h)` The window has been resized by the user, and is now `w` x `h` pixels large.
- `(delete)` : Windowing systems often provides a mean to delete a window. For example, X11 ICCM compliant window managers are able to send a WM_DELETE message to a client. If an event handler is attached to the window, a delete event is added to the event queue, and the window is not destroyed. A program thus gets a chance to display a confirmation dialog before deleting the window.
- `(sendevent x y)` : These event is generated by function `sendevent` . Integer values `x` and `y` are arbitrary.

These few events give enough information for building graphics interfaces. The class library `"libogre/ogre.lsh"` is designed for that purpose.

(eventinfo)

Returns a list `(name shiftp controlp)` that further describes the latest event returned by function `checkevent` . List elements `shiftp` and `controlp` are flags indicating whether the shift or controlp are depressed.

After a mouse event, string `name` is the mouse button name. Possible names are `"Button1"` , `"Button2"` , etc. for single clicks, and `"Button1-Dbl"` , `"Button2-Dbl"` , etc. for double clicks. After a `keypress` event, string `name` is the name of the key.

5.18.10 System Specific Graphic Functions

Lush provides also system specific functions. The name of these functions always begin with the name of the supported graphic system. For example, "x11" is the prefix for X-Windows specific functions and "win" is the prefix for Windows (95 and NT) specific functions.

Specific Graphic Functions for X11

The following functions are only implemented on Unix systems implementing the X Windows standard. This includes "Motif" and "OpenWindows" based systems.

(x11-window [[[x y] w h] name])

See: (new-window [[x y] w h] [name])

This is the low level function called by `new-window` on X Windows systems.

This function creates a window of size `w` and `h` at position `x` and `y` named `name` on the default X Windows screen. During the first call of function `x11-window` , Lush searches the variable `display` and the environment variable DISPLAY for the name of the X Windows server and display to use.

Unlike `new-window` , this function does not store the newly created window in the variable `window` , and thus does not make this window current.

(x11-fontname s)

See: (font [`fontname`])

This function comes with the X11 driver. It converts a legal PostScript(TM) font name, into a X11 font name.

String `s` must be a legal PostScript font name, composed of a font family, an optional font style, and a font size, possibly separated by dashes, like `"Helvetica-18"` , `"Times-Roman24"` , `"Courier-Bold-12"` . This function returns the corresponding font name under the X11. conventions.

Example:

```
? (when x11-fontname
    (x11-fontname "Times-Roman18") )
= ":family=times:pixelsize=18:roman"
```

(x11-depth)

Returns the depth of the current screen.

(x11-lookup-color `name`)

Lookup the (R, G, B) components of a color named `name` .

(x11-configure [`raise x y w h`])

Changes the window location (`x` , `y`) and size (`w` , `h`). Giving the empty list as argument leaves the corresponding characteristic unchanged. When argument `raise` is true, the window is deiconified and raised.

This function works only with X11R4 or greater. When effective, it returns a list `(visible w h)` whose elements indicate the window visibility and size.

(x11-iconify [`w`])

Iconify window `w` (default is current window).

(x11-text-to-clip `text`)

This function stores string `text` into the X11 copy/paste mechanism. This function will some day recognize the copy/paste protocol used and behave accordingly.

It implements today a cut buffer based mechanism suitable for exchanging text with emacs and xterm. We plan to support selection as soon as possible.

(x11-clip-to-text)

This function retrieves a string from the X11 copy/paste mechanism. It returns the empty list `()` if no string is available. This function will some day recognize the copy/paste protocol used and behave accordingly.

It implements today a cut buffer based mechanism suitable for exchanging text with emacs and xterm. We plan to support selection as soon as possible.

Specific Graphic Functions for Windows

[note: this function is not available in the current version of Lush]

The following functions are only implemented on Windows

(wbm-window [[[x y] w h] name])

See: (new-window [[x y] w h] [name])

See: (wbm-toplevel-window [[[x y] w h] name])

This is the low level function called by `new-window` on Windows 95 and NT systems. This function is available under WinLush only.

Function `wbm-window` creates a window of size `w` and `h` at position `x` and `y` named `name` in the WinLush environment. The command line version of Lush cannot create graphic windows yet.

Unlike `new-window` , this function does not store the newly created window in the variable `window` , and thus does not make this window current.

(wbm-toplevel-window [[[x y] w h] name])

See: (new-window [[x y] w h] [name])

See: (wbm-window [[[x y] w h] name])

This function is very similar to function `wbm-window` described above. Instead of creating a child window of the main WinLush window (according to the Windows MDI scheme), function `wbm-toplevel-window` creates a toplevel window that is not attached to the main WinLush window.

(wpr-window [[w h] name])

See: (print-window w h [destination])

See: (wpr-printers)

This is the low level function called by `print-window` on Windows 95 and NT systems.

Function `wpr-window` creates a window descriptor which actually accesses the printer named `name` . The default printer is used if argument `name` is not specified. Function `wpr-printers` should be used to obtain the list of valid printer names.

Unlike `new-window` or `print-window` , this function does not store the newly created window in the variable `window` , and thus does not make this window current.

Arguments `w` and `h` specify the width and height of the coordinate system. The drawings will be scaled to fit the size of the page returned by your printer.

(wpr-printers)

See: (wpr-window [[w h] name])

This function returns a list of printers names suitable for function `wpr-window` . The default printer name comes first.

(win-depth)
Returns the depth of the default display of your computer. This is the number of bits specified in the Display Properties dialog box.

(win-configure [`raise x y w h`]**)**
Changes the window location (`x` , `y`) and size (`w` , `h`). Giving the empty list as argument leaves the corresponding characteristic unchanged. When argument `raise` is true, the window is deiconified and raised.

This function returns a list `(visible w h)` whose elements indicate the window visibility and size.

(win-text-to-clip `text`**)**
This function stores string `text` into the clipboard.

(win-clip-to-text)
This function retrieves a string from the clipboard. It returns () if the clipboard does not currently contain a string.

Determining System Dependent Feature Support

The function `gdriver-feature` implements a small database about the features supported by the common graphic drivers. We suggest you use this function to implement portable programs.

(gdriver-feature `feat opt`**)**
This function provides a portable mean to scan the current graphic system for a feature. It returns the function implementing the feature specified by symbol `feat` on the given window. The result depends on the parameter `opt` .

- When `opt` is symbol `loose` , this function always returns a function. It returns the feature function as soon as a partial implementation exists. It returns function `progn` if the feature is not implemented.

  This option is usefull for calling minor services (e.g. puting text into the clipboard) for which failure has little consequence.

- When `opt` is symbol `strict` , this function returns a function if and only if the feature is fully implemented. It returns the empty list otherwise.

Features are the following.

- `depth` : This feature is a generalization of `x11-depth` .
- `text-to-clip` : This feature is a generalization of `x11-text-to-clip` .
- `clip-to-text` : This feature is a generalization of `x11-clip-to-text` .
- `configure` : This feature is a generalization of `x11-configure` .
- `hide` This feature is a generalizetion of `x11-iconify` .
- `fontname` : This feature is a generalization of `x11-fontname` .
- `lookup-color` : This feature is a generalization of `x11-lookup-color` .

Here is an example of the use which can be made of `gdriver-feature` .

```
? (new-window 400 400 400 400 "test")
= ::Window:X11:8c440
? ((gdriver-feature configure loose) () 300 300 400 400)
= (t 400 400)
```

5.18.11 Producing Encapsulated PostScript Files

Lush can output graphics commands into an Encapsulated PostScript File (EPS). You can then send these files to a PostScript printer or use adequate software packages to modify these files and integrate them into your documents. There is even a Lush command that tranlates these files into a sequence of Lush drawing commands.

Note: Many computers use incompatible conventions for marking the end of lines (Unix machines use the character NL, Macs use the character CR and PC use a sequence CR NL.) Several popular software exhibit a strange behavior when they attempt to read a sequence of characters which appears to them as a very long line. You must probably use a text filter like `dos2unix` to convert your EPS files.

(ps-window [[`w h`] `name`]**)**

See: (print-window `w h` [`destination`])

Function `ps-window` is the low level function called by `print-window` on systems that provide no other alternative. All versions of Lush however support Encapsulated PostScript files.

Function `ps-window` returns a window descriptor which actually writes PostScript commands into a file named `name` . When argument `name` is omitted, the PostScript commands are written to file `"tloutput.ps"` .

Unlike `new-window` or `print-window` , this function does not store the newly created window in the variable `window` , and thus does not make this window current.

Arguments `w` and `h` specify the width and height of the coordinate system. The target rectangle will be scaled to fit a standard A4 or Letter format page.

(gspecial `string`**)**

This function sends a system specific command represented by string `string` . This function is ignored if the graphics driver does not recognize the string `string` . In the case of a window descriptor created with `ps-window` , the string `string` is inserted verbatim into the Encapsulated PostScript file.

(ps-play `filename` [`function`]**)**

See: (ps-window [[[`x y`] `w h`] `name`])

This function interprets PostScript files created with `ps-window` . Function `ps-play` parses the PostScript file `filename` , converts it into a sequence of SN calls, and applies function `function` to these calls. The default value for the optional argument `function` is `eval` . Function `ps-play` thus plays the contents of the PostScript file in the current window.

Exemples:

To play a EPS file into a window, type:

```
? (let ((window (new-window)))
   (ps-play "your_file.ps") )
```

To see the contents of a EPS file, type:

```
? (ps-play "your_file.ps" pprint)
```

Note: This function only interprets PostScript files created by the Lush PostScript driver. It is by no means a complete PostScript interpreter.

(ps-plot `filename x y w h` [`bb-overrule`]**)**

See: (ps-play `filename` [`function`])

This function plots EPS-files created with SN or Lush in the window referred to by the symbol `window` in the region determined by `x y w` and `h` . `x y` correspond to the coordinates of the region's upper-left corner, `w` to its width and `h` to its height. The plot is scaled to fill exactly this region using the EPS `"%%BoundingBox"` directive which can be overruled by the optional `bb-overrule` parameter. If `w` or `h` are 0 or nil, scaling in their respective directions are choosen as to preserve proportions. If both `w` and `h` are 0 or nil, no scaling is performed.

5.18.12 Producing SVG Graphics

SVG (Scalable Vector Graphics) is a W3C standard for vector graphics. Lush can produce SVG graphics into an SVG file that can be subsequently edited with SVG-compatible drawing tools such as inkscape and sodipodi on Linux.

Producing an SVG file from Lush graphic is easily achieved using the `svg-window` function. The resulting window descriptor works like standard lush graphic windows.

Graphic commands evaluated within a `graphics-batch` generate graphics objects that are grouped together. Bitmap images inserted in the SVG file through `rgb-draw-matrix` are saved in separate PNG files in the same directory as the SVG file. If the SVG file name is "/mydir/myfile.svg", the image files will be "/mydir/myfile-XXXX.png", where XXXX is a 4-digit integer.

The implementation of `svg-window` relies on class `SVGWindow` . An instance of this class can be obtained by applying function `lisp-driver-delegate` to the window descriptor. Class `SVGWindow` defines a number of new methods

for drawing objects. This includes manipulating coordinate transforms, stroke width, stroke styles, separate stroke color and fill color, opacity, etc.

(svg-window [w h] name [rw rh units])

Function svg-window creates a window descriptor that produces an SVG file named name .

Arguments w and h specify the width and height of the drawing rectangle in drawing units (default: 512x512). Optional arguments rw rh and units indicate the physical size of the rectangle into which the drawing rectangle will be mapped (not necessarily preserving the aspect ratio). units is the unit in which this physical size is expressed. The default is "px" (pixels). Possible units are "pt" (=1.25px), "pc" (=15px), "mm" (=3.543307px), "cm" (=35.43307px), and "in" (=90px). By default, the real size is w px by h px. hence the resolution is 90 units per inch.

Unlike new-window or print-window , this function does not store the newly created window in the variable window , and thus does not make this window current.

SVGWindow

The implementation of svg-window relies on class SVGWindow . An instance of this class can be obtained by applying function lisp-driver-delegate to the window descriptor. Class SVGWindow defines a number of new methods for drawing objects. This includes manipulating coordinate transforms, stroke width, stroke styles, separate stroke color and fill color, opacity, etc.

```
(let* ((window (svg-window 500 500 "myfile.svg"))
       (wobject (lisp-driver-delegate window)) )
  (draw-rect 100 100 300 300)
  (==> wobject start-group 120 240 -45 1 1)
  (draw-text 0 -12 "slanted text")
  (==> wobject end-group) )
```

(new SVGWindow svg-file w h [real-width real-height units])

Create a new SVGWindow object whose output is written to the file svg-file . The parameters w and h are size of the window in drawing coordinates (i.e. a object drawn at coordinates (0, 0) will be in the upper left corner, and an object with coordinates (w , h) will be in the lower right corner of the drawing area. Optional parameters real-width and real-height give the physical size of a rectangle within which the drawing are will be mapped (not necessarily preserving the aspect ratio). units is a string with the name of the units in which these sizes are expressed. The default is "px" (pixels). Possible units are "pt" (=1.25px), "pc" (=15px), "mm" (=3.543307px), "cm" (=35.43307px), and

"in" (=90px). By default, the real size is `w` px by `h` px. hence the resolution is 90 units per inch. example:

```
;; make a drawing whose real size is 180x90mm
;; but the drawing coordinate range from 0,0 to 2000,1000
(new SVGWindow "stuff.svg" 2000 1000 180 90 "mm")
```

Drawing Methods Specific to SVGWindow

These methods have no equivalent in traditional Lush graphic drivers.

(==> `SVGWindow` fill-opacity `x`)

Set the opacity of fill operations.

(==> `SVGWindow` stroke-opacity `x`)

Set the opacity of strokes.

(==> `SVGWindow` opacity `x`)

Set the general opacity.

(==> `SVGWindow` stroke-width `x`)

Set the stroke width.

(==> `SVGWindow` stroke-color `x`)

Set the color used for strokes and outlines.

(==> `SVGWindow` fill-color `x`)

Set the color used for filled shapes

(==> `SVGWindow` start-group [`tx ty r sx sy skx sky`])

Start a new group of drawing commands. The optional arguments can be used to specify a local coordinate transform. `tx` and `ty` are the horizontal and vertical translations, `r` a clockwise rotation angle in degrees, and `sx` and `sy` are scaling factors, and `skx` and `sky` are skew angles in degrees along the X and Y axes. The order fo transformations is: translation, skew, rotation, scaling. The group is terminated by a call to the `end-group` method. This command can be handy for drawing rotated objects. For example, drawing a vertical text at coordinate 100 200 can be done with:

```
(==> my-SVGWindow start-group 100 200 -90 1 1)
(draw-text 0 0 "blah blah")
(==> my-SVGWindow end-group)
```

(==> `SVGWindow` end-group)

End a group started with `start-group` , possibly restoring the previous coordinate system.

(==> `SVGWindow` gspecial `s`)

Write string `s` into the SVG file. This allows to write raw SVG directly.

(==> `SVGWindow` font [`s`])

(svg-demo "myfile.svg")

This is a simple demo code of the capabilitied SVG windows. Call this function, then edit myfile.svg with sodipodi or inkscape.

5.18.13 Graphical Utilities

A family of Lush functions are provided to perform elementary computations on rectangles. A rectangle is stored as a list `(x y w h)` . The first two elements `x` and `y` are the coordinates of its top left corner; `w` is its width, `h` its height.

(rect-2-ppbrect `r`)

Returns rectangle `r` under a format suitable for using with `new-plot-port` .

(window-rect)

Returns the boundaries of the current window as a rectangle.

(point-in-rect `x` `y` `r`)

The point-in-rect function returns `t` if point (`x` , `y`) is located inside rectangle `r` .

(rect-in-rect `r1` `r2`)

The `rect-in-rect` function returns `t` if rectangle `r2` encloses rectangle `r1` .

(collide-rect `r1` `r2`)

Returns the intersection of `r1` and `r2` . If rectangle `r1` and `r2` do not intersect, `collide-rect` returns the empty list.

(bounding-rect `r1` `r2`)

Returns the smallest rectangle that encloses both rectangles `r1` and `r2` .

5.19 Organizing multiple windows with tilings

A tiling is a partition of the screen area into rectangular regions, called 'slots'. The `tile` and `tile-at` command place windows on the screen in 'slots', according to the current tiling. Slots are numbered. For a 3x4 tiling, for example, the slots are arranged like this:

```
--------------------
|  1 |  2 |  3 |  4 |
--------------------
```

```
|  5 |  6 |  7 |  8 |
---------------------
|  9 | 10 | 11 | 12 |
---------------------
```

Command `new-tiling` creates a new tiling that covers the screen completely. New `new-tiling*` creates a new tiling window in a tiling, and `make-current` changes the current tiling.

If you find yourself using window tilings frequently, you might want to add these lines to you `lushrc.lsh` file:

```
(libload "tiling")
(import all from tiling-)
```

5.19.1 (new-tiling `n` [`m`])

See: tile, tile-at

Create a new empty tiling and return it.

With one argument create a smallest possible tiling with at least `n` slots. With two arguments create an `n` x `m` tiling. If there is no current tiling, make the new tiling current.

5.19.2 (new-tiling* `w h`)

See: tile, tile-at

Create a new empty tiling for windows of width `w` and height `h` .

5.19.3 (new-tiling* `win`)

See: tile, tile-at

Create a new empty tiling for windows like `win` and return it.

Create the biggest possible tiling for placing windows like `win` on the screen. If there is no current tiling, make the new tiling current.

5.19.4 Namespace tiling-

The following commands are defined in namespace `tiling-` .

(tiling-tile `win` [`tl`])

See: tile-at

Move window `win` to next free slot in tiling `tl` .

(tiling-tile-at [n [tl [win]]])

See: tile

Place window `win` in tiling `tl` at slot `n` and return the window. Hide `win` when `tiling` is not the current tiling and raise it otherwise.

When no window is given (or when `win` is `()`), create a new one. When no tiling is given (or when `tl` is `()`), use the current tiling. When no slot number is given (or when `n` is `()`), use the first free slot in tiling.

`Tile-at` does not change the value of global `window` .

(tiling-tile-at! [n [tl [win]]])

See: tile-at, tile

Similar to `tile-at` , but delete old window when slot `n` is occupied.

(tiling-hide [tl])

Iconify all windows in tiling `tl` and return `()` . When no tiling is given use default tiling.

(tiling-raise [tl])

Place all windows in tiling `tl` on top and return `()` . When no tiling is given use default tiling.

(tiling-close n [tl])

Close window in slot `n` of tiling `tl` and return `()` When no tiling is given use default tiling.

(tiling-close all [tl])

Close all windows in tiling `tl` and return `()` . When no tiling is given use default tiling.

(tiling-make-current tl)

Hide the current tiling, make `tl` current and raise it, return the old current tiling.

(tiling-retile [n m [tl]])

Reshape tiling `tl` to `n` x `m` and replace all windows to compactly fill the tiling. Return the reshaped tiling.

Note: If the number `n` * `m` of slots is reduced, there must remain enough slots for all the windows currently in `tl` . When no tiling is given, use the current tiling.

5.20 Events and Timers

5.20.1 Event Queue

See: Event Lists.

See: Ogre.

The Lush interpreter provides a centralized machanism for queuing and dispatching events. Events are represented by arbitrary non-empty lists. The event lists discussed in section "Event Lists" are merely the events generated by the window system.

Function `sendevent` is the simplest way to generate an event. Besides the event itself, function `sendevent` requires an "event handler", that is to say an arbitrary non null lisp object. The event handler can be merely used as a key to identify the target of the event. Yet it is strongly suggested that the event handler should be a lisp object that recognizes method `handle` .

Besides function `sendevent` , events can be generated by defining a timer using `create-timer` , or by associating an event handler with a graphic window using function `set-event-handler` .

Although events can be polled manually using functions `testevent` , `checkevent` , and `waitevent` , it is often more practical to let Lush dispatch the events automatically. Lush silently polls the event queue whenever it is waiting for input on the console. All available events are dispatched by calling the method `handle` of their event handler with the event itself as a single argument. These automatic call allow the implementation of asynchronous graphic interfaces, like those implemented by the Ogre library.

(set-event-handler `w h`**)**

See: Ogre.

Associates the event handler `h` with window `w` . The new event handler for window `w` replaces the old one. Providing the empty list as argument `h` simply removes the previous event handler without defining a new one. Once an event handler has been attached to a window, graphics events occuring on that window, like mouse or keyboard interaction, are posted on the lush event queue and dispatched as usual.

(create-timer `handler delay` [`interval`]**)**

See: Timers

See: (new Timer [`delay` [`interval`]] [`callback`])

Creates a timer that generates an event `(timer timerid)` for handler `handler` every `interval` milliseconds after an initial delay of `delay` milliseconds. Specifying an interval of zero milliseconds creates a one shot timer that fires only once after `delay` milliseconds. This function returns a timer identifier that can be used with function `kill-timer` . Section "Timers" describe a more convenient way to define a timer.

(create-timer-absolute `handler date`**)**

Creates a timer that generates an event `(timer timerid)` for handler `handler` at the specified date. Dates are real numbers representing a number of seconds spent since a system dependent date.

(kill-timer `timerid`**)**

See: (create-timer `handler delay` [`interval`])

Destroys the timer `timerid` .

(sendevent `handler event`**)**

Posts and event `event` with event handler `handler` . Argument `event` must be a non empty list. Argument `handler` must be a non null object. It is customary (but not mandatory) to make sure that the event handler recognizes method `handle` for processing automatically displatched events.

(sendevent `x y`**)**

This obsolete form of sendevent takes two integer arguments and posts event `(sendevent x y)` to the event handler associated with the current window.

(testevent [`h`]**)**

Function `testevent` returns the first event available for handler `h` without removing the event from the queue. When argument `h` is omitted, function `testevent` implicitely uses the event handler associated with the current window.

Function `testevent` works differently when argument `h` is the empty list. It returns the event handler associated with the first pending event, or the empty list if no events are pending.

When no suitable event is pending, function `testevent` returns the empty list without waiting.

(checkevent [`h`]**)**

Function `checkevent` returns the first event available for handler `h` and removes it from the queue. When argument `h` is omitted, function `checkevent` implicitely uses the event handler associated with the current window.

Function `checkevent` works differently when argument `h` is the empty list. It returns the event handler associated with the first pending event, without modifying the queue.

When no suitable event is pending, function `checkevent` returns the empty list without waiting.

(waitevent)

Function `waitevent` first tests the event queue and returns the event handler associated with the first pending event. Otherwise function `waitevent` waits until an event occurs and returns the associated event handler.

(process-pending-events)

Function `process-pending-events` dispatches all pending events. While there are pending events, function `process-pending-events` removes the first event from the queue, checks whether the associated event handler recognizes method `handle` , and, calls method `handle` of the event handler object with the event as single argument.

Function `process-pending-events` returns when no events are available on the event queue. Function `process-pending-events` is implicitely called whenever events become available while Lush is waiting for user input on the console. Long Lush programs can call this function from time to time in order to maintain event driven graphic interface active during the execution of the program.

(==> `eventhandler` handle `event`)

See: (process-pending-events)

Method `handle` of event handler objects is automatically called when lush dispatches events, either because it is waiting for user input on the console, or because function `process-pending-events` has been called.

5.20.2 Timers

(new Timer [`delay` [`interval`]] [`callback`])

Creates a new timer that fires every `interval` milliseconds after an initial delay of `delay` milliseconds. Specifying an interval of zero milliseconds creates a one shot timer that fires only once after `delay` milliseconds.

A timer event is posted into the lush event queue when the timer fires. This event is dispatched when lush waits for user input or when function `process-pending-events` is called. Dispatching the timer event causes function `callback` to be called with the timer as a single argument.

Example:

```
? (new timer 1000 (lambda(c) (printf "One second\\n"))))))
```

Timers only post events if the previously posted timer events have been dispatched timely. This feature prevents the event queue to grow very large when lush is not able to dispatch the events fast enough.

(==> `timer` **set** `delay` [`interval`])

Sets the timer delay `delay` and periodicity `interval` expressed in milliseconds. Passing the empty list as argument `delay` cancels the timer.

(==> `timer` **setcall** `callback`)

Sets the timer callback function to `callback` . This function is called whenever lush dispatches events generated by this timer.

5.20.3 Pseudo Threads

The pseudo threads facility makes implementing and executing timer-triggered code easy. Since no preemption is happening, pseudo threads are not really threads.

The pattern is as a follows: 1. Implement the code that is to be executed repeatedly as a method of some class. 2. Call `new-thread` with an instance of that class and the name of the method.

The timer-invoked method takes no arguments and returns true or false. A timer will keep invoking the object's method until it returns `()` or until `kill-thread` is called. Example:

```
(defclass Countdown object
  count)

(defmethod Countdown Countdown (n)
  (setq count n) )

(defmethod Countdown count-down ()
  (printf "\\%d\\n" count)
  (decr count)
  (> count 0) )

? (setq thr (new-thread (new Countdown 7) 'count-down 1))
= ::timer:8c3a5a0
? 7
6
5
4
(kill-thread thr)
= ()
?
```

(new-thread `o m` [`r`]**)**

Create a new pseudo thread with object `o` and message `m` and target call rate `r` (calls per second, default is 50), and return thread id.

(kill-thread `id`**)**

Kill thread `id` , return `()` .

5.21 Documentation and Help System

Lush has a convenient documentation system that allows users to:

- browse the Lush documentation using the GUI tool `(helptool)`
- quickly access the documentation of a particular function or topic from the Lush prompt by typing `^Atopic` or by calling `(apropos "topic")` .
- consult HTML, LaTex, PostScript, or DjVu versions the Lush manual

The documentation system allows developers to easily include the code and its documentation in the same file. It also makes it easy to organize sections and entries into a complete manual.

5.21.1 Using the Online Manual

`^A`**topic**

Typing `^A` (caret then A or control-A) followed by a topic at the Lush prompt is the easiest and quickest way to obtain help. This command displays a numbered list of those entries and sections of the manual that contain `topic` as a substring. The user is then asked to pick one of the choices by entering the corresponding number. Entering `q` or `enter` returns to the Lush prompt. If the list has only one match, the corresponding manual entry is shown right away. The first time this command is invoked, it parses and reads the manual. The `^A` macro-character calls the `(apropos "topic")` function described below. Example:

```
? ^Aregex-match

------------------------------------------------------------------------
(regex-match <r> <s>)                          [DX]
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Returns t if regular expression r exactly matches the entire string s .
Returns the empty list otherwise. Example:
? (regex_match "(+|-)?[0-9]+(\\\\.[0-9]*)?" "-56")
= t
```

```
= ()
```

(apropos `topic-string` [`force`]**)**

This command displays a numbered list of those entries and sections of the manual that contain `topic` as a substring. The user is then asked to pick one of the choices by entering the corresponding number. Entering `q` or `enter` returns to the Lush prompt. If the list has only one match, the corresponding manual entry is shown right away. The first time this command is invoked, it parses and reads the manual.

If the optional boolean parameter `force` is non-nil, the Lush manual is re-read.

(helptool [`book` [`title`]]**)**

Author(s): Yann LeCun

Helptool is a GUI-based browser and search engine for the Lush manual. It is invoked by simply typing `(helptool)` at the Lush prompt.

Helptool provides an "explorer-like" GUI to browse through the sections and entries of the documentation. The optional argument `book` can be one of the following:

- if absent or nil, the default documentation book is read from `lsh/lush-manual.hlp` (if necessary) and displayed. This book is read only once in a session.
- if equal to t, the default document book is read (or re-read if it was previsouly read), and displayed.
- if equal to a string, a documentation book is read from the file passed in the string (which can be a .hlp or a .lsh file). The second optional argument is used as the title of the book. If the second argument is not present, the first argument will be used as the title. This is a convenient way of browsing the documentation of a Lush program that is not part of the standard package. It is also a convenient way to use Lush as a documentation browser for other projects.

The Helptool window is composed of an "explorer-like" area on the left that contains a hierarchical list of sections and entries, and a document display area on the right that shows the body of the selected entry. Entries can be opened and closed with a mouse click on the pink square icon. The content of an entry can be displayed by simply clicking on it.

Entering a string or a regular expression in the search tool at the top of the window will "filter" entries that contain that string or match that regular expression. Hitting the `enter` key or the `next` button (or the `space` key in the explorer area) will jump from matching entry to matching entry.

Most users will prefer to use the keyboard shortcuts that provide a very fast and convenient way to navigate the manual. The explorer area, the document

area, and the search tool have different keyboard shortcuts. It should be noted that the various areas must be clicked on to get the keyboard focus. The most important key in the explorer area is the space bar, which allows for continuous sequential reading.

Here are the explorer area keyboard shortcuts:

- `down-arrow` or `n` : move to the next entry (without displaying the document)
- `up-arrow` or `p` : move to the previous entry at the same hierarchical level (without displaying the document)
- `right-arrow` : open the current entry (show the subentries) and display the body.
- `left-arrow` : close the current entry (hide the subentries), or move to the parent entry if the current entry is already closed (or if it has no subentries).
- `space` : This provides a simple way to read to manual sequentially. Hitting space will display the body of the current entry in the document area if it is not already being displayed. If it is already being displayed, the document area will be scrolled down. If the document area is already at the bottom, it will go to the next entry that matches the current search string (or to the next entry if the search string is empty).
- `backspace` : scroll up the document area.
- `enter` : go to the next entry matching the search string (or the next entry if the search string is empty) and display the corresponding body in the document area.
- `c` : close all open entries (collapse the entire tree) and go to the title entry.

Examples of how to invoke Helptool:

- `(helptool)` : normal call.
- `(helptool t)` : force re-reading of lush manual
- `(helptool "sdl/sdl.hlp")` : read all entries in file `sdl.hlp` and and display as a book entitled "sdl/sdl.hlp".
- `(helptool "sdl/libsdl.lsh" "LibSDL")` : read all entries in `libsdl.lsh` and display as a book entitled "LibSDL".
- `(helptool "/home/yann/blah.hlp" "my blah")` : read all entries in `/home/yann/blah.hlp` and display as book entitled "my blah".

At startup, helptool only loads the title lines of each section or entry, but not the bodies. The bodies are read on demand and cached. They are re-read if the file where they reside has been modified since last time it was read.

Known bug: Reading the entire Lush manual takes several seconds.

5.21.2 Writing Documentation

Documentation is either extracted from comments in Lush files (with `".lsh"` extension), or extracted from Lush documentation files (with `".hlp"` extension).

Help entries can contain text, and/or can contain other help entries as subtopics. Entries can be organized hierarchically into books that can be browsed graphically using `(helptool)` . Entries in a file are added to the main documentation by inserting the file name at the desired location in one of the `.hlp` files read by `(helptool)` .

Looking at or at any `.lsh` or `.hlp` file in , , or is a good way to learn how to organize Lush documentation files or the comments of a Lush file in order to provide documentation.

Creating a New Section or Entry

Sections or entries are created in .lsh or .hlp files by a title line that starts with the compound macro-character `#?` . Title lines can create new sections, describe variables, functions, classes, and methods.

Here are a few examples of typical title lines:

- `#? *** All about Choucroute Garnie`

  : Add a Section

- `#? ** << choucroute-garnie.lsh`

  : Include .lsh file

- `#? << choucroute-garnie.hlp`

  : Include .hlp file

- `#? choucroute-garnie`

  : Describe Variable

- `#? (garni-la-choucroute <saucisse> <patates>)`

  : Describe Function

- `#? * choucroute`

  : Describe Class

- `#? * (new choucroute)`

  : Describe Constructor

- `#? (==> <choucroute> garni <saucisse> <patates>)`

  : Describe Method

As shown above, the `#?` of each entry can optionally be followed by one or more asterisks. The number of asterisks specifies the hierarchical level of the entry in the manual relative to the other entries in the same file. Entries with fewer asterisks than a previous entry in the same file will become subtopics of that previous entry:

```
#? *** My Section Title.
my section text goes here
#? ** My Subsection Title 1
my subsection 1 text goes here
#? ** My Subsection Title 2
my subsection 2 text goes here
```

Entries that document functions, classes, methods, and variables, should be written according to the following convention:

- **Function:** Function description entries indicate how to call the function. Arguments are indicated by angle brackets. Here are a few examples:

  ```
  #? (car <list>)
  #? (+ <n_1> ... <n_n>)
  #? (new-window [<w> <h>] <name>)
  ```

- **Variable:** Description of variables should simply include the variable name on the title line. It is recommended that the text include a

  ```
  {<type> VAR}
  ```

  directive.

- **Class:** Class descriptions are usually associated with an entry for its constructor, indicating how to create an instance. Method documentation entries show how to call the method and should be subheadings of the constructor entry:

  ```
  #? * (new helpinfo <record>)
  #? (==> <helpinfo> tty-print)
  ```

A special directive in `.hlp` files indicate that entries at that location should be read from another file. The following example inserts all toplevel entries from file `"toto.hlp"` as a section marked with two stars (the stars of the entries in the file `"toto.hlp"` are only used to place the those entries relative to each other).

```
#? ** << toto.hlp
```

Document Body: Introduction

The body (text) of an entry or section can be written following the title line.

In a `.hlp` file, the body of an entry is whatever text is found until the end-of-file or the next entry title line (i.e. the next `#?` at the beginning of a line). An empty line is interpreted as a paragraph boundary (but a line with only spaces is not).

In a `.lsh` file, the body of an entry is whatever bloc of consecutive comment lines (i.e. lines that begin with one or more semicolons) that follow the title line. By convention, the lines should begin with two semicolons. A lines with only semicolons is intepreted as a paragraph boundary.

The body of an entry or section can be written acording to two different syntaxes. The "normal" syntax comprises a set of curly-brace-enclosed tags that have the same name and more or less the same function as the corresponding HTML tags. The "old" syntax (present for backward compatibility) comprises TROFF/NROFF-like tags that must be at the beginning of a line and begin with a dot. Blank lines indicate paragraph boundaries in both syntaxes

Debugging a documentation file or a Lush file with documentation entries can be done easily by calling `(helptool "the-file-to-be-debugged")` . The entry bodies are re-read whenever the file is modified, but the entries themselves (the title lines) are not.

Document Body: the Brace Syntax

The brace document format consists of segments of text enclosed between curly braces with a tag immediately following the opening brace. In this syntax, the following characters or character sequences have a special meaning:

- **two carriage returns (blank line)** : paragraph break
- **tag...** : a brace tag expression
- **,(...)** : read Lush expression in paren and insert it in the document. It will be evaluated at rendering time. If the result is a string, it will be inserted in the text.
- **\(...)** : same as above
- **,$expr** : read Lush object following dollar sign and insert it in the document. Unlike with the comma/paren construct, this allows to insert symbols and string literals. The Lush object will be evaluated at rendering time. If the result is a string, it will be inserted in the text.
- **\$expr** : same as above
- **,,expr** : read and evaluate the Lush expression and substitute the result (preferably a string) in the text. Unlike with with the single comma constructs, the evaluation is performed at read time.

- `...` : display content of bracket as bold/highlighted (mainly used for short Lush code snippets or arguments of the function being described).

To insert text that contain the above combinations requires escaping. This can be done with the **,$"any legal Lush string"** construct.

A brace expression is an expression of the form:

```
{<tag1> word1 {<tag2> word21 word22} word3 .... wordn}
```

A brace expression is read by the Lush reader just like a regular Lush expression. The reader translates the above example into a list like this:

```
(<tag1> -1 1 "word1" 1
        (<tag2> -1 1 "word21" 1 "word22")
        1 "word3" 1 "...." 1 "wordn")
```

Such brace lists produce the appropriate document when evaluated in the right context.

Here are the tags. Three dots indicate that text or other brace expression can be inserted.

Basic text formatting tags:

- `br` : line break
- `p` : paragraph break
- `li ...` : list item. Preferrably (though not necessarily) appears within a `ul` tag.
- `ul li ... ...` : unstructured list with list items
- `div ...` : does nothing, just groups its arguments
- `pre ... /pre` : preformatted text (no need to escape special characters)
- `code ... /code` : preformatted code displayed in blue (no need to escape special characters)
- `img ...` : insert image

Text attribute tags :

- `b ...` : turn to bold font
- `i ...` : set font to italic
- `c ...` : highlight, and turn bold (for code and arguments)

Metatags: The content of these tags is not displayed in the main document but is displayed in the header:

- `author ...` : author
- `symbol ...` : which Lush symbol does this entry describe.
- `location ...` : override the name of the file in which this function/entry is defined. By default, this is automatically set to file being read.
- `keywords ...` : keywords for searching this entry
- `date ...` : date of creation/modification of the object being described. By default, this is automatically set to the last modification time of the file in which the entry is defined (if this file is known).
- `title ...` : override for the name of the entry. By default this is set to the entry title line.
- `type ...` : type of object being described. By default this is guessed from the entry title whenever possible.

Special Tags:

- `ex` **lush-expression** : insert a demonstration of typing the lush-expression at the Lush prompt and getting the result.
- `hlink` : hyperlink to URL or other entry in the document
- `see` : hyperlink to URL or other entry in the document

Conditional tags:

- `if-html ...` : insert only if rendering to HTML
- `if-latex ...` : insert only if rendering to LaTeX
- `if-ogre ...` : insert only if rendering to helptool
- `if-text ...` : insert only if rendering to plain text

Tags that are legal but not yet implemented: these tags are legal but are not functional at the moment and and reserved for future implementations (they currently act as `div`):

- `desc ...` : short one-line description of the entry
- `u ...` : underlined text
- `tt ...` : typewriter font
- `font ...` : select font
- `center ...` : center text
- `h1 ...` : big title
- `h2 ...` : medium title
- `h3 ...` : small title

Document Body: the "Dot-Tag" Legacy Syntax

Predecessors to Lush (SN and TL3) used a format for documentation inspired by TROFF/NROFF with formatting tags that begin with a dot and are placed at the beginning of the line. Much of the Lush documentation is still in this format, which is why we document it here.

Within the text, strings enclosed in angled brackets are displayed in boldface and highlighted in a color different from the regular text. This should be used for short segments of Lush code, file names, arguments of the function being documented, and other litterals.

Tags are divided in three categories: tags with no argument, tags whose argument is on the same line as themselves, and tags whose "argument" (or scope) is the text until the next tag.

Formatting tags that take no arguments:

- **.P or .PP** : paragraph break (blank line plays the same role), or return to paragraph fill mode
- **.BR** : line break (without paragraph skip).

Formatting tags whose argument is on the same line:

- **.HLINK** : hyperlink to another entry of a URL.
- **.SEE** : reference/hyperlink to another entry.
- **.EX** : example, a Lush expression that will be evaluated and whose result is displayed in the document.
- **.IMG ir .EPS** : the path to an image file (in any format that ImageMagick understands).

Formatting tags whose argument is whatever text is between it and the next tag:

- **.LI or .IP** : list item.
- **.PRE or .VP** : pre-formatted text.
- **.CODE** : displayed code.
- **.DIV** : switch to paragraph mode (without a paragraph skip like .P

Conditional inclusion tags (active until the next tag):

- **.IFOGRE or .IFHELP or IFHLP** : include only in helptool output.
- **.IFTEX** : include only in LaTeX output.
- **.IFTXT** : include only in plain text output.
- **.IFHTML** : include only in HTML output

Metadata tags (whose argument is on the same line):

- **.TYPE** : type of entry (e.g. DX, DE, CLASS, MSG....)
- **.DESC** : short one-line description.
- **.FILE** : file where the definition for the entry resides.
- **.AUTHOR or .AUTH** : author(s)
- **.DATE** : date of last modification.

Directive `.EX` lets you run a lisp expression and display it as an example. It is preferable that the expression carries no side effect.

Directive IMG allows to include a picture (in any format):

```
.IMG toto.png
```

5.22 Word Processor

The help system is built around a rudimentary word processor system. This system can be used in Lush programs to format and display formatted text on various devices.

The basic structure manipulated by the word processing system is the brace expression as described in the section describing the help syntax.

A brace expression is an expression of the form:

```
{<tag1> word1 {<tag2> word21 word22} word3 .... wordn}
```

A brace expression is read by the Lush reader just like a regular Lush expression. The reader translates the above example into a list like this:

```
(<tag1> -1 1 "word1" 1
        (<tag2> -1 1 "word21" 1 "word22")
        1 "word3" 1 "...." 1 "wordn")
```

Such brace lists can be rendered appropriately when evaluated in the right context.

Brace lists can be rendered to plain text, to HTML, to LaTeX, or to a graphic device. A set of high level functions to do so is provided as described below. The typical syntax is as follows:

```
(render-brace-XXX arg1 arg2 ... '{<p> this is the text to be rendered})
```

where `XXX` is the device (text, html, latex, or graphics) and `arg1` ... `argn` are the arguments as described in the following sections. The above brace expression is equivalent to:

```
(render-brace-XXX arg1 arg2 ... '(<p> "this" "is" "the" "text" "to" "be" "rendered"})
```

A brace expression can span multiple lines and contain other brace expressions. Here is an example:

```
(render-brace-XXX arg1 arg2 ...
 '{<p> this is the text to be rendered
   {<ul>
    {<li> first item}
    {<li> second item}
   }
   some more text.})
```

5.22.1 (render-brace-text `left-margin` `right-margin` `brace`)

Renders the brace `brace` to the current output as plain text. The lines are wrapped at the `right-margin` . This is quite useful for producing formatted text from within a Lush script (e.g. to display a help text). Here is an example:

```
#!/bin/sh
exec lush "$0" "$@"
!#
(cond
  ((or (member "-h" argv) (member "--help" argv))
    (render-brace-text 0 72
     '{<p> This script does blah blah {<br>}
       The options are as follows:
       {<ul>
        {<li> "-h" or "--help": show this text}
        {<li> "-w" or "--wrong": produce the wrong output}}}))
  ((or (member "-w" argv) (member "--wrong" argv))
     (printf "wrong\\n"))
  (t (printf "right\\n"))))
```

5.22.2 (render-brace-html `left-margin` `right-margin` `brace`)

Renders the brace `brace` to the current output as HTML. The lines are wrapped at the `right-margin` .

5.22.3 (render-brace-latex `section` `brace`)

Renders the brace `brace` to the current output as LaTeX source. `section` is a string containing the section number of the brace within the LaTeX document.

5.22.4 (render-brace-graphics `x y w brace`)

Renders the brace `brace` to the current graphic window at location `x` , `y` setting the width of a text line to `w` pixels.

Chapter 6

Ogre: Object Oriented GUI Toolkit

6.1 Introduction to Ogre

See: Graphics.

See: Events.

Lush provides all the necessary graphic primitives for programming a graphics interface.

- Output primitives include functions for opening windows and functions for producing various elementary drawings. Lush supports multiple fonts, multiple windows and even multiple window systems.

- Input functionalities are also present. Lush gathers all events in a common event queue. There are functions for inspecting this event queue (`checkevent` and `waitevent`). Hook functions are called when events are occuring while Lush is waiting for user commands (`event-hook` and `idle-hook`).

It is rather easy to build simple graphic interfaces using these primitives. For instance, the function `snpaint` implements a simple drawing program for drawing lines and rectangles using low level primitives.

Graphic interfaces however become quickly complex. The class browser described in the last chapter of this document contains various features, like menus, buttons, requesters, selectors and scrollbars. A direct program would be overwhelmingly complex.

This is why people have developped various toolkits which handle automatically a large part of this complexity. The Ogre library is such a toolkit entirely written using the Lush object oriented language. This manual describes how to build programs with this library.

In the Ogre library, each component of a graphics interface, a button for example, is implemented as an object which is an instance of a subclass of class `VisualObject` .

The class of such an object defines both the appeareance of the object and its response to user input. Defining a new type of object (e.g. a menu) is thus just a matter of defining a new subclass of `VisualObject` .

There is a wide variety of such object classes:

- Basic objects are the major components of an interface. For instance, buttons, strings or check boxes are visible and respond to user interaction.
- Container objects are rectangular areas containing other objects and maintaining a certain layout. For instance there is a class `Column` which maintain several objects aligned in a column.
- Window objects are special container object affected to a window on the screen. Every object inserted in a window object appears into the window and is eligible for receiving user events.

There are two programming levels with the Ogre library. At the simplest level, you just use the provided classes for defining your interface. At the second level, you define subclasses of the standard Ogre classes in order to create new kind of graphic components.

6.2 Ogre Tutorial

Here is the transcript of a trivial call to the ogre library. You must first initialize the Ogre library by calling function `ogre` :

```
? (ogre)
 [ogre.lsh] (autoload)
= idle-hook
```

Let us then open a window `ww` containing a message string and a button arranged in a column named `cc` :

```
? (setq ww
    (new windowobject 0 0 400 300 "Essai"
      (setq cc (new column (new string "Press button to beep")
                           (new stdbutton "Beeper"
                                    (lambda(c) (beep))) ) ) ) )
= ::windowobject:e3100
```

When you click on the button, the callback function `(lambda(c) (beep))` is called and produces an audible beep. Let us add now a check box into the column:

```
? (==> cc insert (new checkbox "check me"
                   (lambda(c)
                        (printf "Check box state is \\%l\\n"
                                  (==> c getdata))))))
= ()
```

The callback function takes one argument. This argument is the check box object that we can query using method `getdata` . Let us add now an editable string named `ee` into the column:

```
? (==> cc insert (setq ee (new editstring 18 "hello")))
;; Argument 18 is the field width.
= ()
? (==> ee setdata "hello people")
= ()
? (==> ee getdata)
= "hello people"
```

Again we can manipulate the state of the editable string using method `setdata` and `getdata` . You may have noticed that the button width has changed when we have inserted the editable string. The column indeed manage its contents in order to keep them properly aligned. We can indeed move and resize the column as a whole:

```
? (==> cc move 50 70)
= (50 20 129 84)
? (==> cc resize 250 120)
= (50 20 250 100)
```

We can even add into the column a small object which lets you move the column with the mouse:

```
? (==> cc insert (new dragarea 50 20))
= ()
```

Arguments `50` and `20` are the width and height of this object. Since the drag area is inserted into a column, its width is adjusted to the column width.

Let us add a row with two exclusive buttons:

```
? (==> cc insert (new row (new radiobutton "choice1")
                          (new radiobutton "choice2") ) )
= ()
```

We can get (or set) the complete state of the column with a single message `getdata` (or `setdata`):

```
? (==> cc getdata)
= (() "hello people" () t)
? (==> cc setdata '(t "goodbye" t ()))
= ()
```

The Ogre library provides many more complex object, as shown in the following example:

```
? (setq h (new filerequester ww))
= ::filerequester:f3ef0
? (==> h popup)
= ::filerequester:f3ef0
? (==> h getdata)
= "/home/leon"
```

Let us add now a button for destoying the window.

```
? (==> ww insert (new stdbutton "Bye Bye."
                      (lambda(c) (==> thiswindowobject delete)) ) )
= ()
```

This button ends the tutorial.

6.2.1 Calling Ogre in a Lush Script

Ogre provides a simple way to write GUI applications. The best way to turn an Ogre program into a standalone application is to write an executable Lush script.

Here is an example of a very Lush script that opens up an Ogre window and runs until that window is closed by the user:

```
#!/bin/sh
exec lush "$0" "$@"
!#
;; A simple Ogre GUI demo
(ogre)
(wait (new autowindowobject 10 10 100 100 "Simple Lush GUI Demo"
        (new column
          (new stdbutton "    hit me    " (lambda (c) (printf "OUCH\\n")))
          (new stdbutton "    feed me   " (lambda (c) (printf "CRUNCH\\n"))))))
```

The `wait` function takes one argument. It causes Lush to sit around and keep processing events until the argument is nil. When the argument becomes nil (presumably as the result of processing an event), `wait` returns.

In the above example, the argument to `wait` is the `WindowObject` (i.e. the Ogre window) in which the buttons reside. When the user closes the window, this object is destroyed, hence `wait` returns, terminating the script.

Using `wait` in a script that runs an Ogre application is a necessity because scripts terminate as soon as the evaluation of the content of the script terminates. Since the constructors of Ogre applications return immediately (to allow for Ogre apps to run simultaneously with the Lush main prompt), the script woould open the Ogre window and terminate immediately if we did not use `wait` .

6.3 The Ogre Class Library

All the objects must interact in order to be drawn at the proper location and with the proper ordering. Similarly, user events (e.g. a mouse click) must be dispateched towards the object located below the mouse.

The kernel of the Ogre library ensures these interaction by defining how messages are propagated between the objects in a graphic interface.

6.3.1 Ogre Methods

Event dispatching is mostly performed by the `Container` class. A container dispatches the events messages to its children according to simple rules. Keyboard events are sent to the "active" object. All other events are sent to the object located under the mouse pointer.

Repainting is managed asynchronously. When you want to redraw an object, you must send a message `expose` to this object to tell the Ogre system to schedule a repainting operation. When the system becomes idle, the Ogre library sends a `repaint` message to all objects to repaint with a proper ordering which ensures that the topmost objects are repainted last.

Similarly, you never change the location of an object directly. You send message `move` , `resize` or `moveresize` which tell the Ogre system to relocate an object. The Ogre library then enforces all the rules of the container objects, performs a global recomputation of the location of all objects and places the objects by sending them a message `geometry` .

In general, most operation on the Ogre objects are implemented using two messages.

- The request message (e.g. `expose`) is used by a program to tell the Ogre library that you wish to perform a certain operation.
- The implementation message (e.g. `repaint`) is only called by Ogre to perform the operation and update all the other objects accordingly.

When you want to perform a certain operation, you must call the request message and not the implementation message. The Ogre library will call the implementation message for you at a proper time and with the proper order.

When you define a new class of Ogre objects, you must define the implementation method only. The corresponding request message will be inherited from the superclass.

Here is a list of the most useful message pairs: \begin center \begin tabular —c—l—l— \hline Operation & Request & Implementation \\ \hline \hline Drawing & \typo expose & \typo repaint \\ & & \typo repaint-bw \\ & & \typo repaint-color \\ & & \typo backpaint \\ & & \typo backpaint-bw \\ & & \typo backpaint-color \\ \hline Moving and resizing & \typo move & \typo geometry \\ & \typo moverel & \typo manage-geometry \\ & \typo resize & \typo compute-geometry \\ & \typo moveresize & \\ \hline Making an object visible & \typo insert & \typo realize \\ & \typo remove & \\ \hline \end tabular \end center

Operation	Request	Implementation
Drawing	expose	repaint
		repaint-bw
		repaint-color
		backpaint
		backpaint-bw
		backpaint-color
Moving and resizing	move	geometry
	moverel	manage-geometry
	resize	compute-geometry
	moveresize	
Making an object visible	insert	realize
	remove	

6.3.2 Ogre Class Hierarchy

There are in fact two major kinds of classes:

- "User classes" define the most useful graphics components. For instance, class `StdButton` defines a standard push button, class `Menu` manages popup menus and class `Requester` manages subwindows which pop up to request information from the user.

- "Abstract classes" are located on top of the class hierarchy. They define slots and request methods for managing these interactions in behalf of instances of their subclasses. Abstract classes are provided as a choice of superclasses for defining new classes of Ogre objects.

Here is a display of the current class hierarchy in the Ogre library. Classes displayed with a star are the abstract classes.

```
visualobject *
  control *
    editstring
      editnumber
      editsecretstring
    button *
      stdbutton
        tinybutton
      checkbox
        radiobutton
      filereqbutton
    menuitem
    knob *
      dragarea
      sizearea
    slider *
      hslider
      vslider
      scrollbar *
        hscrollbar
        vscrollbar
    textpane
  container *
    form *
      windowobject *
        autowindowobject *
      requester
        warningrequester
        errorrequester
        yesnorequester
          filerequester
          printrequester
       viewer
       selector
       edittext
     column
       menupopup
     row
     grid
     frame
       framesize
     viewerhole
```

```
emptyspace
darkspace
string
icon
menu
  choicemenu
```

Here is the use of the main abstract classes:

- Class `VisualObject` is the root of the Ogre class tree. It defines how objects are located and repainted. It also defines the default event processing messages.
- Class `Control` is the subclass of the simplest interactive objects. Such objects may be enabled or disabled. They may call a callback function when the user requests it.
- Class `Button` defines common messages handled by an object responding to a single mouse click.
- Class `Knob` defines common messages handled by an object which requests a geometry change of its container. For instance, inserting a `DragArea` in a container allow the user to move the container by clicking on the dragarea object.
- Class `Slider` and `Scrollbar` define the common methods for objects allowing the user enter a numerical value proportional to the location of a visible cursor.
- Class `Container` is the superclass of all objects containing other objects. It defines methods used for dispatching the events to the proper objects as well as methods for enforcing a certain layout.
- Class `Form` is a special container used for defining new objects composed of other objects. For instance, a `Viewer` is a `Form` containing two scrollbars and a container whose contents is moved according to the scrollbar position.
- Class `WindowObject` is the most important class. It defines a container object attached to a graphic window on the screen. Every object inserted in a window object becomes visible and is eligible for receiving events.

6.4 Ogre Utility Functions

6.4.1 Initializing the Ogre library

(ogre)

A single function named Ogre loads and initializes the library. You must call this function on top of each file involved with a graphics interface. Function

Ogre is often defined as an autoload function which loads the Ogre library and calls the actual Ogre function.

This function initializes the library and sets up the event dispatcher. In particular, it creates the object `ogre-task` and defines the functions `event-hook` and `idle-hook` . If the library was already initialized, function Ogre returns immediatly.

ogre-task

See: (event-hook)
See: (idle-hook)
See: (process-pending-events)
The object located in variable `ogre-task` is set up by function `ogre` . This object performs three central tasks in the Ogre library:

- It records all operations requested by the user and delayed by the library. This is the case of most repainting requests.
- It starts a repainting operation whenever function `idle-hook` is called.
- It sends event messages to the proper window object whenever function `event-hook` is called.

When Lush is waiting for user commands on the standard input, the Ogre library manages events asynchronously. The user can thus either type lisp commands or activate graphic interfaces.

When a Lush program is running, however, the Ogre library manages events when the function `process-pending-events` is called. It is a good practice to call this function a few times during long programs.

6.4.2 Error Handling in the Ogre Library

See: Errors in Lush.
See: (debug-hook)
See: Interruptions in Lush.
See: (break-hook)
The Ogre library runs Lush functions whenever an event occurs.

An error condition occurs when these functions are incorrectly designed. The error message is printed as usual, but you are not prompted for a debug toplevel.

An interruption occurs if you type `Ctrl-C` in the Lush console. The break message is printed as usual, but you are not prompted for a break toplevel.

If you wish to be prompted for these debugging utilities, you must redefine functions `ogre-debug-hook` and `ogre-break-hook` . Here is a simple way to achieve this:

```
(setq ogre-debug-hook nice-debug-hook)
(setq ogre-break-hook nice-break-hook)
```

6.4.3 Ogre Color Palette

Rendering a graphic interface strongly depends on the capabilities of your screen.

- A black and white screen is barely able to display a few levels of gray using special dithering patterns. Objects must be rendred using simple and visible graphics.
- On the other hand, a color screen is able to render three dimensional looking objects using several colors for rendering the various shadow level.

Ogre provides a way to test if you are using a black and white or a color screen. It also provides several utility functions for displaying three dimensional looking objects.

color-palette

See: (color [`c`])

If you are using a black and white display, this variable contains the empty list. You can then select three colors using function `color` .

- Color `color-fg` is the foreground color (usually black),
- Color `color-bg` is the background color (usually white)
- Color `color-gray` is a 50% dithered gray shade used for rendering disabled objects.

If you are using a color display, this variable contains an array containing the color numbers used by Ogre for displaying an object. Several variables are used to name these colors:

```
palette-left                (for rendering light)
palette-right               (for rendering shadow)
palette-up                  (for rendering raised objects)
palette-down                (for rendering depressed objects)
palette-disabled            (for rendering disabled texts)
palette-selected            (for rendering selected objects)
```

(new-palette `r g b`)

See: (alloccolor `r g b`)

See: (==> `windowobject` palette `p`)

This function creates a new palette array whose dominant color is defined by its RGB component `r` , `g` and `b` .

You can then store the resulting array into variable `color-palette` for redefining the default palette. You can also pass the new palette to an existing window object using message `palette` .

(getcolor `colname`**)**

See: (color [`c`])

This function returns the color number for the color named `n` in the current palette. Although any number can be passed as argument `n` , we suggest using one of the predefined symbols `palette-left` , `palette-right` , `palette-up` , `palette-down` , `palette-disabled` or `palette-selected` for specifying a color. The resulting color number can then be applied using function `color` .

This function is equivalent to:

```
(color-palette <colname>)
```

(setcolor `colname`**)**

See: (color [`c`])

This function sets the current color to the color named `n` in the current palette. Although any number can be passed as argument `n` , we suggest using one of the predefined symbols `palette-left` , `palette-right` , `palette-up` , `palette-down` , `palette-disabled` or `palette-selected` for specifying a color.

This function is equivalent to:

```
(color (getcolor <colname>))
```

(fill-background `x y w h`**)**

See: (fill-rect `x y w h`)

This function fills a rectangle with the background color `palette-up` of the current palette. The 3d rendering functions are designed for being called on such a background.

(draw-up-rect `x y w h`**)**

See: (draw-rect `x y w h`)

This function draws the border of a rectangle using the colors defined in the current palette. The border only is drawn and appears as a slightly raised line.

(draw-up-round-rect `x y w h`**)**

See: (draw-round-rect `x y w h` [`r`])

This function draws the border of a rectangle with round corners using the colors defined in the current palette. The border only is drawn and appears as a slightly raised line.

(fill-up-rect `x y w h` [`c`]**)**

See: (fill-rect `x y w h`)

This function fills a rectangle using color number `c` and draws a border in order to give the illusion of a rectangle located above the background.

The default value for `c` is the color number indicated by `palette-up` .

(fill-down-rect `x y w h` [`c`]**)**

See: (fill-rect `x y w h`)

This function fills a rectangle with the color number `c` and draws a border in order to give the illusion of a rectangle located below the background.

The default value for `c` is the color number indicated by `palette-down` .

(fill-up-round-rect `x y w h` [`c`]**)**

See: (fill-round-rect `x y w h` [`r`])

This function fills a rectangle with round corners using color number `c` and draws a border in order to give the illusion of a rectangle located above the background.

The default value for `c` is the color number indicated by `palette-up` .

(fill-down-round-rect `x y w h` [`c`]**)**

See: (fill-round-rect `x y w h` [`r`])

This function fills a rectangle with round corners using the color number `c` and draws a border in order to give the illusion of a rectangle located below the background.

The default value for `c` is the color number indicated by `palette-down` .

(fill-up-circle `x y r` [`c`]**)**

See: (fill-circle `x y r`)

This function fills a circle using color number `c` and draws a border in order to give the illusion of a circle located above the background.

The default value for `c` is the color number indicated by `palette-up` .

(fill-down-circle `x y r` [`c`]**)**

See: (fill-circle `x y r`)

This function fills a circle using color number `c` and draws a border in order to give the illusion of a circle located below the background.

The default value for `c` is the color number indicated by `palette-down` .

6.4.4 Ogre Fonts

Font names used by the font function used to be device dependent. A couple of functions have been defined for selecting fonts in a device independent way.

(ogre-font `size` [`serifp` [`monospacep` [`boldp` [`italicp`]]]]**)**

Selects a font of `size` size. The font family is chosen according to flags `serifp` and `monospacep` . The style is chosen according to flags `boldp` and `italicp` .

(font-18)

Convenience function. Same as `(ogre-font 18 () () () ())` .

(font-12b)

Convenience function. Same as `(ogre-font 12 () () t ())` .

(font-12)

Convenience function. Same as `(ogre-font 12 () () () ())` .

(font-8f)

Convenience function. Same as `(ogre-font 12 () t () ())` .

(font-8)

Convenience function. Same as `(ogre-font 8 () () () ())` .

6.5 Visual Objects

Class `VisualObject` is the root of the Ogre class tree and defines the default appearance (nothing) and the default response to user input (nothing too) of a graphics interface component.

Class `VisualObject` however defines most request messages which are inherited by the graphic components. It also defines the default implementation of most implementation messages.

Four slots are defined by class `VisualObject` :

```
(defclass visualobject object
  (rect (0 0 0 0))
  oldrect
  itscontainer
  window )
```

- Slot `window` is the descriptor of the window that contains the object. This slot is set as soon as a window is associated to the object.
- Slot `itscontainer` is set when the object is inserted into a container. It indicates which container (the father) manages this object.

- Slot `rect` is a list `(x y w h)` that indicates the rectangle associated to the object in the window coordinates.

- Slot `oldrect` is a temporary storage used by method `moveresize` to store the last acknowledged geometry of an object. This slot is non null whenever a geometry change has been requested.

6.5.1 (new VisualObject `w h`)

Creates and return a new instance of class `VisualObject` . Arguments `w` and `h` are the width and the height of the object.

This constructor is seldom used for creating new instances. Derived classes (e.g. `Control`) usually call this constructor method within their own constructor method in order to initialize the `VisualObject` part of their instances.

6.5.2 VisualObject Request Messages

Class `VisualObject` defines request messages inherited by most other objects for performing two central tasks:

- Repainting the graphic interface components

- Changing the geometry of graphic interface components.

Unless you have a thorough knowledge of the internals of Ogre, you should neither redefine nor override these messages when defining a subclass of `VisualObject` .

(==> `VisualObject` **expose** [`rect`])

Message `expose` tells the Ogre library that a visual object needs to be repainted. Argument `rect` indicates with more accuracy which rectangle in the current window must be repainted. When argument `rect` is ommited, the entire object rectangle is considered.

Exposure messages are propagated down until they reaches the window object. The clipped rectangle are then added to the damaged area list. Repainting is usually performed when the event queue becomes empty.

(==> `VisualObject` **repair-damaged**)

Message `repair-damaged` starts the repainting operation without delay. When this message is sent to an Ogre object, the Ogre system immediatly repaints the damaged areas.

(==> `VisualObject` moveresize `x y w h`)

Message `moveresize` ask the Ogre library to redefine the rectangle associated to an Ogre object. Arguments `x` and `y` are the coordinates of the topleft corner of the rectangle. Arguments `w` and `h` are respectively the width and the height of the rectangle.

This request is then signaled to the object's container which gets a chance to redefine its own geometry and to enforce a particular layout. When these recomputations are finished, the Ogre library effectively calls method `geometry` to relocate the objects.

(==> `VisualObject` move `x y`)

Message `move` is used for changing the location of the topleft corner of the object's rectangle to coordinates `x` and `y` . It actually calls method `moveresize` .

(==> `VisualObject` moverel `xr yr`)

Message `moverel` is used for adding `xr` and `yr` to the coordinates of the topleft corner of the object's rectangle. It actually calls method `moveresize` .

(==> `VisualObject` resize `w h`)

Message `resize` is used for changing the width and height of the object's rectangle to `w` and `h` . It actually calls method `moveresize` .

(==> `VisualObject` geometry `x y w h`)

See: (==> `Container` manage-geometry)

Method `geometry` performs a geometry change on a given object. Unlike method `moveresize` , the change takes place immediatly with all the overhead of recomputing the global position of all objects.

Method `geoemtry` is normally called by the method `manage-geometry` defined by the container objects for enforcing a particular layout. You may call this method directly (this is a backward compatibility requirement), although it is more efficient to use method `moveresize` .

(==> `VisualObject` front `x y w h`)

See: (==> `Container` manage-geometry)

Although the repainting process becomes inefficient, object rectangles may overlap. Methods `front` performs an immediate geometry change and places the object on top of the object stack.

(==> `VisualObject` back `x y w h`)

See: (==> `Container` manage-geometry)

Although the repainting process becomes inefficient, object rectangles may overlap. Methods `back` performs an immediate geometry change and places the object at the bottom of the object stack.

6.5.3 VisualObject Implementation Methods

Implementation methods are overriden by each subclass of `VisualObject` in order to define the behavior of each type of object. Class `VisualObject` provides a default implementation for four types of implementation methods:

- Method `realize` is called whenever an object is associated to or dissociated from a window. Redefining a realization method allows for precomputing font sizes and color numbers at once.
- The geometry computation method is called whenever the geoemtry requirement of an object may have changed. This method defines the minimum size of a graphic object.
- Repaint methods are called when the Ogre library detects that a part of the object rectangle has been damaged, either because method `expose` has been called or because the object has been moved or uncovered by another object.
- Events methods are called whenever the user performs certain mouse of keyboard actions. They define what actions are performed in response to user events. Event messages are documented in the next section.

(==> `VisualObject` realize `window`)

This method is called whenever an object is associated to a window (realized) or dissociated from a window (unrealized).

- An object is realized when it is inserted into a realized container. A window object is always realized. When an object becomes realized, method `realize` is called with a valid window descriptor as argument `window` . Method `realize` then sets the slot `window` , compute the geometry requirement of the object and calls `expose` in order to paint the object.
- An object is unrealized when it is removed from its container. Method `realize` is then called with an empty list as argument.

The default definition of method `realize` is given below.

```
(defmethod visualobject realize (w)
  (when (setq window w)
```

```
    (==> this compute-geometry)
    (==> itscontainer change-geometry) )
  (==> this expose rect) )
```

Subclasses of `VisualObject` may override this default definition. The new definition however must call the superclass method `realize` in order to perform the essential tasks described above.

Before overriding method `realize` , you should also consider overriding method `compute-geometry` instead of method `realize` .

(==> `VisualObject` compute-geometry)

Method `compute-geometry` is called when the object is realized and when a change of the object state changes the geometry requirements of the object.

This method must either return an empty list or compute the minimal size of the object's rectangle and enforce this minimal size by sending a message `resize` and return the new rectangle `rect` .

The default method `compute-geometry` just returns the empty list. Class `OgreString` for instance overrides this method to compute the size of the string text and resizes the string object to the correct size.

(==> `VisualObject` backpaint)

Method `backpaint` is called by the Ogre library for repainting the background of an object. In the case of a container object, this method is called before repainting the contents of the object.

The default method `backpaint` tests wether you have a black and white or a color display and calls method `backpaint-bw` or `backpaint-color` respectively. It is therefore advisable to override methods `backpaint-bw` and `backpaint-color` instead of `backpaint` .

(==> `VisualObject` backpaint-bw)

Method `backpaint-bw` is called by the Ogre library for repainting the background of an object on a black and white display. In the case of a container object, this method is called before repainting the contents of the object.

The default `backpaint-bw` method just clears the object's rectangle with the background color `color-bg` . This is suitable for most cases.

(==> `VisualObject` backpaint-color)

Method `backpaint-bw` is called by the Ogre library for repainting the background of an object on a color display. In the case of a container object, this method is called before repainting the contents of the container object.

The default `backpaint-color` method just clears the object's rectangle with the background color defined in the current palette. This is suitable for most cases.

(==> `VisualObject` repaint)

Method `repaint` is called by the Ogre library for repainting the foreground of an object. In the case of a container object, this method is called after repainting the contents of the container object.

The default method `repaint` tests wether you have a black and white or a color display and calls method `repaint-bw` or `repaint-color` respectively. It is therefore advisable to override methods `repaint-bw` and `repaint-color` instead of `repaint` .

(==> `VisualObject` repaint-bw)

Method `repaint` is called by the Ogre library for repainting the foreground of an object on a black and white display. In the case of a container object, this method is called after repainting the contents of the container object.

The default method `repaint-bw` does nothing.

(==> `VisualObject` repaint-color)

Method `repaint` is called by the Ogre library for repainting the foreground of an object on a color display. In the case of a container object, this method is called after repainting the contents of the container object.

The default method `repaint-color` does nothing.

6.5.4 Event Methods

See: (==> `Control` activate `d`)

See: Events.

Whenever the user performs certain mouse of keyboard actions, the Lush kernel generates an event as explained in the preceding chapter.

When the Ogre library detects an event in a window, it builds an ordered list of objects eligible for handling the event. An event message is then send to the object with highest priority. If the event method returns the symbol `ignored` , the library proceed with the next object in the ordered list. This process stops as soon as an object accepts the event (i.e. until an event message returns a value different from symbol `ignored` .)

Starting with the highest priority, the objects eligible for receiving event messages are:

- The toplevel container,
- The chain of embedded containers (if any) managing the topmost object,
- The topmost object located below the mouse cursor,
- The current actived control object of the window. Activation is discussed later in section “Control”.

Class `VisualObject` defines a default method for the event messages. These default methods just reject the event by returning symbol `ignored` .

In practice, the default event rejection mechanism ensures that mouse events are handled by the object located below the mouse cursor and keyboard events are handled by the actived objects. It is however possible to change these settings by overriding the event methods of objects with a higher priority.

(==> `VisualObject` **mouse-down** `x1 y1`)

See: (eventinfo)

Method `mouse-down` is called when the mouse button is depressed. Arguments `x1` and `y1` indicate the location of the mouse cursor in the current window when the mouse button has been depressed.

You can use function `eventinfo` to find the name of the mouse button and the state of the shift and control keys. Names returned by function `eventinfo` are quite machine dependent however.

(==> `VisualObject` **mouse-drag** `x1 y1 x2 y2`)

Method `mouse-drag` is called whenever the mouse is moved while the mouse button is depressed. Arguments `x1` and `y1` indicate the location of the mouse cursor in the current window when the mouse button was first depressed. Arguments `x2` and `y2` indicate the coordinates of the mouse cursor after the move.

(==> `VisualObject` **mouse-up** `x1 y1 x2 y2`)

Method `mouse-up` is called when the mouse button is released. Arguments `x1` and `y1` indicate the location of the mouse cursor in the current window when the mouse button was first depressed. Arguments `x2` and `y2` indicate the coordinates of the mouse cursor when the button was released.

(==> `VisualObject` **keypress** `c x y`)

Method `keypress` is called whenever an ASCII key is hit. Argument `c` is a string which contains a single character for that key. Arguments `x` and `y` are the location of the mouse cursor when the key was hit.

(==> `VisualObject` **arrow-left** `x y`)

See: (eventinfo)

Method `arrow-left` is called whenever the user types the left arrow key. Arguments `x` and `y` are the location of the mouse cursor when the key was hit.

You can use function `eventinfo` to find the exact name of the key and the state of the shift and control keys. Key names returned by function `eventinfo` are quite machine dependent however.

(==> `VisualObject` arrow-right `x y`)

See: (eventinfo)

Method `arrow-right` is called whenever the user types the right arrow key. Arguments `x` and `y` are the location of the mouse cursor when the key was hit.

You can use function `eventinfo` to find the exact name of the key and the state of the shift and control keys. Key names returned by function `eventinfo` are quite machine dependent however.

(==> `VisualObject` arrow-up `x y`)

See: (eventinfo)

Method `arrow-up` is called whenever the user types the up arrow key. Arguments `x` and `y` are the location of the mouse cursor when the key was hit.

You can use function `eventinfo` to find the exact name of the key and the state of the shift and control keys. Key names returned by function `eventinfo` are quite machine dependent however.

(==> `VisualObject` arrow-down `x y`)

See: (eventinfo)

Method `arrow-down` is called whenever the user types the down arrow key. Arguments `x` and `y` are the location of the mouse cursor when the key was hit.

You can use function `eventinfo` to find the exact name of the key and the state of the shift and control keys. Key names returned by function `eventinfo` are quite machine dependent however.

(==> `VisualObject` help `x y`)

See: (eventinfo)

Method `help` is called whenever the user types the system dependent help key. Arguments `x` and `y` are the location of the mouse cursor when the key was hit.

The help key under Windows is function key F1. The help key under X11 can be configured using program `xmodmap` .

You can use function `eventinfo` to find the exact name of the key and the state of the shift and control keys. Key names returned by function `eventinfo` are quite machine dependent however.

(==> `VisualObject` fkey `x y`)

See: (eventinfo)

Method `fkey` is called whenever the user types a key which does not correspond to an ASCII character and yet cannot be processed as an arrow key or a help key. You must then use function `eventinfo` to analyze the keyname and process the event.

Since key names returned by function `eventinfo` are quite machine dependent, your event handling procedure should use this name as a hint rather than expecting well defined values.

Remark: All operating systems define certain hot keys for various purposes. Lush cannot override these assignments. Under Windows for instance, keys F9, F10 and CTRL-F4 are directly processed by the operating system and never passed to WinLush.

(==> `VisualObject` size `w h`)

Method `size` is called whenever the geometry of an object changes in response to a user interaction or to a change in a related object.

6.6 Control Objects

See: Visual Objects.

See: Event Methods.

Class `Control` is the abstract class for interactive graphics objects like buttons, check boxes, menu items, etc...

Since class `Control` is a subclass of class `VisualObject` , all the properties and methods of class `VisualObject` also apply to class `Control` . In addition, class `Control` provides more support for defining the interactive graphics objects.

- Class `Control` provides standard methods for enabling or disabling a control object. Slot `disabled` contains a non nil value when a control is disabled.
- Class `Control` provides standard methods for changing the activation status of a control object.
- Class `Control` defines standard methods for changing the appearance of an object. Slot `text` contains whatever data represents the appearance of an object.
- Class `Control` defines standard methods for storing and handling the state resulting of user interaction. Slot `data` contains whatever data represent this state.
- Class `Control` stores a callback function in slot `callback` . This function is called when the user performs certain action defined in by implementation of the specific control object.

6.6.1 (new Control `w h f`)

See: Ogre Callbacks.

Creates and return a new instance of class `Control` . Arguments `w` and `h` are the width and the height of the control object. Argument `f` is the callback function for this control object.

This constructor is seldom used for creating new instances. Derived classes (e.g. `Button`) usually call this constructor method within their own constructor method in order to initialize the `Control` part of their instances.

6.6.2 Enabling or Disabling a Control Object

When disabled, the control object ignores all user interaction. This is signalled by having the control rendered with a specific appearance. A disabled button on a black and white display is rendered with a dithering pattern. The label of disabled button on a color display is rendered with a light gray color instead of black.

The enabled/disabled status of a control object is stored in the slot `disabled` defined by class `Control` .

- Slot `disabled` contains `()` if the object is enabled.
- Slot `disabled` contains a number (the disable count) is the object is disabled.

Implementation methods usually test slot `disabled` before performing their task. Repainting methods must change the rendering colors according to the enabled/disabled status of the object is disabled. Event methods must ignore event messages when the object is disabled.

Two request methods, `enable` and `disable` , are defined by class `Control` for changing the enabled/disabled status of an object:

(==> `Control` disable)

If message `disable` is sent to an enabled control object, the object is disabled. If message `disable` is send to an already disabled control object, the object's disable count is increased.

(==> `Control` enable)

When message `enable` is sent to a control object, the object's disable count is decreased by one. The object is enabled when this count reaches zero.

The disabled count feature proves useful for disabling an object either permanently or temporarily:

- For permanently enabling or disabling a control object whose state is unknown, enable it first and possibly disable it.
- For temporarily disabling an object, just disable it. It will retrieve its previous state when you will enable it again.

(new DisableLock `c1...cn`)

It is often useful to disable a button during the execution of some Lush program and to restore its initial state. This task could be achieved by sending messages `disable` and `enable` .

If however an error occurs during the program execution, the button would remain disabled. The best way to solve this problem consists in creating a lock object defined by class `DisableLock` .

If you wish to temporarily disable object `c1` to `cn` , proceed as follows:

```
(let ((lock (new DisableLock c1...cn)))
        ;;; call your Lush program here
        (......) )
```

When created, the lock object sends a message `disable` to the objects `c1` to `cn` . The lock is destroyed when you leave the `let` instruction. The lock destructor then sends a messages `enable` to our objects `c1` to `cn` .

If an error occurs during the execution of the Lush program, the lock is destroyed by the garbage collector and the objects retrived to their initial state.

6.6.3 Activation of a Control Object

The active control object receives all the event messages that were not processed by the objects located below the mouse cursor. Only one object might be actived in a given window at a given instant.

Activation is especially useful for redirecting keyboard events toward objects containing some editable text. Having the keyboard events processed by the object located under the mouse often seems unnatural because the keyboard does not move physically like a mouse.

The user activates such an editable object with a mouse click in the object rectangle. It is customary in Ogre that the editable objects ignore the keyboard events unless they are active. Most keyboard events are then unconditionnally directed to the active object without regard to the position of the mouse cursor.

On the other hand, it is often useful to implement accelerator keys for performing tasks usually achieved by menu items or buttons. Accelerator key events must be defined by the containers managing the related objects. This definition ensures that accelerator keys events are not sent to the active object but handled directly.

The activation status of a control object is stored in slot `activated` (sic) defined by class `Control` . This slot contains a non nil value if the control object is active. Implementation methods of editable objects usually test slot `activated` before performing their task. Repainting methods must indicate the activation status of an object. Event methods must ignore event messages unless the object is actived.

Two request methods are defined by class `Control` for testing or changing the activation status of an object. These methods ensure that one object only

is actived in a given window.

(==> `Control` activate `d`)

Changes the activation status of the target object to `d` . If argument `d` is `()` , the target object is deactived. If argument `d` is `t` , the target object is actived and the previous active object is desactived.

6.6.4 Appearance of a Control Object

Class `Control` defines a slot named `text` which contains whatever data is useful for defining the appearance of a control object. In most cases so far, this slot contains a single string which is either a button label or a checkbox caption.

Two request methods are defined by class `Control` for obtaining and changing the contents of this slot:

(==> `Control` settext `d`)

This request methods changes the contents of the slot `text` of the target object to the value given by argument `d` .

Since this change causes a general change in the object appearance, the object is sent a message `compute-geometry` for redefining its geometry requirements and a message `expose` for updating the display. Sending message `settext` sometimes triggers a global relocation of all objects in the window.

(==> `Control` gettext)

This request method returns the contents of slot `text` . It is prefered to use this method rather than accessing directly the slot `text` because complex control objects may use other slots for controlling the appearance of the object.

6.6.5 State of a Control Object

Class `Control` defines a slot named `data` which contains whatever data is useful for storing the state of a control object resulting from the user interaction. For instance, slot `data` of a check box contains a boolean value. Similarly, slot `data` of an editable string contains the current value of the string.

Three methods `hasdata` , `setdata` and `getdata` are used for obtaining and changing the state of a control object. These methods are more implementation method than request methods. Several subclasses of `Control` override these methods in order to pre-process or post-process the state information.

These methods are mostly useful because they provide an abstract way to save and restore the state of a collection of control objects. For instance, a container object can save or restore the state of all its descendants with a single message.

(==> `Control` setdata `d`)

This method changes the state of a control object to `d`

The default method sets the slot `data` of the target object to the value given by argument `d` . Since this change usually changes the object appearance, the object is sent a message `expose` for updating the display.

(==> `Control` getdata)

This method returns the state of a control object.

The default method just returns the contents of slot `data` . It is prefered to use this method rather than accessing directly the slot `data` because complex control objects may use other slots for controlling the state of the object.

(==> `Control` hasdata)

If an object contains some state information, this method returns the object itself. If an object contains no state information, this method returns the empty list.

Method `hasdata` is used to test if an object provides an adequate implementation for methods `setdata` and `getdata` . The default method `hasdata` indicated that there is some state information in the control object. Certain subclasses, like push button, override this method in order to signal that they carry no state information.

6.6.6 Ogre Callbacks

See: Window Objects.

See: Popup Requesters.

See: Forms.

See: Menus.

Each control object has a callback function. Callback functions are executed when certain event messages are received. For example, a push button executes the callback function when it is depressed.

Callbacks functions are called with one argument which is the caller object. They are executed within the caller scope and therefore directly access the slots of the caller object. In addition certain local variables are set:

- Variable `thiswindowobject` always refers to the toplevel container of the control object. See the discussion on class `WindowObject` for more information.
- Variable `thisrequester` possibly refers to the closest requester containing the control object. See the discussion on class `Requester` for more information.
- Variable `thisform` possibly refers to the closest form containing the control object. See the discussion on class `Form` for more information.

- Variable `thismenu` possibly refers to the menu object containing the menu item which has launched the callback function. See the discussion on class `Menu` for more information.

Three methods are implemented by class `Control` for handling callback functions.

(==> `Control` setcall `f`)

Message `setcall` redefines the callback function of a target object to be function `f` . Trouble will surely occur if the argument `f` is not a function with one argument or an empty list.

(==> `Control` execute)

Message `execute` starts the callback function with the object itself as argument. The object is disabled while the callback function is running.

(==> `Control` trigger)

Message `trigger` asks the object to simulate an user action. When a push button receives a trigger message, for example, it is drawn as depressed, executes the callback function and is redrawn as usual.

The default implementation defined by class `Control` just calls method `execute` . Subclasses of `Control` usually override method trigger by sending false event messages to the object.

6.7 Container Objects

See: Visual Objects.

Class `Container` is the abstract class for container objects. A container object manages several objects called its sons. Sons are only visible through the rectangle of the container. The container dispatches event messages, propagates repainting messages, controls the geometry of its sons and performs a couple of other critical tasks.

Since class `Container` is a subclass of class `VisualObject` , all the properties and methods of class `VisualObject` also apply for class `Container` . Class `Container` however defines new slots and new methods for handling container objects.

- Slot `contents` contains a list whose elements `(son x y w h)` point to the managed objects `son` and cache the coordinates for their rectangles.

- Class `Container` defines private method for implementing the core of the Ogre library. These methods deal with the repainting process and the geometry management.

- Class `Container` defines public request methods for handling several control objects as a whole. Method `enable` and `disable` allow you to disable or enable all the control objects located in a container. Method `getdata` returns a list containing the states of all control objects located in the container. This list is built by sending messages `getdata` to the control objects. Method `setdata` takes such a list as argument and sets the state of all control objects located in the container.

Subclasses of `Container` are very frequently defined for defining structuring container whose sons are arranged according to a certain layout. Classes `Row` and `Column` are examples of such subclasses.

6.7.1 (new Container `x y w h ...contents...`)

Creates a new container object. Arguments `x` , `y` , `w` and `h` describe the initial rectangle of the container object. The arguments denoted as `...contents...` are graphical objects initially managed by the container object.

This constructor is seldom used for creating new instances. Derived classes (e.g. `Row`) usually call this constructor method within their own constructor method in order to initialize the `Control` part of their instances.

6.7.2 Repainting

When it is time to repaint the contents of a container, the container object receives a message `backpaint` for rendering its background. The objects managed by the container are then rendered on top of this background. Finally the container object receives a message `repaint` for drawing above the managed objects.

During this operation, the clipping rectangle is the container object's rectangle. Only the portion of the sons which overlap the container's rectangle are rendered.

6.7.3 Inserting an Removing Objects

Three request method are implemented for inserting new objects into a container or removing inserted objects from a container.

(==> `Container` **insert** `what`)

Inserts object `what` on top of the objects located in the container.

When an object is inserted into a container, the coordinates of the topleft corner of the container's rectangle are added to the coordinates of the object's rectangle. Therefore, if the initial rectangle of an object is located at cordinates `(0, 0)` , the object appears on the topleft corner of the container.

If there is a policy for the layout of the container, the new layout is computed and each object in the container is moved, resized and repainted.

(==> `Container` remove `what`)

Removes object `what` from the container.

When an object removed from a container, the coordinates of the topleft corner of the container's rectangle are subtracted from the coordinates of the object's rectangle. Inserting the object again thus inserts the object at the same location with respect to the container's topleft corner.

(==> `Container` removall)

Removes all objects managed by the container.

6.7.4 Geometry Management

Class `Container` implements the core of the geometry management system of the Ogre library. Containers are used to enforce certain geometry policies on their sons. These structuring containers liberate the programmer from precomputing the position of each object for all the interface configurations. For instance:

- Class `Container` enforces a very weak policy. If you move an instance of class `Container` , the managed objects move with their container.
- Class `Row` aligns the objects in a row and ensures that all objects have the same sufficient height.
- Class `Column` aligns the objects in a column and ensures that all objects have the same sufficient width.
- Class `Grid` aligns the objects on a grid with a predefined number of columns. Objects located in a same column have the same sufficient width and objects located in a same row have the same sufficient height.

Containers define the geometry policy using only two implementation methods, `compute-geometry` and `manage-geometry` . Here are the steps involved in a geometry computation:

- 1. An object managed by a container receives a geometry change request (i.e. `moveresize` , `move` or `resize`).
- 2. The container of this object receives a message `compute-geometry` in order to recompute its size requirements by looking at the slots `rect` of the managed objects. It might then call method `resize` in order to enforce its new minimal size.
- 3. The final rectangle of the container is stored in slot `rect` of the container. The container then receives a message `manage-geometry` to get a chance to assign a final rectangle to all managed objects by sending them a message `geometry` .

The hidden part of the iceberg lies in the global computation of the object's location. If the container decides to change its geometry requirements during step 2, the container of the container then receives a message `compute-geometry` and gets a chance to participate to this geometry discussion. When this second container sets the first container's geometry, it calls the first container's method `geometry` which calls the first container's method `manage-geometry` and effectively performs step 3.

Important Note: The geometry management system has been significantly revamped in Lush. It is now faster and skinnier. Old programs still work correctly unless they redefine method `geometry` of class `WindowObject` .

(==> `Container` **compute-geometry)**

The implementation method `compute-geometry` is called when the container is realized and when a change in their object state changes the geometry requirements of the object.

This method must compute the minimal size of the container on the basis of the information stored in the slots `rect` of the managed objects or cached in the list of managed objects stored in slot `contents` of the container. This method may enforce a minimal size by sending a message `resize` .

The default method `compute-geometry` just returns the empty list.

Example:

Class `Row` defines the following method `compute-geometry` which computes the minimal size of a row:

```
(defmethod row compute-geometry ()
  (when window
    (let* ((height (sup (cons 0 (all (((son x y w h) contents)) h))))
           (width (sum (all (((son x y w h) contents)) w))) )
      (==> this resize (+ width (* hspace (length contents))) height)
      rect ) ) )
```

(==> `Container` **manage-geometry)**

The implementation method `manage-geometry` is called when it is time to locate the sons of the container on the basis of the final rectangle assigned to the container.

This method must compute the final rectangle of all the managed objects according to the container's rectangle (found in slot `rect` of the container) and according to the managed object initial rectangles (found in slot `rect` of the managed objects and cached in the list of managed objects located in slot `contents` of the container.) Method `manage-geometry` then sends a message `geometry` to the managed objects in order to assign a final rectangle to these objects.

The default method `manage-geometry` defined in class `Container` does nothing. A secondary mechanism just moves the managed objects in order to

maintain their position with respect to the topleft corner of the container's rectangle.

Example:

Class `Row` defines the following method `manage-geometry` which arranges the managed objects in a row starting on the left of the container's rectangle.

```
(defmethod row manage-geometry ()
  (let (((x y w h) rect))
    (incr x (div hspace 2))
    (each ((i contents))
      (let ((w (nth 3 i)))  ;; get the object's width
         (==> (car i) geometry x y w h)
         (incr x (+ w hspace)) ) ) ) )
```

(new GeometryLock `c`)

A significant amount of time is spent in geometry management when many objects are inserted into a realized container. It is then advisable to postpone the geometry until all objects have been inserted.

Such a task is accomplished by creating an object instance of class `GeometryLock` as follows:

```
(let ((lock (new GeometryLock c)))
        ;;; insert your objects here into container c
        (==> c insert ...) )
```

When created, the lock sets a flag in container `c` which suspend the usual geometry management. When the lock is destroyed, the flag is cleared and the geometry management is started once. If an error occurs during the object insertion, the lock is destroyed by the garbage collector and the container is left with an acceptable state.

6.7.5 Control Management

Five request methods are defined by class `Container` for handling as a whole several control objects managed by a container or by its sons.

(==> `Container` disable)

When a container receives a message `disable` , all the control objects managed by the container or by its sons receive a message `enable` . This is useful for controlling the enabled/disabled state of several control objects at once.

```
(defmethod container disable ()
  (each ((i contents))
    (==> (car i) disable) ) )
```

(==> `Container` enable)

When a container receives a message `enable` , all the control objects managed by the container or by its sons receive a message `enable` . This is useful for controlling the enabled/disabled state of several control objects at once.

```
(defmethod container enable ()
  (each ((i contents))
    (==> (car i) enable) ) )
```

(==> `Container` hasdata)

Method `hasdata` returns the list of stateful control objects managed by the container of by any of its sons. This is achived by concatenating the result of running method `hasdata` on all the objects managed by the container.

```
(defmethod container hasdata ()
  (flatten (all ((i contents)) (==> (car i) hasdata))) )
```

(==> `Container` getdata)

Method `getdata` returns the list of states returned by all control objects managed by the container of by any of its sons.

```
(defmethod container getdata ()
  (all ((i (==> this hasdata)))
    (==> i getdata) ) )
```

(==> `Container` setdata `d`)

Method `getdata` sets the states of all control objects managed by the container of by any of its sons using list `d` which is usually returned by a call to method `getdata` .

```
(defmethod container setdata (d)
  (each ((i (==> this hasdata))
         (j d) )
    (==> i setdata j) ) )
```

6.8 Container Flavors

This chapter presents various flavors of predefined containers. Class `Form` define a container abstract class for implementing composite objects. Class `WindowObject` defines a toplevel container associated to a window on your screen. Structuring containers help organizing the layout of a graphic interface.

6.8.1 Forms

See: Ogre Callbacks.

Class `Form` is an abstract class used to group several cooperating objects into a single Ogre component. This component can then be inserted into a window or a requester.

Class `Form` is a trivial subclass of class `Container` . Class `Form` adds three properties to standard containers.

- Method `compute-geometry` makes a form object large enough to hold all its sons while keeping their initial relative positions.
- Method `manage-geometry` places the sons of a form object at the center of the form object while keeping their relative position.
- When Ogre calls the callback function of a descendant of a form object, it sets the temporary variable `thisform` to the form object.

These properties are handy for designing complex graphics objects composed of several components as subclasses of `Form` . Several classes, like class `WindowObject` , class `Requester` and class `Viewer` , are implemented as subclasses of `Form` .

Moreover, a form object is sometimes considered as an extended control object. For instance, we can gather a slider and an editable numeric field into a form object and use this object like an usual control object.

This is achieved by defining a subclass of `Form` whose constructor builds the elements of the composite object and keeps a pointer to these object in some of its slots. This subclass then must define methods `hasdata` , `setdata` , `getdata` , `execute` and `setcall` like those of a control object. Method `hasdata` usually returns `this` . The other methods are usually forwarded to the components of the composite object.

(new Form `...contents...`)

Creates a new form object managing objects `...contents...` . There is no need to provide a default size for a form object since the form size is derived from the component sizes and location.

6.8.2 Window Objects

See: Events.

Class `WindowObject` is a subclass of class `Form` which implements the link between a physical window and its managed ogre objects.

- The constructor of class `WindowObject` creates a window on the screen with the same size than the window object rectangle. It call then method `realize` to tell its sons that a physical window is now attached to the graphic interface.

- Class `WindowObject` define methods for handling the window events and redirecting event messages to the proper descendant objects.
- Class `WindowObject` define slots and methods to manage the repainting process. The window object remembers which portions of the window has been damaged or exposed (using message `expose`) and sends repainting messages when appropriate.
- Class `WindowObject` defines methods for synchronizing the window object rectangle and the physical window geometry.

A window object is a toplevel container. Inserting a window object into another container is an almost certain cause of trouble.

Whole graphics interface are often defined as subclasses of `WindowObject` . Such subclasses usually contain new slots for storing interface specific data. Callback functions then send messages to `thiswindowobject` . These messages then are executed within the interface scope.

(new WindowObject `x y w h name ...contents...`**)**

Creates a window object managing objects `...contents...` .

When a window object is created, the constructor creates a window on the screen named `name` with width `w` and height `h` . If both arguments `x` and `y` are `0` , the position of this window is defined by the default rule of your computer. If both arguments are positive, they define the position of the topleft corner of the window.

Exemple:

```
;;; First of all, initialize Ogre !
? (ogre)
= idle-hook
;;; create a window with a button
? (setq win (new WindowObject 100 100 300 300 "Example"
                         (setq b (new stdbutton "Hello"              ;; its name
                                              (lambda(c) (beep)) )) )) ;; its callback
= ::WindowObject:06f00
;;; Now we remove the button b from win
? (==> win remove b)
```

Note: The event handler of the newly created window is the corresponding Ogre window object. You can thus interactively obtain the Ogre window object for a given window by typing `(setq w (waitevent))` and clicking into the corresponding window.

(new AutoWindowObject `x y w h name contents`**)**

Lush provides an additional class named `AutoWindowObject` . Unlike class `WindowObject` , class `AutoWindowObject` manages a single object `contents`

and dynamically resizes the window to the size of the managed object.

The constructor arguments `w` and `h` are only hints for sizing the window. They may be empty lists instead of numbers. The size of the window is eventually given by the size of its contents.

WindowObject Request Methods

(==> `windowobject` palette `p`)

See: Ogre Color Palette.

Message `palette` redefines the color palette used by Ogre for repaining the objects managed by the windowobject. This method sets slot `color-palette` of the window object and calls `expose` in order to repaint the window.

Argument `p` can be the empty list (for black and white display) or a palette returned by function `new-palette` (for color display). The color palette selection is enforced even if you do not use an adequate screen.

(==> `windowobject` setmodal `m`)

Message `setmodal` affects the redirection of event messages by defining a model object. When a modal object is defined, all events are redirected to this object. This is useful for implementing requesters that prevent the user to reach the other components of the interface.

- When argument `m` is the empty list, message `setmodal` undefines the current modal object.
- When argument `m` is one of the objects managed by the window object, message `setmodal` makes this object modal.

(==> `windowobject` getmodal)

Returns the current modal object in the window object or the empty list if no modal object is defined.

(==> `windowobject` read-event)

See: Event Lists.

Method `read-event` returns the next available event in the window associated to the window object. Events are returned as standard Lush event lists. If no events are pending, method `read-event` blocks.

(==> `windowobject` manage-event `event`)

See: Event Lists.

Method `manage-event` processes a Lush event list `event` and sends a corresponding event message to the proper object in the window object.

WindowObject Implementation Methods

When defining a graphic interface as a subclass of class `WindowObject` you may override three methods:

- Method `delete` is called when you destroy the window using a the window manager feature (i.e. with the mouse).

- Methods `compute-geometry` and `manage-geometry` work as usual. They are however called in special cases allowing a dynamical change of the window size.

(==> `windowobject` delete)

Method `delete` is called when the user deletes the onscreen window using the window manager features. In the same spirit, we suggest you to call method `delete` whenever you destroy a window object.

The default method `delete` just deletes the window object. You may override this method, for instance, to pop up a confirmation dialog to the user.

(==> `windowobject` manage-geometry)

Beside its normal use, method `manage-geometry` is also called when you resize the window with the mouse. The mouse action is actually converted into a `size` event which is processed using the usual geometry management functions.

The default method `manage-geometry` does nothing. Overriding the method `manage-geometry` in a window object is often useful for adjusting the size of the window object components to the onscreen window size.

(==> `windowobject` compute-geometry)

Method `compute-geometry` usually computes the minimal size of the object and enforces this geometry using method `resize` . In the case of a window object, the actual size of the onscreen window is also adjusted.

The default method `compute-geometry` does nothing. You may override this method to adjust automatically the size of the onscreen window to the mimial size of its contents.

6.8.3 Structuring Containers

See: Container Objects.

Structuring containers are used for designing the layout of a graphics interface. They arrange their sons according to a certain policy. Ogre provides three class of structuring containers: class `Row` , class `Column` and class `Grid` .

These containers inherit all the slots and methods of an ordinary container. They just define specific methods `compute-geometry` and `manage-geometry` to enforce their layout policy.

In addition, two classes are provided for padding a row or a column with some space, class `EmptySpace` and class `DarkSpace` .

(new Row `...contents...`)

See: (==> `Container` insert `what`)

See: (==> `Container` remove `what`)

See: Control Management.

Returns a new row object managing the objects specified by `...contents...` .

Class `Row` defines a structuring container which aligns its contents horizontally from left to right. The height of a row is determined by the managed object whose required height is largest.

An empty space is inserted between the objects. This space is defined by slot `hspace` of the row object and defaults to `4` points.

(new Column `...contents...`)

See: (==> `Container` insert `what`)

See: (==> `Container` remove `what`)

See: Control Management.

Returns a new column object managing the objects specified by `...contents...` .

Class `Column` defines a structuring container which aligns its contents vertically from top to bottom. The width of a column is determined by the managed object whose required width is largest.

An empty space is inserted between the objects. This space is defined by slot `vspace` of the column object and defaults to `4` points.

Example:

```
;; Creates a window with three buttons
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
                (new Column
                  (new StdButton "One" (lambda(d) (print 1)))
                  (new StdButton "Two" (lambda(d) (print 2)))
                  (new StdButton "Three" (lambda(d) (print 3))))))
= ::WindowObject:06f00
```

(new Grid `cols ...contents...`)

See: (==> `Container` insert `what`)

See: (==> `Container` remove `what`)

See: Control Management.

Returns a new grid object with `cols` columns and managing the objects specified by `...contents...` .

Class `Grid` defines a structuring container which aligns its contents in a grid with `cols` columns. The grid is filled left to right. The width of each column is determined by the object of the column whose required width is largest. The height of each row is determined by the object of the row whose required height is largest.

Empty spaces are inserted between rows and columns. These spaces are defined by slot `vspace` and `hspace` of the grid object and defaults to `4` points.

(new EmptySpace `w` [`h`] **)**

Extra spaces may be added into a structuring container by inserting an emptyspace object. Such an object has no response to user events and is uniformly filled with the background color.

Class `EmptySpace` is a class of emptyspace objects. The constructor expression above returns a new emptyspace object of minimal width `w` and minimal height `h` . When argument `h` is omitted, a square minimal space is assumed.

Of course, the minimal geometry specifications interact with the geometry management of structuring conteners. It is seldom necessary to define all the sizes of an emptyspace object.

Exemple:

```
;;; Creates a window with three buttons
;;; with a 20 pixels space between button 2 and button 3
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
                (new Column
                  (new StdButton "One" (lambda(d) (print 1)))
                  (new StdButton "Two" (lambda(d) (print 2)))
                  (new EmptySpace 20)
                  (new StdButton "Three" (lambda(d) (print 3))) )))
= ::WindowObject:06f00
```

(new DarkSpace `w` [`h`] **)**

Similarly you can insert a darkpace object into a structuring container. On a black and white display, a darkspace is displayed as a black area. On a color display, a darkspace is displayed as a raised area.

Class `DarkSpace` is a class of emptyspace objects. The constructor expression above returns a new emptyspace object of minimal width `w` and minimal height `h` . When argument `h` is omitted, a square minimal space is assumed.

Of course, the minimal geometry specifications interact with the geometry management of structuring conteners. Specifying a single argument of `4` , for instance, will define a minimal width and height of 4 points. If such a darkspace object is inserted in a column, the width of object will be increased up to the column width, effectively displaying a 4 points thick line.

Exemple:

```
;;; Creates a window with three buttons
;;; with a 3 pixels line between button 2 and button 3
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
                (new Column
                  (new StdButton "One" (lambda(d) (print 1)))
```

```
                (new StdButton "Two" (lambda(d) (print 2)))
                (new DarkSpace 3)
                (new StdButton "Three" (lambda(d) (print 3))))))
= ::WindowObject:06f00
```

6.9 Buttons

Buttons are small interactive area which can be used for triggering an action or storing boolean states. Ogre provides four class of buttons:

- Class `StdButton` and `TinyButton` for push buttons,
- Class `CheckBox` for boolean state buttons,
- Class `RadioButton` for exclusive buttons.

All these classes are subclasses of `Control` . Buttons thus inherit all the slots and methods defined in class `Control` .

6.9.1 Push Buttons

A push button invokes a specific action when the user clicks the mouse button while the pointer is located above the button image. Two kinds of push buttons are defined by the Ogre library.

(new StdButton `label call`**)**

See: (font-12b)
See: (==> `Control` settext `d`)
See: (==> `Control` gettext)
See: (==> `Control` setcall `f`)
Class `StdButton` implements a standard push button. The label of a standard push button is displayed using the font set by the function stored in slot `textfont` , which default to a 12 points bold font `font-12b` .

The constructor expression returns a push button whose name is given by string `label` . This label may be changed by sending a message `settext` . When the button is depressed, the callback function `call` is executed. The button remains disabled while the callback function has not returned.

Example:

```
;;; A window with a 3 state label
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
                (new StdButton "One"
                        (lambda(caller)
                                (==> caller settext
```

```
                                        (selectq (==> caller gettext)
                                                ("One" "Two")
                                                ("Two" "Three")
                                                (t     "One") ) ) ) ) ) )
= :WindowObject:06f00
```

(new TinyButton `label call`**)**

See: (font-12b)
See: (==> `Control` settext `d`)
See: (==> `Control` gettext)
See: (==> `Control` setcall `f`)

Class `TinyButton` implements a push button with a reduced size. A tiny push button is slightly smaller than a standard push button because the label is written using a smaller font `font-12` .

The constructor expression returns a push button whose name is given by string `label` . This label may be changed by sending a message `settext` . When the button is depressed, the callback function `call` is executed. The button remains disabled while the callback function has not returned.

6.9.2 Check Boxes

A check box is composed of a small square button followed by a descriptive text. When the user clicks on the square or on the text, the state of the small square changes.

Class `CheckBox` implements check box buttons. Since class `CheckBox` is a subclass of class `Control` , all the slots and methods defined for control objects are inherited.

(new CheckBox `label call`**)**

See: (==> `Control` settext `d`)
See: (==> `Control` gettext)
See: (==> `Control` setdata `d`)
See: (==> `Control` getdata)
See: (==> `Control` setcall `f`)

Returns a new check box whose descriptive text is given by string `label` . When a check box is depressed, its state changes and the callback function `call` is executed.

You can query or change the state of a check box (`t` or `()`) by sending messages `getdata` and `setdata` . The descriptive text can be modified using `settext` .

Example:

```
;;; A window with a check box that controls
;;; the enable/disable state of a push button
```

```
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
          (new column
                (setq thebutton
                        (new StdButton "beep"
                                (lambda(c) (beep)) ))
                (new CheckBox "Disable button"
                                (lambda(c)
                                        (if (==> c getdata)
                                                (==> thebutton disable)
                                          (==> thebutton enable) ) ) ) ) ) )
```

(new ImageButton `up-image down-image disabled-image call`**)**

See: (==> `Control` setcall `f`)

Class `ImageButton` implements a push button whose appearance is an image drawn with `rgb-draw-matrix` . Each of the first three arguments must be an idx2 or an idx3 of ubytes containing the pixel data (greyscale if an idx2, RGB if an idx3 with at least 3 elements in the last dimension). `up-image` is the image shown when the button is up, `down-image` when it is pushed, `disabled-image` when it is disabled. The A (alpha) component, if present, is ignored. `call` is the callback function called when the button is clicked.

Here is an example that shows how to create down and disabled image of an `ImageButton` from an up image.

```
(libload "libimage/image-io")
(libload "libimage/rgbaimage")
(ogre)
(setq up (image-read-rgba "my-button-image.png"))
(setq down (copy-array up))
(rgbaim-contbright up down -1 0)
(setq disabled (copy-array up))
(rgbaim-contbright up disabled 0.5 40)
(setq btn (new ImageButton up down disabled (lambda (c) (printf "coucou\\n"))))
(setq window (new windowobject 10 10 200 200 "test" btn))
```

(new StdButtonBg `text up-image down-image disabled-image call`**)**

See: (==> `Control` setcall `f`)

This works just like an `StdButton` except that the image of the button is an RGBA or grayscale image passed as argument (instead of the boring blueish rounded rectangle of `StdButton`). The text is displayed on top of the image. The three images passed as argument will be used respectively as the up button image, the clicked (down) button image, and the disabled button image. The text is drawn in black over the background image, so the image colors should not be too dark. The background images are automatically sized/resized to fit

the text (without necessarily preserving the aspect ratio). The images must be idx2 (for grayscale images) or idx3 with the last dimension equal to 4 (RGBA images). Tha A (alpha) component is ignored.

Here is an example that shows how to create an `StdButtonBg` from one of the standard icons:

```
(libload "libimage/image-io")
(libload "libimage/rgbaimage")
(setq icondir (concat-fname lushdir "lsh/libogre/icons"))
(ogre)
(setq btn
  (new StdButtonBg
    "Click Me"
    (image-read-rgba (concat-fname icondir "button-brushed-metal-02-up.png"))
    (image-read-rgba (concat-fname icondir "button-brushed-metal-02-down.png"))
    (image-read-rgba (concat-fname icondir "button-brushed-metal-02-disabled.png"))
    (lambda (c) (printf "coucou\\n"))))
(setq window (new windowobject 10 10 200 200 "test" btn))
(==> btn disable)
(==> btn enable)
```

6.9.3 Exclusive Buttons

Exclusive button are two-state buttons which cooperate with the other buttons in the same container in order to ensure that at most one button is in a positive state.

Class `RadioButton` implements exclusive buttons. Since class `RadioButton` is a subclass of class `CheckBox` , all the slots and methods defined for checkbox and control objects are inherited.

(new RadioButton `label call`)

See: (==> `Control` settext `d`)
See: (==> `Control` gettext)
See: (==> `Control` setdata `d`)
See: (==> `Control` getdata)
See: (==> `Control` setcall `f`)

Returns a new check box whose descriptive text is given by string `label` . You can query or change the state of a check box (`t` or `()`) by sending messages `getdata` and `setdata` . The descriptive text can be modified using `settext` .

You must insert a radiobutton in a container containing only radiobuttons. Whenever the user clicks on a radiobutton, the state of all the other radiobuttons in the container becomes `()` , while the state of the clicked radiobutton becomes `t` and the callback function `call` is called.

Example:

```
;;; A window with a three radio buttons
;;; building a three state choice
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
                ;; create a common callback function
                (let ((call (lambda(d)
                                        (printf "\\%s has been selected\\n"
                                                (==> d gettext) ) )))
                        (new column
                              (new radiobutton "choice 1" call)
                              (new radiobutton "choice 2" call)
                              (new radiobutton "choice 3" call) ) ) ) )
```

Remarks:

- All radiobuttons in a same container are exclusive.
- A container of radiobuttons should not contain objects which are not instances of class `RadioButton` .

6.9.4 File Requester Button

A file requester button is composed of a small square button which contains an icon. When the button is depressed, a file requester is poped up.

A file requester button is usually associated with a string editor, which displays the file selected in the requester.

See: File Requester.

See: Editable Strings.

(new FileReqButton `areqmsg aedstring`**)**

Returns a new file requester button. Argument `areqmsg` is the introductory message of the requester associated with the button. Argument `aedstring` is the string editor associated with the button.

6.10 Character String Objects

Ogre provides several classes for handling character string:

- Class `OgreString` provides a way to display fixed text.
- Class `EditString` provides a way to display a text string that the user can edit using the mouse and the keyboard.
- Class `EditNumber` provides a way to display a numerical text string that the user can edit using the mouse and the keyboard.

- Class `EditSecretString` provides a way to display a secret text string that the user can edit using the mouse and the keyboard.
- Classes `EditText` and `TextPane` implement a multiline text editor. Class `TextPane` is the basic class for the editor or viewer. Class `EditText` manages a text editor and two scrollbars.

6.10.1 Fixed Strings

Class `OgreString` is used for creating a caption text which does not respond to user interaction. Such objects are useful for displaying explanatory text or short message. Class `OgreString` is a subclass of class `VisualObject` . This is not a control object because it has no user interaction capabilities.

(new OgreString `text`)

See: (==> `Control` settext `d`)
See: (==> `Control` gettext)
Creates a new string object displaying text `text` . The text displayed by a string object may be changed using `settext` like the text of a control object.

- Argument `text` may be a string containing or not newline characters. When `text` contains newline characters, a multiline text is displayed.
- Argument `text` may also be a list of strings, one per line.

The text is displayed using a font set by the fonction stored in slot `textfont` of the string object. This font default to a standard 12 points font.

6.10.2 Editable Strings

See: Regular Expressions.
See: Activation of a Control Object.
See: (==> `Control` setdata `d`)
See: (==> `Control` getdata)
See: (==> `Control` setcall `f`)
An editable string is a single line text editor.
An editable string object becomes actived when the user clicks the mouse button above its rectangle. Keypress and arrow events then are bound to editing functions according to a keymap stored in global variable `EditStringKeymap` .
The text edited by an editable string is controlled by the value of the slot `regex` of the editable string. This slot contains a regular expression which must match the text at all times. If an editing action results in a non-matching text, the action is discarded and a beep is emited.

- Class `EditString` implements the editable strings. Since class `EditString` is a subclass of class `Control` , all the slots and methods defined for control objects are inherited.

- Class `EditNumber` implements an editable string intended for entering or displaying numbers.
- Class `EditSecretString` implements an editable string intended for entering secret texts by displaying any character as an 'x'.

(new EditString `minsize` [`defaulttext`])

Creates a new editable string displaying at least `minsize` characters. The initial value of the text is specified by the optional string argument `defaulttext` .

Example:

```
;;; A window with two editable strings
? (setq win
        (new WindowObject 100 100 300 300 "Essai"
                (new grid 2
                       (new string "First string:")
                       (new editstring 16 "number one")
                       (new string "Second string:")
                       (new editstring 16 "number two") ) ) )
```

EditStringKeymap

The global variable `EditStringKeymap` contains the keymap for EditString objects as a list of key bindings. Every key binding has the form `(char action)` where `char` is a one character string and `action` is a symbol identifying a specific editing action.

Every typed character is matched against the keymap.

- If a binding matches the typed character, message action with no argument is sent to the editable string.
- If no binding matches, the character is inserted into the data string at the current cursor location. The cursor is then advanced by one position.

The default keymap is loosely modeled after the Emacs text editor:

```
(setq EditStringKeymap
 '(     ("\\n"          execute)         ;; <LineFeed>
        ("\\r"          execute)         ;; <Enter> or <Return>
        ("\\b"          backspace        ;; <ctrl-H> or <Backspace>
        ("\\x7f"        backspace)       ;; <Delete>
        ("\\^D"         delete-char)     ;; <Ctrl-D>
        ("Delete"      delete-char)     ;; Key <Delete>
        ("\\^A"         begin-of-line)   ;; <Ctrl-A>
        ("Home"        begin-of-line)   ;; Key <Home>
        ("\\^E"         end-of-line)     ;; <Ctrl-E>
```

```
("End"          end-of-line)     ;; Key <End>
("\\^F"          arrow-right)     ;; <Ctrl-F> and <Right>
("\\^B"          arrow-left)      ;; <Ctrl-B> and <Left>
("\\^P"          arrow-up)        ;; <Ctrl-P> and <Up>
("\\^N"          arrow-down)      ;; <Ctrl-N> and <Down>
("\\^K"          kill)            ;; <Ctrl-K>
("\\^Y"          yank) ) )        ;; <Ctrl-Y>
```

The binding second elements, actions, are the names of methods defined by class `EditString` to perform the various editing tasks. The valid actions are:

- Action `execute` calls the callback function. You can install a callback function using message `setcall` because an editable string is a control object.
- Action `backspace` deletes the character preceding the cursor.
- Action `delete-char` deletes the character located right after the cursor.
- Action `begin-of-line` moves the to cursor the beginning of the line.
- Action `end-of-line` moves the cursor to the end of the line.
- Action `arrow-left` moves the cursor one character left.
- Action `arrow-right` moves the cursor one character right.
- Action `arrow-up` recalls the previous string in the history buffer. Editable strings maintain a buffer of the last 20 strings typed by the user.
- Action `arrow-down` recalls the next string in the history buffer.
- Action `kill` deletes the text between the cursor and the end of the string. This text is copied into the clipboard.
- Action `yank` inserts text from the clipboard. The kill and yank actions implement in fact a simple Copy/Paste mechanism between editable strings.

6.10.3 Editable Numbers

See: Regular Expressions.
See: (==> `Control` setdata `d`)
See: (==> `Control` getdata)
See: (==> `Control` setcall `f`)

(new EditNumber `minsize` [`defaultvalue`]**)**

Class `EditNumber` is a subclass of class `EditString` used for editing numeric fields. It checks that the text is a valid number using a regumar expression stored in slot `regex` at construction time. In addition, the methods `setdata` and `getdata` have been modified to handle numbers instead of strings.

Argument `minsize` is the minimal number of characters dislayed in the object. The initial value of the number is specified by the optional number argument `defaultvalue` .

When sent to an editable numeric string, message `setdata` requires a numerical argument. Similarly, message `getdata` returns a number or the empty list when no text has been entered.

6.10.4 EditSecretString

This subclass of `EditString` works exactly like `EditString` but always display the string character as a string of `"x"` .

6.10.5 Multiline Text Editor

See: (==> `Control` setdata `d`)
See: (==> `Control` getdata)
See: (==> `Control` setcall `f`)
See: Forms.

(new EditText `w h` [`default`]**)**

Creates a new edittext object (i.e. a text editor with both a vertical and horizontal scrollbar). Argument `w` is the number of visible columns. Argument `h` is the number of visible lines.

Class `EditText` is a subclass of `Forms` that defines composite object containing a `TextPane` object and two scroll bars. The text editor is actually implemented by class `TextPane` described hereafter.

Class `EditText` forwards messages `setdata` , `getdata` , `read-file` and `write-file` to this underlying `TextPane` object.

See: (new TextPane `w h` [`def editp vs hs`])
See: (==> `TextPane` getdata)
See: (==> `TextPane` setdata `arg`)
See: (==> `TextPane` read-file `fname`)
See: (==> `TextPane` write-file `fname`)

(new TextPane `w h` [`def editp vs hs`]**)**

Creates a new `TextPane` object. Class `TextPane` is a subclass of `Control` . It implements a simple multi-line text editor.

- Argument `w` is the number of visible columns.

- Argument `h` is the number of visible lines.
- Argument `def` specifies a default contents as a string or a list of strings. Single strings are searched for TAB and NL characters and splitted into multiple strings (one per line).
- Flag `editp` tells if the string is editable or just viewable.
- If scrollbars are attached to the text pane, you must pass them as arguments `vs` and `hs` . The text pane will then synchronize the scrollbars with the actual contents of the text pane area.

TextPaneKeymap

This A-list contains the associations between key strokes and editing methods executed by a `TextPane` object. The default keymap is loosely based on the Emacs key bindings.

```
(setq TextPaneKeymap
 '(     ("\\n"     execute)
        ("\\r"     execute)
        ("\\b"     backspace)
        ("\\x7f"   backspace)
        ("\\^A"    begin-of-line)
        ("\\^B"    arrow-left)
        ("\\^D"    delete-char)
        ("\\^E"    end-of-line)
        ("\\^F"    arrow-right)
        ("\\^K"    kill)
        ("\\^N"    arrow-down)
        ("\\^P"    arrow-up)
        ("\\^V"    page-down)
        ("\\^Y"    yank)
        ("Delete" delete-char)
        ("C-Home" begin-of-text)
        ("C-End"  end-of-text)
        ("Home"   begin-of-line)
        ("End"    end-of-line)
        ("Prior"  page-up)
        ("Next"   page-down)
        ("\\x1b"   metakey "\\x1b")
        ("\\x1b<"  begin-of-text)
        ("\\x1b>"  end-of-text)
        ("\\x1bv"  page-up)
 ) )
```

The first element of each association describes the keystroke or keystroke combination. This string can contain an ASCII character, a string containing a keystroke combination, or a function key name. The leading character of keystroke combinations must be associated to method `metakey` . The function key names may be preceded by `"C-"` and `"S-"` to indicate that the control or shift key must be depressed.

The rest of the association specify which action should be called when the corresponding keystroke is typed by the user. The following actions are supported:

- Action `execute` calls the callback function. You can install a callback function using message `setcall` because an editable string is a control object.
- Action `backspace` deletes the character preceding the cursor.
- Action `delete-char` deletes the character located right after the cursor.
- Action `begin-of-line` moves the to cursor the beginning of the line.
- Action `end-of-line` moves the cursor to the end of the line.
- Action `begin-of-text` moves the caret at the beginning of the text.
- Action `end-of-text` moves the caret at the end of the text.
- Action `arrow-left` moves the cursor one character left.
- Action `arrow-right` moves the cursor one character right.
- Action `arrow-up` moves the cursor one line up.
- Action `arrow-down` moves the cursor one line down.
- Action `page-up` moves the cursor one screen up.
- Action `page-down` moves the cursor one screen down.
- Action `kill` deletes the text between the cursor and the end of the string. This text is copied into the clipboard.
- Action `yank` inserts text from the clipboard. The kill and yank actions implement in fact a simple Copy/Paste mechanism between editable strings.
- Action `metakey` is used to declare the prefix of a keystroke combination. The action argument will be combined with the next keystroke and the combined string will be searched in the keymap.

(==> `TextPane` setdata `arg`)

Sets the text in a TextPane. Argument `arg` specifies a default contents as a string or a list of strings. Single strings are searched for TAB and NL characters and splitted into multiple strings (one per line).

(==> `TextPane` getdata)

Returns a list of strings representing all lines of the text pane.

(==> `TextPane` read-file `fname`)

Reads the contents of text file `fname` into the text pane.

(==> `TextPane` write-file `fname`)

Writes the contents of the text pane into file `fname` .

6.11 Icons

An icon is a simple object which displays a greyscale or colormapped picture. The Icon class is somewhat superseded by the Ogrimage class.

6.11.1 (new icon `mat` &optional [`sx` [`sy` [`map`]]])

Returns a new icon.

Argument `mat` is a 2D matrix containing the data to be represented as rectangles using 64 gray levels. By default, the values lower than 0 will be black and the values beyond 1 will be white.

Arguments `sx` and `sy` are the width and height of each rectangle.

Argument `cmap` may be a 1D integer matrix defining 64 colors. When it is defined, its colors are used instead of gray levels.

See: (gray-draw-matrix `x y mat minv maxv apartx aparty`)

See: (color-draw-matrix `x y mat minv maxv apartx aparty cmap`)

6.12 OgrImage

An OgrImage is a simple object which displays a color or greyscale picture. An OgrImage can be used advantageously in combination with the ImageViewer class.

6.12.1 (new OgrImage `m` [`autoscale`])

Return a new OgrImage that will display picture `m` . `m` must be an IDX2 (greyscale image) or IDX3 whose last dimension must be 1 (grayscale), 3 (RGB), or 4 (RGBA). If `autoscale` is true, `OgrImage` will scale the image data to utilize the full dynamic range of the displaying device.

6.12.2 (==> `OgrImage` toggle-autoscale)

Change state of autoscale flag and repaint the image.

6.12.3 (==> OgrImage incr-zoom)

Increment the zoom level.

6.12.4 (==> OgrImage decr-zoom)

Decrement the zoom level.

6.12.5 (==> OgrImage get-selected)

return the rectangle last selected by the user by clicking and dragging the mouse on the image.

6.13 ImageViewer

An ImageViewer is a scrollable viewing area for an OgrImage.

6.13.1 (new ImageViewer w h m [scroll])

Create a new image viewer which will display image `m` in a scrollable viewing area of size `w` , `h` . If the optional `scroll` argument is true, the image scrolling will follow scrollbar dragging, whereas if `scroll` is nil or absent, the scrolling is only performed when the mouse button is released. Here is an example:

```
(libload "libimage/image-io")
(setq m (image-read-rgb (concat-fname lushdir "lsh/libimage/demos/sample.jpg")))
(ogre)
(setq w (new WindowObject 10 10 400 400 "asd" (new ImageViewer 340 340 m t)))
```

6.13.2 (==> ImageViewer get-selected)

return the rectangle last selected by the user by clicking and dragging the mouse on the image.

6.14 Menus

A menu is a popup column associated to a menu button. When the mouse is depressed on the menu button, a popup column appears and the user may drag the mouse over an item in the column. When the mouse button is released, the item is selected, its callback function is executed and the popup column disappears.

Menus are implemented with three classes:

- Class `Menu` defines the menu button. A menu button appears as a bold string preceded by a small upside down triangle. When the menu button is depressed, an instance of class `MenuPopup` is inserted into the menu button's window.
- Class `MenuPopup` is a subclass of class `Column` . It arranges the menu items and manages event messages by selecting the appropriate menu item.
- Class `MenuItem` is a subclass of class `Control` . Menu items may receive `settext` messages for changing their label and `setdata` messages for setting slot `data` . When slot `data` is not the empty list, a check mark appears on the left of the menu item.

The constructor of class `Menu` sets up all these objects in a single call.

6.14.1 Standard Menus

(new Menu `menuname label1 call1` **...** `labelN callN` **)**

Creates a menu button labelled `menuname` linked to a menu popup composed of items defined by the pairs `label1` , `call1` to `labelN` , `callN` . Argument `labeli` is the name of the `i` th menu item. Argument `calli` is the callback associated to the `i` th menu item.

Example:

```
;;; A window with a menu composed of 3 items
? (setq win
      (new WindowObject 100 100 300 300 "Essai"
          (new menu "Test Menu"
                ;; the first item toggles its check mark
                "Toggle"
                (lambda(item) (==> item setdata (not (==> item getdata))))
                ;; the second item activates the bell
                "Bell"
                (lambda(item) (beep))
                ;; the third one destroy this interface
                ;; remenber that <thiswindowobject> always
                ;; refer to the closest window object.
                "Quit"
                (lambda(item) (==> thiswindowobject delete)) ) ) )
```

(==> `Menu` **finditems** `label`**)**

Message `finditems` returns the menu items matching argument `label` .

- If `label` is the empty list `()` , message `finditems` returns a list containing all the menu items.

- If `label` is a number, message `finditems` returns a list containing the `label` -th menu item.
- If `label` is a string, message `finditems` returns a list of menu items whose label string is equal to `label` .

(==> `Menu` disable `...labels...`)

See: (==> `Menu` finditems `label`)

When a menu receives a message `disable` , it sends a message `disable` to all menu items identified by the arguments `...labels...` . Therefore, the corresponding menu items are disabled.

Each argument of message `disable` is interpreted like the argument `label` of message finditems. It can be the empty list, a number or a string.

(==> `Menu` enable `...labels...`)

See: (==> `Menu` finditems `label`)

When a menu receives a message `disable` , it sends a message `disable` to all menu items identified by the arguments `...labels...` . Therefore, the corresponding menu items are disabled.

Each argument of message `disable` is interpreted like the argument `label` of message finditems. It can be the empty list, a number or a string.

(new MenuItem `label call`)

Creates a new menu item labelled `label` . Argument `call` indicates the call-back function for this menu item. Menu items may be inserted into a menu popup by sending a message `insert` to the menu object.

(==> `Menu` insert `menuitem`)

Message `insert` inserts a new menu item `menuitem` in the menu popup.

(==> `Menu` remove `menuitem`)

Message `insert` removes new menu item `menuitem` from the menu popup.

6.14.2 Choice Menus

This class implements a multiple choice item. This object display a menu mark and the name of the currently selected item. When the user clicks on it, a menu popup is displayed and the user can select another item.

Choice menus supports the methods of standard menus, in addition to their own methods.

(new ChoiceMenu [`items` [`callback`]])

Creates a new choice menu that lets the user choose among the items specified in list of strings `items` . When argument `items` is unspecified, selection is impossible.

(==> `choicemenu` **setitems** `items`**)**

Changes the items that the user can select with the choice menu to be the strings of list `items` . As a side effect, the current selection is reset to the empty list.

(==> `choicemenu` **setdata** `d`**)**

Selects string `d` in the choice menu.

(==> `choicemenu` **getdata)**

Returns the string selected in the choice menu.

(==> `choicemenu` **setcall** `callback`**)**

Sets the callback function

(==> `choicemenu` **disable)**

Disables a choicemenu.

(==> `choicemenu` **enable)**

Enables a choicemenu.

6.15 Popup Requesters

A requester is a prepared form containing a collection of graphical objects. When the requester receives a message `popup` , the form is displayed in the middle of the window and receives all event messages until it receives a message `popdown` .

Requesters are useful for asking the user to provide additional information about an action trigerred by a button or a menu item. The callback function of the button or menu item then just send a message `popup` to the requester which performs all the remaining tasks.

- Class `Requester` implements the basic mechanism of a requester.
- Class `ErrorRequester` and `WarningRequester` implement requesters specialized for displaying error messages.

- Class `YesNoRequester` defines a requester with a standard state and two standard buttons for accepting or cancelling an action. These requesters are useful for entering extra information about an action trigerred by a button or a menu item.

- Class `PrintRequester` implements a standard requester for selecting a printer.

- Class `FileRequester` implements a standard requester for entering a filename for reading or writing.

6.15.1 Requester

Class `Requester` is a subclass of `Form` which implements the basic mechanisms of a requester. When the requester receives a message `popup` , the requester is inserted into a predefined window object until reception of a message `popdown` .

Example:

```
;;; A window with a button that pops a requester up.
;;; The requester contains an editstring and two buttons.
;;; 1- Create the window
? (setq win
          (new WindowObject 100 100 300 300 "Essai"
                (setq thebutton
                        (new StdButton "Pop the requester up"
                                (lambda(d) (==> therequester popup)) ))))
= ::WindowObject:06f00
;;; 2- Create the requester
? (setq therequester
        (new Requester win
          (new Column
                ;; The first button pops the requester down
                ;; Variable <thisrequester> always refers to the closest
                (new StdButton "Pop the requester down"
                        (lambda(d)
                          (==> thisrequester popdown)) )
                ;; The second button changes the label of the popup button
                (new EditString 8 "Pop it up again")
                (new StdButton "Change label"
                        (lambda(d)
                          (==> thebutton settext
                                ;; Collective <getdata> on the requester!
                                (car (==> thisrequester getdata)))))))))
= ::Requester:0702c
```

(new Requester `support ...contents...`**)**

Creates a new requester containing objects `...contents...` . This requester will be inserted into the window object containing object `support` upon recpetion of a message `popup` .

(==> `Requester` **popup)**

When a requester receives message `popup` , it is inserted in the middle of the window object defined at requester creation time.

All event messages are then directed to the requester. Since no other component of the interface toplevel is active, it is advisable to insert a button whose action consists in sending message `popdown` to the requester itself.

(==> `Requester` **popdown)**

A popped requester is removed from its window object when it receives a message `popdown` . The usual event dispatching is then restored.

(==> `Requester` **popuplock)**

Message `popuplock` behaves very much like message `popup` . Unlike message `popup` , message `popuplock` blocks until the requester receives a message `popdown` . When message `popuplock` returns, the window object state is normal again.

Example:

```
;;; Pop up the requester defined above in this section and block:
? (==> therequester popuplock)
;;; Lush only returns when you activate the 'popdown' button
= ()
```

(==> `Requester` **popuphard)**

Message `popuphard` is an even more drastic form of `popuplock` . It pops up the requester and blocks until the requester is popped down like message `popuplock` .

While message `popuphard` is active, no other window object accepts user events. If the user click in another Ogre window, the computer beeps and put the window containing the requester above all other window on the screen.

Message `popuphard` is useful for requesting an information which requires immediate attention from the user.

(==> `Requester` **setsupport** `win`**)**

Message `setsupport` changes the support window of a requester.

6.15.2 Warning Requester

Class `WarningRequester` is a subclass of `Requester` . A warning requester contains only an `OgreString` object. The actual text of the string is defined when you pop the warning requester up.

(new WarningRequester `support`**)**

Creates a new warning requester for displaying text in window object `support` .

(==> `WarningRequester` **popup** `text`**)**

Message `popup` first sets the string's text to `text` and pops the requester up. The requester may be popped down by sending the usual message popdown.

Using a warning requester with messages `popup` and `popdown` allows for displaying messages indicating what is being computed.

(==> `WarningRequester` **popuplock** `seconds text`**)**

Message `popuplock` first sets the string's text to `text` and pops the requester up during `seconds` seconds. When this delay is elapsed, the requester is popped down and method `popuplock` returns.

Using a warning requester with popuplock allows for displaying a temporary warning message.

6.15.3 Error Requester

Class `ErrorRequester` is a subclass of `Requester` . An error requester contains an `OgreString` object and a button labelled `"Ok"` for popping the requester down. The actual text of the string is defined when you pop the error requester up.

Such a requester is very useful for signaling an error to the user. The user must depress the button `"Ok"` to remove the requester. This button actually sends a message `popdown` to the error requester.

Example:

```
;;; Assuming window object <win> already exists:
? (setq error-dialog (new ErrorRequester win))
= ::ErrorRequester:07120
? (==> error-dialog popup "Error message")
= ()
```

(new ErrorRequester `support`**)**

Creates a new error requester for displaying error messages in window object `support` .

(==> `WarningRequester` popup `text`)

Message `popup` first sets the string's text to `text` and pops the requester up. The requester may be popped down by sending the usual message popdown.

Using a warning requester with messages `popup` and `popdown` allows for displaying messages indicating what is being computed.

(==> `WarningRequester` popuplock `text`)

Message `popuplock` first sets the string's text to `text` and pops the requester up. The requester is popped down and method `popuplock` returns when the user depresses button `"Ok"` .

6.15.4 Yes/No Requester

Class `YesNoRequester` is a subclass of `Requester` for implementing yes/no requesters. A yes/no requester contains a user defined part, a positive button and a negative button.

Like usual requesters, yes/no requesters are popped up when they receive a message `popup` or `popuplock` . Both the positive and the negative button pop the yes/no requester down.

When you create the yes/no requester however, you specify a callback function.

- When the positive button is depressed, the yes/no requester is popped down and the callback function is executed.
- When the negative button is depressed, the yes/no requester object is popped down and the state of the user defined part of the requester is restored to the state recorded when the requester was popped up. The callback function is not executed.

If you have popped up the requester using message `popuplock` , you can also test which button has been depressed by looking at the value returned by message `popupdown` . This value is the empty list if the negative button has been depressed.

Slots `yesbutton` and `nobutton` of a yes/no requester contain the positive and negative buttons. These buttons may be programmatically actived by sending them a message `trigger` .

(new YesNoRequester `support dialog yes no call`)

Creates a new yes/no requester for displaying on the window object of object `support` .

Argument `dialog` is the user defined part of the requester. String `yes` is the label of the positive button. String `no` is the label of the negative button. Argument `call` is a callback function or the empty list.

Example:

```
;;; A button that pops up a yesnorequester
;;; for changing its label...
? (setq win
        (new WindowObject 100 100 400 200 "Essai"
                (setq thebutton
                        (new StdButton "Hello"
                                (lambda(c) (==> theyesnoreq popup)) ))))
= ::windowobject:07010
? (setq theyesnoreq
        (new YesNoRequester win
                ;; The user defined part
                (new Column
                        (new OgreString "Change button label")
                        (new EditString 8 "new label") )
                ;; The yes and no labels
                " Ok " "Cancel"
                ;; The callback
                (lambda(caller)
                        (==> thebutton settext
                                (car (==> caller getdata)) )))))))
= ::yesnorequester:07040
```

(==> YesNoRequester settext yes no default)

The label of the positive and negative buttons can be changed by sending a message `settext` to the yes/no requester. String `yes` becomes the label of the positive button. String `no` becomes the label of the negative button.

There is usually a default button indicated by a wider outline. This button is triggered if the user hits the carriage return key. This default button is usually the positive one, but may be changed with argument `default` of message `settext` .

- If `default` is `'yes` , the positive button is the default,
- if `default` is `'no` , the negative button becomes the default,
- if default is the empty list, no default button is set.

(==> YesNoRequester setcall callback)

Sets the callback function for the requester. Argument `callback` can be a function with one argument or the empty list.

(==> YesNoRequester ask yes no default)

See: (==> `YesNoRequester` settext `yes no default`)

See: (==> `Requester` popuphard)

This message should be used with yes/no requesters whose user defined component is a single `OgreString` object.

When such a requester receives message `ask` , it first sets the text of the string object to `text` and the button labels to `yes` and `no` . The default button is indicated by argument `default` .

The requester is then popped up using `popuphard` . The user can then press either the positive or the negative button. When the requester is popped down, message `ask` returns `t` or `()` according to the user answer.

Example:

```
;;; Creates a confirmation dialog in an exiting win <win>.
? (setq confirm-dialog
        (new YesNoRequester win
                (new string "msg")          ; dummy message
                "yes" "no"                  ; dummy labels
                () )                        ; no callback
= ::yesnorequester:070c0
? (==> confirm-dialog ask
        "Should I really do that"           ; the message
        "Proceed" "Please don't"            ; the button labels
        'no )                               ; the default
;;; Waiting for your answer...
= t or ()
```

6.15.5 Print Requester

See: (print-window `w h` [`destination`])

A print requester lets the user choose a valid destination for printing a graphics. Function `getdata` will return a string that can be used verbatim as the `destination` argument of function `print-window` .

(new PrintRequester `w` [`callback`]**)**

See: Yes/No Requester.

Creates a new print requester for displaying in the window object `w` . The optional argument `callback` defines a callback function that will be called if the user presses button "Ok" (as usual with Yes/No Requesters.)

(==> `PrintRequester` **getdata)**

See: (print-window `w h` [`destination`])

Returns a string that can be used directly as argument `destination` of function `print-window` .

6.15.6 File Requester

A file requester lets the user choose a filename using a list of files existing in various directories. The file requester contains a message string, a file selector, an editable filename string and two buttons labelled `"Ok"` and `"Cancel"` .

Starting from the current filename, the selector displays the contents of the directory. A first click on a selector item selects a file or directory and copies its name in the editable filename string. Alternatively, the user can type a filename in the filename field. The button `"Ok"` is disabled if the user types an invalid filename.

This selection is validated by a second click, by pressing the enter key or by depressing the button `"Ok"` .

- If the selected item is a directory, the contents of this directory is displayed and the user can select again an item.
- If the selected item is a file, the requester is popped down and the callback function is executed.

Class `FileRequester` is a subclass of class `YesNoRequester` which implements the standard Ogre file requester.

(new FileRequester `w` [`message flag filter callback`])

See: Yes/No Requester.

Creates a new file requester for displaying in the window object `w` . The contents of the message string is indicated by the optional argument `message` . The callback function is indicated by the optional argument `callback` or may be set using message `setcall` .

Special behavior are selected by argument `flag` :

- When `flag` is `'no-newfile` , it is forbidden to enter the name of a non existent file.
- When `flag` is `'ask-newfile` , validating the name of a non existent file pops up a confirmation requester. This is useful before creating a new file.
- When `flag` is `'ask-oldfile` , validating the name of an existent file pops up a confirmation requester. This is useful before overwriting an existing file.

  Argument `filter` is a function with one argument for testing the filenames. It returns a non nil value if its argument is a valid file name for this file request. For instance, this function might test an extension or examine the file header. Files rejected by the function `filter` are not displayed in the selector. If the user types such an invalid filename in the filename field, validation is prevented by disabling the button `"Ok"` .

(==> `FileRequester` setdata `fname`)

Sets the current file name of a `FileRequester` object to file or directory `fname` .

(==> `FileRequester` getdata)

Gets the full current file name of a `FileRequester` object.

(==> `FileRequester` getdir)

Gets the name of the directory displayed in the selector of a `FileRequester` object.

(==> `FileRequester` getbase)

Returns the base name of the current filename selected in a `FileRequester` object. If the selected filename is a directory, this message returns the empty list.

(==> `FileRequester` setparm `message flag` [`filter`])

Changes the parameters of a `FileRequester` object.

(==> `FileRequester` ask `message flag` [`filter` [`fname`]])

Changes the parameters of a `FileRequester` object, sets the current filename to file or directory `fname` , pops up the requester and waits until the request is complete.

- If the user validates a file name, a non nil value is returned. The filename can be accessed by sending a `getdata` message.
- If the user selects the Cancel button, the empty list is returned.

(ogre-ask-file `winobj message flag` [`filter` [`fname`]])

This function popups a default filerequester in window `win` .
Other arguments are the same as for method `ask` of class `FileRequester` .
See: (==> `FileRequester` ask `message flag` [`filter` [`fname`]])

6.16 Movable and Resizable Objects

Class `DragArea` and `SizeArea` define objects used for moving or resizing their container. Special container classes `Frame` and `FrameSize` define containers which may be moved and resized by the user.

6.16.1 (new DragArea w h)

Creates a new dragging area with minimal width `w` and minimal height `h` .

A dragging area appears as a rectangle with a gray outline. When you depress the mouse inside a dragging area, you can drag its container to another place.

Two slots affect the behavior of a dragging area:

- The moved object always remains inside rectangle `constraintrect` . This rectangle defaults to the empty list `()` which means that the object remains confined within its container's rectangle.
- Slot `magnet` should contain a list `(mx my)` . The moved object keeps aligned on an invisible grid whose columns are mx pixels wide and whose rows are my pixels high. It defaults to the empty list `()` which means that objects may be moved at any pixel position.

6.16.2 (new SizeArea w h)

Creates a new sizing area with minimal width `w` and minimal height `h` .

A sizing area appears as two overlapping gray squares. When you depress the mouse inside a sizing area, you can change the size of its container. Just drag the mouse until you touch a container boundary. This container boundary then follows the mouse until you release the mouse button.

Two slots affect the behavior of a size area.

- The resized object always remains confined within the rectangle `constraintrect` . This rectangle defaults to the empty list `()` which means that the object remains confined within its container's rectangle.
- Slot `formfactor` should contain a list `(fx fy)` . The width of resized object remains an integer multiple of `fx` , the height of the resized object remains an integer multiple of `fy` , and both coefficients are equal. Slot `formfactor` defaults to the empty list `()` which means that no form factor constraint apply.

6.16.3 (new Frame x y w h ...contents...)

Creates a new frame at location `x` and `y` with minimal width `w` and minimal height `h` . The frame initally manage objects `contents` .

Class `Frame` is a subclass of class `Container` . A frame always contains a dragging area located in the background of the container. A frame thus is a container that the user can move.

Example:

```
? (setq win (new windowobject 0 0 400 400 "Essai"
                (new Frame 50 30 200 140
```

```
                        (new stdbutton "Beep"
                                (lambda(c) (beep)) ) ) ))
```

6.16.4 (new FrameSize x y w h ...contents...)

Creates a new sizable frame at location `x` and `y` with minimal width `w` and minimal height `h` . The frame initally manage objects `contents` .

Class `FrameSize` is a subclass of class `Frame` . A sizable frame always contains a dragging area located in the background of the container. A sizable frame also contains a small size area in its bottom right corner. A sizable frame thus is a container that the user can move and resize.

Example:

```
? (setq win (new windowobject 0 0 400 400 "Essai"
                (new FrameSize 50 30 200 140
                        (new stdbutton "Beep"
                                (lambda(c) (beep)) ) ) ))
```

6.17 Sliders and Scrollbars

Sliders and scrollbars are graphical objects for entering numerical values.

- Sliders are rendered as symbolized potentiometers. Sliders are mostly used as a way to enter numerical values.
- Scrollbars are rendered as a gray area with a white handle. Scrollbars are mostly used for controlling other objects.

The user can grab and move the handle with the mouse. The user might also maintain the mouse button depressed on either side of the handle. In this case, the handle moves slowly towards the mouse pointer.

The standard callback function is called whenever the user releases the mouse button. An additional callback might be set up with message `setdrag` . This callback is called whenever the user moves the handle.

Sliders and scrollbars are implemented by a set of specialized subclasses of class `Control` .

```
Control                 abstract class for all controls
  Slider                abstract class for all sliders and scrollbars
    HSlider               horizontal sliders
    VSlider               vertical sliders
    Scrollbar           abstract class for all sliders and scrollbars
      HScrollbar          horizontal scrollbars
      VScrollbar          vertical scrollbars
```

Example: Example:

```
? (setq win (new windowobject 100 100 400 200 "Sliders & Scrollbars"
                 (new row
                   (new grid 2
                    (new emptyspace 100 100)
                    (new vslider 0 100 ())
                    (new hslider 0 100 ())
                    (new emptyspace 10 10) )
                   (new grid 2
                    (new emptyspace 100 100)
                    (new vscrollbar 100 ())
                    (new hscrollbar 100 ())
                    (new emptyspace 10 10) ) ) ) )
```

6.17.1 Sliders

Sliders are used for entering numerical values using an analog handle. Horizontal sliders are implemented by class `HSlider` . Vertical sliders are implemented by class `VSlider` . Both class `HSlider` and `VSlider` are subclasses of class `Slider` which implement a number of useful methods.

(new HSlider `mini maxi` [`callback`])

Returns a horizontal slider for entering integer values in the range `mini` to `maxi` . The callback function `callback` is called whenever the user releases the mouse button.

(new VSlider `mini maxi` [`callback`])

Returns a vertical slider for entering integer values in the range `mini` to `maxi` . The callback function `callback` is called whenever the user releases the mouse button.

(==> `Slider` **setrange** `mini maxi`**)**

See: (==> `Scrollbar` setrange `mini maxi` [`prop`])

Message `setrange` redefines the minimal and maximal values allowed in a slider or a scrollbar.

(==> `Slider` **setstep** `step`**)**

Message `setstep` redefines the possible increments of the values allowed in a slider or a scrollbar.

The initial increment is always 1. This initial increment ensures that the slider or scrollbar is limited to integer values. Specifying the empty list as an increment means that any value in the legal range are allowed.

(==> `Slider` setdrag `call`)

Message `setdrag` sets up a secondary callback function which is called whenever the handle moves. Installing such a callback function is useful for displaying echo to the user.

6.17.2 Scrollbars

Scrollbars are a special kind of slider which are used for controling which information is displayed in other objects.

Scrollbars are implemented by classes `HScrollbar` and `VScrollbar` which are indirect subclasses of class `Slider` . All the methods defined by class `Slider` are thus inherited by scrollbars.

(new HScrollbar `maxi` [`callback`])

Returns a horizontal scrollbar for entering integer values in the range `0` to `maxi` . The callback function `callback` is called whenever the user releases the mouse button.

(new VScrollbar `maxi` [`callback`])

Returns a vertical scrollbar for entering integer values in the range `mini` to `maxi` . The callback function `callback` is called whenever the user releases the mouse button.

(==> `Scrollbar` **setrange** `mini maxi` [`prop`])

See: (==> `Slider` setrange `mini maxi`)

Message `setrange` redefines the minimal and maximal values allowed in a scrollbar. When the optional argument `prop` is provided, the maximal value is reduced by `prop` and the value `prop` is used for defining the size of the knob. This is useful for scrolling lists of texts.

6.18 Composite Objects

This chapter describes a few standard objects composed of several elementary objects. Although composite objects usually are subclasses of `Form` , they obey the protocols defined for class `Control` .

6.18.1 Viewers

A viewer object display a rectangular portion of a particular object (the viewer's contents). The size of the visible part of this object depends on the viewer size. The user control which part of the object is visible by sending messages `setpos` or by using the optional scrollbars provided by the viewer.

Class `Viewer` implements viewer objects.

(new Viewer `w h contents` [`hp vp`]**)**

Creates a new viewer on object `contents` . The viewer is `w` points wide and `h` points high. The optional argument `hp` (resp. `vp`) is a boolean value (`t` or `()`) controling whether an horizontal (resp. vertical) scrollbar is displayed or not.

Example:

```
? (setq win (new WindowObject 0 0 400 200 "Essai"
                (new Viewer 300 150
                        (new column (new stdbutton "One" ())
                                    (new emptyspace 60 60)
                                    (new stdbutton "Two" ())
                                    (new stdbutton "Three" ())
                                    (new stdbutton "Four" ()) )
                         t t) ) )
```

(==> `Viewer` **setpos** `h v`**)**

Message `setpos` defines the visible portion of the contents of a viewer. This portion is a rectangle whose top left corner is located `h` pixels to the right and `v` pixels below the top left corner of the contents of the Viewer object.

(==> `Viewer` **sethpos** `h`**)**

Message `sethpos` just change the horizontal coordinate of the visible portion of the contents of a viewer.

(==> `Viewer` **setvpos** `v`**)**

Message `setvpos` just change the vertical coordinate of the visible portion of the contents of a viewer.

(==> `Viewer` **setcontenu** `object`**)**

Message `setcontenu` changes the object viewed through a Viewer object. Argument `object` must be a valid graphic component.

6.18.2 Selectors

A selector is an object which allow the user to select one or several strings within a list of strings. Only a few strings are visible at a given time. A scrollbar located along the right side of the Selector object controls which strings are visible. Selected strings are hilighted.

Class `Selector` implements a selector object. The complete behavior of a selector is controlled by three slots of the selector object: `multiple` , `call1` and `call2` .

- When slot `multiple` contains a non nil value, the user might select multiple items. In this case, message `getdata` sets and message `setdata` returns the list of the selected strings.

- If slot `multiple` contains the empty list, one item at most can be selected at a given time. Message `getdata` sets and message `setdata` returns the selected string or the empty list.

Two callback functions are called when the user selects items.

- Callback `call1` is called whenever the user selects an item by clicking the mouse button over an unselected string.

- Callback `call2` is called whenever the user selects an item a second time by clicking the mouse button over a hilighted string.

(new Selector `nvisible` [`callback` [`items`]])

Creates a new selector able to display `nvisible` strings at once.

Argument callback of the constructor controls the values of these flags.

- If argument `callback` is `t` , the user might select multiple items.

- If argument `callback` is a function, one item at most can be selected at a given time. The specified callback function is called whenever the user selects a new item.

- If argument `callback` is a non-nil list, one item at most can be selected at a given time. The list is assumed to be made of two functions or nil values. The first one is used whenever the user selects a new item. The second one is used whenever the user selects again a selected item.

Argument `items` optionally gives a list of strings initially displayed in the selector.

(==> `Selector` **getdata)**

Message `getdata` returns the selected strings in a selector:

- If slot `multiple` contains a non nil value, the user can select multiple items. In this case, message `getdata` returns the list of the selected strings.

- If slot `multiple` contains the empty list, the user can select at most one item. Message `getdata` then returns the selected string or the empty list if no string is selected.

(==> `Selector` getdatanum)

Instead of returning the strings themselves, like message `getdata` , message `getdatanum` returns the order number of the selected strings in a selector:

- If slot `multiple` contains a non nil value, the user can select multiple items. In this case, message `getdatanum` returns the list of the order numbers of selected strings.
- If slot `multiple` contains the empty list, the user can select at most one item. Message `getdata` then returns the order number of the selected string or the empty list if no string is selected.

(==> `Selector` setdata `data`)

Message `setdata` sets which strings are selected in a selector. The selected strings are displayed on a highlited background.

- If slot `multiple` contains a non nil value, argument `data` must be a list of strings or a list of numbers indicating which items must be selected.
- If slot `multiple` contains the empty list, argument data is the empty list (to desselect the selected item), a string or a number indicating which string must be selected.

(==> `Selector` setpos `pos`)

Message `setpos` makes the `pos` -th string appear on top of the selector and updates the scrollbar in accordance.

(==> `Selector` setitems `items`)

Message `setitems` redefines a new list of strings `items` displayed in a selector and clears the current selection.

Since the minimal size of a selector depends on the width of the largest string in the item list, sending message `setitems` can trigger a geometry negociation and readjust the location of all interface components.

6.19 A Complete Example

Whole graphics interface are often defined as subclasses of class `WindowObject` . Such subclasses usually contain new slots for storing interface specific data. Callback functions then send messages to `thiswindowobject` . These messages then are executed within the interface scope.

6.19.1 The Class Browser "classtool"

This section describes a complete application of the Ogre library. This application is nothing but the standard Tlisp class browser. This class browser is invoked by typing the command `(classtool)` .

(classtool [`cl`]**)**

Invokes the Lush class browser on class `cl` . The class browser displays the subclasses, superclasses, slots and methods of a class. Class `object` is assumed if no argument is specified.

The class browser interface is composed of a menu and six selectors:

- Selector "Sub-Classes" displays the names of the subclasses of class `cl` . A mouse click on one of these subclasses changes the current class to the selected class.
- Selector "Slots" displays the names of all slots defined by class `cl` .
- Selector "Method" displays the names of all method selectors defined for class `cl` .
- Selector "Super-Classes" displays the names of the successive superclasses of `cl` . A mouse click on one of these subclasses changes the current class to the selected class.
- Selector "Inherited Slots" displays the names of all slots defined by the successive superclasses of class `cl` . A mouse click on a method item prints the definition of the selected method.
- Selector "Inherited Methods" displays the names of all methods inherited from the successive superclasses of class `cl` . A mouse click on a method item prints the definition of the selected method.

The top of the class browser contains a menu and an information string. The information string displays the number of slots, the number of inherited slots, the number of methods and the number of inherited methods for the current class. The name of the menu is always the name of the current class, displayed in large characters.

The menu itself contains four items:

- Selecting item `"Show Class"` prints the definition of class `cl` .
- Selecting item `"Show Subtree"` prints an indented list of the subclasses of class `cl` .
- Selecting item `"Refresh"` reads again the information for class `cl` and reflects possible changes on the display.

- Selecting item `"Select"` pops up a requester which lets the user enter the name of a class. The class browser jumps to the selected class when the user presses button `"Ok"` . If the typed class name is not a valid class name, a message is displayed.

6.19.2 The Program "classtool"

Here is a review of the main components of the "classtool" program. You can look at the complete listing of this program in file:

```
<lushdir>/lib/classtool.lsh
```

\vspace 5mm \bf i) Initial definitions.

The first executable line of file `"classtool.lsh"` initializes the Ogre library by calling function `ogre` . This is necessary to ensure that the Ogre class library is properly loaded and initialized.

```
(ogre)
```

Then we define a subclass `c-classtool` of class `WindowObject` . This class contains several slots for referencing the major components of the interface.

```
(defclass c-classtool windowobject
  the-menu                ;; the menu
  the-string              ;; the information string
  the-i-classes           ;; the superclass selector
  the-i-slots             ;; the inherited slots selector
  the-i-methods           ;; the inherited methods selector
  the-classes             ;; the subclass selector
  the-slots               ;; the slots selector
  the-methods             ;; the method selector
  the-error-requester     ;; a signaling requester
  the-class-requester     ;; the requester
  cl )                    ;; the current class
```

\vspace 5mm \bf ii) Constructor Method.

We define then the constructor of class `c-classtool` . This constructor first calls the constructor of its superclass `WindowObject` and defines the contents of the window object. This very long call sets up the major components of the classtool interface.

All the interface is a single column which contains:

- A row implements the menu bar. The menu bar contains only one menu object (stored in slot `the-menu`) and an information string (stored in slot `the-string`).

The menu defines the five items documented above. The callback functions of the menu items do not perform very much. They merely send an appropriate action message to the interface itself (acceded through variable `thiswindowobject`) or pop up a suitable the requester.

```
(setq the-menu
    (new Menu "object"
       "Show Class"
       (lambda(c) (==> thiswindowobject display-action))
       "Show Subtree"
       (lambda(c) (==> thiswindowobject subtree-action))
       "Refresh"
       (lambda(c) (==> thiswindowobject refresh-action))
       "Select"
       (lambda(c) (==> the-class-requester popup))
       "Quit"
       (lambda(c) (==> thiswindowobject delete)) ) )
```

- A dark space object provides a clear separation between the menu bar and the rest of the interface. Although the minimal space allocated for the dark space 3 points wide, the column layout policy makes this space as wide as the column itself.

- A grid with 3 columns contains all the other objects. It includes the selector titles (string objects) and the selector themselves which are stored in appropriate slots of the classtool object. The callback functions of the selector do not perform much. They just send messages `classes-action` or `methods-action` to the classtool object itself.

  The grid also include three emptyspace object which specify a minimal size for the grid columns. This technique avoids troublesome geometry changes because it ensures that the selectors are already wide enough for displaying most names.

  The constructor of class `c-classtool` then adjust the font used in the menu title by directly poking into the object slot `textfont` . It calls then method `compute-geometry` to ensure that the object size is adjusted for the new font.

```
(setq :the-menu:textfont font-18)
(==> the-menu compute-geometry)
```

The constructor of class `c-classtool` then creates two requesters.

- The error requester `the-error-requester` is used later for displaying error messages.

```
(setq the-error-requester
      (new ErrorRequester this) )
```

- The class requester is popped up when you select menu item `"select"` for directly entering a class name. It defines a callback function which just sends a message `select-action` to the classtool object.

  We poke a new regular expression in the slot `regex` of the editable string in order to ensure that the text typed by the user is a valid symbol.

```
(setq the-class-requester
      (new YesNoRequester this
           (new column
                (new OgreString "Type a class name")
                (new DarkSpace 3)
                (let ((x (new EditString 20)))
                  (setq :x:regex "[A-Za-z]?[-_|A-Za-z0-9]*") x) )
           " Ok " "Cancel"
           (lambda(c) (==> thiswindowobject select-action)) ) ) )
```

\vspace 5mm \bf iii) Method "setclass".

Method `setclass` is then defined. This method collects the class information for the selected class and updates the information displayed in the selectors.

It first checks that its argument is a valid class and pops up the error requester if this check is negative.

```
(defmethod c-classtool setclass(c)
  (if (not (and c (classp c)))
      (==> the-error-requester popup "This is not a valid class")
```

If the check is positive, method `setclass` displays a message `"working"` in the message string and force an immediate display update using message `repair-damaged` .

```
    (==> the-string settext "<<working>>")
    (==> this repair-damaged)
```

The menu title is then changed to the class name using method `settext` . Since the display update is delayed until all events are processed, this change becomes visible when all selectors are updated.

Method `setclass` then collects the class information into six lists: the subclass list (`cc`), the slot list (`cs`), the method list (`cm`), the superclass list (`ic`), the inherited slot list (`is`) and the inherited method list (`im`). We take a particular care of separating the various inherited classes by a dummy entry in the inherited lists.

This information is then loaded into the selectors using method `setitems` . The message string is then updated to the class statistics.

```
(==> the-classes setitems cc)
(==> the-slots setitems cs)
(==> the-methods setitems (sort-list cm >))
(==> the-i-classes setitems ic)
(==> the-i-slots setitems is)
(==> the-i-methods setitems im)
(==> the-string settext
     (sprintf "  \\%l: \\%d+\\%d slots, \\%d+\\%d methods"
              cn (length cs) isc (length cm) imc) ) ) ) )
```

\vspace 5mm \bf iv) Action methods.

We define then the various action methods which are called by the menu items, by the selectors, or by the class selection requester.

Method `classes-action` is called when the user selects a class in the subclass or superclass requester. This method just sends a message `setclass` to switch to the new class.

```
(defmethod c-classtool classes-action(c)
  (let ((cn (==> c getdata)))
    (==> this setclass (apply scope (list (named cn)))) ) )
```

Method `methods-action` is called when the user selects a method in the method selector or the inherited method selector. Method `methods-action` first makes sure that the user did not select a dummy entry used for separating the classes in the inherited method selector. It uses then function `pretty-method` to print the definition of the selected method.

```
(defmethod c-classtool methods-action(c)
  (let ((m (==> c getdata))
        (cl cl))
    (when (<> (left m 5) "=====")
          (setq m (named m))
          (while (and cl ~(member m (methods cl)))
            (setq cl (super cl)) )
          (when cl
                (==> this repair-damaged)
                (==> c setdata ())
                (print)
                (pretty-method cl m) ) ) ) )
```

Method `refresh-action` is called by the menu item `"Refresh"` . It just calls method `setclass` on the current class (from slot `cl`) in order to reload the class information.

```
(defmethod c-classtool refresh-action()
  (==> this setclass cl) )
```

Method `display-action` is called by the menu item `"Show Class"` . It just prints the class definition when adequate.

```
(defmethod c-classtool display-action()
  (if (not (super cl))
      (beep)
    (print)
    (pprint (nconc (list 'defclass (classname cl) (classname (super cl)))
                   (slots cl) )) ) )
```

Method `subtree-action` is called by the menu item `"Show Subtree"` . It just prints an indented list of the subclasses of the current class.

```
(defmethod c-classtool subtree-action()
  (let ((subtree (lambda(cl tb)
                   (tab tb)
                   (print (classname cl))
                   (each ((c (subclasses cl)))
                         (subtree c (+ tb 2)) ) ) ))
    (print)
    (subtree cl 2) ) )
```

Method `select-action` is called when the class selection requester pops down. It gets the class name typed by the user (using `getdata`), normalize the name and calls method `setclass` again.

```
(defmethod c-classtool select-action()
  (let ((cn (car (==> the-class-requester getdata))))
    (if (regex-match "\\\\|.*\\\\|" cn)
        (setq cn (regex-subst "^\\\\|(.*)\\\\|$" "\\%0" cn))
      (setq cn (downcase cn)) )
    (==> this setclass (apply scope (list (named cn)))) ) )
```

\vspace 5mm \bf v) Miscellaneous.

We override then method `keypress` in order to define an accelerator key. Depressing the space bar pops up the class selection requester. All other keypress events are handled by the default method which just ignore these events.

```
(defmethod c-classtool keypress(c x y)
  (selectq c
    (" " (==> the-class-requester popup))
    (t (==> this (windowobject . keypress) c x y)) ) )
```

Finally the hook function `classtool` creates an instance of class `c-classtool` and sets the initial class displayed in the browser

```
(de classtool( &optional (cl object) )
    (when (symbolp cl)
          (setq cl (eval cl)) )
    (when (not (classp cl))
          (error t "Not a class" cl) )
    (let ((w (new c-classtool)))
      (==> w setclass cl) ) )
```

In addition, an autoload function `classtool` is defined in file `"stdenv.lsh"` which loads file `"classtool.lsh"` and calls this function `classtool` .

Chapter 7

Dynamic Loader/Linker

One of the coolest features of Lush is its dynamic loader/linker. The dynamic loader/linker allows to load object files (.o, .so, or .a) and make their functions easily accessible from the interpreter. Functions in dynamically loaded object files can be written in any language (though C is preferred).

Object files can be dynamically loaded into Lush using the `(mod-load "mylibrary.o")` construct.

Let's say you have written a C file called `tt` titi.c `/tt` with the following content:

```
float sq(float x)
{
  return x*x;
}
```

You have compiled the file and produced the object file `titi.o` . Calling `sq` from the Lush interpreter is as simple as the following. First, dynamically load the object file into Lush

```
? (mod-load "titi.o")
```

then, write a compiled lisp function whose only purpose is to call `tt` sq `/tt` . To be able to call C from Lush:

```
? (de square (x) ((-float-) x)
      (cheader "extern float sq(float);")
      (float #{ sq( $x ) #} )
? (dhc-make "junk" square)
```

The function square can now be called:

```
? (square 5)
= 25
```

Dynamically loaded Modules can be .o object files, but also .a files (static libraries), or .so files (shared objects). Functions in dynamically loaded modules can call any external function or variable defined or used by Lush as well as external functions and variable defined in other modules. In particular, the C library functions are accessible. A function defined in a module however is not always executable. Indeed, its module might call an undefined function, or a function defined by another non executable module. In fact, four situations occur:

- Initialized and executable modules are the only accessible modules. All functions referenced by these modules have been found, and the initialization routine (e.g. `init_essai`) has been succesfully executed, creating descriptors for the new lisp functions defined in the modules. At this point, all new lisp functions defined by such a module are accessible and work as expected.

- Uninitialized modules reference some undefined functions, or some functions defined by a non executable module. Therefore, the initialization function has not been executed, and the descriptors for the new lisp functions defined by such a module have not been created.

- Modules may be initialized but non executable. Such a situation occurs when a module has been initialized and executable, but is no longer executable, because it uses some function or global variable which is no longer defined, because its module has been unloaded.

- Finally, certain modules do not define an initialization function. Such modules just define C functions used by other modules. We say that such a module is in a unknown state.

7.1 (mod-load `filename`)

This function loads a piece of binary code into LUSH. It can be used to load various kind of files containing object code:

- Shared libraries. These files usually have extension `".so"` , `".sl"` , `".dll"` or `".dylib"` depending on the operating system. Use `(getconf "SOEXT")` to determine which extension is valid on your system.

- Object files. These files usually have extension `".o"` or `".obj"` . Use `(getconf "OBJEXT")` to determine which extension is valid on your system.

- Library archives containing a collection of object files. Function `mod-load` only loads the components of a `.a` file that provide definitions for currently undefined symbols. You may want to use functions `mod-create-reference` to create fake undefined symbols and selectively load parts of the library before actually using them. WARNING: C++ static constructors are not called when a `.a` file is loaded. If you experience problems calling functions defined in a dynamically loadad `.a` files written in C++, you should first convert the `.a` file to a `.so` file (static constructors in `.so` files are called).

WARNING: Lush on Mac OS X has some peculiarities. First, `mod-load` loads object files by first creating a equivalent "bundle" file. These bundle files are stored in directory `"/tmp"` . Second, `mod-load` has no support for loading library archive. The best course of action is to first transform them into dynamic library (dylibs).

7.2 (mod-unload `filename`)

This function removes a piece of binary code previously loaded with `mod-load` . Only object files (extension `".o"`) and library archives (extension `".a"`) can be safely removed. It is currently not possible to unload a shared library module (usually a file with extension `".so"`) because these files are dealt with using operating system facilities that seldom provide unloading support.

This operations encompasses three steps:

- Destroying all lisp functions previously defined by the module. Calling such functions will cause an error in the future.
- Relinquishing the memory utilized by the module.
- Checking the executability of all loaded modules, and mark the lisp functions defined by a module as partially linked, if this modules is no longer executable.

7.3 (find-shared-library `name` [`extlist`])

Returns the pathname of a shared library named `name` . Shared libraries are searched in the directories specified by variable `shared-library-path` which is initialized by `"stdenv.lsh"` .

The optional argument `extlist` is a list of possible filename extensions. The default value is either null (when `name` already contains an extension) or the system dependent filename extension for shared libraries.

```
? (find-shared-library "libm")
= "/usr/lib/libm.so"
```

7.4 (find-static-library name [extlist])

Returns the pathname of a static library named `name` . Static libraries are searched in the directories specified by variable `static-include-path` which is initialized by `"stdenv.lsh"` .

The optional argument `extlist` is a list of possible filename extensions. The default value is either null (when `name` already contains an extension) or the system dependent filename extension for static libraries.

```
? (find-static-library "libm")
= ()
```

7.5 (find-shared-or-static-library name [extlist])

Returns the pathname of a library named `name` . A shared library is first searched with the supplied extension list `extlist` . Otherwise a static library is searched.

```
? (find-shared-or-static-library "libm")
= "/usr/lib/libm.so"
```

7.6 (mod-list)

This function returns the list of the currently loaded modules.

Example:

```
? (mod-list)
= ("/home/leonb/lush/src/lush" "/home/leonb/test/essai.o")
```

7.7 (mod-undefined)

This function returns a list with the names of all undefined C functions and global variables in the current modules.

Example:

```
? (mod-undefined)
= ("compute_squares" "numbers_of_squares")
```

7.8 (mod-status)

Displays a summary of all loaded modules.

7.9 (mod-inquire filename)

This function returns a list describing the status of a loaded module defined by the object file `filename` . The first element of this list is a string describing the states of a module. When the initialization function has been called, the names of the new lisp functions are provided in the remaining part of this list.

7.10 (mod-create-reference string1 ... stringN)

Creates a fake undefined symbol that will be considered when loading libraries (archive files like `"foo.a"`). The loader indeed only loads the library components which define symbols currently undefined. You may want to use functions `mod-create-reference` to create fake undefined symbols and load certain parts of the library before actually using them.

7.11 (mod-compatibility-flag boolean)

The old loaded was based on the DLD-3.2.3 library. This library had significant bugs in the code checking the executability of a module. The new loader implements these checks properly. This righful code may prevent you to load your old files. You can use function `mod-compatibility-flag` with a non nil argument to loosen the checks until the new system is almost as buggy as the old one.

7.12 Low-Level Module Functions

Most of the above functions are in fact written in Lisp using a set of lower level functions. LUSH modules are represented by lisp object of class `Module` . The following functions manipulate these objects.

7.12.1 (module-list)

Returns the list of all currently loaded modules. Unlike `mod-list` this function returns the module objects instead of the module filenames.

7.12.2 (module-filename m)

Returns the filename associated with module object `m` .

7.12.3 (module-executable-p m)

Test if the code for module `m` is executable.

7.12.4 (module-unloadable-p `m`)

Test if module `m` can be removed from memory.

7.12.5 (module-initname `m`)

Returns the name of the C initialization function for module `m` .

7.12.6 (module-depends `m`)

Returns the list of all the initialized modules that depends on module `m` . This function is useful to evaluate the consequences of a call to `module-unload` .

7.12.7 (module-never-unload `m`)

Make sure that the module `m` will not be unloaded. Attemps to unload the module will cause an error.

7.12.8 (module-defs `m`)

Returns an alist describing all the primitives defined by module `m` . This alist is populated the first time the module becomes executable. It is the user's responsibility to make this primitives available for general use by defining conveniently named variables.

7.12.9 (module-load `filename` [`hookfunction`])

Loads the binary code file `filename` and returns a module object.

Function `hookfunction` is called with two arguments when the state of the module changes. The first argument is a selector symbol representing the nature of the state change. The second arguments is the module object itself.

The selector symbol can take the following values:

- `init` The hook function is called with selector `init` just after calling the initialization function of the module. Most useful hook functions will scan the list of definitions returned by `module-defs` and define global symbols to access the newly defined primitives.
- `exec` The hook function is called with selector `exec` whenever the executability of the module changes because of loading/unloading another module.
- `unlink` The hook function is called with selector `unlink` just before unloading the module. This is a good time to revert the changes made during the module initialization.

7.12.10 (module-unload `m`)

Unloads the binary module `m` .

7.13 Extending the Interpreter

While most users will prefer limit themselves to writing lisp functions (possibly with inline C code) and compiling them, some adventurous users may need to extend the interpreter more directly. This section describes how to do that by making use of the dynamic loader/linker.

Here is an example of a file, named `"essai.c"` , which defines a new lisp function written in C for computing the square of the hypotenuse of triangles.

```
/* ------ Beginning of File "essai.c" ------ */
#include "header.h"
/* This is the function that does the work */
double hypotenuse(x,y)
double x,y;
{
  double z = x*x + y*y;
  printf("hypot(\\%f,\\%f)=\\%f\\n",x,y,z);
  return z;
}


/* This is the interpreter interface function */

DX(xhypotenuse) {
  ARG_NUMBER(2);
  ALL_ARGS_EVAL;
  return NEW_NUMBER( hypotenuse(AREAL(1),AREAL(2)));
}

/* This is the initialization routine.
 * Its name is formed by prepending "init_" to the file name
 */
void init_essai()
{
  dx_define("hypotenuse",xhypotenuse);
}
/* These two (optional) definitions to guarantee that
 * this module will only be loaded by compatible versions of LUSH.
 */
int majver_essai = 40; /* LUSH_MAJOR */
int minver_essai = 10; /* LUSH_MINOR */
/* ------ End of File "essai.c" ------ */
```

Integrating this function into the LUSH interpreter can be achieved in two ways:

- Copying this file in the `"src"` directory of Lush, adding `"essai.o"` to the list of objects in the `"Makefile"` , calling `"init_essai()"` from file `"toplevel.c"` and recompiling everything. This will build a new version of Lush with the new function.
- Compiling this file separately and loading the resulting object file into Lush at run-time. This solution is named dynamic linking of an external module. Modules are loaded with the function `mod-load` , and unloaded with the function `mod-unload` . These functions allocate the necessary memory, relocate the machine code, and resolve the external references.

The compilation is performed by the command:

```
globina\\% gcc -c -I/home/leonb/lush/include essai.c
```

The resulting `"essai.o"` file can be loaded into LUSH with the following command.

```
? (mod-load "essai.o")
= "/home/user/subdir/essai.o"
? (hypotenuse 3 4)
hypot(3.000000,4.000000)=25.000000
= 25
```

7.14 Debugging Modules with gdb

Although GDB is unaware of Lush's dynamic linking capabilities, it contains a convenient function `add-symbol-file` to load symbols from an object file. The lush dynamic loaded contains a C function `dld_print_gdb_commands` to print the GDB commands necessary to load all the required symbols. This function can be called from the GDB prompt.

Example:

```
globina\\% gdb lush
... GDB starts
(gdb) run
... LUSH starts
? ;;; The following override forces compilation with -g:
? (setq dhc-make-overrides (alist-add "OPTS" "-g" dhc-make-overrides))
= (("OPTS" . "-g"))
? ;;; Define a function
? (de mydiv(a b) ((-double-) a b) (/ a b))
= mydiv
? (dhc-make () mydiv)
```

```
...
? (mydiv 2 3)
= 0.6667
? (mydiv 2 0)

Program received signal SIGFPE, Arithmetic exception.
0x0885dcbf in ?? ()
(gdb) call dld_print_gdb_commands(1)
     add-symbol-file /home/leonb/lush/C/i686-pc-linux-gnu/mydiv.o 0x885dc84 \\
         -s .text 0x885dc84 -s .data 0x885dd44 -s .bss 0x885ddf8 -s .rodata 0x885ddf8
$1 = 10
(gdb) add-symbol-file /home/leonb/lush/C/i686-pc-linux-gnu/mydiv.o 0x885dc84 \\
         -s .text 0x885dc84 -s .data 0x885dd44 -s .bss 0x885ddf8 -s .rodata 0x885ddf8
add symbol table from file "/home/leonb/lush/C/i686-pc-linux-gnu/mydiv.o" at
      .text_addr = 0x885dc84
      .text_addr = 0x885dc84
      .data_addr = 0x885dd44
      .bss_addr = 0x885ddf8
      .rodata_addr = 0x885ddf8
(y or n) y
Reading symbols from /home/leonb/lush/C/i686-pc-linux-gnu/mydiv.o...done.
(gdb) where
#0  0x0885dcbf in C_mydiv (L1_a=2, L1_b=0) at /home/leonb/lush/C/mydiv.c:28
#1  0x0885dd00 in X_mydiv (a=0xbfffbaa8) at /home/leonb/lush/C/mydiv.c:44
#2  0x0810e8f0 in dh_listeval ()
(gdb) up
#1  0x0885dd00 in X_mydiv (a=0xbfffbaa8) at /home/leonb/lush/C/mydiv.c:44
44        ret.dh_real = C_mydiv (a[1].dh_real, a[2].dh_real);
(gdb) down
#0  0x0885dcbf in C_mydiv (L1_a=2, L1_b=0) at /home/leonb/lush/C/mydiv.c:28
28          L_Tmp0 = (L1_a / (real) L1_b);
(gdb) print L1_b
$2 = 0      ### HAHA. Bug was found.
```

GDB contains decent support to debug code contained in shared libraries (i.e. files with extension `".so"`). Another possibility for debugging modules consists in building them as shared libraries.

Example:

```
globina\\% gcc -shared -o essai.so -I/home/leonb/lush/include  essai.c
globina\\% gdb lush
... GDB starts
(gdb) run
... LUSH starts
? (mod-load "essai.so")
= "/home/leonb/essai.so"
```

```
<CTRL-C>
(gdb) br xhypotenuse
Break in xhypotenuse
(gdb) cont
? (hypotenuse 3 4)
Breakpoint 1, 0x4001a8c1 in xhypotenuse () from /home/leonb/essai.so
(gdb) ....
```

There are two limitations:

- Shared libraries must be completely linked. They can only reference symbols contained in the main lush executable and its libraries (not in other dynamically loaded modules)
- Shared libraries cannot be unloaded and replaced by another version.

7.15 Compiling and Loading C Code

It is often useful to compile and load C subroutines that either define new Lush primitives, or provide support functions to compiled lush functions.

Class `LushMake` provides a familiar interface for controlling the compilation of these subroutines and for loading them into the Lush system.

- First you create a `LushMake` object. Optionally you can specify the directory containing the source files, a different directory for the object files, and additional compilation options.
- Then you add compilation rules very similar in spirit to makefile rules.
- Finally you invoke methods `make` and `load` .

```
(let ((lm (new LushMake)))
  ;; Define the rules
  (==> lm rule "foo.o" '("foo.c" "foo.h"))
  (==> lm rule "bar.o" '("bar.f" "foo.h")
          "$F77 $DEFS $LUSHFLAGS -c $SRC -o $OBJ" )
  ;; Compile and load
  (==> lm make)
  (==> lm load) )
```

A more extensive example can be seen in `"packages/sn28/sn28common.lsh"` .

7.15.1 (new LushMake [srcdir [objdir]])

Create a new `LushMake` object.

The source files will be searched under directory `srcdir` . The default is to search source files under the directory containing the file being loaded. The object files will be created in architecture dependent subdirectories of `objdir` . The default is to create these subdirectories inside the source directory.

7.15.2 (==> `lushmake` **setdirs** `srcdir` [`objdir`])

Sets the source directory `srcdir` (where source files are searched) and the object directory `objdir` (where architecture dependent subdirectories are created to hold the object files). The default value for `objdir` is equal to `srcdir` .

7.15.3 (==> `lushmake` **setflags** `flags`)

Sets additional compilation flags. These additional compilation flags will be appended to the contents of the rule variable `"$LUSHFLAGS"` .

7.15.4 (==> `lushmake` **rule** `target deps` [`command`])

Define a compilation rule in the spirit of makefiles.

Argument `target` is the name of the object file to create relative to an architecture dependent subdirectory of the object directory.

Argument `deps` is a list containing the names of the files on which the target depends. The target is rebuilt if any of these files is more recent than the current target.

The optional argument `command` is a string containing the command to execute to rebuild `target` . The default command is provided in variable `dhc-make-command` and should be adequate for C programs. Environment variables in the command are expanded using `getenv` or `getconf` . The following additional variables are also defined:

- `$LUSHFLAGS` : Compilation flags composed by concatenating the contents of variable `dhc-make-lushflags` and the additional flags defined using method `setflags` .
- `$INCS` : Compilation flags specifying the directories to search for include files. The value of this variable is generated by `dhc-generate-include-flags` and augmented with `"-I$SRCDIR"` .
- `$SRCDIR` : The pathname of the source directory.
- `$OBJDIR` : The pathname of the object directory.
- `$SRC` : The pathname of the first file listed in argument `deps` .
- `$OBJ` : The pathname of the target file `target` .

See: dhc-make-command
See: dhc-make-lushflags
See: (dhc-substitute-env `str` [`htable`])
See: (dhc-generate-include-flags [`includepath`])

7.15.5 (==> `lushmake` **show** [`...target...`])

Show what commands need to be executed to bring targets `...target...` up to date. Calling this method without arguments processes all targets defined by the rules.

7.15.6 (==> `lushmake` **make** [`...target...`])

Makes sure that the targets listed as argument are up-to-date relative to their dependencies. This method finds the relevant rule and checks each dependency file. It first checks whether the dependency files themselves must be rebuilt. The rule command is then invoked if any of the dependency is more recent than the target.

Calling this method without arguments processes all targets defined by the rules.

7.15.7 (==> `lushmake` **load** [`...targets...`])

Makes sure that all listed targets are up-to-date and loads all listed object files using `mod-load` . In addition all the relevant dependencies will be recorded using `libload-add-dependency` .

Calling this method without arguments processes all object file targets defined by the rules.

Chapter 8

CLush: Compiled Lush

Author(s): Leon Bottou, Yann LeCun

[under construction]

One of Lush's most interesting features is its Lisp-to-C compiler and the ability to freely intermix Lisp and C code in a single function. Unlike many other interpreted/compiled languages, and contrary to many well established principles of language design, the two dialects that the Lush interpreter and the Lush compiler implement are two very different languages, though they share the same syntax. The compiled dialect is strongly typed, lexically bound, and has no garbage collector, while the interpreted dialect has loose typing, is dynamically bound, and is garbage collected.

In fact, the Lush compiler is not designed to replace the interpreter, but rather to complement it. Lush applications are generally a combination of compiled and interpreted code. The compiled part is often itself a combination of Lush and C code. The compiled code will generally contain the "expensive" and heavily numerical parts of an application, where performance is the main requirement, while the interpreted part will contain the high-level logic, the user interface, the memory management, etc, where flexibility is more important than pure performance.

8.1 A Simple Example

[under construction]

Numerous examples of compiled functions with and without in-line C-code are available in the libraries (in directories lsh and packages).

8.2 Using libload and dhc-make

Lush contains convenient functions that implement capabilities similar to make. Properly using these functions ensures that the Lush interpreter loads the most

recent version of the functions as necessary, and that all functions are judiciously recompiled when functions they call have been redefined.

Users should just follow the following four rules:

- Every Lush file should use the function `libload` to load all required files (i.e. files defining functions or classes required by this lush file).
- The compilation of Lush functions and classes should be performed by inserting one or several calls to `dhc-make` at the end of the lush file that defines those functions and classes.
- Users should identify the main Lush file of the project, that is to say the file that directly or indirectly loads all the Lush files of your project. This is usually the file with the higher level functions. It is sometimes convenient to create a Lush file for the sole purpose of being the main Lush file.
- When you want to load the most recent version of all your project files, you just need to load the main Lush file using either `libload` or the macro-character `|^L|` . This will load all modified files, reload all files that depends on modified files, and recompile all functions and classes that need recompilation, either because they have been modified, or because they depend on modified classes or functions.

See: (libload `libname` [`opt`])
See: (dhc-make `fname fspec1` ... `fspecN`)

8.3 Compilation Functions

8.3.1 High-Level Compilation Functions

(dhc-make `fname fspec1...fspecN`**)**

Function `dhc-make` controls the compilation of classes and functions specified by the `fspecs` arguments.

Although this function can be used from the command line, it mostly is used within a Lush file to control the compilation of classes and functions defined in this file. We will assume now that `dhc-make` is invoked from a Lush file.

The first argument `fname` is either the empty list or a string representing filenames of the C and object files output by the Lush compiler.

- Passing the empty list instructs `dhc-make` to generate a suitable filename derived from the name of the Lush file containing the `dhc-make` command. Generated C files will be created in a subdirectory `"C"` of the directory contaning the Lush file. Generated object files will be created in a subdirectory of `"C"` whose platform dependent name is provided by expression `(getconf "host")` .

- Passing a string specifies a particular base name. Function `dhc-make` adds the suitable suffixes for C files or object files, replacing any other suffix possibly present in `fname` . Argument `fname` may specify a directory for both the C and object files. Otherwise the C and object files will be generated as explained above.

Each of the following arguments `fspec` is either a function name or a list composed of a class name and method names. The following example, for instance, first compiles function `foo` , then compiles methods `rectangle` and `area` of class `rectangle` , and finally compiles function `bar` .

```
(dhc-make ()
          foo
          (rectangle rectangle area)
          bar )
```

The order of the `fspec` arguments is important because no function, class or method can be compiled before compiling all functions, classes, or methods it references. This order can be different from the function and class definition order in the Lush file.

Before starting the compilation, function `dhc-make` tests whether the target files already exist and are up-to-date:

The C file will be generated if any of the following conditions is true.

- no such C file exists,
- the Lush file is more recent than the existing C file,
- the Lush file depends on another Lush file which is more recent than the existing C file. These dependencies are those collected by function `libload` .

The object file will be generated if any of the following conditions is true:

- no such object file exists,
- the C file is more recent than the existing object file, as happen, for instance, when the C file was regenerated during this invocation of `dhc-make` .

Finally `dhc-make` performs a `mod-load` of the object file and checks that the compiled functions are now executable. It returns the name of the object file.

Note: Function `dhc-make` can also be invoked from the command line. When this is the case, it always generates fresh C and object files, and derives their names from the last `fspec` argument.

(dhc-make-class `fname classname1` [`classname2 ...`]**)**

See: (dhc-make `fname fspec1 ... fspecN`)

Compile each class with all its methods in the order in which the methods were defined.

(dhc-make-with-libs `fname libs fspec1...fspecN`**)**

See: (dhc-make `fname fspec1 ... fspecN`)

This function is similar to `dhc-make` but takes an additional argument `libs` to handle cases where the compiled functions depend on external C libraries. This argument can take two values:

- Argument `libs` may be a list of library names. Function `dhc-make-with-libs` works like `dhc-make` but also loads all the specified libraries after loading the compiled functions, and before checking whether the compiled functions are executable.
- Argument `libs` may be the symbol `t` . Function `dhc-make-with-libs` then works like `dhc-make` but does not cause an error if the compiled functions are not executable because they contain unresolved references. The caller is expected to load additional modules using `mod-load` in order to satisfy these unresolved references.

(dhc-make-with-c++ `fname libs fspec1...fspecN`**)**

See: (dhc-make `fname fspec1 ... fspecN`)

See: (cinline `format var1 ... varn`)

This function is similar to `dhc-make-with-libs` but uses the C++ compiler instead of the C compiler to generate the object file. This makes no difference except that one can use C++ constructs when inlining C code with `cinline` . This is particularly handy when interfacing external C++ libraries.

(dhc-make-all `fname fspeclist libs`**)**

See: (dhc-make `fname fspec1 ... fspecN`)

See: (dhc-make-with-libs `fname libs fspec1 ... fspecN`)

This is the elementary function called by `dhc-make` , `dhc-make-with-libs` and `dhc-make-with-c++` .

Argument `fspeclist` is a list containing the `fspec1 ... fspecN` arguments of function `dhc-make` . Argument `fname` is similar to the `fname` argument of functions `dhc-make` or `dhc-make-with-libs` . Argument `libs` is similar to the `libs` argument of function `dhc-make-with-libs` .

8.3.2 Customizing the Behavior of dhc-make

The following variables control the behavior of `dhc-make` .

dhc-make-force

Setting this variable to a non null value forces the recompilation of all files, even when they have not been modified.

The following example, for instance, forces the compilation of functions `foo` and `bar` .

```
(let ((dhc-make-force t))
   (dhc-make () (foo bar)) )
```

dhc-make-lushflags

This variable contains a string containing the compilation flags for invoking the C or C++ compiler. The following example, for instance, insert additional options `"-w -DMMX"` for the C compiler:

```
(let ((dhc-make-lushflags (concat dhc-make-lushflags " -w -DMMX")))
   (dhc-make () (foo bar)) )
```

The initial value is derived from the Makefile variable `LUSHFLAGS` . It might reference other predefined varibales prefixed with character `$` . These variables are expanded using `dhc-substitute-env` .

dhc-make-command

This variable is a string containing the command for generating object files. Variables prefixed with `$` will be expanded using `dhc-substitute-env` . Four additional variables are defined:

- `$LUSHFLAGS` Compilation flags defined by `dhc-make-lushflags` or by the optional argument of `dhc-make-o` .
- `$INCS` Flags specifying all the include directories specified by variable `c-include-path` . With this option, the C compiler can locate all include files that can be located using function `find-c-include` .
- `$SRC` The pathname of the source file.
- `$OBJ` The pathname of the object file.

See: (find-c-include `name`)

dhc-make-overrides

This variable contains an a-list of additional variable definitions for interpreting `dhc-make-lushflags` and `dhc-make-commands` . Definitions provided in `dhc-make-overrides` take precedence over the defaults provided by Lush.

The following example, for instance, invokes the C++ compiler instead of the C compiler:

```
(let ((dhc-make-overrides
         (cons (cons "CC" (dhc-substitute-env "$CXX")) dhc-make-overrides) ))
  (dhc-make () (foo bar)) )
```

8.3.3 Querying File Dependencies

The following functions can be used to determine whether `dhc-make` will invoke the compiler.

(dhc-make-get-dependencies [`lushfile`])

Returns the list of Lush files on which file `lushfile` is dependent. These are the files directly or indirectly loaded using `libload` . When argument `lushfile` is not provided, `file-being-loaded` is assumed.

(dhc-make-test `fname`)

This function tests whether a subsequent invocation of `dhc-make` with the same `fname` argument needs to generate a new version of the C file. When there is no need to generate a new C file, function `dhc-make-test` makes sure that the object file is up-to-date and returns the name of the object file. Otherwise it returns the empty list.

This function is useful to avoid processing long files when we are only interested in precompiled functions. See `"packages/lapack/lapack-s.lsh"` for an example.

8.3.4 Low-Level Compilation Functions

Most users will rarely use the functions described in this section, but they come in handy in certain cases.

(dhc-make-o-filename `src`)

Returns a suitable filename for the object file

(dhc-generate-c `filename` '([`func1` [`funcn`]]))

Translate lisp functions and classes `func1` ... `funcn` to C code and produces a source file suitable for a file named `filename` . Argument `filename` should be provided without the ".c" suffix and must be a legal C identifier as it is used for the initialization function in the C code.

(dhc-substitute-env `str` [`htable`])

Returns a copy of string `str` after substituting all environment variables. Substitution values are searched in `htable` when available, then passed to functions `getconf` and `getenv` .

(dhc-make-c `fname fsymblist`**)**

Compile functions or classes `fsymblist` into a new source file `fname` . Argument `fname` must be provided without the `".c"` suffix.

(dhc-make-o `src-file` [`obj-file` [`lushflags`]]**)**

Compile C source file `src-file` generated by the dh compiler into object file `obj-file` . Argument `lushflags` is an optional string containing compiler options.

(dhc-make-c-maybe `snname fname fsymblist`**)**

Compile functions or classes `fsymblist` producing the file `fname` . Argument `fname` must be provided without the suffix ".c". Compilation will only occur if the existing `fname` was created before the file `snname` or any file loaded from `snname` using `libload` .

(dhc-make-o-maybe `src-file` [`obj-file` [`cflags`]]**)**

Same as `dhc-make-o` but only recompiles if source file is newer than object file

8.4 What is Compilable, and What is Not

[under construction]

There is a simple test to determine if a particular function is compilable or not: (compilablep `func`). This may returns `'yes` , `'maybe` or `()` . The value `'maybe` is returned when the argument is a macro whose compilability cannot be determined without expansion.

Basically the kinds of things that can be compiled by the CLush compiler are the kinds of things that you expect to be compilable in a "conventional" language like C. All the really "Lispish" stuff like `eval` , `lambda` , etc is not compilable (those things don't exist in C).

There are three types of things that cannot be compiled:

- Things that construct functions and code at run time (e.g. `lambda` , `mlambda`), or things that evaluate dynamically generated code (e.g. `eval`).
- Things that allocate dynamic structures that would require garbage collection
- Things that call non-compilable functions.

8.4.1 (compilablep `function`)

Tests if a function is compilable. Returns t, maybe or (). The value `'maybe` is returned when the argument is a macro whose compilability cannot be determined without expansion.

8.5 CLush Specific Functions and Constructs

8.5.1 Numerical Type Declarations

(declare (-double-) `var1` **...** `varn`**)**

declare double precision floating point variables.

(declare (-float-) `var1` **...** `varn`**)**

declare single precision floating point variables.

(declare (-int-) `var1` **...** `varn`**)**

declare integer variables (32 bits).

(declare (-short-) `var1` **...** `varn`**)**

declare short integer variables (16 bits).

(declare (-byte-) `var1` **...** `varn`**)**

declare signed byte variables (8 bits).

(declare (-ubyte-) `var1` **...** `varn`**)**

declare unsigned byte variables (8 bits).

(declare (-str-) `var1`**...**`varN`**)**

declare character string variables.

(declare (-gptr- [`spec`]**)** `var1` **...** `varn`**)**

This declares the symbols `var1` ... `varn` as C-style pointers. In the absence of a `spec` the pointers will be equivalent to universal pointers (`void *`). An optional string `spec` can be specified to explicitely declare the C type of the pointer. Examples:

```
(declare (-gptr- "int *") integer-pointer)
(declare (-gptr- "FILE *") file-pointer)
(cpheader "<my_header_file.h>")
```

```
(declare (-gptr- "MyObjectType *") myobject-pointer)
```

Argument `spec` may also be a list containing a Lisp class name. This will define the type of the variable as a pointer to the C version of that class. Example:

```
(declare (-gptr- (myclass)) myclass-pointer)
```

The C type of `myclass-pointer` will be `"struct CClass_myclass *"` .

(declare (-obj- (class)) var1...varN)

This syntax declares the variables `var1` ... `varN` as being instances of the class `class` .

(declare (-idx0- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx0 (scalars) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx1- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx1 (vector) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx2- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx2 (matrix) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx3- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx3 (3-tensor) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx4- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx4 (4-tensor) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx5- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx5 (5-tensor) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx6- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx6 (6-tensor) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

(declare (-idx7- (type)) var1...varN)

This syntax declares the variables `var1` ... `varN` as idx7 (7-tensor) of type `type` , where `type` is one of the Lush numerical types (-ubyte-, -double-, -float-, etc).

8.5.2 Class Declaration and Compilation

Classes can be compiled with some restrictions. In short:

- The types of all the slots of a compiled class must be declared.
- It is preferrable to include the constructor in the list of methods being compiled.
- The constructor of a compiled class **must** set initialize all the slots with an object of the type declared in the class definition.
- The parent class of a compiled class must be compiled prior to the class
- Methods defined in a subclass that overload a method of the parent class must have the **same prototype** as the method they overload (i.e. the same arguments with the same type, the same return type, and the same temporary variables).

Here is an example of compiled class declaration syntax (taken from `packages/gblearning/modules.l`):

```
(defclass mle-cost sn-module
  ((-obj- (logadd-layer)) logadder)
  ((-obj- (idx1-ddstate)) logadded-dist)
  ((-idx1- (-int-)) label2classindex))
```

Essentially, the type of each slot must be declared using the same syntax as parameters of a function.

Compiling a class with `dhc-make` and its variants is done by passing a list whose first element is the class name and the remaining elements are the methods to be compiled. Here is an example:

```
(dhc-make ()
  (sn-module)
  (mle-cost mle-cost fprop bprop bbprop))
```

It is not required to compile all the methods of a class. A compiled class can have a mixture of compiled and non-compiled methods.

8.5.3 Type Conversions

Functions are provided to convert objects from one type to another. Their behavior is somewhat different in interpreted and compiled mode.

Numerical Type Conversions

These functions convert one numerical or object type to another. In compiled mode, they behave like C casts. In interpreted mode, the argument is casted to the corresponding C type, and then converted back to a `double` since `double` is the only numerical type manipulated by the interpreter.

(to-bool `arg`**)**
Convert `arg` to a boolean.
(to-char `arg`**).**
Convert `arg` to a char value.
(to-uchar `arg`**).**
Convert `arg` to an unsigned char value.
(to-int `arg`**)**
Convert `arg` to an integer.
(to-float `arg`**)**
Convert `arg` into a single precision floating point number.
(to-double `arg`**)**
Convert `arg` into a double precision floating point number.

Other Type-casts

(to-gptr `arg`**)**
See: to-mptr
Yield address of C-representation of `arg` .
This is not really a type conversion but yields the address of an object and is the counterpart to C's &-operator. It is generally unsafe to use the address returned by `to-gptr` for accessing an object because the interpreter is unaware of this reference and will reclaim any managed objects that appears unreferenced.

(to-mptr `arg`**)**
See: to-gptr
Yield address of C-representation of managed object `arg`
This is similar to the `to-gptr` function but only succeeds if `arg` is a managed object. In addition, the result of `to-mptr` is an `MPTR` object, which is considered a reference by the interpreter. This means that `arg` will not be reclaimed as long as the `MPTR` exists.

(to-str `mptr`**)**
Create a STRING object from managed address `mptr` .

(to-idx `p` [`rank` [`elem-type`]]**)**
Cast pointer `p` into an index. Optionally, check index rank and element type.

(to-obj [`class`] `arg`**)**
Interpret `arg` as an object of class `class` . Argument `arg` must be a gptr or an object. Argument `class` is mandatory in compiled mode.

- When argument `arg` is an object, this function checks that the object class is a subclass of `class` and returns the unmodified object.

- The compiler converts the GPTR into a pointer to an object of the specified class. Class membership is checked at run time.

8.5.4 Inline C Code

CLush allows to freely intermix Lisp and C code within a single function. This can be done in several ways, the simplest of which is the "hash-brace" construct.

Hash-Brace Construct

The macro character pair `# c-code #` allows easy embedding of C code inside a Lisp function. The embedded C code is delimited by `# and #` . Functions containing hash-brace segments cannot be executed by the interpreter and must be compiled.

Any C code can be written within a hash-brace construct. Lisp expressions and variables can be evaluated and refered to in the Lisp code by preceding them with a dollar sign.

- `$legal-c-identifier` refers to the lisp variable of the same name
- `$legal-lisp-variable` refers to the lisp variable (the braces are required if the lisp variable name is not a legal C name, e.g. if it contains dashes).
- `$(lisp expression)` refers to the result returned be the lisp expression.

Examples:

```
[under construction]
[insert juicy hash-brace examples here]
```

The hash-brace macro calls the lower-level compiler macro `cinline` .

Example:

```
(de add-to-integer( intg arg )
  ((-int-) intg arg)
  #{ $intg += $arg; #}
  intg )
```

expands to:

```
(de add-to-integer (intg arg)
  ((-int-) intg arg)
  (cinline " \\%s += \\%s; " intg arg)
  intg )
```

See: (cinline `format` [`arg1` [`arg2` ... [`argn`]]])

C Directives and Macros

(cpheader `s1 s2` **...)**

This directive can be used to insert lines in the C file generated by the CLush compiler. This is primarily used to include header files or to define macros. Examples:

```
(cpheader "#include <stdio.h>")
```

The lines are inserted at the beginning of the C file, before all the Lush header files.

(cheader `s1 s2` **...)**

Same as `cpheader` , but inserts the lines after the Lush header files instead of before.

(cinit `s1 s2` **...)**

Insert lines of code in the module's init function. This may be used to initialize module-global variables.

(cinline `format var1...varn`**)**

Insert C code in the C file generated by the Lush compiler. The `format` is an printf-like format where instances of "%s" are substituted by the C name of the lisp variables `var1` ... `varn` . Example:

```
;; set element 2 of v to x.
(de choucroute (v x)
  (declare (-idx1- (-int-)) v)
  (declare (-double-) x)
  (cinline "(IDX_PTR(\\%s,int))[2] = (int)(\\%s);" v x) ())
```

The C function generated by the Lush compiler looks like this:

```
extern_c char
C_choucroute (struct idx *L1_v, real L1_x)
{
  TRACE_PUSH ("C_choucroute");
  {
    (IDX_PTR (L1_v, int))[2] = (int) (L1_x);
    TRACE_POP ("C_choucroute");
    return 0;
  }
}
```

Function `cinline` is rarely used, as most users will prefer to use the hash-brace construct.

Locating Include Files and Libraries

c-include-path

See: (find-c-include `name`)

See: dhc-make-command

This variable is initialized by the Lush startup code and contains a list of directories containing potential include files. Include files along this path can be searched using function `find-c-include` .

Furthermore, the compilation command, defined by variable `dhc-make-command` , typically uses the macro `"$INCS"` to specify all these directories to the compiler. Appending directories to `c-include-path` is then the easiest way to handle additional directories for include files.

shared-library-path

See: (find-shared-library `name` [`extlist`])

This variable is initialized by the Lush startup code and contains a list of directories containing potential shared libraries. Function `find-shared-library` should be used to locate libraries located along this path, or in standard system locations.

static-library-path

See: (find-static-library `name` [`extlist`])

This variable is initialized by the Lush startup code and contains a list of directories containing potential shared libraries. Function `find-static-library` should be used to locate libraries located along this path.

(find-c-include `name`**)**

Returns the pathname of a C include file named `name` . Include files are searched in the directories specified by variable `c-include-path` which is initialized by `"lushenv.lsh"` .

```
? (find-c-include "stdio.h")
= "/usr/include/stdio.h"
```

Here is an example of how to use `find-c-include` to precompute an include file:

```
(defconstant qqq (sprintf "#include \\"\\%s\\"" (find-c-include "stdio.h")))
= qqq
? (de asd () (cpheader #.qqq) #{ printf("asd\\n") #} ())
= asd
? (dhc-make "junk" asd)
```

8.5.5 Compiler Directives

(ifcompiled `compiled-expr interpreted-expr`**)**

This allows to execute different codes in interpreted mode and in compiled mode.

(ifdef `symb expr1 expr2`**)**

[under construction]

#@`expr`

Expressions prefixed by #@ are not evaluated unless `dhc-debug-flag` is set to `t` . This can be used to insert debugging code.

dhc-debug-flag

Turns on execution of code prefixed with —#@—

8.5.6 Dynamic Allocation in Compiled Code

[under construction] This will eventually become a tutorial on how to use the `pool` class defined in `libstd/dynamic.lsh` . The reference manual for `dynamic.lsh` is available in the standard library section of this manual.

See: Dynamic Allocation with Pools.

8.6 Interfacing Existing C/C++/FORTRAN Libraries to Lush

[under construction]

8.6.1 Interfacing C/C++ Code

Using Lush's inline C capability and the `mod-load` function, interfacing existing C code to Lush is extremely simple.

Calling C/C++ functions defined in a library

If the code is available as a dynamic library `libasd.so` or a static library `libasd.a` , one merely needs to include the library and the header file in a call to `dhc-make-with-libs` . Let's assume that our library defines a function `cblah` that we would like to call in a Lush function `blah` . Here is an example:

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(dhc-make-with-libs ()
  '("libasd.so")
  #{ #include <asd.h> #}
  blah)
```

This will only work if `libasd.so` is in the system's library search path, and if `asd.h` is in the standard header file search path.

Let's say that `libasd.so` and `asd.h` were placed in the same directory as our Lush file, we can do the following:

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(libload "libc/make")
(let* ((current-dir (dirname file-being-loaded))
       (dhc-make-lushflags (concat dhc-make-lushflags (sprintf " -I\\%s" current-dir)))

  (dhc-make-with-libs ()
    (list (concat current-dir "/libpng.so"))
    #{ #include "asd.h" #}
  blah))
```

Calling C/C++ functions defined in an object file

If the code is available as an object file `/wherever/asd.o` , one merely needs to do `(mod-load "/wherever/asd.o")` . The C functions and global variables defined in `asd.o` are automatically visible from inline C code in Lush.

Here is a simple example of a Lush file that dynamically loads `asd.o` and defines a function `blah` that calls the C function `cblah` defined in `asd.o` :

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(mod-load "/wherever/asd.o")
(dhc-make ()
  #{ double cblah(double); #}
  blah)
```

Rather than explicitely defining the C prototype of `cblah` in the `dhc-make` call, one may prefer to include the header file `asd.h` that corresponds to `asd.o` . This can be done easily:

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(mod-load "/wherever/asd.o")
(dhc-make ()
  #{ #include "/wherever/asd.h" #}
  blah)
```

Naturally, rather than refering to absolute paths, we may prefer to put `asd.o` and `asd.h` in the same directory as our Lush file. In that case, we need to tell Lush and the C compiler where to find everything:

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(let* ((current-dir (dirname file-being-loaded))
       (dhc-make-lushflags (concat dhc-make-lushflags (sprintf " -I\\%s" current-dir))))
  (mod-load (concat current-dir "/asd.o"))
  (dhc-make ()
    #{ #include "asd.h" #}
  blah))
```

Redefining `dhc-make-lushflags` like this will add the directory where the Lush file resides to the path in which the C compiler looks for header files.

Naturally, all of this assumes that `asd.o` has previously been compiled. Lush conveniently provides a "make"-like utility to compile C files and specify dependencies called `LushMake` (see the corresponding documentation. Using `LushMake` as shown below will automatically generate `asd.o` when the Lush file is loaded, whenever `asd.c` or `asd.h` have been modified:

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(libload "libc/make")
(let* ((current-dir (dirname file-being-loaded))
       (dhc-make-lushflags (concat dhc-make-lushflags (sprintf " -I\\%s" current-dir)))
       (lm (new LushMake current-dir)))
  ;; compile asd.c to asd.o and mod-load asd.o
  (==> lm setflags (sprintf "-I\\%s" current-dir))
  (==> lm rule "asd.o"  '("asd.c" "asd.h"))
  (==> lm load)

  (dhc-make ()
    #{ #include "asd.h" #}
  blah))
```

Perhaps the functions defined in `asd.o` call functions from a library that must be linked-in, say `libpng.so` . These libraries can easily be added into using `dhc-make-with-libs` instead of `dhc-make` :

```
(de blah (x) ((-double-) x) (to-double #{ cblah($x) #}))

(libload "libc/make")
(let* ((current-dir (dirname file-being-loaded))
       (dhc-make-lushflags (concat dhc-make-lushflags (sprintf " -I\\%s" current-dir)))
       (lm (new LushMake current-dir)))
  ;; compile asd.c to asd.o and mod-load asd.o
  (==> lm setflags (sprintf "-I\\%s" current-dir))
```

```
(==> lm rule "asd.o"  '("asd.c" "asd.h"))
(==> lm load)

(dhc-make-with-libs ()
  '("libpng.so")
  #{ #include "asd.h" #}
blah))
```

If our code in `asd.c` is written in C++ instead of C, we must replace the call to `dhc-make-with-libs` with `dhc-make-with-c++` .

8.6.2 Interfacing FORTRAN Code

[This section is under construction]

Basically, this works like calling C code. FORTRAN libraries can be loaded

8.7 Making Standalone Executables

Author(s): Yann LeCun

Lush provides a rudimentary way to generate standalone executable programs from a set of compiled Lush functions.

8.7.1 (make-standalone `lushfile` `cdir` `executable` `main-func`)

generate a standalone C program that contains all the functions in lush file `lushfile` . All the necessary C source files will be written in directory `cdir` . The directory will be created if necessary. Generating the executable program can be done by cd-ing to `cdir` and typing "make". The resulting executable file will be called `executable` . The argument `main-func` is a string that indicates the name of the Lush function that should become the "main" function of the resulting C program.

This main function must be defined as:

```
(de mymain (ac av)
  ((-int-) ac)
  ((-gptr- "char **") av)
  ...body...
  (to-int some-number))

(dhc-make () mymain)
```

8.7.2 low-level support functions for make-standalone

(lushc.lsh-to-c `fname`)

get the name of the .c file that corresponds to a .lsh file. This does not check if the C file actually exists, it merely returns its supposed path. This simply turns a string of the form "blah/asd.lsh" into a string of the form "blah/C/asd.c".

(lushc.o-to-c `oname`)

get the name of the compiler-generated .c file that corresponds to a .o file. This does not check if the .c file actually exists, it merely returns its supposed path. This simply turns a string of the form "blah/C/i686-pc-linux-gnu/asd.o" into a string of the form "blah/C/asd.c".

(lushc.c-to-o `cname`)

get the name of the .o file that corresponds to a compiler-generated .c file. This does not check if the files actually exist. This simply turns a string of the form "blah/C/asd.c" into a string of the form "blah/C/i686-pc-linux-gnu/asd.o".

(lushc.find-module `oname`)

return the module that corresponds to a .o file. `oname` must be an absolute path. The module must be mod-loaded beforehand. This function returns nil if no suitable module is found.

(lushc.get-parent-modules `themodule`)

get all the modules on which a module is directly dependent. return them as a set (a htable).

(lushc.get-ancestors-m `themodule`)

get all the modules on which a module is dependent, directly or indirectly. Return the result as a set (a htable).

8.8 More on the Lisp-C Interface

[under construction]

8.8.1 DH: the Compiled Function Class

(dhinfo-t `dhfunc`)

[under construction]

(dhinfo-c `dhfunc`**)**

[under construction]

8.8.2 Classinfo

(classinfo-t `class`**)**

[under construction]
Similar to the SN3 function but takes a class as argument instead of now obsolete cclass objects.

(classinfo-c `class`**)**

[under construction]
Similar to the SN3 function but takes a class as argument instead of now obsolete cclass objects.

8.8.3 Controling the Lisp/C Interface

(lisp-c-map [`arg`]**)**

This function displays the internal data structure of the interface between Lisp and Compiled code. The LISP-C interface maitains a sorted avl tree of all objects currently allocated. This table relates the address of the compiled structure, the type of the structure, the way the object was created and possibly the Lisp representation of this object.

- Calling this function without arguments returns the number of entries in the table.
- Calling this function with `arg` equal to `0` or `()` prints all the table. You better check the size of the table before doing this.
- Calling this function with `arg` equal to `1` prints all the table entries that needs to be synchronized before calling a compiled function.
- Calling this function with `arg` equal to `2` prints all the table entries that represent temporary objects.
- Calling this function with `arg` equal to an object, a storage, an index, or a strng prints the table entry (if any) associated with this object.

You can redirect the output of this command using function `writing` .
Example:

```
? (lisp-c-map)
= 12
? (lisp-c-map ())
```

```
  =                L 0x1e29b0 str              "A string for class c1"
 <                 L 0x2f10a0 idx              ::INDEX1:<3>
<                  L 0x2f12e0 obj:c2           ::c2:2f1330
 =                 L 0x2f1370 idx              ::INDEX1:<3>
=                   L 0x2f1c98 srg              ::FSTORAGE:static@2fe820
  =                L 0x2f1e18 str              "A second string for class c1"
 =                 L 0x2ff050 str              "A third string for class c1"
  =                L 0x2ff068 obj:c1           ::c1:2f1b08
=                  L 0x2ff080 idx              ::INDEX1:<3>
  =                L 0x2ff0a8 srg              ::FSTORAGE:static@2fe82c
 =                 L 0x31cb20 obj:c1           ::c1:1e2980
  =                L 0x31cb38 srg              ::FSTORAGE:ram@1e2968:<3>
= 12
The first character describes the position of the entry in the balanced
tree.  The second character tells wether the object was created by LISP (L)
or by compiled code (C) (e.g. Pool). Each entry displays then the address
of the compiled object and the lisp representation of the object. An object
created by C code may have no lisp representation until it is referenced by
a represented object or until the proper GPTR is casted by function <obj>.
```

(lisp-c-dont-track-cside)

Restrict the synchronization of the lisp and C data structures in a way similar to previous versions of the system (SN3.x). C objects managed by compiled code will appear as gptrs in Lisp code instead of being translated into equivalent Lisp objects.

(lisp-c-no-warnings `..exprs..`**)**

Evaluates expressions `..exprs..` like function `progn` without displaying the usual diagnostic about the synchronization of the lisp and C data structures.

8.9 Compiler Internals

8.9.1 (dhc-generate-c `filename '([func1 [funcn]])`)

Translate lisp functions and classes `func1` ... `funcn` to C code and produces a source file suitable for a file named `filename` . Argument `filename` should be provided without the ".c" suffix and must be a legal C identifier as it is used for the initialization function in the C code.

8.9.2 (compilablep `function`)

Tests if a function is compilable. Returns t, maybe or (). The value `'maybe` is returned when the argument is a macro whose compilability cannot be determined without expansion.

8.9.3 (dhc-substitute-env str [htable])

Returns a copy of string `str` after substituting all variable names prefixed by a `"$"` character. Substitutions are first searched in the optional `htable` , and finally passed to functions `getconf` and `getenv` .

8.9.4 (dhc-generate-include-flags [includepath])

Generate include flags for the compilation command taking into account all the directories specified in list `includepath` . This list defaults to the value of variable `c-include-path` .

8.9.5 varlushdir

This variable is defined by `"stdenv.lsh"` when the owner of the lush directory is different from the owner of the lush process. Compiled versions of the system lush files will be created in a shadow directory tree located under the `varlushdir` directory.

8.9.6 (dhc-make-cdir cdir [create])

This function returns `cdir` when `varlushdir` is the empty list. When `varlushdir` is the name of an existing directory, this function checks if `cdir` is a subdirectory of the Lush directory `lushdir` . If this is the case, it returns the name of a subdirectory under `varlushdir` with the same name. When argument `create` it `t` , function `dhc-make-cdir` creates all the requitred directories under `varlushdir` .

8.9.7 (dhc-make-c-filename src)

Returns a suitable filename for the C file generated from the LSH file `src`

8.9.8 (dhc-make-o-filename src)

Returns a suitable filename for the object file generated from the C or LSH file `src`

8.9.9 (dhc-make-get-dependencies [snname])

Returns a list of files that are loaded directly or indirectly when loading file `snname` . This function takes advantage of the information collected by function `libload` . Argument `snname` defaults to the currenly loaded file.

8.9.10 dhc-make-lushflags

Compilation flags used by `dhc-make-o` . This is similar to the variable LUSHFLAGS in the makefiles. Variables prefixed with `$` will be expanded using `dhc-substitute-env` .

8.9.11 dhc-make-command

Compilation command used by dhc-make-o. Variables prefixed with `$` will be expanded using `dhc-substitute-env` . Four additional variables are defined:

- `$LUSHFLAGS` Compilation flags defined by `dhc-make-lushflags` or by the optional argument of `dhc-make-o` .
- `$LUSHDIR` The lush main directory.
- `$INCS` Flags specifying all the include directories specified by variable `c-include-path` . With this option, the C compiler can locate all include files that can be located using function `find-c-include` .
- `$SRC` The pathname of the source file.
- `$OBJ` The pathname of the object file.

8.9.12 dhc-make-overrides

This variable contains an a-list of variable definitions for overriding those usually derived from `getconf` and `getenv` .

8.9.13 dhc-make-force

Setting this variable to a non null value forces the recompilation of all files, even when they have not been modified.

8.9.14 dhc-make-essential-libs

This variable contains a list of archive libraries that should be reloaded when undefined symbols remain. This is useful on some systems.

8.9.15 (dhc-make-o `src-file` [`obj-file` [`lushflags`]])

Compile C source file `src-file` generated by the dh compiler into object file `obj-file` . Argument `lushflags` is an optional string containing compiler options.

8.9.16 (dhc-make-rebuild-p `target dependencies`)

Returns true if file `target` needs to be rebuilt. This is the case if filr `target` does not exist or if any of the files listed in `dependencies` is newer than file `target`

8.9.17 (dhc-make-o-maybe src-file [obj-file [cflags]])

Same as `dhc-make-o` but only recompiles if source file is newer than object file

8.9.18 (dhc-make-c fname fsymblist)

Compile functions or classes `fsymblist` into a new source file `fname` . Argument `fname` must be provided without the `".c"` suffix.

8.9.19 (dhc-make-c-maybe snname fname fsymblist)

Compile functions or classes `fsymblist` producing the file `fname` . Argument `fname` must be provided without the suffix ".c". Compilation will only occur if the existing `fname` was created before the file `snname` or any file loaded from `snname` using `libload` .

8.9.20 (dhc-make-test fname)

Test if both the C file named `fname` and the associated object file are up-to-date relative to the currenly loaded file and its dependencies. Returns either () or the name of the up-to-date object file. When argument `fname` is the empty list, a suitable filename is constructed from the currenly loaded file using the same rules as `dhc-make` .

8.9.21 (dhc-make-with-c++ fname library-list f1 [f2 ...[fn]])

Same as `dhc-make-with-libs` but compiles with the c++ compiler instead of the c compiler.

Note that the dynamic loader is not currently able to execute static initializers possibly present in the generated code. The lush compiler obviously does not generate such construct. On the other hand everything can happen when using inlined C++ code.

8.9.22 (dhc-nolast l)

Remove last element of list.

8.9.23 (dhc-remove-dup l)

Remove duplicates in a list.

8.9.24 (dhc-remove-eqdup l)

Remove pointer duplicates from a list.

8.9.25 (dhc-remove-nth n l)

Remove `n` th element from list `l`

8.9.26 (dhc-postincr symb)

Increments variable `symb` by one and returns the value of the variable *before* incrementing.

8.9.27 dhc-debug-flag

Turns on execution of code marked with —#@—

8.9.28 —#@—

Expressions prefixed by #@ are not evaluated unless `dhc-debug-flag` is set to `t` .

8.9.29 (dhc-add-c-declarations str [str2 ...])

To be used in a `dhm-c` macro. Adds string `str` into the declaration part of the current block of C code.

8.9.30 (dhc-add-c-statements str [str2 ...])

To be used in a `dhm-c` macro. Adds string `str` to the statement part of the current block of C code.

8.9.31 (dhc-add-c-epilog str)

To be used in a `dhm-c` macro. Adds string `str` to the epilog part of the current block of C code.

8.9.32 (dhc-add-c-externs str)

To be used in a `dhm-c` macro. Adds string `str` to the C code segment declaring external symbols.

8.9.33 (dhc-add-c-metaexterns str)

To be used in a `dhm-c` macro. Adds string `str` to the C code segment declaring external symbols used in meta information.

8.9.34 (dhc-add-c-header str)

To be used in a `dhm-c` macro. Adds string `str` to the C code segment declaring files included after the standard LUSH include files.

8.9.35 (dhc-add-c-pheader str)

To be used in a `dhm-c` macro. Adds string `str` to the C code segment declaring files included *before* the standard LUSH include files.

8.9.36 (dhc-class-to-struct-decl type)

Returns a string containing the C structure declaration for objects of class `type` .

8.9.37 (dhc-class-to-vtable-decl type)

Returns a string containing the C structure declaration for the virtual table of class `type` .

8.9.38 (dhc-type-to-c-decl type)

Returns a short string with the C equivalent of type `type` .

8.9.39 (dhc-declare-temp-var type [clue])

Uses `tmpnames-seed` to generate a unique temp variable of type `type` . Returns the C string representation.

8.9.40 dhc-debug-stack

Variable containing the source causing the error

8.9.41 (dhc-error string [arg])

Called when we get a syntax error in the source

8.9.42 (dhc-check-symbol source)

Take an argument source and check if it is a symbol. If it is return the corresponding object of class dhc-symbol.

8.9.43 (dhc-internal-error str)

Called when we detect inconsistent internal state

8.9.44 dhc-type

A dhc-type is a structure which contains almost all the information which is available at compile time of an expression. Information which is not meant to be used in the construction of the DHDOC should be kept away of dhc-type (that info can be used in the t-node structure instead). For instance `u-bump`

and `u-access` have an impact on the DHDOC (whether a variable is writable, whether an argument is a temporary) but the flag `ignore-return-value` in a progn is not type information and has nothing to do in the dhc-type construct.

8.9.45 (==> `dhc-type` print)

A type printer, for debugging purposes

8.9.46 (==> `dhc-type` hashcode)

returns a hashcode for a type. only to be called after the type is fully unified.

8.9.47 (==> `dhc-type` access [`v`])

set or return the access type of a type. Possible values: 'read or 'write

8.9.48 (new dhc-type `class` [`a1`] [`a2`])

- Declaration of simple types: (new dhc-type `type`). Where `type` is a symbol from like, 'dht-float 'dht-bool, 'dht-nil, etc...
- Declaration of arrays: (new dhc-type 'dht-out `ndim` [`type`]) Where `type` can be any type (default is (new dhc-type 'dht-float)). `ndim` is an integer.
- Declaration of object: (new dhc-type 'dht-obj `type-list`) where `type-list` is a list of cons of the form (name . type)
- Declaration of functions: (new dhc-type 'dht-func `type-list`) where `type-list` is a list of valid types corresponding to the arguments

8.9.49 (dhc-desc-to-type `desc`)

Convert a type description to a type. Argument `desc` is a type description similar to those returned by `dhinfo-t` and `classinfo-t` .

8.9.50 (new dhc-symbol `name` `lex` [`fmt`])

return an symbol object which knows the lisp name `name` , the C name, the scope level (passed in `lex` . `lex` is zero if the symbol is an argument). `fmt` is by default "L%d_%s" where %d refers to the scope level (`lex`) and %s to the C name of the symbol (computed from the lisp name with dhc-lisp-to-c-name.

8.9.51 (dhc-search-symtable `symbolname` `table`)

Search a symbol table

8.9.52 (dhc-add-to-symtable table symbolobject)

Return a new symbol table with one more element

8.9.53 (dhc-add-symbol-table symb lex)

Add a symbol named `symb` at lexical level `lex` into the current symbol table.

8.9.54 (dhc-add-global-table symb)

Add a symbol to the global table

8.9.55 (new t-node t-node-list type [source] [symbol])

Returns a t-node (type node). T-nodes contains all the type information of a LISP expression. If `source` is a symbol, then the slot `symbol` should contain an object of class `dhc-symbol` .

A t-node contains 5 slots:

```
<tn-list> <type> <source> <symbol> <ignore>
```

The slot `type` holds the return type of the expression which is parsed. is a t-node. If the expression is terminal, the slot tn-list should be the empty list and the slot source should contain the corresponding source.

Examples:

```
Parsing the expression  (+ a b)
returns the t-node      <<+:(dhm)>,<a:(flt)>,<b:(flt)>:(flt)>
In this t-node the slots are:
  tn-list : a list of 3 t-nodes    = (<+:(dhm)> <a:(flt)> <b:(flt)>)
  type    : a return dhc-type node = (flt)
  source  : the original source    = (+ a b)
For the first element of the above t-node
i.e. (car :t_node_above:tn-list)
  tn-list : a list of t-nodes    = ()
  type    : a return dhc-type node = (dhm)
  source  : the original source    = +
```

8.9.56 (dhc-make-t-node expr)

Return a treetype (a t-node) for `expr` but leave all the types unknown.

8.9.57 (dhc-copy-source-tree source)

Utility for dhc-get-treetype. Copies a source tree list (except leaves). Returns the copy.

8.9.58 (dhc-get-treetype source)

This function returns the treetype of a expression.

```
? (dhc-get-treetype '(+ 3 4))
= <<+:(dhm)>,<3:(number)>,<4:(number)>:(number)>
? (dhc-get-treetype '(lambda (a) (-flt- a) a))
= <<lambda:(dhm)>
   ,<<a:(flt)>:(unk)>
   ,<<-flt-:(dhm)>,<a:(flt)>:(flt)>
   ,<a:(flt)>
   :(func ((flt)) () (flt))>
```

8.9.59 (dhc-get-type source)

This function returns the dhc-type of expression `source`

8.9.60 (dhm-t func (source) . body)

Installs a new DHM-T for the function named `func` .

The DHM-T function will be called when parsing a list expression starting with function `func` . This list expression is passed as argument to the DHM-T function. It should construct and return a t-node for the provided source code.

8.9.61 (dhm-t-declare model f1 ... fn)

Associates the dhm-t for function `model` with all functions `f1` to `fn` .

8.9.62 (dhm-c symb (source treetype retplace) . body)

Installs a new DHM-T for the function named `func` .

The DHM-C function will be called when generating C code for a list expression starting with function `func` . This list expression is passed as argument `source` of the DHM-C function. Argument `treetype` is the t-node prepared by the DHM-T function. The function should call the `dhc-add-c-xxx` functions to compose the C code and return a string representing a C expression for the return value. When argument `retplace` is non nil, it contains the name of a variable where the return value must be stored. The dhm-c function should then return this name.

8.9.63 (dhm-c-declare model f1 ... fn)

Associates the dhm-c for function `model` with all functions `f1` to `fn` .

8.9.64 (get-dhm-t symb)

Return the dhm-t for function `symb`

8.9.65 (get-dhm-c `symb`)

Return the dhm-c for function `symb`

8.9.66 (dhm-p `func` (source) . `body`)

Installs a new DHM-P for the function named `func`

The DHM-P function will be called when preprocessing a list expression starting with function `func` . The list expression is passed as `source` . It should construct a copy of `source` , using function `dhc-pp` on all parts of `source` that can be subject to macro-expansion

There are relatively few DHM-P functions because the SN3 compiler lacked this facilities. In the absence of a DHM-P function, all arguments of a function call are considered for macro-expansion. This is obviously wrong for functions like `let` or `quote` .

8.9.67 (get-dhm-p `symb`)

Return the dhm-p for function `symb`

8.9.68 (dhc-pp `body`)

Macro preprocessor for the dhc compiler. return a version of lisp expression `body` with all macros (both dm and dmd) expanded.

8.9.69 (dhc-parse-replacement-source-t `source` `newsource`)

A dangerous hack to allow source replacement from within a dhm-t. Modifies the source tree being parsed and calls the dhm-t for the "newsource" The "source" argument should be the input argument of the calling dhm-t. See the dhm-t for 'length' for usage. No corresponding dhm-c is needed.

8.9.70 (process-numerical-args-t `arglist` `rettype`)

Process the arguments of a numerical expression, and determine the type of the returned value. The arguments are passed in `arglist` , and the type of the return value in `rettype` .

8.9.71 u-nodes

[under construction]

Chapter 9

Standard Libraries

This section describes all the libraries that are distributed with Lush in the `lsh` directory, but are not loaded by default into the interpreter environment.

9.1 C-Style Input/Output

functions this section provide a convenient interface to the C stdio library. Functions are provided to open and close C-style file descriptors and pipes. The most commonly used stdio functions are also provided, including ftell, fseek, fprintf, rewind, fputc, fgetc, and several others. C-style file descriptors manipulated by these functions are merely generic pointers (-gptr-). As a consequence, many of these function as "unsafe", i.e. calling them with an invalid file pointer may cause the interpreter to crash.

9.1.1 (fprintf `file-pointer args...`)

```
((-gptr-) file) ; the file pointer returned by a fopen
MACRO
DESCRIPTION: same as printf, but writes to a file rather than to standard output
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
```

9.1.2 (fwrite-str `file-pointer s`)

```
((-gptr-) file) ;; the file pointer returned by a fopen
((-str-) s)     ;; the string to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes string <s> to file <file-pointer>
```

See: Compilable file I/O

9.1.3 (stdout)

```
RETURNS: (-gptr-) the file pointer associated with the standard output
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.4 (stdin)

```
RETURNS: (-gptr-) the file pointer associated with the standard input
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.5 (fopen `filename type`)

```
((-str-) filename) ; the name (or path) of the file to be opened:
((-str-) type)     ; the opening type: "rb" for reading, "wb" for creating and writing
RETURNS: (-gptr-) a file pointer to the open file (the pointer is null in case of faili
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: same as the C function fopen
```

See: Compilable file I/O

9.1.6 (fclose `file-pointer`)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: same as the C function fclose
```

See: Compilable file I/O

9.1.7 (popen filename type)

```
((-str-) filename) ; the name (or path) of the pipe to be opened:
((-str-) type)     ; the opening type: "rb" for reading, "wb" for creating and writing, "a" for a
RETURNS: (-gptr-) a file pointer to the open pipe (the pointer is null in case of failiure)
CREATED: Yann LeCun 08/29/96
COMPILABLE: Yes
DESCRIPTION: same as the C function popen
```

See: Compilable file I/O

9.1.8 (pclose file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a popen
RETURNS: ()
CREATED: Yann LeCun 08/29/96
COMPILABLE: Yes
DESCRIPTION: same as the C function pclose
```

See: Compilable file I/O

9.1.9 (ftell file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: The current position in the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: same as C function ftell
```

See: Compilable file I/O

9.1.10 (fseek file-pointer pos)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-real-) pos)     ; an absolute position in the file:
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: moves to the given position in the file
```

See: Compilable file I/O

9.1.11 (fseek-from-end file-pointer pos)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-real-) pos)     ; a position in the file relative to the end
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: moves to the given position in the file
```

See: Compilable file I/O

9.1.12 (fseek-from-current file-pointer pos)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-real-) pos) ; a position in the file relative to the current position:
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: moves to the given position in the file
```

See: Compilable file I/O

9.1.13 (fgetc file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-int-) The next byte read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Reads a byte (character) from the file (same as C function fgetc)
```

See: Compilable file I/O

9.1.14 (fputc file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-int-) val)     ; contains the byte to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes a byte (character) to the file (same as C function fputc)
```

See: Compilable file I/O

9.1.15 (fread-ubyte file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-ubyte-) the next ubyte read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.16 (fwrite-ubyte file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-ubyte-) val)     ; contains the ubyte to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes a ubyte to the given file
```

See: Compilable file I/O

9.1.17 (fread-byte file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-byte-) the next byte read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.18 (fwrite-byte file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-byte-) val)     ; contains the byte to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes an byte (4 bytes on Sun-OS or Solaris) to the given file
```

See: Compilable file I/O

9.1.19 (fread-short file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-short-) the next short (2 bytes) read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.20 (fwrite-short file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-short-) val)
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes a short (2 bytes) to the given file
```

See: Compilable file I/O

9.1.21 (fread-int file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-int-) the next int (4 bytes on Sun-OS or Solaris) read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.22 (fwrite-int file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-int-) val)     ; contains the int to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes an int (4 bytes on Sun-OS or Solaris) to the given file
```

See: Compilable file I/O

9.1.23 (fread-flt file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-float-) the next float (4 bytes) read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.24 (fwrite-flt file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-float-) val):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes a float (4 bytes) to the given file
```

See: Compilable file I/O

9.1.25 (fread-real file-pointer)

```
((-gptr-) file)   ; the file pointer returned by a fopen
RETURNS: (-real-) the next real (8 bytes) read from the file
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
```

See: Compilable file I/O

9.1.26 (fwrite-real file-pointer val)

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-real-) val):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: Writes a real (8 bytes) to the given file
```

See: Compilable file I/O

9.1.27 (reverse_n ptr sz n)

```
((-gptr-) ptr) ;; pointer to the block of memory that must be reversed
((-int-) sz)   ;; size of each item to reverse
((-int-) n)    ;; number of items to reverse
RETURNS: ()
CREATED: Pascal Vincent 04/17/96
COMPILABLE: Yes
DESCRIPTION: Reverses order of bytes in <n> items of size <sz>
             starting at memory location <ptr>
             This is a tool for writing/reading file formats that are portable
             across systems with processors that represent long-words
             differently in memory (Sparc vs Intel-Pentium for ex.)
             It can be called from inline-C as C_reverse8(ptr,n);
```

9.1.28 (fscan-int file-pointer)

```
(-gptr- file-pointer) ;; the file poiner returned by fopen
DESCRIPTION: Same as fscanf(file_pointer,"\\%d",&result) in C.
RETURNS: int
CREATED: Yoshua Bengio 23 August 1996
```

9.1.29 (fscan-flt file-pointer)

```
(-gptr- file-pointer) ;; the file poiner returned by fopen
DESCRIPTION: Same as fscanf(file_pointer,"\\%d",&result) in C.
RETURNS: float
CREATED: Yoshua Bengio 23 August 1996
```

9.1.30 (fscan-str file-pointer)

```
(-gptr- file-pointer) ;; the file poiner returned by fopen
DESCRIPTION: same as fscanf(file_pointer,"\\%s",result).
             The result must hold in a string of size < 1024 bytes.
RETURNS: string
CREATED: Yoshua Bengio 23 August 1996
```

9.1.31 (fgets file-pointer max-size)

```
(-gptr- file-pointer) ;; the file poiner returned by fopen
(-int- max-size)      ;; maximum allowed size of line in bytes
DESCRIPTION: same as C's fgets(result,max_size,file_pointer).
```

```
RETURNS: string
CREATED: Yoshua Bengio 23 August 1996
```

9.1.32 (file-size file-name)

returns the size in byte of file `file-name` author: Leon Bottou.

9.1.33 (rewind f)

```
((-gptr-) f):
CREATED: Oct 97
DESCRIPTION:
```

9.1.34 (skip-comments start f)

```
((-char-) start):
((-gptr-) f):
CREATED: Oct 97
DESCRIPTION:
skip comments lines starting with <start>
```

9.2 Abstract datatypes

9.2.1 Integer pairs

A most compact way to store two 32 bit integers is treating the 64 bits as a double floating point number. This way a pair of two integers may be returned by a C function without using pointers or objects. This facility is used in the implementation of some abstract datatypes. See also datatype `IPairSet` .

(ipair `i j`**)**

Create a pair of two integers `i` and `j` .

(ipi1 `ip`**)**

First integer in pair `ip` .

(ipi2 `ip`**)**

Second integer in pair `ip` .

(iprot `ip`**)**

Return `(ipair (ipi2 ip) (ipi1 ip))` .

9.2.2 Queue (FIFO)

A queue is an abstract data type which can hold an arbitrary number of items. Items may be added ("enqueued") or taken out of the queue ("dequeued"). Items can only be taken out of the queue in the order they were added.

```
? (let ((q (make-queue)))
    (enqueue q 2 "three" "four")
    (dequeue q)
    (dequeue q))
= "three"
```

While syntactically not an object, a queue behaves like an object in that the functions `enqueue` and `dequeue` change the queue in-place (`enqueue` and `dequeue` are destructive functions).

This queue implementation cannot be compiled. Use `Deque` instead when a compilable queue implementation is required.

(make-queue)

Create an empty queue.

(queue-length `q`**)**

Number of items in queue `q` .

(queue-empty-p `q`**)**

True if queue `q` is empty.

(enqueue `q item-1` **...** `item-n`**)**

Add one or more items to `q` queue in-place, return the queue.

(dequeue `q`**)**

Take an item out of the queue, return the item.

(peek-queue `q`**)**

Return next item in queue `q` without altering the queue.

9.2.3 Stack (LIFO)

A stack is an abstract data type which can hold an arbitrary number of items. Items may be "pushed" onto the stack or "popped" off the stack. Items pushed last are popped first.

```
? (let ((s (make-stack)))
    (push s 2)
    (push s "three")
    (push s "four")
    (pop s))
= "four"
```

The stack functions are implemented as macros and may be compiled if the stack is specialized to hold items a particular item-type only.

(make-stack `n` [`item-type`]**)**

Create an empty stack with initial capacity `n` for items of type `item-type` (default `atom`). Any storage class symbol is a valid `item-type` (e.g., `short` , `int` , `gptr` , etc.).

(emptyp `stack`**)**

True if `stack` is empty.

(push `stack item`**)**

Push an item onto stack.

(pop `stack`**)**

Pop item from stack.

(peek-stack `stack`**)**

Return next item on `stack` without popping it.

(clear-stack `stack`**)**

Make stack empty.

(clear-stack* `stack`**)**

Make pointer-stack empty.

9.2.4 Double-ended Queue

The `IDeque` datatype supports adding and removing items from front or back of the queue. It may be used as a stack or as a queue or both. This deque implementation supports the iterator protocol.

(new IDeque `n`**)**

Create a new deque with initial capacity for `n` items.

(==> `IDeque` number-of-items)

Number of items in deque.

(==> `IDeque` clear)

Clear deque. After calling this method the deque will be empty.

(==> `IDeque` push `i`**)**

Add item `i` to front of deque, return `()` .

(==> <IDeque push-all `v`**)**

Add all items in vector `v` to front of deque, return `()` .

(==> `IDeque` pop)

Remove item from front of deque and return it.

(==> `IDeque` next)

Remove item from front of deque and return it.

(==> `IDeque` peeknext)

Return item at the front of deque.

(==> `IDeque` puhslast `i`**)**

Add item `i` to back of deque, return `()` .

(==> `IDeque` puhslast-all `v`**)**

Add all items in vector `i` to back of deque, return `()` .

(==> `IDeque` poplast)

Remove item from back of deque and return it.

(==> `IDeque` peeklast)

Return item from back of deque.

9.2.5 Partition

A partition represents the partition of an arbitrary set of items into disjoint sets (called 'blocks'). Inserting new items, merging blocks, and determining equivalence (same block membership) of items are fast operations.

(new IPartition `n`)

Create a new partition with initial capacity for `n` items.

(==> `IPartition` new-block)

Create new singleton block and return item.

(==> `IPartition` make-new-blocks `n`)

Create `n` new singleton blocks and return item of last created singleton.

(==> `IPartition` clear)

Clear partition. After calling this method the partition object will be empty.

(==> `IPartition` find `i`)

Find index of proxy for item `i` .

(==> `IPartition` unify-blocks `i` `j`)

Merge the block containing item `i` with the block containing item `j` ; return `()` .

(==> `IPartition` same-block `i` `j`)

True if items `i` and `j` are in the same block.

(==> `IPartition` number-of-items)

Total number of items in `IPartition` .

(==> `IPartition` items)

Return all items in all blocks as an iterator.

(==> `IPartition` number-of-blocks)

Number of blocks in `IPartition` .

(==> `IPartition` size-of-block `i`)

Number of items in the block that includes item `i` .

(iterate `IPartition`)

Iterate over all blocks in arbitrary order (returns proxy items).
See: (==> `IPartition` blocks-small-to-large)
See: (==> `IPartition` blocks-large-to-small)

(==> `IPartition` blocks-small-to-large)

Iterate over all blocks from smallest to largest.

(==> `IPartition` blocks-large-to-small)

Iterate over all blocks from largest to smallest.

9.2.6 Heap / Priority Queue

A heap can hold arbitrary many items. Each item has an associated key (a double number). Inserting new items and finding the item with smallest key are fast operations.

(new Heap `n`)

Create a new min-heap with initial capacity for `n` objects.
Class `Heap` supports the iterator protocol.

(==> `Heap` insert `k` `i`)

Insert new item `i` with key/priority `k` into the heap.

(==> `Heap` insert-all `ks` `is`)

Insert all items in `is` with corresponding key in `ks` .

(==> `Heap` number-of-items)

Number of items in the heap.

(==> `Heap` key)

Return current minimum key.

(==> `Heap` peeknext)

Return the minimum-key item without removing it from the heap.

(==> `Heap` next)

Remove the minimum-key item from the heap and return it.

(==> `Heap` drop-lt `min-key`)

Remove all items with key strictly less than `min-key` and return key of new minimum item in heap or `NaN` when heap is empty.

(==> `Heap` clear)

Clear heap. After calling this method the heap object is empty.

(new IntHeap)

Create a new empty min-heap for integers.

(new SmallIntHeap `n`)

Create a new empty min-heap for integers in the range [0.. `n-1`].

A `SmallIntHeap` supports deletion (method `remove`) and updating of item keys. Unlike an `IntHeap` , a `SmallIntHeap` keeps only one instance of an item in the heap (items are unique). Use method `insert` to insert an new item or to change an item's key.

This heap implementation requires that the possible item values are in a fixed range [0.. `n-1`], where `n` must be specified at heap creation time. A `SmallIntHeap` uses memory linear in `n` . Like class `Heap` , `SmallIntHeap` supports the iterator protocol.

(==> `SmallIntHeap` remove `i`)

Remove item `i` from heap, return `t` on success and `()` when `i` was not in the heap.

9.2.7 Sets

The implementation of the set datatype is based on a self-adjusting binary search tree. Set classes support the iterator protocol. Iteration over sets proceeds from smallest to largest item.

(==> `OrderedSet` number-of-items)

Number of items in set.

(==> `OrderedSet` clear)

Clear set and return self. After invoking this method the set is empty.

(==> `OrderedSet` insert `i`)

Insert item `i` into set, return `t` on success and `()` when `i` was already in set.

(==> `OrderedSet` insert-all `is`)

Insert all items in vector `is` and return `()` .

(==> `OrderedSet` remove `i`)

Remove item `i` from set, return `t` on success and `()` when `i` was not in set.

(==> `OrderedSet` remove-range `from to`)

Remove items in interval [`from` .. `to`] from set, and return the number of items removed.

(==> `OrderedSet` remove-range* `from to`)

Remove items in interval [`from` .. `to`] and return them in a new set.

(==> `OrderedSet` member `i`)

True if item `i` is in the set.

(==> `OrderedSet` mininum)

Minimum item in set.

(==> `OrderedSet` maximum)

Maximum item in set.

(==> `OrderedSet` random)

Return a randomly chosen item from the set.

(do-set (`i s` [`from`]) `.` `body`)

Iterate in order over all items `i` in set `s` , optionally starting with item `from` .

(new IntSet)

Make a new empty set of integer items.

(new ShortSet)

Make a new empty set of short integer items.

(new CharSet)

Make a new empty set of char items.

(new UCharSet)

Make a new empty set of unsigned char items.

(new FloatSet)

Make a new empty set of float items.

(new DoubleSet)

Make a new empty set of double float items.

(new GptrSet)

Make a new empty set of gptr items.

(new StrSet)

Make a new empty set of string items.

(new IPairSet)

Make a new empty set of integer pair items.

9.2.8 Small Integer Sets

This set implementation is more efficient than the general `IntSet` . It requires that the possible values are in a fixed range [0.. `n` -1], where `n` must be specified at object creation time. A `SmallIntSet` requires memory linear in `n` . Another difference to an `IntSet` is that iterating over an `SmallIntSet` does not yield the set elements in order.

(new `SmallIntSet` `n`)

Create a `SmallIntSet` object with possible values in the range [0.. `n` -1].

(==> `SmallIntSet` number-of-items)

Number of items in set.

(==> `SmallIntSet` clear)

Clear set and return self. After invoking this method the set object is empty.

(==> `SmallIntSet` insert `i`)

Insert item `i` into set, return `t` on success and `()` when `i` was already in set.

(==> `SmallIntSet` insert-all `is`)

Insert all items in vector `is` and return `()` .

(==> `SmallIntSet` remove `i`)

Remove item `i` from set, return `t` on success and `()` when `i` was not in set.

(==> `SmallIntSet` member `i`)

True if item `i` is in the set.

(==> `SmallIntSet` minimum)

Minimum item in set.

(==> `SmallIntSet` maximum)

Maximum item in set.

(==> `SmallIntSet` random)

Return a randomly chosen item from the set.

(==> `SmallIntSet` complement)

Turn set into its complement and return `()` .

(==> `SmallIntSet` to-array)

Return all elements as a vector.

9.2.9 IntGraph

This class allows dynamic creation and manipulation of graphs. Vertices are identified with integers, edges with integer-pairs. The identifiers for vertices may be chosen freely, the identifiers for edges result from pairing two vertex identifiers.

An `IntGraph` object may represent directed or undirected graphs. Directed edges are called *arcs* , and undirected edges are called *edges* . For example, method `insert-edge` inserts an edge, method `insert-arc` inserts an arc into

the graph. Some methods make only sense for undirected graphs. The result of applying a method for undirected graphs to a graph that is not undirected is undefined.

(new `IntGraph`)

Create a new empty graph.

(==> `IntGraph` number-of-vertices)

(==> `IntGraph` number-of-edges)

(==> `IntrGraph` clear)

(==> `IntGraph` number-of-arcs)

(==> `IntGraph` new-vertex)

Insert a new vertex into graph and return it.

(==> `IntGraph` insert-vertex `v`)

Insert vertex `v` into graph, return `t` on success and `()` when `v` was already in the graph.

(==> `IntGraph` insert-vertices `vs`)

Insert vertices (integers) in table `vs` into graph and return `()` .

(==> `IntGraph` insert-vertex-range `v0` `v1`)

Insert vertices [`v0` .. `v1`], return `()` .

(==> `IntGraph` has-vertex `v`)

True when `v` is a vertex in the graph.

(==> `IntGraph` remove-vertex `v`)

Remove vertex `v` and all edges incident with `v` . Return `t` on success and `()` when `v` was not an existing vertex.

(==> `IntGraph` eliminate-vertex `v`)

Eliminate vertex `v` and return the set of new arcs added to the graph. Raise an error if vertex `v` does not exist.

(==> `IntGraph` merge-vertices `v0 v1`)

Merge vertices `v0` and `v1` or raise an error when there is no edge linking `v0` and `v1` . Return the set of new arcs added to the graph.

The resulting vertex is identified with `v0` , `v1` is not a vertex of the graph after completing this operation.

(==> `IntGraph` deficiency `v`)

Return deficiency set of vertex `v` (an arc table).

(==> `IntGraph` degree `v`)

Return out-degree of vertex `v` .

(==> `IntGraph` neighbors `v`)

Return all neighbors of vertex `v` as a set.

(==> `IntGraph` degrees)

Return out-degree for all vertices as a vector.

(==> `IntGraph` insert-edge `v0 v1`)

Insert edge `(v0,v1)` and return `t` when it is a new edge.

(==> `IntGraph` insert-edge* `v0 v1`)

Insert edge `(v0,v1)` and return `t` when it is a new edge.

Unlike `insert-edge` this method does not check existence of the vertices `v0` and `v1` .

(==> `IntGraph` has-edge `v0 v1`)

True when edge `(v0,v1)` is in the graph.

(==> `IntGraph` remove-edge `v0` <v1)

Remove edge `(v0,v1)` and return `t` when it was in the graph.

(==> `IntGraph` insert-arc `v0 v1`)

Insert arc `(v0,v1)` and return `t` when it is a new arc. Raise an error when either vertex is not in the graph.

(==> `IntGraph` has-arc `v0 v1`)

True when arc `(v0,v1)` is in the graph.

(==> `IntGraph` remove-arc `v0` <v1)

Remove arc `(v0,v1)` and return `t` when it was in the graph.

(==> `IntGraph` insert-edges `edges`)

Insert edges in table `edges` into graph and return `()` .

(==> `IntGraph` insert-arcs `arcs`)

Insert arcs in table `arcs` into graph and return `()` .

(==> `IntGraph` from-igraph `ig`)

(==> `IntGraph` undirected)

True if an undirected graph.

(==> `IntGraph` vector-dimension `ord`)

Return the vector dimension of the vertex ordering `ord` .

(==> `IntGraph` perfect `ord`)

True if `ord` is a perfect elimination ordering for this graph.

9.3 Image Processing Libraries

9.3.1 Displaying images

(new ImageShowWindow `title image thumbnails` [`named-funcs`])

Create a new ImageShowWindow object which contains an ImageViewer object and a toolbar (Column of IPTriggerButton objects). The string `title` gives the window title, `image` is an array. Optional argument `named-funcs` is an alist of (`label` . `function`) pairs. When provided, a button will be inserted into the toolbar for each entry in `named-funcs` .

See: image-show

image-disp-autoscale

See: image-disp

Boolean variable. When true, `image-disp` will linearly scale graylevel images to use the full dynamic range of the display device (default: `t`).

(image-disp `img` [`win` [`mag`]])

See: image-show, mat-disp

Display image in window.

With one argument, open a new window and display image `img` . With two arguments, use the existing window `win` for display. With three arguments, magnify the image by `mag` before display.

image-show-autoscale

See: image-disp

Boolean variable. When true, `image-show` will linearly scale graylevel images to use the full dynamic range of the display device (default: `t`).

(image-show `arg` [`label func` [**...**]])

See: image-disp

Open new image viewer and display image(s). Return the viewer.

When `arg` is an image, show it. When `arg` is an image filename, load the image and show it. When `arg` is a directory name, load all images in that directory and launch an interactive image viewer.

Following `arg` you may pass an alternating sequence of labels and functions. Each such function takes one image and returns one image. For each label-button pair the image viewer includes an extra button. Pressing a button triggers application of the associated function to the currently displayed image and the viewer is updated with the resulting image.

If a label character is followed by a dot ('.'), this character becomes the accelerator key for the corresponding button. Examples:

```
(image-show "lena.gif")
(image-show "lena.jpg" "transp.ose" mat-transpose "f.lipud" mat-flipud)
```

9.3.2 Reading/Writing Image Files to/from IDX

Functions to read/write image files into/from an IDX. Image files can be in any format that ImageMagick's convert command can handle. The type of the input image is determined automatically.

(image-info `f`**)**

Determine image characteristics from the image file and return them as an `htable` object. The entries in the hash table have the following meaning:

```
-----------------------------------------------------------------
Key             | Meaning
-----------------------------------------------------------------
```

```
format          | Image format
width           | Width in number of pixels
height          | Height in number of pixels
n-images        | Number of "scenes" or frames
n-colors-used   | Number of different colors in the image
                | (Not the same as the size of the colorspace!)
depth           | Number of bits per channel
label           | An embedded label string ("" if not present)
comment         | An embedded comment string ("" of not present)
colorspace      | Color space ("RGB", "Gray", "CMYK", ...).
palette-image-p | True if image is saved as a palette image
has-matte-p     | True if image has matte data
```

A palette image is a storage format where for each pixel there is an index into a color table (or "palette"). The color table is also included in the image file. Currently, reading in the index values nor reading in the color table is supported by Lush. Instead, when you load a palette image, the data gets automatically translated into RGB or grayscale values by `image-read-rgb` , `image-read-rgba` , or `image-read-ubim` , respectively.

The size of the color space (that is, the number of different possible color values representable in that space) is `2^depth` times `number of channels` (where depth is the bit depth of each channel). For instance, there are 3*2^8 different colors in RGB with bit depth 8. Note, however, that in a palette image the number of different possible colors is restricted to the size of the color table.

See: image-read-rgb, image-read-rgba, image-read-ubim

(image-read-rgb `f`)

read an image file (in any format), and return an rgbimage with the image in it (IDX3 of ubytes with last dimension=3). Accepted format are whatever ImageMagick's convert command can convert to PPM. The type of the input image is determined automatically. In case it is not, the the type can be prepended to the filename with a colon. Command line options prepended to the argument are passed to convert.

Example:

```
(setq img (image-read-rgb "myimage.png"))
(setq img (image-read-rgb "TIFF:myimage.dunno"))
(setq img (image-read-rgb "-geometry 50x50\\% myimage.jpg"))
```

(image-read-rgba `f`)

read an image file (in any format), and return an rgbaimage with the image in it (IDX3 of ubytes with last dimension=4). The alpha channel is filled if the image being read has one. An alpha value of 0 means opaque and 255 means fully transparent (this is backward to many conventions in other places in the

Lush library). Accepted format are whatever ImageMagick's convert command can convert to PPM. The type of the input image is determined automatically. In case it is not, the the type can be prepended to the filename with a colon Command line options prepended to the argument are passed to convert.

Example:

```
(setq img (image-read-rgba "-geometry 50x50\\% myimage.png"))
(setq img (image-read-rgba "-geometry 50x50\\% PNG:myimage"))
```

(image-read-ubim `f`)

read an image file (in any format), and return a ubimage with the image in it (IDX2 of ubytes) Accepted format are whatever ImageMagick's convert command can convert to PPM. Conversion to gryscale is performed by Lush, since convert doesn't convert to PGM.

(image-read `filename`)

Read an image (in any format) and return as scalar image (ubimage) if it is gray-scale, as RGBA if it contains alpha channel data, and as RGB otherwise. `image-read` finds images using `image-find` .

(image-file-p `filename`)

True if file `filename` is an image file.

(image-find `name`)

Find an image or a set of images. If `name` is a filename, search for the image in the current directory and in Lush's images directory and return the full filename. If `name` is a directory name, identify all image files in `name` and return their full filenames in a list.

(image-write-rgb `f im`)

Writes an image stored in an IDX3 of ubytes into an image file (in any format). The type of the output image file is determined by a string prepended to the filename with a colon. Command line options prepended to the argument are passed to the convert command

Example:

```
(image-write-rgb "TIFF:myimage.tiff" m))
```

(image-write-ubim `f im`**)**

Writes an image stored in an IDX2 of ubytes into an grayscale image file (in any format). The type of the output image file is determined by a string prepended to the filename with a colon. Command line options prepended to the argument are passed to the convert command

Example:

```
(image-write-ubim "myimage.png"))
```

(as-image `m`**)**

Rescale and quantize double-matrix `m` to an uchar-image.

The values of `m` are shifted and rescaled so that the values in the resulting uchar-matrix use the whole range 0..255.

9.3.3 Reading/Writing PPM/PGM/PBM Image Files

Routines for reading and writing PBM/PGM/PPM image files into/from idx. Another set of functions built on top of these is used to read/write image files in other formats than PPM/PGM/PBM.

(pnm-header `f`**)**

```
((-gptr- "FILE *") f):
Return the descriptive string of a PNM image file
(P6, P4, or P5), and skip the file descriptor
past the comments.
```

(mmap-idx3-storage `f u d0 d1 d2`**)**

```
((-gptr- "FILE *") f):
((-idx3- (-ubyte-)) u):
((-int-) d0 d1 d2):
Maps file f (at the current position given by FTELL) into idx3 u
does not check that f size is OK.
```

(mmap-idx2-storage `f u d0 d1`**)**

```
((-gptr- "FILE *") f):
((-idx2- (-ubyte-)) u):
((-int-) d0 d1):
Maps file f (at the current position given by FTELL) into idx2 u
does not check that f size is OK.
```

(pnm-fread-into-rgbx `f out`)

```
((-gptr- "FILE *") f):
```

read an image from a PBM/PGM/PPM file into an idx3 of ubytes (RGB or RGBA). `f` must be a valid file descriptor (C pointer). `out` is appropriately resized if required. The last dimension is left unchanged but must be at least 3. Appropriate conversion is performed. extra color components (beyond the first 3) are left untouched.

See: pnm

(pnm-read-into-rgbx `f out`)

read a PBM/PGM/PPM file into an idx3 of ubytes. The output idx is appropriately resized if necessary. The last dimension must be at least 3.

(pnm-fread-rgb `f`)

((-gptr- "FILE *") f) `f` must be an open file descriptor. read a PBM/PGM/PPM file and return an idx3 of ubytes with the image in it. Conversion to RGB is performed if necessary.

(pnm-read-rgb `s`)

read a PBM/PGM/PPM file and return an idx3 of ubytes with the image in it. Conversion to RGB is performed if necessary.

(pnm-fread-rgba `f`)

((-gptr- "FILE *") f) `f` must be an open file descriptor. read a PBM/PGM/PPM file and return an idx3 of ubytes with the image in it. Conversion to RGB is performed if necessary. The alpha channel is set to 0.

(pnm-read-rgba `s`)

read a PBM/PGM/PPM file and return an idx3 of ubytes with the image in it. Conversion to RGBA is performed if necessary. The alpha channel is set to 0.

(pnm-fread-into-ubim `f out`)

```
((-gptr- "FILE *") f):
((-idx2- (-ubyte-)) out):
```

read an image from a PBM/PGM/PPM file into an idx2 of ubytes (one byte per pixel). `f` must be a valid file descriptor (C pointer). `out` is appropriately resized if required. Appropriate conversion is performed (e.g. if the file contains an RGB image).

(pnm-read-into-ubim `f out`)

read a PBM/PGM/PPM file into an idx2 of ubytes (greyscale image). The output idx is appropriately resized if necessary. Appropriate conversions to greyscale are performed

(pnm-fread-ubim `f`)

((-gptr- "FILE *") f) `f` must be an open file descriptor. read a PBM/PGM/PPM file and return an idx2 of ubytes with the image in it. Conversion to greyscale is performed if necessary.

(pnm-read-ubim `f`)

read a PBM/PGM/PPM file and return an idx2 of ubytes with the image in it. Conversion to greyscale is performed if necessary.

(ppm-mmap `f`)

```
((-gptr- "FILE *") f):
maps a ppm RGB image (file descriptor <f>)
into memory and return an idx3 of ubytes
with the data.
```

(pgm-mmap `f`)

```
((-gptr- "FILE *") f):
maps a pgm image (file descriptor <f>)
into memory and return an idx2 of ubytes
with the data.
```

(ppm-fwrite-rgb `f m`)

writes RGB image `m` in file descriptor `f` as a PPM/PPM file. `m` is and idx3 of ubytes. `f` must be a file descriptor.

(ppm-write-rgb `f m`)

writes RGB image `m` in file `f` as a PPM/PPM file. `m` is and idx3 of ubytes.

(pgm-fwrite-ubim `f m`)

writes an ubimage `m` in file `f` as a PGM file. `m` is and idx2 of ubytes, and `f` is an open file descriptor.

(pgm-write-ubim `f m`**)**

writes greyscale ubimage `m` in file `f` as a PGM file. `m` is and idx2 of ubytes.

9.3.4 Reading/Writing PBM Image Files

reading and writing PBM image files (one bit per pixel, bitonal images) from and into lists of runs, and idx.

(pbm-write-runs `f runs nlin ncol`**)**

```
((-str-) f):
((-idx2- (-int-)) runs):
((-int-) nlin ncol):
CREATED: P. Haffner, Oct 97
DESCRIPTION:
writes <runs> in file <f> as a PBM file.
```

(pbm-write-ubim `f m`**)**

writes binary image `m` in file `f` as a PBM file. `m` is and idx2 of ubytes.

(pbm-write-ubim-val `f m v`**)**

writes binary image `m` in file `f` as a PBM file. `m` is and idx2 of ubytes. mask (value on) all values `v`

(pbm-read-ubim `f white black`**)**

read binary image `m` from PBM file `f` . code black as `black` and white as `white`

9.3.5 Reading/Writing Run-Length-Encoded Image Files

read and write RLE images. RLE images are a compact way of representing binary images by a list of "runs" of black pixels.

(rle-read-ubim `f white black`**)**

read binary image `m` from RLE file `f` . code black as `black` and white as `white`

(rle-read-runs `f runs img-dim`**)**

```
((-str-) f):
((-idx2- (-int-)) runs):
((-obj- (img-dim-C)) img-dim):
CREATED: P. Haffner, Feb 98
MODIFIES: <runs> and <img-dim>
RETURNS: ()
DESCRIPTION:
read runs IDX <runs> from RLE file <f>.
dimension 0 of <runs> is the number of runs
dimension 1 of <runs> must be >= 3 with
0: Y position
1: X1 position
2: X2 position
populates object img-dim with both dimensions
```

See: img-dim-C
EXAMPLE: (setq runs-dim (new img-dim-C 0 0)) (setq runs (int-array 1 4)) (a 4th optional dimension for connected componanet analysis)
(rle-read-runs filename runs runs-dim)

(rle-write-runs `f runs nlin ncol`**)**

```
((-str-) f):
((-idx2- (-int-)) runs):
((-int-) nlin ncol):
CREATED: P. Haffner, Oct 97
DESCRIPTION:
writes <runs> in file <f> as a RLE file.
```

9.3.6 RGBA Images of ubytes

basic image processing on RGBA images where each pixel component is an unsigned byte.

rgbaimage

a rgbaimage is an idx3 of ubytes intended to store images. The last dimension is 4, and contains the R, G, B, and Alpha channels respectively.

Geometric Transforms

A few functions to enlarge, reduce, resize, and rotate images.
(rgbaim-resize `im width height mode`**)**

resize an rgba image to any size using bilinear interpolation Appropriate local averaging (smoothing) is performed for scaling factors smaller than 0.5. If one of the desired dimensions is 0, an aspect-ratio-preserving scaling is performed on the basis of the other dimension. When both `width` and `height` are non zero, the last parameter, `mode` determines how they are interpreted.

- if either `width` or `height` is zero, `mode` is ignored.
- mode=0: fit the image into a `width` by `height` frame while preserving the aspect ratio
- mode=1: scale the image to `width` by `height` possibly changing the aspect ratio
- mode=2: `width` and `height` are interpreted as scaling ratios

The sizes of the output image are rounded to nearest integers smaller than the computed sizes, or to 1, whichever is largest.

(rgbaim-enlarge `in nlin ncol`**)**

enlarges image `in` with integer ratios `nlin` (vertical) `ncol` (horizontal). the enlarged image is returned. The horizontal (resp. vertical) size of the returned image is equal to the integer part of the horizontal (resp vertical) size of `in` divided by `ncol` (resp `nlin`). returns (copy-array `in`) when enlarge rate is 1

(rgbaim-enlarge-into `in out nlin ncol`**)**

enlarges image `in` with integer ratios `nlin` (vertical) `ncol` (horizontal) and write result in `out` . The horizontal (resp. vertical) size of the output image is equal to the integer part of the horizontal (resp vertical) size of `in` divided by `ncol` (resp `nlin`).

(rgbaim-subsample `in nlin ncol`**)**

subsamples image `in` with integer ratios `nlin` (vertical) `ncol` (horizontal). the subsampled image is returned. The horizontal (resp. vertical) size of the returned image is equal to the integer part of the horizontal (resp vertical) size of `in` divided by `ncol` (resp `nlin`). returns (copy-array `in`) when subsample rate is 1

(rgbaim-subsample-into `in out nlin ncol`**)**

subsamples image `in` with integer ratios `nlin` (vertical) `ncol` (horizontal) and write result into `out` . The horizontal (resp. vertical) size of the output image is equal to the integer part of the horizontal (resp vertical) size of `in` divided by `ncol` (resp `nlin`).

(rgbaim-rot90-left `inp`**)**

rotate image `inp` by 90 to the left (counter-clockwise) and return the result.

(rgbaim-rot90-right `inp`**)**

rotate image `inp` by 90 to the right (clockwise) and return the result.

(rgbaim-rot180 `inp`**)**

rotate image `inp` by 180 degrees and return the result.

(rgbaim-rot180-inplace `in`**)**

in-place 180 degree rotation of an image (caution: input image is destroyed).

(rgbaim-crop `in x y w h`**)**

crop rectangle (`x` , `y` , `w` , `h`) from image `in` and return the result (a copy).

(rgbaim-warp-quad `in out background mode x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3`**)**

```
((-flt-) x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3):
((-int-) background mode):
((-idx2- (-ubyte-)) in out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
warp rgbaimage <in> through a geometric transformation that maps
a quadrilateral into a rectangle (bilinear transform).
The quadrilateral is specified by <x1> <y1> <x2> <y2> <x3> <y3> <x4> <y4>
(points are numbered clockwise starting from upper left).
The rectangle is specified by the upper left and lower right points
which are respectively <p1><q1> and <p3><q3>.
The result is put in <out>.  Clipping is automatically performed.
pixels outside of the bounds of <in> are assumed to have the value
<background> (which must be an idx1 of ubyte of size 4).
<mode>=0: no antialiasing, <mode>=1: antialiasing with
bilinear interpolation (2 times slower).
```

(rgbaim-warp `in out background pi pj`**)**

```
((-idx1- (-ubyte)) background):
((-idx3- (-ubyte-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of ubytes. <background> is the value assumed outside
of the input image. <pi> and <pj> are integers idx whose dimensions must
be identical to the first two dimensions of <out>.
An output pixel at coordinate (x,y) takes the value of the input pixel at
coordinate <pj[x,y]>,<pi[x,y]>. If the values in <pi> and <pj> are non
integers the pixels are computed by bilinearly interpolating the input
```

```
image. The <pi> and <pj> matrices contain 32-bit integers which are
interpreted as 16 bit of integer part and 16 bits of fractinal part.
Integers values are assumed to fall in the center of each pixel,
so the upper left-hand corner of an image is at coordinate (-0.5, -0.5).
```

(rgbaim-warp-fast `in out background pi pj`**)**

```
((-int-) background):
((-idx2- (-ubyte-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of ubytes. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates
This is essentially identical to rgbaim-warp, escept no bilinear
interpolation is performed (only a nearest neighbor rule is used).
```

(rgbaim-rotscale `src sx sy dst dx dy angle coeff bg`**)**

rotate, scale, and translate image `src` and put result in `dst` . point `sx` , `sy` in `src` will be mapped to point `dx` , `dy` in `dst` . Image will be rotated clockwise by `angle` degrees and scaled by `coeff` . Pixels that fall off the boundary are clipped and pixels in the destination that are not determined by a source pixel are set to color `bg` (which must be an idx1 of ubytes of size 4). It is generally preferable to call rgbaim-rotscale-rect before hand to get appropriate values for `dx` , `dy` and for the size of `dst` so that no pixel is clipped. Here is an example:

```
(let* ((w (idx-dim m 1))
       (h (idx-dim m 0))
       (wh (int-array 2))
       (cxcy (float-array 2))
       (bg (ubyte-array 4)))
   (rgbaim-rotscale-rect w h hotx-src hoty-src angle coeff wh cxcy)
   (let ((z (ubyte-array (wh 1) (wh 0) 4)))
     (rgbaim-rotscale m hotx-src hoty-src z (cxcy 0) (cxcy 1) angle coeff bg)))
```

(rgbaim-rotscale-rect `w h cx cy angle coeff wh cxcy`**)**

Given an input image of width `w` , height `h` , with a "hot" point at coordinate `cx` and `cy` , this function computes the width, height, and hot point coordinates of the same image rotated by `angle` and scaled by `coeff` so as to ensure that no pixel in the rotated/scaled image will have negative coordinates (so the image will not be clipped). `wh` and `cxcy` which must be idx1 of floats with two elements. This function should be called before rgbaim-rotscale.

Low-Level Geometric Transform Functions

These are unlikely to be used directly by most users.

(rgbaim-interpolate-bilin `background pin indimi indimj inmodi inmodj ppi ppj rez`**)**

Author(s): Y. LeCun

```
(-gptr- pin rez background):
(-int-  indimi indimj inmodi inmodj ppi ppj):
SIDE EFFECTS: <rez>
DESCRIPTION:
returns a bilinearly interpolated RGBA pixel value for coordinate
<ppi> <ppj>. The image data is pointed to by <pin>, with
<indimi> <indimj> <inmodi> <inmodj> being the dimensions and modulos.
This function clips automatically if <ppi> <ppj> are outside of the
bounds by assuming the outside values are equal to <background>.
pixel values are ubytes, while coordinates are 32 bit fixed point
with 16 bit integer part and 16 bit fractional part.
The function does not use floating point arithmetics.
```

Color Processing

(rgbaim-contbright `in out c b`**)**

correct contrast and brightness of image `in` and put result in `out` (`in` and `out` must be the same size). `c` and `b` are floats that will affect the brightness and contrast respectively. `c` =1 and `b` =0 leave the image unchanged. Each output pixel is computed as follows: output = (input-128)*c + b + 128 (clipped between 0 and 255). An image can be conveniently inverted by setting `c` to -1 and `b` to 0.

(rgbaim-luminance `in out`**)**

compute luminance of each pixel and put result in image `out` the formula used is lum = 0.299 * R + 0.587 * G + 0.114 * B

(rgbaim-lum2rgba `in out`**)**

transform a grey-level image `in` (idx2 of flts) into an RGBA image. `in` and `out` must have identical sizes, though no check of that is done

(rgbaim-uvw `in out`**)**

transform an RGBA image into an UVWA image, where the V component contains the luminance information, and the U and W components collectively contain the chrominance information. In the UVW color coding scheme the euclidean distance is meant to reflect the subjective "perceptual" distance. See Digital Image Processing by W. Pratt, page 66.

(rgbaim-saturate `in min max out`**)**

Saturate pixel component values in image `in` to the range [`min` , `max`]. Result is put in `out` .

Blitting

(rgbaim-blit `x y in out`**)**

blit `in` into `out` at position `x` , `y` with alpha blending. an alpha value of 0 in `in` means opaque, 255 means totally transparent.

(rgbaim-blitcolor `x y in r g b out`**)**

blit color defined by `r` , `g` , `b` using idx2 `in` as a stencil into RGBA image `out` at position `x y` The values in `in` must be between 0 and 255. if the value is 255, no paint is put in; if the value is 0, the pixel takes the value specified by `r` , `g` , `b` .

Histograms

(rgbaim-histo32 `rgbaim ppal count`**)**

compute histogram of image `rgbaim` . `ppal` must be a 32768 by 3 matrix of floats which will contain the list of pixel values found. `count` must be a 32768 matrix which will contain the corresponding pixel counts. pixel colors are sorted by luminance. This function really computes a 32768 color palette by assigning each pixel to the cube it belongs to in the 32x32x32 RBG color cube. The color prototype assigned to each cube is the mean of the pixels belonging to the cube.

Low-Level Histogram Functions

(rgbaim-init-lum2rgb)

build a table of 32x32x32 elements containing RGB cell indices in ascending luminosity.

(rgbaim-lum2rgb `n`**)**

converts a luminosity index to a 5-bit RGB cell number.

Various Unusual Subsampling Functions

(rgbaim-subsample+ `in nlin ncol`**)**

same as `rgbaim-subsample` , but adds one row and one column for lazy pixels

(rgbaim-subsample-med3 `rgbaim`**)**

subsample images by 3x3 using a median filter.

(rgbaim-med3-subopt `rgbaim`**)**

3x3 subsampling using the median smart median optimised in C for 3x3 squares: 1.3 sec

(rgbaim-med3-slow rgbaim)

3x3 subsampling using the median brute force median, with a 9 element quicksort

9.3.7 RGBA Images of floats

basic image processing on RGBA images where each pixel component is a float.

rgbafimage

a rgbafimage is an idx3 of flts intended to store images. The last dimension is 4, and contains the R, G, B, and Alpha channels respectively.

(rgbafim-subsample `in nlin ncol`**)**

subsamples image `in` with integer ratios `nlin` (vertical) `ncol` (horizontal). the subsampled image is returned. The horizontal (resp. vertical) size of the returned image is equal to the integer part of the horizontal (resp vertical) size of `in` divided by `ncol` (resp `nlin`).

(rgbafim-luminance `in out`**)**

compute luminance of each pixel and put result in image `out` the formula used is lum = 0.299 * R + 0.587 * G + 0.114 * B

(rgbafim-lum2rgba `in out`**)**

transform a grey-level image `in` (idx2 of flts) into an RGBA image. `in` and `out` must have identical sizes, though no check of that is done

(rgbafim-chromin-broken `in z k out`**)**

attempts to compute the chrominance info in `in` (normalized colors with almost identical luminance) and put the result in `out` . Both `in` and `out` are idx3 of flts. `z` is a multiplier for the average pixel component value. `k` is a fudge factor that prevents the normalisation from blowing up if the color is black or almost black.

(rgbafim-chrominance `in z k out`**)**

attempts to compute the chrominance info in `in` (normalized colors with almost identical luminance) and put the result in `out` . Both `in` and `out` are idx3 of flts. `z` is a multiplier for the average pixel component value. `k` is a fudge factor that prevents the normalisation from blowing up if the color is black or almost black.

(rgbafim-uvw `in out`**)**

transform an RGBA image into an UVWA image, where the V component contains the luminance information, and the U and W components collectively contain the chrominance information. In the UVW color coding scheme the euclidean distance is meant to reflect the subjective "perceptual" distance. See Digital Image Processing by W. Pratt, page 66.

(rgbafim-fromuvw `in out`**)**

converts back from UVWA to RGBA. It is advasable to saturate the resulting image, as numerical errors may cause the pixel values to exceed the range 0-255.

(rgbafim-saturate `in min max out`**)**

Saturate pixel component values in image `in` to the range [`min` , `max`]. Result is put in `out` .

(rgbafim-blit `x y in out`**)**

blit `in` into `out` at position `x` , `y` with alpha blending. an alpha value of 0 in `in` means opaque, 255 means totally transparent.

(rgbafim-blitcolor `x y in r g b out`**)**

blit color defined by `r` , `g` , `b` using idx2 `in` as a stencil into RGBA image `out` at position `x y` The values in `in` must be between 0 and 255. if the value is 255, no paint is put in; if the value is 0, the pixel takes the value specified by `r` , `g` , `b` .

(rgbafim-init-lum2rgb)

build a table of 32x32x32 elements containing RGB cell indices in ascending luminosity.

(rgbafim-lum2rgb `n`**)**

converts a luminosity index to a 5-bit RGB cell number.

(rgbafim-histo32 `rgbafim ppal count`**)**

compute histogram of image `rgbafim` . `ppal` must be a 32768 by 3 matrix which will contain the list of pixel values found. `count` must be a 32768 matrix which will contain the corresponding pixel counts. pixel colors are sorted by luminance. This function really computes a 32768 color palette by assigning each pixel to the cube it belongs to in the 32x32x32 RBG color cube. The color prototype assigned to each cube is the mean of the pixels belonging to the cube.

(rgbafim-interpolate-bilin `background pin indimi indimj inmodi inmodj ppi ppj rez`**)**

```
(-gptr- pin rez background):
(-int-  indimi indimj inmodi inmodj ppi ppj):
SIDE EFFECTS: <rez>
AUTHOR: Y. LeCun
```

```
COMPILABLE: Yes
DESCRIPTION:
returns a bilinearly interpolated RGBA pixel value for coordinate
<ppi> <ppj>. The image data is pointed to by <pin>, with
<indimi> <indimj> <inmodi> <inmodj> being the dimensions and modulos.
This function clips automatically if <ppi> <ppj> are outside of the
bounds by assuming the outside values are equal to <background>.
pixel values are flts, while coordinates are 32 bit fixed point
with 16 bit integer part and 16 bit fractional part.
```

(rgbafim-warp in out background pi pj**)**

```
((-idx1- (-flt)) background):
((-idx3- (-flt-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of flts. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
Pixel values are antialiased using bilinear interpolation.
```

(rgbafim-warp-fast in out background pi pj**)**

```
((-int-) background):
((-idx2- (-flt-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of flts. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
This is essentially identical to warp-rgbafimage, except no antialiasing
is performed (it goes a lot faster, but is not nearly as nice).
```

(rgbafim-warp-quad `in out background mode x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3`**)**

```
((-flt-) x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3):
((-int-) background mode):
((-idx2- (-flt-)) in out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
transforms rgbafimage <in> (idx2 of flt) mapping quadrilateral <x1> <y1> <x2>
<y2> <x3> <y3> <x4> (points are numbered clockwise starting from upper left)
to rectangle whose upper left and lower right points are <p1><q1>, <p3><q3>.
result is put in <out> (idx2 of flt).  Clipping is automatically performed.
pixels outside of the bounds of <in> are assumed to have the value
<background>.  <mode>=0: no antialiasing, <mode>=1: antialiasing with
bilinear interpolation (2 times slower).  execution time on sparc 10 is about
5 ms in mode 0 and 10 ms in mode 1 for a 32x32 target image.
```

9.3.8 Greyscale Images of ubytes

linear coordinate transform using bilinear interpolation.

ubimage

a ubimage is an idx2 of ubytes intended to store image.

(ubimage2flt `in out`**)**

```
((-idx2- (-ubyte-)) in):
((-idx2- (-flt-)) out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Converts ubimage <in> into idx2 of flt <out>.
Values are scaled down by 1/255.
```

(flt2ubimage `in out`**)**

```
((-idx2- (-ubyte-)) out):
((-idx2- (-flt-)) in):
```

```
RETURNS: Null
SIDE EFFECTS: None
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
converts flt idx2 <in> to an ubimage <out>.
a pixel values are scaled up by 255.
```

(ubimage2fltimage `in out bkgd ink`**)**

```
((-idx2- (-ubyte-)) in):
((-idx2- (-flt-)) out):
((-flt-) bkgd ink):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: C. Burges
COMPILABLE: Yes
DESCRIPTION:
Generalization of Yann's ubimage2flt for arbitrary ink and background.
Converts ubimage <in> into idx2 of flt <out>.
```

(fltimage2ubimage `in out bkgd ink`**)**

```
((-idx2- (-flt-)) in):
((-idx2- (-ubyte-)) out):
((-flt-) bkgd ink):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: C. Burges
COMPILABLE: Yes
DESCRIPTION:
Generalization of Yann's flt2ubimage for arbitrary ink and background.
Converts flt idx2 <in> to a ubimage <out>.
```

(ubim-resize `im width height mode`**)**

resize a greyscale image to any size using bilinear interpolation Appropriate local averaging (smoothing) is performed for scaling factors smaller than 0.5. If one of the desired dimensions is 0, an aspect-ratio-preserving scaling is performed on the basis of the other dimension. When both `width` and `height` are non zero, the last parameter, `mode` determines how they are interpreted.

- if either `width` or `height` is zero, `mode` is ignored.
- mode=0: fit the image into a `width` by `height` frame while preserving the aspect ratio

- mode=1: scale the image to `width` by `height` possibly changing the aspect ratio
- mode=2: `width` and `height` are interpreted as scaling ratios

The sizes of the output image are rounded to nearest integers smaller than the computed sizes, or to 1, whichever is largest.

(ubim-subsample `in nlin ncol`**)**

subsamples image `in` with integer ratios `nlin` (vertical) `ncol` (horizontal). the subsampled image is returned. The horizontal (resp. vertical) size of the returned image is equal to the integer part of the horizontal (resp vertical) size of `in` divided by `ncol` (resp `nlin`).

returns (copy-array `in`) when subsample rate is 1

(ubim-zoom `ubim zoomx zoomy`**)**

Zoom an ubyte image `ubim` by the given integer zoom factors.

(ubim-invert `im`**)**

reverse polarity of an ubimage. each pixel is replaced by 255 minus itself

(ubim-interpolate-bilin `background pin indimi indimj inmodi inmodj ppi ppj`**)**

```
(-gptr- pin):
(-int-  background indimi indimj inmodi inmodj ppi ppj):
RETURNS: (-ubyte-)
SIDE EFFECTS: None
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
returns a bilinearly interpolated pixel value for coordinate
<ppi> <ppj>. The image data is pointed to by <pin>, with
<indimi> <indimj> <inmodi> <inmodj> being the dimensions and modulos.
This function clips automatically if <ppi> <ppj> are outside of the
bounds by assuming the outside values are equal to <background>.
pixel values are ubytes, while coordinates are 32 bit fixed point
with 16 bit integer part and 16 bit fractional part.
The function does not use floating point arithmetics.
```

(ubim-warp in out background pi pj**)**

```
((-int-) background):
((-idx2- (-ubyte-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of ubytes. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
Pixel values are antialiased using bilinear interpolation.
```

(ubim-warp-fast in out background pi pj**)**

```
((-int-) background):
((-idx2- (-ubyte-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of ubytes. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
This is essentially identical to warp-ubimage, except no antialiasing
is performed (it goes a lot faster, but is not nearly as nice).
```

(ubim-warp-quad in out background mode x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3**)**

```
((-flt-) x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3):
((-int-) background mode):
((-idx2- (-ubyte-)) in out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
```

```
transforms ubimage <in> (idx2 of ubyte) mapping quadrilateral <x1> <y1> <x2>
<y2> <x3> <y3> <x4> <y4> (points are numbered clockwise starting from upper left)
to rectangle whose upper left and lower right points are <p1><q1>, <p3><q3>.
result is put in <out> (idx2 of ubyte).  Clipping is automatically performed.
pixels outside of the bounds of <in> are assumed to have the value
<background>.  <mode>=0: no antialiasing, <mode>=1: antialiasing with
bilinear interpolation (2 times slower).  execution time on sparc 10 is about
5 ms in mode 0 and 10 ms in mode 1 for a 32x32 target image.
```

9.3.9 Greyscale Images of shorts

functions to manipulate images where each pixel is a 16-bit short.

shimage

a shimage is an idx2 of shorts intended to store image. the pixel values are interpreted as fixed point integers, with the upper byte being the integer part, and the lower byte being the fractional part.

(**shimage2flt** in out)

```
((-idx2- (-short-)) in):
((-idx2- (-flt-)) out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Converts shimage <in> into idx2 of flt <out>.
Values are scaled down by 1/256.
```

(**flt2shimage** in out)

```
((-idx2- (-short-)) out):
((-idx2- (-flt-)) in):
RETURNS: Null
SIDE EFFECTS: None
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
converts flt idx2 <in> to an shimage <out>.
a pixel values are scaled up by 256.
```

(shimage2fltimage in out bkgd ink**)**

```
((-idx2- (-short-)) in):
((-idx2- (-flt-)) out):
((-flt-) bkgd ink):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: C. Burges
COMPILABLE: Yes
DESCRIPTION:
Generalization of Yann's shimage2flt for arbitrary ink and background.
Converts shimage <in> into idx2 of flt <out>.
```

(fltimage2shimage in out bkgd ink**)**

```
((-idx2- (-flt-)) in):
((-idx2- (-short-)) out):
((-flt-) bkgd ink):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: C. Burges
COMPILABLE: Yes
DESCRIPTION:
Generalization of Yann's flt2shimage for arbitrary ink and background.
Converts flt idx2 <in> to a shimage <out>.
```

(shim-interpolate-bilin background pin indimi indimj inmodi inmodj ppi ppj**)**

```
(-gptr- pin):
(-int-  background indimi indimj inmodi inmodj ppi ppj):
RETURNS: (-short-)
SIDE EFFECTS: None
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
returns a bilinearly interpolated pixel value for coordinate
<ppi> <ppj>. The image data is pointed to by <pin>, with
<indimi> <indimj> <inmodi> <inmodj> being the dimensions and modulos.
This function clips automatically if <ppi> <ppj> are outside of the
bounds by assuming the outside values are equal to <background>.
pixel values are shorts, while coordinates are 32 bit fixed point
with 16 bit integer part and 16 bit fractional part.
The function does not use floating point arithmetics.
```

(shim-warp in out background pi pj**)**

```
((-int-) background):
((-idx2- (-short-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of shorts. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
Pixel values are antialiased using bilinear interpolation.
```

(shim-warp-fast in out background pi pj**)**

```
((-int-) background):
((-idx2- (-short-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of shorts. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
This is essentially identical to warp-shimage, except no antialiasing
is performed (it goes a lot faster, but is not nearly as nice).
```

(shim-warp-quad in out background mode x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3**)**

```
((-flt-) x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3):
((-int-) background mode):
((-idx2- (-short-)) in out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
```

```
transforms shimage <in> (idx2 of short) mapping quadrilateral <x1> <y1> <x2>
<y2> <x3> <y3> <x4> (points are numbered clockwise starting from upper left)
to rectangle whose upper left and lower right points are <p1><q1>, <p3><q3>.
result is put in <out> (idx2 of short).  Clipping is automatically performed.
pixels outside of the bounds of <in> are assumed to have the value
<background>.  <mode>=0: no antialiasing, <mode>=1: antialiasing with
bilinear interpolation (2 times slower).  execution time on sparc 10 is about
5 ms in mode 0 and 10 ms in mode 1 for a 32x32 target image.
```

9.3.10 Greyscale Images of floats

basic image processing on greyscale images where each pixel is a float.

(fim-subsample `in nlin ncol`**)**

subsample image `in` by integer ratios `nlin` vertically and `ncol` horizontally (with averaging).

(fim-interpolate-bilin `background pin indimi indimj inmodi inmodj ppi ppj`**)**

```
(-gptr- pin):
(-int-  background indimi indimj inmodi inmodj ppi ppj):
RETURNS: (-float-)
SIDE EFFECTS: None
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
returns a bilinearly interpolated pixel value for coordinate
<ppi> <ppj>. The image data is pointed to by <pin>, with
<indimi> <indimj> <inmodi> <inmodj> being the dimensions and modulos.
This function clips automatically if <ppi> <ppj> are outside of the
bounds by assuming the outside values are equal to <background>.
pixel values are floats, while coordinates are 32 bit fixed point
with 16 bit integer part and 16 bit fractional part.
The function does not use floating point arithmetics.
```

(fim-warp `in out background pi pj`**)**

```
((-float-) background):
((-idx2- (-float-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
```

```
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of floats. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
Pixel values are antialiased using bilinear interpolation.
```

(fim-warp-fast `in out background pi pj`**)**

```
((-int-) background):
((-idx2- (-float-)) in out):
((-idx2- (-int-)) pi pj):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Warps an image using <pi> and <pj> as tabulated coordinate transforms.
<in> and <out> are idx2 of floats. <background> is the value assumed outside
of the input image. <pi> and <pj> are tabulated coordinates which can
be filled up using compute-bilin-transform or similar functions.
This is essentially identical to warp-fimage, except no antialiasing
is performed (it goes a lot faster, but is not nearly as nice).
```

(fim-warp-quad `in out background mode x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3`**)**

```
((-flt-) x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3):
((-float-) background)
((-int-) mode):
((-idx2- (-float-)) in out):
RETURNS: Null
SIDE EFFECTS: <out>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
transforms fimage <in> (idx2 of float) mapping quadrilateral <x1> <y1> <x2>
<y2> <x3> <y3> <x4> (points are numbered clockwise starting from upper left)
to rectangle whose upper left and lower right points are <p1><q1>, <p3><q3>.
result is put in <out> (idx2 of float).  Clipping is automatically performed.
pixels outside of the bounds of <in> are assumed to have the value
<background>.  <mode>=0: no antialiasing, <mode>=1: antialiasing with
bilinear interpolation (2 times slower).  execution time on sparc 10 is about
```

```
5 ms in mode 0 and 10 ms in mode 1 for a 32x32 target image.
```

9.3.11 Computing Image Warping Maps

compute displacement tables for 2D geometric transformations of images, such as bilinear warps that transform any quadrilateral into any rectangle.

(compute-bilin-transform `dispi dispj x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3`)

```
((-idx2- (-int-)) dispi):
((-idx2- (-int-)) dispj):
((-flt-) x1 y1 x2 y2 x3 y3 x4 y4 p1 q1 p3 q3):
RETURNS: Null
SIDE EFFECTS: <dispi> <dispj>
AUTHOR: Y. LeCun
COMPILABLE: Yes
DESCRIPTION:
Tabulates a bilinear transformation that maps the quadrilateral defined
by the <xi> <yi> to a rectangle whose upper left point is <p1> <q1> and
lower right point is <p3> <q3>.
<x1> <y1> is the upper left point on the quadrilateral, and the points
are numbered clockwise.
<dispi> and <dispj> must be idx2 of int. On output, element (i,j)
of <dispi> and <dispj> will be filled respectively with the abscissa
and ordinate of the point in the quadrilateral that maps to point (i,j).
Values in these idx2 are interpreted as 32bit fixed point numbers
with 16 bit integer part and 16 bit fractional part.
<dispi> and <dispj> can subsequently be used as input to
warp-shimage, or warp-shimage-fast.
This function makes minimal use of floating point arithmetics.
```

9.3.12 Connected Component Analysis

elementary routines for performing a connected component (CC) analysis on images. The basic procedure consists in:

- Creating a new `CCAnalyzer` object for a given image
- Extracting the runs using method `run-analysis` .
- Extracting the connected components using method `cc-analysis` .

There are a few other methods for performing various cleanup on the runs and on the CCs. There is also a method for reconstructing the image from the cleaned CCs.

Another trick: There is a matrix named `ubcut` in the `CCAnalyzer` object. Every pixel set in this matrix will force the RUN and CC code to ignore connexity on certain pixels. For instance, you can force arbitrary cuts by setting the pixels of a 4-connex line in this matrix. This has to be done before the call to `run-analysis` .

– Leon Bottou 9/96

CCAnalyzer

Class for connected component analysis.

(new CCAnalyzer `grayimage`)

Class CCanalyzer holds all the information necessary for the connected component analysis of an image `grayimage` .

(==> `ccanalizer` run-analysis `threshold`)

Computes a run length representation of the binary image obtained by thresholdind the gray image `ubmatrix2d` at threshold `thres` . The run information is stored into slot `runs` of the `CCAnalizer` object.

Each line of matrix `runs` is a 4-vector containing the following information:

- At index `(RUN-Y)` : The Y coordinate of the run.
- At index `(RUN-X1)` : The X coordinate of the first pixel of the run.
- At index `(RUN-X2)` : The X coordinate of the last pixel of the run.
- At index `(RUN-ID)` : The ID of the blob this run belongs to. The blob information is filled by the CA extraction code.

This function breaks runs on each non zero pixel of the image located in slot `ubcut` of `ccanalyzer` . The default value is all zeroes.

This function returns `t` on success.

(==> `ccanalyzer` cc-analysis)

Perform the connected component analysis of the image represented by the runs matrix contained in the `ccanalyzer` object. This algorithm usually considers 8-connexity. This is altered when the cut image `ubcuts` contains a non zero value indicating a forced cut.

The results are stored in the cc descriptor matrix `ccdesc` of the object. Each line of this matrix represent a connected component (CC).

- At index `(CC-NRUN)` : The number of runs in the CC.
- At index `(CC-FRUN)` : The index of the first run of the CC in the CC ordered run matrix `ccruns` stored in the object.

- At index `(CC-NPIX)` : The number of non background pixels in the CC.
- At index `(CC-WPIX)` : The sum of the pixel values (gray level) of the CC. These values are initially zero. Use function `cc-measure-gray` to compute these values.
- At indices `(CC-LEFT)` , `(CC-TOP)` , `(CC-RIGHT)` and `(CC-BOTTOM)` : The boundign box of the CC.

This function returns `t` on success. You may postprocess the returned CC with `cc-measure-gray` , `cc-remove-specs` , and other functions in this file.

(==> `ccanalyzer` remove-long-runs `maxlen`)

This function removes the runs whose length is longer than `maxlen` . Long runs are sometimes good indicator for underlines. Note that nothing is changed in the image. You must use `reconstruct-from-runs` or `reconstruct-from-cc` for that.

(==> `ccanalyzer` reconstruct-from-runs)

Reconstruct matrix `ubimg` from the run representation. This code saves the gray level information already present in `ubimg` . It just sets all pixels not included in a run to zero.

(==> `ccanalyzer` cc-measure-gray)

Compute the average gray level of each CC and store it at index `(CC-WPIX)` of the CC descriptor matrix `ccdesc` .

(==> `ccanalyzer` remove-small-cc `minpix` `mingray` `minweight`)

Remove small connected components

- whose size is less than `minpix` pixels,
- or whose average gray level is less than `mingray` ,
- or whose summed gray level is less than `minweight` .

Note that nothing is changed in the image or the run matrix. You must use `reconstruct-from-cc` for that.

(==> `ccanalyzer` reconstruct-from-cc)

Reconstruct matrix `ubimg` from the connected components in `ccdesc` . This code saves the gray level information already present in `ubimg` . It just sets all pixels not included in a run to zero.

(==> `ccanalyzer` **overlay-cc** `ccid ccimg originx originy`)

Overlays connected component `ccid` into ubyte matrix `ccimg` . Matrix `ccimg` is assumed to have its top left corner at position `originx originy` in the intial image. Connected components that do not fit matrix `ccimg` are silently clipped. Returns `t` when something was drawn.

(==> `ccanalyzer` **draw-cc** `x y`)

Draws image at location `x y` by painting each connected component with a random color. Unfortunately, random colors are sometimes too close. [INTERPRETED ONLY]

(==> `ccanalyzer` **pick-cc** `x y`)

returns the index of the connected component that contains the point `x` , `y` , or -1 if no component contains the point.

(==> `ccanalyzer` **image-height**)

returns height of image on which CCA is performed

(==> `ccanalyzer` **image-width**)

returns width of image on which CCA is performed

(==> `ccanalyzer` **run-histo** `histo hosto offset`)

compute histogram of runs. `histo` is an N by P matrix, and `hosto` an N vector. this method will accumulate the histogram of run lengths for each scan line between `offset` and `offset` +N-1. on output, elements i, j of histo will be incremented by the number of runs on line i- `offset` that are of length j. element i of `hosto` will incremented by the number of runs on scanline i- `offset` that are larger or equal to P.

(==> `ccanalyzer` **bbox-histo** `histo`)

computes 2D histogram of bounding box heights and width and increment 2D matrix `histo` by it. Element i, j of `histo` will be incremented by the number of CC whose bounding box is of height i and of width j.

(==> `ccanalyzer` **ccbottoms** `thres`)

return a matrix with x and y of horixontal center of bottom of each CC. CCs whose number of black pixels are smaller than `thres` are ignored.

(==> CCAnalyzer get-cc-fltim cc-id cc-fltim cc-com cc-off)

```
  ((-int-) cc-id) : index of the cc
  ((-idx2- (-flt-)) cc-fltim) : image in float
  ((-idx1- (-flt-)) cc-com) : center of mass,  in float [X,Y]
  ((-idx1- (-int-)) cc-off) : offset [X,Y]
ADDED: Patrick Haffner, July 97


Given the index of a CC, returns:
1) the corresponding gray level float image, dilated (WITH OFFSET)
2) the center of mass of the bitmap (WITHOUT OFFSET)
3) the offset used to align the float image to the upper right corner.
WARNING: fltim expands all the edges of the bounding box
by 1 pixel, UNLESS this cannot be done because we hit the
global image border.
```

9.3.13 Morphological Operations

Author(s): Leon Bottou 09/96

Erosion, dilation, distance transforms, thresholding, and masking on on images of ubytes.

(ubim-internal-disttrans `im dist`**)**

Returns the internal distance transformation matrix of binary image `im` . The elements of the output ubyte matrix `dist` contains the distance to the closest background pixel in the image (background level is zero for this purpose). Distances are clipped to 255. The dimensions of `dist` must be 2 larger than that of `im` . `dist` will be resized to the appropriate size if necessary.

(ubim-external-disttrans `im dist`**)**

computes the external distance transformation matrix of binary image `im` . The elements of the output ubyte matrix `dist` contains the distance to the closest non-background pixel in the image (background level is zero for this purpose). Distances are clipped to 255. The dimensions of `dist` must be 2 larger than that of `im` . `dist` will be resized to the appropriate size if necessary.

(ubim-positive-threshold `im thres`**)**

Applies a threshold on ubyte image `im` . All values smaller than or equal to threshold are zeroed. Other values are set to 255.

(ubim-negative-threshold `im thres`**)**

Applies a threshold on ubyte image `im` . All values smaller than or equal to threshold are set to 255. Other values are set to 0

(ubim-erode `im factor`**)**

Perform in place erosion of ubyte image `im` at distance `factor` .

(ubim-dilate `im factor`**)**

Perform in place dilation of ubyte image `im` at distance `factor` .

(ubim-mask `im mask`**)**

Sets to zero all bytes of ubyte image `im` whose corresponding pixel in `mask` is zero. This is a AND operation.

9.3.14 Morphological Operation on Images of shorts

Erosion/dilation morphological operations on images of shorts.

CAUTION with the so-called "margin" parameter. a margin of 1 is sufficient in most cases, however the margin pixels MUST be filled with 0 (background/white) for the functions to work properly. (`dilero` does it automatically).

(ubim-fillborder `shim c`**)**

fill a one-pixel border of the ubimage `shim` with value `c` . This function is useful for various morphological operations.

(shim-fillborder `shim c`**)**

fill a one-pixel border of the shimage `shim` with value `c` . This function is useful for various morphological operations.

(shim-dilation `im margin`**)**

```
((-idx2- (-short-)) im)  ;; image to be dilated:
((-int-) margin):
RETURNS:
CREATED: Y. LeCun, modified P.Haffner
COMPILABLE: Yes
DESCRIPTION:
```

(shim-erosion `im oldim mode margin`**)**

```
((-idx2- (-short-)) im)      ;; image to be eroded:
((-idx2- (-short-)) orig-im) ;; original image, only used in mode 2:
((-int-) mode):
((-int-) margin)  ;; erosion margin, typical values are 1 or 2:
RETURNS:
CREATED: Y. LeCun, modified P.Haffner
COMPILABLE: Yes
DESCRIPTION:
erosions with hexagonal tesselation
<dist> is an idx2 of shorts with dimensions identical to <im>+ border.
It will contain the distance transform on output.
<mode> shoul be
                1 ==> erosion
                2 ==> erosion and hole restoration
```

(idx-s2xors2 `m1 m2`**)**

```
((-idx2- (-short-)) m1 m2):
RETURNS: (-idx2- (-short-)) m1x2
CREATED: P.Haffner
COMPILABLE: Yes
DESCRIPTION: performs a logical XOR betweem m1 and m2.
m1x2 pixel is 0 when m1 and m2 pixels are equal, 0 otherwise.
```

(dilero `runs margin`**)**

```
(-idx2- runs):
(-int- margin):
RETURNS: an -idx2- matrix of runs
CREATED: P.Haffner
COMPILABLE: Yes
DESCRIPTION:
morphological closing operator.
successively applies dilation and erosion functions
<margin> represents the extend of the initial dilation
- 0 nothing happens
- 1 using an hexagonal tesselation, all pixel next to  a black pixel becomes black.
- 2 distance is extended to two pixels.
```

9.3.15 Miscellaneous Image Macros and Conversions

basic macros and functions to manipulate dimensions and internal formats of images

(rgbaim2rgbafim `in`**)**

convert RGBA image of ubytes into RGBA image of floats.

(rgbafim2rgbaim `in`**)**

convert RGBA image of floats into RGBA image of ubytes.

(pixel2rgbim `in`**)**

converts an integer array of pixel values into a RGB image

img-dim-C

class that holds the two dimensions of an image d0: height d1: width
(new img-dim-C `d0 d1`**)**
(equal-dim `dimA dimB`**)**

```
((-obj- (img-dim-C)) dimA dimB):
CREATED: Oct 97
DESCRIPTION:
t if dimA and dimB are equal
```

(align-dimensions `img-dim align-block`**)**

```
((-obj- (img-dim-C)) img-dim):
(-int- align-block):
CREATED: Oct 97
DESCRIPTION:
return new dimension object
- dimensions larger than img-dim
- multiple of align-block
```

9.3.16 Constants Used in Run-Length Encoded Images

Indexes to acces tables of runs

```
RUN-Y  (f) 0
RUN-X1 (f) 1
RUN-X2 (f) 2
```

```
RUN-ID (f) 3
CC-NRUN   (f) 0
CC-FRUN   (f) 1
CC-WPIX   (f) 2
CC-NPIX   (f) 3
CC-LEFT   (f) 4
CC-TOP    (f) 5)1
CC-RIGHT  (f) 6
CC-BOTTOM (f) 7
```

9.3.17 Converting RLE Image to Greyscale Pixelmap

(nt-runs2ubim runs ubim**)**

```
((-idx2- (-int-)) runs):
((-idx2- (-ubyte-)) ubim):
CREATED: P.Haffner, August 97
TEMPORARIES: ?
DESCRIPTION:
maps runs into a ubyte image
```

(static-runs2ubim runs ubim white black**)**

```
((-idx2- (-int-)) runs):
((-idx2- (-ubyte-)) ubim):
((-int-) white black):
CREATED: P.Haffner, August 97
TEMPORARIES: ?
DESCRIPTION:
```

(nt-runs2ubim-dil runs ubim**)**

```
((-idx2- (-int-)) runs):
((-idx2- (-int-)) ubim):
CREATED: P.Haffner, August 97
TEMPORARIES: ?
DESCRIPTION:
1-dilattion while applying <nt-run2ubim>
```

(static-runs2ubim-dil runs ubim**)**

```
((-idx2- (-int-)) runs):
((-idx2- (-ubyte-)) ubim):
```

```
CREATED: P.Haffner, August 97
TEMPORARIES: ?
DESCRIPTION:
```

(static-runs2ubim-subsample `runs ubim x-sub y-sub`**)**

```
((-idx2- (-int-)) runs):
((-idx2- (-ubyte-)) ubim):
((-int-) x-sub y-sub):
CREATED: P.Haffner, August 97
TEMPORARIES: ?
DESCRIPTION:
<nt-runs2ubim> with subsampling
```

See: (nt-runs2ubim `runs ubim`)

(runs2ubim `runs white black`**)**

```
((-idx2- (-int-)) runs):
((-int-) white black):
CREATED: P.Haffner, August 97
TEMPORARIES: ?
DESCRIPTION:
```

(runs2ubim-subsample `runs x-sub y-sub`**)**

```
((-idx2- (-int-)) runs):
((-int-) x-sub y-sub):
CREATED: P.Haffner, August 97
DESCRIPTION:
Subsample on the fly --> grey levels
```

9.3.18 Color Clustering and Quantization

Color Clustering

Functions to find representative colors from a precomputed color histogram. these functions just find the colors, they do not actually quantize the images. This is done in rgbaimage, and rgbafimage.

(cqu-init-proto `ppal proto`**)**

initialize prototypes for K-means. assumes that `ppal` contains the color values sorted by luminosity (as returned by rgbaim-histo32). puts the prototypes regularly spaced along a line between the darkest and lightest color. `ppal` : Nx3 matrix of floats containing a list of RGB color cells `proto` : Px3 matrix of floats, contains the initalized prototypes on output

(cqu-kmeans `ppal count assign split proto label weight variance`**)**

performs a Kmeans algorithms on the color values contained in `ppal` . INPUT: `ppal` : Nx3 matrix of floats containing a list of RGB color present in the picture `count` : N matrix of floats, number of pixels with the corresponding RGB color `assign` : N matrix of floats containing the label of the prototype to which the color cell is currently assigned. `split` : an int: only those color cells whose initial assigned label is equal to `split` will be taken into account. `proto` : Px3 matrix of floats, contains initialized RGB prototypes `label` : P matrix of ints, contains labels for the prototypes OUTPUT: `proto` : contains the updated prototypes on output `assign` : the color cells whose assigned value was `split` will be assigned the value in `label` correspinding to their closest prototype. `weight` : P matrix of float, contains the number of pixels assigned to each prototype `variance` : P matrix of float, contains the variance associated with the prototypes

Color Segmentation and Quantization on RGBA Images

a set of function for detecting the background and foreground colors in a document image and for transforming them into grey-level images suitable for recognition, segmentation, etc... essentially sets the background to 0, the foreground to 256, the rest in the middle.

(rgbaim-project-greys `rgbaim ubim polarity bsatur fsatur colors`**)**

transforms an rgb image into a grey image where the foreground is 256, the background is 0, and the greys in between are between 256 and 0 the resulting image can be used for segmentation and recognition. `rgbaim` an idx3 of ubytes containing the input RGBA image. `ubim` an idx2 of ubytes that will contain the result it should be the same size as `rgbaim polarity` determines the polarity of the image: 0 means light background and dark foreground, 1 means dark background and light forground, -1 should be passed if the polarity is unknown. In that case, the color cluster with the most numerous pixels is assigned to the background. pixel colors in `rgbaim` are projected on a straight segment between two color prototypes computed with K-means. The output pixels are given a grey value that depends on their position along that segment. The function that maps positions on the line to grey-levels is piece-wise linear saturation-type function. It is controlled by two points A and B. below A, points are assigned the value 0, above B they are assigned 256. in between they are mapped linearly. the position of point A is determined by the argument `bsatur` a value of 0 puts A on the background prototype, a value of 0.5 puts it half way between the 2 prototypes. Point B is controlled similarly: 0 puts it at the foreground prototype and 0.5 puts it halfway. `colors` is a 2x3 matrix of rgb color clusters for the background and foreground respectively.

(rgbaim-project-colors `rgbaim rgbaim2 polarity bsatur fsatur colors`**)**

project the colors of an image onto a line joining two center clusters. `rgbaim` an idx3 of ubytes containing the input RGBA image. `rgbaim2 contains the result image on output. it should be the same size as rgbaim polarity determines the polarity of the image: 0 means light background and`

dark foreground, 1 means dark background and light forground, -1 should be passed if the polarity is unknown. In that case, the color cluster with the most numerous pixels is assigned to the background. pixel colors in rgbaim are projected on a straight segment between two color prototypes computed with K-means. The output pixels are given a grey value that depends on their position along that segment. The function that maps positions on the line to grey-levels is piece-wise linear saturation-type function. It is controlled by two points A and B. below A, points are assigned the value 0, above B they are assigned 256. in between they are mapped linearly. the position of point A is determined by the argument bsatur a value of 0 puts A on the background prototype, a value of 0.5 puts it half way between the 2 prototypes. Point B is controlled similarly: 0 puts it at the foreground prototype and 0.5 puts it halfway. colors is a 2x3 matrix of rgb color clusters for the background and foreground respectively.

(rgbaim-cluster-colors rgbaim ubim polarity bsatur fsatur proto weight variance**)**

See rgbaim-project-greys for an explanation of the arguments. This function performs color clustering, using the K-means algorithm. Results in ubim .

proto : n_colors x 3 matrix = colors of the clusters weight : n_colors vector = weights of the Gaussian mixture for each cluster variance : n_colors vector = variance parameter of each cluster Gaussian distribution

See: (rgbaim-greypage rgbaim ubim polarity bsatur fsatur)

(rgbaim-quickquant rgbaim ubim ppal assign**)**

quick color quantization of the pixels in rgbaim to the colors in palette ppal . Each pixel in the output image is a short whose value is the "label" of the palette color that is closest to the input pixel. assign is a vector of ints with the same first dimension as ppal which contains the "label" for each palette color. This function is relatively fast, but is approximate (the palette color associated with an input color is not always the closest). In particular, this function makes little sense if two palette colors are in the same cell in the 32x32x32 RGB cube (which means ppal should probably have less than 2^15 colors).

low-level color quantization functions.

(rgbaim-greyquant rgbaim ubim proto polarity bsatur fsatur**)**

transforms and RGB image into a grey image with uniform foreground and background. proto is a 2x3 matrix of floats containing two color prototypes for background and foreground colors. rgbaim an idx3 of ubytes containing the input RGBA image. ubim an idx2 of ubytes that will contain the result it should be the same size as rgbaim polarity determines the polarity of the image: 0 means light background and dark foreground, 1 means dark background and light forground, -1 should be passed if the polarity is unknown. In that case, the color cluster with the most numerous pixels is assigned to the background. pixel colors in rgbaim are projected on a straight segment between two color prototypes computed with K-means. The output pixels are given a grey value that depends on their position along that segment. The function that maps positions on the line to grey-levels is piece-wise linear saturation-type function.

It is controlled by two points A and B. below A, points are assigned the value 0, above B they are assigned 256. in between they are mapped linearly. the position of point A is determined by the argument `bsatur` a value of 0 puts A on the background prototype, a value of 0.5 puts it half way between the 2 prototypes. Point B is controlled similarly: 0 puts it at the foreground prototype and 0.5 puts it halfway. `colors` is a 2x3 matrix of rgb color clusters for the background and foreground respectively.

(rgbaim-projectcolors `rgbaim rgbaim2 proto polarity bsatur fsatur`**)**

Basically does the same thing as `rgbaim-greyquant` , but builds a color image instead of a gey image.

See: (rgbaim-greyquant `rgbaim ubim proto polarity bsatur fsatur`)

Color Segmentation on RGBA float Images

a set of function for detecting the background and foreground colors in a document image and for transforming them into grey-level images suitable for recognition, segmentation, etc... essentially sets the background to 0, the foreground to 256, the rest in the middle.

(rgbafim-cluster-colors `rgbafim ubim polarity bsatur fsatur proto weight variance`**)**

See arguments of grey-page. This function performs the actual color clustering, using the K-means algorithm. Results in

`proto` : n_colors x 3 matrix = colors of the clusters `weight` : n_colors vector = weights of the Gaussian mixture for each cluster `variance` : n_colors vector = variance parameter of each cluster Gaussian distribution

See: (rgbafim-greypage `rgbafim ubim polarity bsatur fsatur`)

(rgbafim-project-greys `rgbafim ubim polarity bsatur fsatur colors`**)**

transforms an rgb image into a grey image where the foreground is 256, the background is 0, and the greys in between are between 256 and 0 the resulting image can be used for segmentation and recognition. `rgbafim` an idx3 of flts containing the input RGBA image. `ubim` an idx2 of flts that will contain the result it should be the same size as `rgbafim polarity` determines the polarity of the image: 0 means light background and dark foreground, 1 means dark background and light forground, -1 should be passed if the polarity is unknown. In that case, the color cluster with the most numerous pixels is assigned to the background. pixel colors in `rgbafim` are projected on a straight segment between two color prototypes computed with K-means. The output pixels are given a grey value that depends on their position along that segment. The function that maps positions on the line to grey-levels is piece-wise linear saturation-type function. It is controlled by two points A and B. below A, points are assigned the value 0, above B they are assigned 256. in between they are mapped linearly. the position of point A is determined by the argument `bsatur` a value of 0 puts A on the background prototype, a value of 0.5 puts it half way between the 2 prototypes. Point B is controlled similarly: 0 puts it at the foreground prototype and 0.5 puts it halfway. `colors` is a 2x3 matrix of rgb color clusters for the background and foreground respectively.

(rgbafim-project-colors `rgbafim rgbafim2 polarity bsatur fsatur colors`**)**

project the colors of an image onto a line joining two center clusters. `rgbafim` an idx3 of flts containing the input RGBA image. `rgbafim2 contains the result image on output. it should be the same size as rgbafim polarity determines the polarity of the image: 0 means light background and dark foreground, 1 means dark background and light forground, -1 should be passed if the polarity is unknown. In that case, the color cluster with the most numerous pixels is assigned to the background. pixel colors in rgbafim are projected on a straight segment between two color prototypes computed with K-means. The output pixels are given a grey value that depends on their position along that segment. The function that maps positions on the line to grey-levels is piece-wise linear saturation-type function. It is controlled by two points A and B. below A, points are assigned the value 0, above B they are assigned 256. in between they are mapped linearly. the position of point A is determined by the argument bsatur a value of 0 puts A on the background prototype, a value of 0.5 puts it half way between the 2 prototypes. Point B is controlled similarly: 0 puts it at the foreground prototype and 0.5 puts it halfway. colors is a 2x3 matrix of rgb color clusters for the background and foreground respectively.`

(rgbafim-quickquant `rgbafim ubim ppal assign`**)**

quick color quantization using result of cqu-kmeans `rgbafim` is the RGBA image to be quantized `ubim` is the output image (of floats). It must be the same size as `rgbafim` . `ppal` is the color histogram as computed by rgbafim-histo32 `assign` is the vector of color labels for each color in the histogram this is "quick" only for fairly large images.

(rgbafim-greyquant `rgbafim ubim proto polarity bsatur fsatur`**)**

transforms and RGB image into a grey image with uniform foreground and background. `proto` is a 2x3 matrix of floats containing two color prototypes for background and foreground colors. transforms an rgb image into a grey image where the foreground is 256, the background is 0, and the greys in between are between 256 and 0 the resulting image can be used for segmentation and recognition. `rgbafim` an idx3 of flts containing the input RGBA image. `ubim` an idx2 of flts that will contain the result it should be the same size as `rgbafim` `polarity` determines the polarity of the image: 0 means light background and dark foreground, 1 means dark background and light forground, -1 should be passed if the polarity is unknown. In that case, the color cluster with the most numerous pixels is assigned to the background. pixel colors in `rgbafim` are projected on a straight segment between two color prototypes computed with K-means. The output pixels are given a grey value that depends on their position along that segment. The function that maps positions on the line to grey-levels is piece-wise linear saturation-type function. It is controlled by two points A and B. below A, points are assigned the value 0, above B they are assigned 256. in between they are mapped linearly. the position of point A is determined by the argument `bsatur` a value of 0 puts A on the background

prototype, a value of 0.5 puts it half way between the 2 prototypes. Point B is controlled similarly: 0 puts it at the foreground prototype and 0.5 puts it halfway. `colors` is a 2x3 matrix of rgb color clusters for the background and foreground respectively.

(rgbafim-projectcolors `rgbafim rgbafim2 proto polarity bsatur fsatur`**)**

Basically does the same thing as `rgbafim-greyquant` , but builds a color image instead of a gey image.

See: (rgbafim-greyquant `rgbafim ubim proto polarity bsatur fsatur`)

9.3.19 Tools for Image Segmentation

Region Masks

One way to represent the segmentation of an image is by a region mask. A region mask is a matrix with the same number of rows and columns as the image. Each cell of the mask holds an integer which identifies the region the corresponding pixel belongs to. The following set of functions deal with such region masks.

(normalize-mask! `mask`**)**

Reassign region indices so they are in the range 1.. `nr` , and return the number of regions `nr` .

(remove-isolated-regions! `mask`**)**

Remove regions which have only one neighboring region. Return the new number of regions in mask.

The resulting region is normal.

(max-region-index `mask`**)**

Largest index used in `mask` .

(total-boundary-length `mask`**)**

Compute the total boundary length of this mask.

Boundaries to the surrounding region 0 is not included in the tally.

(region-bounding-boxes `mask`**)**

Compute bounding boxes for all regions and return in a table.

Each entry in the table consists of four numbers, the indices of the upper left cell and of the lower right cell in the bounding box.

(region-adjacency-graph `mask`**)**

Create region-adjacency graph from `mask` and return it.

(region-adjacency-graph* `mask`**)**

Create augmented region-adjacency graph from `mask` and return it.

Vertex `0` in the resulting RAG identifies the imaginary region surrounding the mask. That is, having region 0 as neighbor means a region is at the mask boundary.

(region-size-histogram `mask`**)**

Create a histogram of region sizes in number of cells.

The result is an int-vector mapping region index to region size.

(colorize-mask `mask`**)**

Colorize the region map `mask` with five colors and return a palette image `cmask` .

To create an RGB image from the result `cmask` , do

```
(array-take <palette> 0 <cmask>)
```

(make-seeds! `img mask nw sel`**)**

Move window of size `nw` x `nw` over the grayscale image `img` (minimum `nw` =2) and compute variance estimates. Pick the lowest-variance points as seeds. Write seeds to `mask` and return number of seeds.

The number of seeds depends on the window size: The minimum distance between seeds is `nw` . Argument `sel` is a structuring element the seed points are dilated with. When `sel` is nonempty, its size must be `nw` x `nw` .

(segment-srg! `img mask`**)**

Segment grayscale image `img` by Seeded Region Growing.

`Img` is a two-dimensional double array. `Mask` is an integer array of the same size as the `img` . All entries in `mask` are zero except for the cells chosen as seeds. Different nonzero `mask` -entries with the same value are considered belonging to the same region.

`Segment-srg!` updates the `mask` argument and returns a histogram of region sizes.

(oversegment `img` [`minreg` [`nw`]])

Oversegment grayscale image `img` and return a region mask.

Parameter `minreg` is a lower bound on the region size (in pixels). `Nw` is the window size used in the search for seed points. The larger `nw` , the fewer seed points will be selected.

9.4 Plotting Library

See: Plotting Functions.

This class library offers an alternate set of graph plotting functions besides those provided by the core interpreter (see Plotting Functions.) It contains the `Plotter` and the `PlotterCurve` classes providing functions for drawing multiple curves and modifying their appearance. Here is a sample code to plot various curves:

```
(libload "libplot/plotter")
;; create a new plotter with default size (open a window if necessary)
(setq p (new Plotter))
;; plot log function in blue from 0 to 10.
(==> p PlotFunc "log" log 0 10 0.1 (alloccolor 0 0 1))
```

```
;; show it on the screen
(==> p redisplay)

;; add a piece of a red ellipse
(==> p PlotXY "lips" sin cos 0 5 0.1 (alloccolor 1 0 0))
(==> p redisplay)

;; plot a green polygonal sinusoid
(setq x (range 0 10 0.5))
(setq y (all ((v x)) (sin v)))
(==> p plotlists "sine" x y (alloccolor 0 1 0) closed-circle)
(==> p Redisplay)

;; add a grid
(==> p SetGrid 1)
(==> p Redisplay)

;; move the plotter to 100,0 and set its size to 300,200
(==> p move 100 0)
(==> p setsize 300 200)
(cls)
(==> p redisplay)

;; plot log function in a postscript file
(let ((window (ps-window "/tmp/curve.ps")))
  (setq psp (new Plotter))
  (==> psp PlotFunc "log" log 0 10 0.1 (alloccolor 0 0 1))
  (==> psp redisplay))

;; plot log function in an editable graphic window using comdraw
(setq window (comdraw-window))
(setq cp (new Plotter))
(==> cp PlotFunc "log" log 0 10 0.1 (alloccolor 0 0 1))
(==> cp setgrid 1)
(==> cp redisplay)
```

9.4.1 Plotter

Class Plotter This is the main class to define `Plotter` objects in Lush. Plotter objects have slots to define properties of the axes such as scales, labels, legend, title, ticks and logscale. It also stores the curves, lines and texts that will be drawn on the Plotter. `rect` slot of the Plotter specifies the rectangular area of the Plotter on the current current window. Slots of the Plotter object can be modified by methods of the class. They are computed automatically or set to their default values if not speicifed by the use.

(new Plotter [[x y] w h])

The constructor `Plotter` creates a new `Plotter` object at pixel positions (x, y), with width `w` and height `h` on the current window. If the `window` object is empty, it pops up a new window. `x,y, w` and `h` are optional. If `x` and `y` are omitted, they are both set to 0. If `w` and `h` are omitted, the size of the `Plotter` is equal to the size of the current window. For example:

```
(setq myplotter (new Plotter 600 600))
```

(==> `Plotter` **move** `x y`)

move the upper left-hand corner of the plotter to screen position `x y` . No screen update is performed until the next call to redisplay.

(==> `Plotter` **setsize** `w h`)

set the width and height of the plotter object to `w` and `h` . No screen update is performed until the next call to redisplay.

(==> `Plotter` **Set** `attribute value`)

Sets the attribute of the `Plotter` object to the given value. The `attribute` argument is a string and can be "xticks", "yticks", "xscale", "yscale", "xlabel", "ylabel", "legend", "title", "grid", "xscale2", "yscale2", "xticks2", "yticks2", "xlabel2", "ylabel2". If an invalid attribute name is given error is generated. The `value` argument should match the attribute's data type. For "xticks", "yticks", "xticks2" and "yticks2" the value should be a list of numbers, for "xscale" and "yscale" it should be a list of two numbers, for "legend", "title", "xlabel", "ylabel", "xlabel2" and "ylabel2", it should be a string. Example:

```
(==> myplotter Set "xticks" (range 0 10))
(==> myplotter Set "xscale" (list 0 10))
(==> myplotter Set "xlabel "x")
```

Individual methods are also provided to set and change the attributes of the Plotter object.

(==> `Plotter` **SetRectNth** `i m n`)

Divides the current window into `m` by `n` rectangles and sets the rectangular area of the `Plotter` to the `i` th one. The first rectangle is the one on the upper left corner. When the rectangle of a `Plotter` object is set, the graphics plot is restricted to that rectangular area.

For example:

```
;;divides the current window vertically and
;;plots the first object on the left and the second object on the right.
(==> plotter1 SetRectNth 1 1 2)
(==> plotter2 SetRectNth 2 2 1)
```

(==> `Plotter` SetXScale `xmin xmax`)

Sets the `xscale` slot of the `Plotter` . `xmin` argument is the minimum axis limit and `xmax` is the maximum axis limit on x. If a scale for x is not set, LUSH selects the axis limits based on the curve points.

(==> `Plotter` SetYScale `ymin ymax`)

Sets the `yscale` slot of the `Plotter` . `ymin` argument is the minimum axis limit and `ymax` is the maximum axis limit on y. If a scale for y is not set, LUSH selects the axis limits based on the curve points.

(==> `Plotter` SetXTicks `x`)

Sets the `xticks` slot of the `Plotter` . LUSH computes the ticks marks on the x axis based on the x scale. However the tick marks on the x axis can be specified with the `SetXTicks` method. Argument `x` is a list of numbers

(==> `Plotter` SetYTicks `y`)

Sets the `yticks` slot of the `Plotter` . LUSH computes the ticks marks on the y axis based on the y scale. However the tick marks on the y axis can be specified with the `SetYTicks` method. Argument `y` is a list of numbers.

(==> `Plotter` SetXLabel `x`)

Adds axis label to the x axis. Argument `x` is a string.

(==> `Plotter` SetYLabel `y`)

Adds axis label to the y axis. Argument `y` is a string.

(==> `Plotter` SetLegend `l`)

Adds a legend text to the up right corner of the `Plotter` rectangle. Argument `l` is a string.

(==> `Plotter` SetTitle `t`)

Adds a title text to the top of the `Plotter` rectangle. Argument `t` is a string.

(==> `Plotter` SetGrid `x`)

Specifies whether to draw grid lines or not. Argument `x` is either 0 or 1. When `x` is 1 grid lines are drawn, when `x` is 0 grids are not drawn.

(==> `Plotter` SetXScale2 `xmin xmax`)

Using the `Plotter` library it is possible to draw graphs with double x axes. `SetXScale2` method sets the scale of the second x axis. `xmin` argument is the minimum axis limit and `xmax` is the maximum axis limit on the second x axis. If no scale is specified for the second x axis, it is not drawn.

(==> `Plotter` SetYScale2 `ymin ymax`)

Using the `Plotter` library it is possible to draw graphs with double y axes. `SetYScale2` method sets the scale of the second y axis. `ymin` argument is the minimum axis limit and `ymax` is the maximum axis limit on the second y axis. If no scale is specified for the second y axis, it is not drawn.

(==> `Plotter` SetXTicks2 `x`)

Using the `Plotter` library it is possible to draw graphs with double x axes. `SetXTicks2` method sets the tick marks of the second x axis. Argument `x` is a list of numbers indicating the tick marks.

(==> `Plotter` SetYTicks2 `x`)

Using the `Plotter` library it is possible to draw graphs with double y axes. `SetYTicks2` method sets the tick marks of the second y axis. Argument `y` is a list of numbers indicating the tick marks.

(==> `Plotter` SetXLabel2 `x`)

Using the `Plotter` library it is possible to draw graphs with double x axes. `SetXLabel2` method sets the label of the second x axis. Argument `x` is a string.

(==> `Plotter` SetYLabel2 `y`)

Using the `Plotter` library it is possible to draw graphs with double y axes. `SetYLabel2` method sets the label of the second y axis. Argument `y` is a string

(==> `Plotter` SetXLogScale)

Using the `Plotter` library it is possible to draw logplots. `SetXLogScale` method converts the scale of the x axis to log.

(==> `Plotter` SetYLogScale)

Using the `Plotter` library it is possible to draw logplots. `SetYLogScale` method converts the scale of the y axis to log.

(==> `Plotter` ResetXLogScale)

Using the `Plotter` library it is possible to draw logplots. `ResetYLogScale` method converts the scale of the x axis from log to regular.

(==> `Plotter` ResetXLogScale)

Using the `Plotter` library it is possible to draw logplots. `ResetYLogScale` method converts the scale of the x axis from log to regular.

(==> `Plotter` NewCurve `name`)

Adds a new `PlotterCurve` object to the `Plotter` . `name` is the name of the curve and of type string. Curves are stored as alists in the `Plotter` objects. Each `PlotterCurve` is associated with a name and with the `NewCurve` method added to the list of curves in the `Plotter` .

(==> `Plotter` ClearCurve `curvename`)

Deletes the `PlotterCurve` object named `curvename` form the `Plotter` .

(==> `Plotter` AddPoint `curvename x y`)

Adds the data point (`x, y`) to the `PlotterCurve` object named `curvename` to the `Plotter` . If a curve named `curvename` does not exist, error is returned. For example:

```
(==> myplotter AddPoint "S" 0 10)
```

(==> `Plotter` SetCurveColor `curvename c`)

Sets the color of the `PlotterCurve` object named `curvename` to color `c` . Argument `c` is a color number (as returned by `alloc-color`). If a curve named `curvename` does not exist, error is returned.

(==> `Plotter` SetCurveSymbol `curvename symbolfunc`)

Sets the marker symbol of the `PlotterCurve` object named `curvename` to symbol `symbolname` . Argument `symbolfunc` must be a two-argument function whose role is to draw a symbol at the screen coordinates passed as arguments. A set of pre-defined such functions are provided: `open-square` , `closed-square` , `open-circle` , `closed-circle` , `open-up-triangle` , `open-down-triangle` , `closed-up-triangle` , `closed-down-triangle` , `straight-cross` and `oblique-cross` .

(==> `Plotter` SetCurveSymbolSize `curvename symbolsize`)

Sets the size of the marker of the `PlotterCurve` named `curvename` to `symbolsize` . `symbolsize` is an integer normally between 1 and 5. If `symbolsize` is set to 0 no markers are drawn at data points. For example:

```
(==> myplotter SetCurveSymbolSize "s" 3)
```

(==> `Plotter` SetCurveLine `curvename line`)

Specifies whether to draw lines between markers of the `PlotterCurve` object or not. Argument line is 0 or 1. If `line` is 0 only markers at data points are drawn, if line is 1, markers are connected. By default, line is 1.

(==> `Plotter` SetCurveXAxis `curvename xaxis`)

In case of use of double x axes, associates the `PlotterCurve` object to the x axis specified by the argument `xaxis` . If `xaxis` argument is 0 the x axis at the bottom is used (default), if it is 1, x axis at the top is used to draw the curve. The scales for the second (top) axis should be set.

(==> `Plotter` SetCurveYAxis `curvename yaxis`)

In case of use of double y axes, associates the `PlotterCurve` object to the y axis specified by the argument `yaxis` . If `yaxis` argument is 0 the y axis on the left is used (default), if it is 1, x axis on the right is used to draw the curve. The scales for the second (right) axis should be set.

(==> `Plotter` PlotFunc `curvename func xmin xmax xstep` [`color`] [`symbol`])

Plot a scalar real function. Samples are computed for are all values between `xmin` and `xmax` by step of `xstep` . `color` is a color identifier (as returned by `alloccolor`), and symbol is a function of two arguments (x and y coordinates) that is called each time a data point is to be plotted. Predefined such functions include nil (no symbol is plotted), open-square, closed-square, open-circle, closed-circle, open-up-triangle, open-down-triangle, closed-up-triangle, closed-down-triangle, straight-cross and oblique-cross, or any user-defined functions Example:

```
(libload "libplot/plotter")
(setq p (new Plotter))
(==> p PlotFunc "log" log 0 10 0.1 (alloccolor 1 0 0))
(==> p redisplay)
```

(==> `Plotter` PlotXY `curvename funcx funcy tmin tmax tstep [color] [symbol]`)

Plot a parameterized curve (X(t), Y(t)). Samples are computed for are all values of t between `tmin` and `tmax` by step of `tstep` . `color` is a color identifier (as returned by `alloccolor`), and symbol is a function of two arguments (x and y coordinates) that is called each time a data point is to be plotted. Predefined such functions include nil (no symbol is plotted), open-square, closed-square, open-circle, closed-circle, open-up-triangle, open-down-triangle, closed-up-triangle, closed-down-triangle, straight-cross and oblique-cross, or any user-defined functions

Example:

```
(libload "libplot/plotter")
(setq p (new Plotter))
(==> p PlotXY "log" log 0 10 0.1 (alloccolor 1 0 0))
(==> p redisplay)
```

(==> `Plotter` PlotLists `curvename xlist ylist [color] [symbol]`)

Plot a list of points whose x and y coordinates are corresponding elements in `xlist` and `ylist` . In other words `xlist` is the list of x coordinates of the data `ylist` is the list of y coordinates of the datapoints. `curvename` is a curve identifier (generally a string). If the curve `curvename` already exists, it is replaces. `color` is a color identifier (as returned by `alloc-color`), and symbol is a function of two arguments (x and y coordinates) that is called each time a data point is to be plotted. Predefined such functions include open-square, closed-square, open-circle, closed-circle, open-up-triangle, open-down-triangle, closed-up-triangle, closed-down-triangle, straight-cross and oblique-cross. Example:

```
(==> myplotter PlotLists "log" (range 1 100) (mapcar log x))
```

(==> `Plotter` AddText `x y txtstring`)

Adds a new text to the `Plotter` object. Texts are stored as list of lists in the `Plotter` . Each element of the text list is a list of the coordinates of the corresponding text and its string. Arguments `x` and `y` are real coordinates (coordinates on the plotter)

(==> `Plotter` AddLine `x1 y1 x2 y2`)

Adds a new line to the `Plotter` object. Lines are stored as list of lists in the `Plotter` . Each element of the line list is a list of the coordinates of the line. Arguments `x1, x2, y1` and `y2` are real coordinates (coordinates on the Plotter).

(==> `Plotter` GetClick)

Returns the real coordinates of the point clicked on the `Plotter` . This method is not supported in a Comdraw window

(==> `Plotter` GetDistance)

Computes the the length of the vector drawn by dragging and dropping the mouse pointer on the `Plotter` . The distance is in real coordinates (coordinates of the plotter). This method is not supported in a Comdraw window.

(==> `Plotter` Redisplay [`location`])

This is fundamental method of a `Plotter` object. Every curve, line, text, etc is stored in the `Plotter` object and is drawn after a `Redisplay` command. To see the modifications on the window, `Redisplay` should be called. The argument `location` is optional. It is a list of x and y pixel coordinates, width and height of the `Plotter` . If specified as (list x y w h), it displays the `Plotter` in the rectangular area (x y w h).

For example:

```
;;locates and reprints the plotter in the rectangle
;;specified by (100,100,500,500) on current window
(==> myplotter Redisplay 100 100 500 500)
```

9.4.2 Low-Level Utility Functions

(compute-xscale `pobject`)

Computes the scale of the x axis for a given `Plotter` object from the data points in each curve.

(compute-yscale `pobject`)

Computes the scale of the y axis for a given `Plotter` object from the data points in each curve.

(set-axes `axes xscale yscale xticks yticks xlabel ylabel legend title grid` [`xlogscale`] `ylogscale`])

Plots the axes given the scales of the x and y axis, axis ticks, axis labels, legend and the title. Labels, legend and title can be empty strings. The `axes` argument is a list of xaxismin, yaxismin, xaxismax and yaxismax. xaxismin and yaxismin are the pixel coordinates of the origin of the xaxes, and xaxismax and yaxismax are the pixel coordinates of the top right corner of the axes. The `grid` argument is either 1 or 0, specifies whether to have grid on the plotter or not. `xlogscale`

and `ylogscale` are optional and are used if a logscale is desired on one of the axes.

For example:

```
(set-axes (list 100 600 800 100)
  (list 0 10) (list 0 10) (range 0 10) (range 0 10) "x" "y" "" "graph" 1)
```

9.5 Computational Geometry in the Plane

9.5.1 Planar Meshes

`Mesh` objects represent points in the plane and edges between pairs of points. Class `Mesh` is a light-weight class imposing no constraints on points and edges. It's main purpose is to provide visualization capabilities for other geometry classes (methods `plot` and `display`).

(new Mesh `points edges`)

Create a `Mesh` object, do not copy the argument arrays. If there are P points and E edges, then `points` is a Px2 double array containing the point's coordinates, and `edges` is an `Ex2` integer array containing pairs of indices into the point array. The adjacency matrix is initially empty.

(==> `Mesh` bounding-rect-mesh)

Return a new `Mesh` that represents your bounding rectangle. The resulting `Mesh` objects consists of four points and four edges.

(==> `Mesh` bounding-rect)

Return bounding rectangle as `(x y w h)` .

(==> `Mesh` edge-length e)

Return length of edge `e` .

(==> `Mesh` edges-length)

Return length for all edges as a vector.

(==> `Mesh` edges-vector)

Return all edges as vectors (result is a Ex2 double array).

(==> `Mesh` weight)

Sum of Euclidean edge lengths.

(==> Mesh translate x y)

Add (x y) to all point coordinates, return () .

(==> Mesh rotate phi)

Rotate points by phi in counter-clockwise direction, return () .

(==> Mesh rotate-about phi x y)

Rotate points by phi about point (x y), return () .

(==> Mesh scale s)

Rescale coordinates by factor s , return () . The scale factor s must be positive.

(==> Mesh display ...kwdargs...)

Render the mesh using Lush graphics, return the window object. Many aspects of the rendering may be controlled using optional keyword arguments:

```
-------------------------------------------------------------------
Keyword      | Meaning
-------------------------------------------------------------------
edge-color   | edge color (default is 'blue)
edge-colors  | array of color numbers (see <alloccolor>; default is nil)
             | an edge will not be displayed when its color number is -1
edge-width   | edge width (default is 1)
point-color  | point color (default is 'black'
point-colors | array of color numbers (see <alloccolor>; default is nil)
             | a point will not be displayed when its color number is -1
point-size   | point marker size (default is 1)
point-sizes  | array of point sizes
plot-points  | if true, plot points (default is t)
plot-edges   | if true, plot edges (default is t)
plot-arrows  | plot edges as arrows (default is nil)
window       | window instance to use for display (default is ())
xrange       | interval on x-axis (e.g., [0 1], default is () or from window)
yrange       | interval on y-axis (e.g., [0 1], default is () or from window)
origin       | either 'NW or 'SW (default is 'SW or from window)
scale        | factor for point coordinates (default is 1 or from window)
margin       | margin around display region in percent of extent
             | (default is 0.05)
```

When (Mesh . display) creates a new window, it stores the values for options xrange , yrange , origin , scale , and margin with it. If the window is passed to (Mesh . display) again in a subsequent call (using option

`window`), it takes the stored values as defaults for these options. Origin `NW` is useful when using array indices as coordinates.

(==> `Mesh` plot . `kwdargs`)

Render the mesh using Gnuplot, return the plotter object. `Plot` creates a 2D plot of the mesh. Many aspects of the plot may be controlled using optional keyword arguments:

```
-----------------------------------------------------------------
Keyword      | Meaning
-----------------------------------------------------------------
edge-color   | edge color (default is 'blue)
edge-width   | edge width (default is 1)
point-color  | point color (default is 'black')
point-shape  | point marker shape (default is 'circle)
point-size   | point marker size (default is 1)
plot-points  | if true, plot points (default is t)
plot-edges   | if true, plot edges (default is t)
xrange       | interval on x-axis (e.g., [0 1], default is ())
yrange       | interval on y-axis (e.g., [0 1], default is ())
margin       | margin around plotting region in percent of extent
             | (default is 0.05)
plotter      | gnuplot instance to use for plotting (default is ())
preamble     | command string that gets sent to gnuplot instance
             | prior to anything else (default is ())
```

(==> `Mesh` splot)

Render the 3d view of mesh using gnuplot, return the plotter object.

`Splot` creates a 3D plot of the mesh. In order to enable mouse actions `splot` does not activate multiplot mode. To activate multiplot mode either use the preamble option or provide a plotter object in multiplot mode (option `plotter`). Many aspects of the plot may be controlled using optional keyword arguments:

```
-----------------------------------------------------------------
Keyword      | Meaning
-----------------------------------------------------------------
edge-color   | edge color (default is 'blue)
edge-width   | edge width (default is 1)
point-color  | point color (default is 'black'
point-shape  | point marker shape (default is 'circle)
point-size   | point marker size (default is 1)
plot-points  | if true, plot points (default is t)
```

```
plot-edges   | if true, plot edges (default is t)
xrange       | interval on x-axis (e.g., [0 1], default is ())
yrange       | interval on y-axis (e.g., [0 1], default is ())
zdata        | z value component for each point (default is zeros)
zrange       | interval on z-axis (e.g., [0 1], default is ())
margin       | margin around plotting region in percent of extent
             | (default is 0.05)
plotter      | gnuplot instance to use for plotting (default is ())
preamble     | command string that gets sent to gnuplot instance
             | prior to anything else (default is ())
```

(simplify `Mesh p [method]`**)**

Create a coarser mesh with a subset of the current set of points.
See: Line simplification

(join `Mesh Mesh`**)**

Create a new mesh by joining the two argument meshes.

(take-mesh `GraphIndicator Mesh`**)**

Create a new Mesh object representing the indicated submesh.

9.5.2 Simple Polygons

A `SimplePolygon` object represents a simple polygon, that is, a closed connected region with polygonal boundary and no holes.

(new SimplePolygon `points`**)**

Create new polygon object with `points` and an edge for each pair of points in the given order. Note that some methods (e.g., `orientation` , `flip-orientation`) consider the edges to be directed.

(==> `SimplePolygon` **area)**

Area of polygon.

(==> `SimplePolygon` **orientation)**

Orientation of polygon. -1 : clockwise 1 : contour-clockwise

(==> `SimplePolygon` **flip-orientation)**

Reverse all edges return `()` .

(==> `SimplePolygon` lower-left-point)

Index of the lower left point.

(==> `SimplePolygon` upper-right-point)

Index of the upper right point.

(==> `SimplePolygon` diameter)

Diameter of polygon (estimate).

(==> `SimplePolygon` perimeter)

Perimeter of polygon.

(==> `SimplePolygon` refine `max-edge-length`)

Add points on the boundary where edges are longer than `max-edge-length` ; return `()` .

(simplify `SimplePolygon p` [`method`])

Create a coarser polygon with a subset of the current set of points.
See: Line simplification

9.5.3 Polygons

A `Polygon` object represents a polygon, that is, a closed connected region with polygonal boundary. In terms of class relationships, a `Polygon` is a `SimplePolygon` with holes, where each hole itself may be represented as a `SimplePolygon` (see `Polygon` method `holes`).

(new Polygon `points holes`)

Create a new polygon object with `points` and an edge for each pair of points in the given order, accept between points with index in holes and its successor, respectively.

(==> `Polygon` holes)

Yield all holes as `SimplePolygon` objects.

(==> `Polygon` remove-holes)

See: polygon-without-holes
Remove all holes from the polygon and return `()` .

(==> <Polygon remove-small-holes p)

Remove holes with perimeter smaller than `p` and return `()` .

(contour-polygon `ubim-image` [`max-edge-length`])

Find the contour in a binary image and return it as a `SimplePolygon` . The background is assumed to be zero and the bitmap of the contour is assumed to have no gaps. The algorithm recursively breaks edges of length greater than `max-edge-length` into two smaller edges. The default value for `max-edge-length` is 6.1.

(simplify `Polygon p` [`method`])

Create a coarser polygon with a subset of the current set of points.
See: Line simplification

(==> `SimplePolygon` **plot .** `kwdargs`)

Render the `SimplePolygon` using gnuplot, return the plotter object. Additional keywords (see method `(Mesh . plot)`):

```
-------------------------------------------------------------------
Keyword      | Meaning
-------------------------------------------------------------------
fill         | fill density, a value in [0,1] (default is 0)
fill-color   | fill color (default is 'blue)
```

(polygon-without-holes `poly`)

Return the outer contour of polygon `poly` .

9.5.4 Planar Triangulations

A `SimpleTriangulation` represents a triangulation in the plane. In terms of class dependency, a `SimpleTriangulation` is a `Mesh` plus explicit triangles information. It is a light-weight class meant to be used for data exchange and visualization purposes.

(new SimpleTriangulation `points edges triangles`)

Create a `SimpleTriangulation` object, do not copy the argument arrays. If there are P points, E edges, and T triangles, then `points` is a Px2 double array containing the point's coordinates, `edges` is an `Ex2` integer array containing pairs of indices into the point array, and `triangles` is a `Tx3` integer array containing indices into the point array.

(==> `SimpleTriangulation` triangle-angles `tr`)

Give vector of triangle angles (in the order of corresponding triangle points).

(==> `SimpleTriangulation` co-circularity `e`)

For an interior edge `e` consider the enclosing quadrilateral and sum the interior angles at the two edge-opposite points. Return this sum or raise an error if `e` is not an interior edge. If the result equals Pi, then the four points of the quadrilateral lie on a circle. If the result is greater than Pi, then flipping the edge increases the minimum angle of the two triangles (i.e., flipping the edge would bring the triangulation closer to a Delaunay triangulation).

(==> `SimpleTriangulation` find-triangle `x y`)

Find triangle that includes point `(x y)` , return -1 if there is no such triangle, return -2 when the point lies on an edge.

(==> `SimpleTriangulation` common-edge `tr1 tr2`)

Return common edge of triangles `tr1` and `tr2` , return -1 when there is no common edge.

(==> `SimpleTriangulation` dual-graph `bem`)

Create the triangulation's dual graph. Every node of the dual graph corresponds to a triangle and two nodes are connected by an edge if the corresponding triangles are adjacent.

The optional argument `bem` is a "boundary edge map". When two adjacent triangles share triangulation edge `e` , and `(bem e)` is nonzero, then the triangles will not be adjacent in the dual graph. To create a triangulation graph without explicit boundary edge map, use the empty vector `[i]` for `bem` .

(==> `SimpleTriangulation` triangles-midpoint)

Return midpoint for all triangles.

(==> `SimpleTriangulation` triangles-nbe `bem`)

Compute number of boundary edges for each triangle (0-3) with the given (ubyte) boundary edge map `bem` .

(==> `SimpleTriangulation` circumcircles)

Compute circumcircles of triangles and as Tx3 matrix.

9.5.5 Axis-based shape descriptors

An axis-based shape description consists of a shape skeleton plus some function defined on the skeleton. The implementations provided here compute approximations of Blum's Symmetric Axis Transform (SAT; often referred to s Medial Axis Transform), and Prasad's Chordal Axis Transform (CAT), which is a variant of Brady's Smoothed Local Symmetries descriptor. Axis-based shape descriptors are important for operations on shapes like smoothing, matching of shapes, and decomposing shapes into parts.

(new AxialDescriptor `points edges p-width`**)**

An `AxialDescriptor` is a `Mesh` with `double` value `p-width` assigned to every point.

(medial-axis-transform `poly` [`proto-only`]**)**

Compute the Medial Axis Transform of polygon `poly` . When `proto-only` is `t` , compute only the proto-skeleton.

(chordal-axis-transform `poly` [`proto-only`]**)**

Compute the Chordal Axis Transform of polygon `poly` . When `proto-only` is `t` , compute only the proto-skeleton.

L. Prasad, R. L. Rao: "A Geometric Transform for Shape Feature Extraction", Proceedings of SPIE's 45th Annual Meeting, 2000, San Diego, CA, Vol. 4117, pp 222-233, 2000.

(delaunay-axis-transform `poly` [`proto-only`]**)**

Compute the Delaunay Axis Transform of polygon `poly` . When `proto-only` is `t` , compute only the proto-skeleton.

9.5.6 (delaunay `arg`)

Compute Delaunay or Constrained Delauny triangulation. If `arg` is an Nx2 array of point coordinates, compute the Delaunay triangulation of the point set and return a `SimpleTriangulation` object. If `arg` is a `Mesh` object, compute the constrained Delaunay triangulation of the points and edges and return a `SimpleTriangulation` object.

Example:

```
(libload "geometry/demos") ;; load a-points
(==> (delaunay a-points) display)
```

`Delauny` uses J.R. Shewchuk's triangle library. See [http://www.cs.cmu.edu/ quake/triangle.html]for more information.

9.5.7 (convex-hull points)

Compute convex hull of point set and return as SimplePolygon object.
Example:

```
(libload "geometry/demos") ; load a-points
(==> (convex-hull a-points) display)
```

Convex-hull uses J.R. Shewchuk's triangle library. See [http://www.cs.cmu.edu/ quake/triangle.html]for more information.

9.5.8 Line simplification

Function simplify takes a Mesh or a Polygon and yields an object of the same kind approximating the input object with a subset of the original points. This operation is also known as "line generalization" in cartography.

Simplify offers different approximation methods, and the meaning of the numerical parameter depends on the method. By default Lowe's method [2] is used.

```
-----------------------------------------------------------------
Key           | Method
-----------------------------------------------------------------
dp            | Douglas-Peucker algorithm [1]. The parameter is an
              | upper bound on the one-sided Hausdorff distance
              | between the approximation and the original object.
              |
lowe          | Lowe's scale-invariant algorithm [2]. The parameter
              | is the minimum distance between two distinguishable
              | points.
```

References:

[1] D.H. Douglas, T.K. Peucker: "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature," The Canadian Cartographer, 10, pp. 111-122, 1973.

[2] D.G. Lowe: "Three-dimensional object recognition from single two-dimensional images," Artificial Intelligence, 31(3), pp. 355-395, 1987.

(new PolylineSimplifier points)

9.6 Shell Commands

This set of functions is designed to facilitate the writing of "shell scripts" in Lush. It includes functions that are more or less equivalent to common shell commands such as ls, cd, cp, mv, rm etc. Those commands manipulate and return lists of strings. Other functions are provided to manipulate those lists of strings.

9.6.1 String List Utilities

(glob `regex l`**)**

Return a list of the elements of `l` that match the regular expression `regex` . `l` must be a list of strings, and `regex` a string containing a valid Lush regular expression.

(split-words `s`**)**

split a string of words (space-separated strings) into a list or those words.

(merge-words `l` [`c`]**)**

turns a list of strings into a single string composed of the concatenation of the elements of `l` interspesed with spaces. if a string is provided in `c` , it is interspesed instead of spaces.

(read-lines `f`**)**

See: lines
read file `f` and return a list of strings, each of which is a line of the file.

(write-lines `l`**)**

write each string in list `l` as a line.

#P`expr`

equivalent to (write-lines `expr`)

9.6.2 File Information

The functions in this section provide a simple interface to the do-it-all function `fileinfo` . The all return NIL if the file passed as argument does not exist or is unaccessible. Some functions in this section return dates as strings. Those date strings can be compared with the usual comparison operators. [(fileinfo `filename`)]

(file-type `f`**)**

return the type of file `f` as an atom equal to 'reg for a regular file, 'dir for a directory, 'chr for character device, etc.

(file-size `f`**)**

return the size of file `f` in bytes

(file-mode `f`)

return the permissions of file `f` as an integer.

(file-uid `f`)

return the userid of the owner of file `f`

(file-gid `f`)

return the groupid of file `f`

(file-atime `f`)

return the last access time of file `f` as a string of the form "YYYY MM DD hh mm ss". The string can be turned into a list with split-words.

(file-mtime `f`)

return the last modification time of file `f` as a string of the form "YYYY MM DD hh mm ss". The string can be turned into a list with split-words.

(file-ctime `f`)

return the creation time of file `f` as a string of the form "YYYY MM DD hh mm ss". The string can be turned into a list with split-words.

(file-newer? `f1 f2`)

returns t if the modification time of file `f1` is later than that of file `f2` , or if `f2` does not exist. Here is how this function can be use for a "make"-like function:

```
(when (file-newer? src dst) (produce-dst-from-src src dst))
```

9.6.3 Filename Manipulations

a number of functions to manipulate file names, and paths are provided as part of the core interpreter, including `basename, dirname, concat-fname, relative-fname, tmpname` . The present set of function adds to that set. The present functions are not as portable as the the ones in the core interpreter, as they assume Unix-style or URL-style path constructs (slash-separated directories) [(basename `path` [`suffix`])][(dirname `path`)][(concat-fname [`filename1`] `filename2`)]

(filename-get-suffixes `f`)

returns all the suffix(es) of a file name, that is all the characters after the last slash and after (and not including) the the last dot. Return the empty string if no suffix is found. A trailing dot is not considered a suffix.

```
? (filename-get-suffixes "dirname/basename.suffix")
= ".suffix"


? (filename-get-suffixes "dirname/basename.suffix1.suffix2")
= ".suffix1.suffix2"


? (filename-get-suffixes "dirname.notasuffix/basename.suffix1.suffix2")
= ".suffix1.suffix2"


? (filename-get-suffixes "dirname.notasuffix/basename")
= ""


? (filename-get-suffixes "dirname.notasuffix/basename.")
= ""
```

(filename-chop-suffixes `f`)

remove all the suffix(es) from a file name, that is all the characters after the last slash and starting at the first dot. A trailing dot is not considered a suffix.

```
? (filename-chop-suffixes "dirname/basename.suffix")
= "dirname/basename"


? (filename-chop-suffixes "dirname/basename.suffix1.suffix2")
= "dirname/basename"


? (filename-chop-suffixes "dirname.notasuffix/basename.suffix1.suffix2")
= "dirname.notasuffix/basename"


? (filename-chop-suffixes "dirname.notasuffix/basename")
= "dirname.notasuffix/basename"


? (filename-chop-suffixes "dirname.notasuffix/basename.")
= "dirname.notasuffix/basename."
```

(filename-get-suffix `f`)

returns the last suffix of a file name, that is all the characters after the last slash and after (and not including) the last dot. Return the empty string if no suffix is found. A trailing dot is not considered a suffix.

```
? (filename-get-suffix "dirname/basename.suffix")
= ".suffix"


? (filename-get-suffix "dirname/basename.suffix1.suffix2")
= ".suffix2"


? (filename-get-suffix "dirname.notasuffix/basename.suffix1.suffix2")
= ".suffix2"


? (filename-get-suffix "dirname.notasuffix/basename")
= ""


? (filename-get-suffix "dirname.notasuffix/basename.")
= ""
```

(filename-chop-suffix `f`)

remove the last suffix from a file name, that is all the characters after (and including) the last dot. A trailing dot is not considered a suffix.

```
? (filename-chop-suffix "dirname/basename.suffix")
= "dirname/basename"


? (filename-chop-suffix "dirname/basename.suffix1.suffix2")
= "dirname/basename.suffix1"


? (filename-chop-suffix "dirname.notasuffix/basename.suffix1.suffix2")
= "dirname.notasuffix/basename.suffix1"


? (filename-chop-suffix "dirname.notasuffix/basename")
= "dirname.notasuffix/basename"


? (filename-chop-suffix "dirname.notasuffix/basename.")
= "dirname.notasuffix/basename."
```

9.6.4 Variables

home

user home directory

dirstack

directory stack manipulated by pushd and popd

tmpdir

directory used by the function sh to store temporary files.

9.6.5 Directories

(pwd)

return current directory (equivalent to (chdir)).

(cd [d]**)**

change current directory to `d` , or to user home if `p` is not present.

(pushd d**)**

Temporarily change current directory to `d` . Returning to the previous directory can be done with popd.

(popd)

return to the current directory before the last pushd.

9.6.6 Shell Commands

This set of functions is designed to facilitate the writing of "shell scripts" in Lush. It includes functions that are more or less equivalent to common shell commands such as ls, cd, cp, mv, rm etc. Those commands manipulate and return lists of strings. Other functions are provided to manipulate those lists of strings.

(rm f**)**

remove file `f` (be careful).

(cp from to**)**

Copy file `from` to file `to` and return `to` .

(mv `from to`**)**

Move file from `from` to `to` and return `to` .

(ls `d1 d2` **...** `dn`**)**

return a list of all the files in `d1` (in case-insensitive lexicographic order), followed by all the files in `d2` , etc... Invisible files (".", "..", ".xxx", etc) are not included.

(ls-a `d1 d2` **...** `dn`**)**

return a list of all the files in `d1` (in case-insensitive lexicographic order), followed by all the files in `d2` , etc... Invisible files (".", "..", ".xxx", etc) are included.

(sh `cmd` [`l`]**)**

run shell command `cmd` and return the standard output as a list of strings (one string per line). The optional argument `l` is a list of strings that will be written to a temporary file (one line per string) and fed to the standard input of command `cmd`

(sh-find `dir pattern`**)**

calls the standard unix command find to find all the files whose name fit the pattern `pattern` (this is a unix-style shell regex, not a Lush regex). Example:

```
(sh-find "." "*.lsh")
```

9.7 Stopwatch: Timer with Microsecond Accuracy

stopwatch is a simple class to measure time with microsecond accuracy. This does not generate events (see the timer/event section for that), but merely provides a way to tell time. This class uses the gettimeofday system call and can be used in compiled code.

9.7.1 (new stopwatch)

create a new stopwatch

9.7.2 (==> `stopwatch` get)

get the time in seconds since the last call to reset, or since the creation of the stopwatch. The time is returned in seconds (with one microsecond accuracy) as a double precision floating point number.

9.7.3 (==> stopwatch reset)

resets the reference time of the stopwatch relative to which subsequent times will be measured.

9.8 Profiling for Lush

Gathering performance data for your Lush functions

Profiling a function is as easy as wrapping your definition with `profile` .

Example:

```
(profile
   (de test-func () (sleep 2)))
(test-func)
(profile-stats-all)
```

You can globally disable profiling once you're finished debugging by setting `*disable-profiling*` to `t` before you include `"profile.lsh"` anywhere in your code.

9.8.1 (profile-stats fn-name)

Print out profiling statistics for `fn-name` .

9.8.2 (profile-clear-stats fn-name)

Reset profiling statistics for `fn-name` .

9.8.3 (profile-stats-all)

Print out statistics for all profiled functions.

9.8.4 (profile-clear-stats-all)

Clear all profiling statistics.

9.8.5 (profile fn-def)

Wrap the function definition `fn-def` to include profiling code. `fn-def` can be a `de` or a `defmethod` form.

Profiling works for both compiled and interpreted code, so feel free to `dhc-make` your `fn-def`

9.9 Tensor/Matrix/Vector/Scalar Libraries

9.9.1 Generic IDX Macros and Functions

Resizing Macros and Functions

A set of macros and functions to resize vectors, matrices and tensors.

(midx-m1resize `m n0`**)**

a macro redims idx1 `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. In interpreted mode, the function idx1-resize can be called.

See: (idx1-resize `m n0`)

(midx-m2resize `m n0 n1`**)**

a macro that redims idx2 `m` in-place

(midx-m3resize `m n0 n1 n2`**)**

a macro that redims idx3 `m` in-place

(midx-m4resize `m n0 n1 n2 n3`**)**

a macro that redims idx4 `m` in-place

(midx-m5resize `m n0 n1 n2 n3 n4`**)**

a macro that redims idx5 `m` in-place

(idx-f1resize `m n0`**)**

redims idx1 of float `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-f2resize `m n0 n1`**)**

redims idx2 of float `m` in-place

(idx-f3resize `m n0 n1 n2`**)**

redims idx3 of float `m` in-place

(idx-f4resize `m n0 n1 n2 n3`**))**

redims idx4 of float `m` in-place

(idx-i1resize `m n0`**)**

redims idx1 of int `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-i2resize `m n0 n1`**)**

redims idx2 of int `m` in-place

(idx-i3resize `m n0 n1 n2`**)**

redims idx3 of int `m` in-place

(idx-i4resize `m n0 n1 n2 n3`**))**

redims idx4 of int `m` in-place

(idx-s1resize `m n0`**)**

redims idx1 of short `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-s2resize `m n0 n1`**)**

redims idx2 of short `m` in-place

(idx-s3resize `m n0 n1 n2`**)**

redims idx3 of short `m` in-place

(idx-s4resize `m n0 n1 n2 n3`**))**

redims idx4 of short `m` in-place

(idx-b1resize `m n0`**)**

redims idx1 of byte `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-b2resize `m n0 n1`**)**

redims idx2 of byte `m` in-place

(idx-b3resize `m n0 n1 n2`**)**

redims idx3 of byte `m` in-place

(idx-b4resize `m n0 n1 n2 n3`**))**

redims idx4 of byte `m` in-place

(idx-u1resize `m n0`**)**

redims idx1 of ubyte `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-u2resize `m n0 n1`**)**

redims idx2 of ubyte `m` in-place

(idx-u3resize `m n0 n1 n2`**)**

redims idx3 of ubyte `m` in-place

(idx-u4resize `m n0 n1 n2 n3`**))**

redims idx4 of ubyte `m` in-place

(idx-g1resize `m n0`**)**

redims idx1 of gptr `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-g2resize `m n0 n1`**)**

redims idx2 of gptr `m` in-place

(idx-g3resize `m n0 n1 n2`**)**

redims idx3 of gptr `m` in-place

(idx-g4resize `m n0 n1 n2 n3`**))**

redims idx4 of gptr `m` in-place

(idx-d1resize `m n0`**)**

redims idx1 of double `m` in-place with the new size `n0` . Unlike redim, it does not create a temporary variable. Unlike idx-set-dim, this macro increases the size of the storage if necessary. very efficient. exists also in macro form.

See: (midx-m1resize `m n0`)

(idx-d2resize `m n0 n1`**)**

redims idx2 of double `m` in-place

(idx-d3resize `m n0 n1 n2`**)**

redims idx3 of double `m` in-place

(idx-d4resize `m n0 n1 n2 n3`**))**

redims idx4 of double `m` in-place

Macros for common operations

(midx-m2oversample `input nlin ncol output`**)**

macro for 2D oversampling

Obsolete Iterators for Inline C code

These function are kept for backward compatibility, but a largely superseded by cidx-bloop described in the Core section.

(cinline-idx1loop `m type c j p1` [p2**...**])

man, that's a tough one to explain. don't try this at home anyway. this thing is like a loop for cinline. It executes the cinline instructions in `p1` ... for each element of matrix `m` (must be an idx1). `c` , and `j` must be strings containing valid C variable identifiers, and `type` is a string that must contain a valid C type (like "float"). during the loop, C variable `c` will point to the current element of `m` , and C variable `j` will contain the index of that element. So a macro that fills the elements of a vector with its index plus a constant n can be written:

```
(cinline-idx1loop mat "flt" "p" "i" (cinline "*p = i+\\%s;" n))
```

(cinline-idxloop2 `m1 type1 m2 type2 c1 c2 j p1` [p2**...**])

same as cinline-idx1loop, but loops simultaneously on two vectors

(cinline-idxloop3 `m1 type1 m2 type2 m3 type3 c1 c2 c3 j p1` [p2**...**])

same as cinline-idx1loop, but loops simultaneously on three vectors

(cinline-idx2loop `m type c i j` [l1**...**])

same as cinline-idx1loop but loops over all elements of an idx2. `i` and `j` are strings containing the C name of the row and column indices.

(cinline-idx2loop2 `m1 type1 m2 type2 c1 c2 i j` [l**...**])

same as cinline-idx1loop2 but loops over all elements of an idx2. `i` and `j` are strings containing the C name of the row and column indices.

(cinline-idx2loop3 `m1 type1 m2 type2 m3 type3 c1 c2 c3 i j` [l**...**])

same as cinline-idx1loop3 but loops over all elements of an idx2. `i` and `j` are strings containing the C name of the row and column indices.

9.9.2 IDX of Single Precision Floating Point Numbers

Basic operations on vectors, matrices, and tensors of floats

(idx-f2timesf2 `A B C`**)**

2d x 2d matrix multiply: C = A * B

(idx-f1reverse `vector`**)**

Reverses in-place the order of the elements of a `vector` . not very efficient, but I didn't have time to re-write a faster one.

(idx-f1fill `m f`**)**

fill idx1 `m` with value `f`

(idx-f2fill `m f`**)**

fill idx2 `m` with value `f`

(idx-f3fill `m f`**)**

fill idx3 `m` with value `f`

(idx-f1dotc `m f q`**)**

multiply elements of `m` by float `f` , put result in `q`

(idx-f2dotc `m f q`**)**

multiply elements of `m` by float `f` , put result in `q`

(idx-f3dotc `m f q`**)**

multiply elements of `m` by float `f` , put result in `q`

(idx-f1dotcacc `m f q`**)**

multiply elements of `m` by float `f` , accumulate result in `q`

(idx-f2dotcacc `m f q`**)**

multiply elements of `m` by float `f` , accumulate result in `q`

(idx-f3dotcacc `m f q`**)**

multiply elements of `m` by float `f` , accumulate result in `q`

(idx-f1addc `m f q`**)**

add elements of `m` to float `f` , put result in `q`

(idx-f2addc `m f q`**)**

add elements of `m` by float `f` , put result in `q`

(idx-f3addc `m f q`**)**

add elements of `m` by float `f` , put result in `q`

(idx-f1lincomb `x cx y cy z`**)**

performs a linear combination of vectors `x` and `y` using coefficients `cx` and `cy` . Put result in `z` .

(idx-f1tanh `x1 y1`**)**

Computes the regular tanh of `x1` element by element, result in `y1` .

(idx-f1inv `x1 y1`**)**

puts inverse of each element of `x1` into corresponding elements in `y1`

(idx-f1sign `x y`**)**

Elements of `y` are set to the sgn function of elements of `x` .

(idx-f1clip `x y`**)**

(idx-f1sup `m`**)**

returns largest element in `m`

(idx-f1max `m` [`r`]**)**

if `r` is present, put largest element of idx1 `m` into idx0 `r` otherwise returns largest element.

(idx-f2max `m` [`r`]**)**

if `r` is present, put largest element of idx1 `m` into idx0 `r` otherwise returns largest element.

(idx-f1inf `m`**)**

returns smallest element in `m`

(idx-f1min `m` [`r`])

if `r` is present, put smallest element of idx1 `m` into idx0 `r` otherwise returns smallest element.

(idx-f2min `m` [`r`])

if `r` is present, put smallest element of idx1 `m` into idx0 `r` otherwise returns smallest element.

(idx-f1indexmax `m`)

returns index (int) of largest element in `m`

(idx-f1indexmin `m`)

returns index (int) of smallest element in `m`

(index-of-max `m`)

return index of largest element in idx1 `m`

(idx-f1fill-with-index `m`)

fill the elements of idx1 `m` with their index.

(idx-f1avg `vector`)

Returns the average of the numbers in the input `vector` .

(idx-f1prod `vec`)

returns the product of the elements of the input vector `vec` ,

(idx-f1logsum `m`)

computes a "log-add" of the elements in `m` , i.e. -log(1/n Sum_i (exp(- m_i))) where n is the dimension of `m` . This is the wrong logsum advocated by Yann. Use idx-f1logadd instead.

(idx-f2logsum `m`)

same as idx-f1logsum, but in 2 dimensions -log(1/n Sum_ij (exp(- m_ij))) where n is the number of elements in `m` This is the wrong logsum advocated by Yann. Use idx-f2logadd instead.

(idx-f1logadd `m`)

computes a "log-add" of the elements in `m` , i.e. -log(Sum_i (exp(- m_i)))

(idx-f2logadd `m`)

same as idx-f1logadd, but in 2 dimensions -log(Sum_ij (exp(- m_ij)))

(idx-f1logaddb `m beta`)

computes a "log-add" with inverse temperature `b` of the elements in `m` , i.e. -1/beta log (Sum_i (exp(- beta*m_i)))

(idx-f2logaddb `m beta`)

same as idx-f1logaddb, but in 2 dimensions -1/beta log(Sum_ij (exp(- beta*m_ij)))

(idx-f1logdotf1 `m p`)

computes a "log-add" of term-by-termt products of elements in `m` and `p` , i.e. -log(p_i Sum_i (exp(- m_i))) `p` should be a normalized probability vector for this to be meaningful

(idx-f2logdotf2 `m`)

same as idx-f1logdotf1, but in 2 dimensions

(idx-f1blogdotf1 `m p out`)

This is the "bprop" corresponding to idx-f1logdotf1 `m` is the input (preferably positive costs), `p` is the prior vector, and on output `out` is the vector of partial derivatives of (idx1-f1logdotf1 `m p`) with respect to each element of `m` .

(idx-f1subextf1 `x y z`)

Computes the external subtraction z[i, j] = x[i] - y[j]

(idx-f1subf0 `x1 y0 z1`)

Substracts (`y0`) from elements of vector `x1` . Results in vector `z1` .

(idx-f1mulacc `x1 y1 z1`)

z1[i] += x1[i] * y1[i]. not efficient. Could be improved a lot.

(idx-f1entropy `m c`)

computes the entropy of a vector (- sum Pi log Pi) this does a normalization and adds a constant `c` to all the elements (so it does not crash of they are 0)

(idx-f1subsample-fast `in nlin`**)**

subsamples vector (considered as a signal) by integer factor `nlin` the returned output is a vector whose size is int(N/ `nlin`) where N is the size of `in` . there is a idx-f1subsample defined in libconvol.sn, it is a little slower and takes 3 parameters.

(idx-f4dotf3 `x y z`**)**

z[i] = sum_jkl x[i, j, k, l]*y[j, k, l]

(idx-f3extf1 `x y z`**)**

External (or cross-) product of 3d `x` with 1d `y` to yield 4d `z` . z[i, j, k, l] = x[i, j, k]*y[l]

(idx-f1extf3 `x y z`**)**

External (or cross-) product of 1d `x` with 3d `y` to yield 4d `z` . z[i, j, k, l] = x[i]*y[j, k, l]

(idx-f3dotf3 `a b c`**)**

`c` is an idx0 which will contain the sum of all the products of elements from `a` and `b` . (this redundant with idx-dot).

(idx-f3dotf2 `x y z`**)**

z[i] = sum_jk x[i, j, k]*y[j, k]

(idx-f4dotf1 `x y z`**)**

z[j, k, l] = sum_i x[i, j, k, l]*y[i]

(idx-f4dotf1acc `x y z`**)**

z[j, k, l] += sum_i x[i, j, k, l]*y[i]

(idx-f1extf2 `x y z`**)**

External (or cross-) product of 1d `x` with 2d `y` to yield 3d `z` . z[i, j, k] = x[i]*y[j, k]

(idx-f3dotf1 `x y z`**)**

z[j, k] = sum_i x[i, j, k]*y[i]

(idx-f3dotf1acc `x y z`**)**

z[j, k] += sum_i x[i, j, k]*y[i]

(make-f2-place-target `n`**)**

return an `n` x `n` matrix with 1s on the diagonal and -1s everywhere else.

(idx-f1range `v`**)**

Return min and max of float-vector `v` .

9.9.3 IDX of Double Precision Floating Point Numbers

Basic operations on vectors, matrices, and tensors of doubles

(idx-d2timesd2 `A B C`**)**

2d x 2d matrix multiply: C = A * B

(idx-d1reverse `vector`**)**

Reverses in-place the order of the elements of a `vector` . not very efficient, but I didn't have time to re-write a faster one for SN3.1

(idx-d1fill `m f`**)**

fill elements of `m` with double `f` .

(idx-d2fill `m f`**)**

fill idx2 `m` with value `f`

(idx-d3fill `m f`**)**

fill idx3 `m` with value `f`

(idx-d1dotc `m f q`**)**

multiply elements of `m` by double `f` , put result in `q`

(idx-d2dotc `m f q`**)**

multiply elements of `m` by double `f` , put result in `q`

(idx-d3dotc `m f q`**)**

multiply elements of `m` by double `f` , put result in `q`

(idx-d1dotcacc `m f q`**)**

multiply elements of `m` by double `f` , accumulate result in `q`

(idx-d2dotcacc `m f q`**)**

multiply elements of `m` by double `f` , accumulate result in `q`

(idx-d3dotcacc `m f q`**)**

multiply elements of `m` by double `f` , accumulate result in `q`

(idx-d1addc `m f q`**)**

add elements of `m` to double `f` , put result in `q`

(idx-d2addc `m f q`**)**

add elements of `m` by double `f` , put result in `q`

(idx-d3addc `m f q`**)**

add elements of `m` by double `f` , put result in `q`

(idx-d1lincomb `x cx y cy z`**)**

performs a linear combination of vectors `x` and `y` using coefficients `cx` and `cy` . Put result in `z` .

(idx-d1tanh `x1 y1`**)**

Computes the regular tanh of `x1` element by element, result in `y1` .

(idx-d1inv `x1 y1`**)**

puts inverse of each element of `x1` into corresponding elements in `y1`

(idx-d1sign `x y`**)**

Elements of `y` are set to the sgn function of elements of `x` .

(idx-d1clip `x y`**)**

(idx-d1sup `m`**)**

returns largest element in `m`

(idx-d1max `m` [`r`])

if `r` is present, put largest element of idx1 `m` into idx0 `r` otherwise returns largest element.

(idx-d2max `m` [`r`])

if `r` is present, put largest element of idx1 `m` into idx0 `r` otherwise returns largest element.

(idx-d1inf `m`)

returns smallest element in `m`

(idx-d1min `m` [`r`])

if `r` is present, put smallest element of idx1 `m` into idx0 `r` otherwise returns smallest element.

(idx-d1range `v`)

Return min and max of double-vector `v` .

(idx-d2min `m` [`r`])

if `r` is present, put smallest element of idx1 `m` into idx0 `r` otherwise returns smallest element.

(idx-d1indexmax `m`)

returns index (int) of largest element in `m`

(idx-d1indexmin `m`)

returns index (int) of smallest element in `m`

(index-of-max `m`)

return index of largest element in idx1 `m`

(idx-d1fill-with-index `m`)

fill the elements of idx1 `m` with their index.

(idx-d1avg `vector`)

Returns the average of the numbers in the input `vector` .

(idx-d1prod `vec`**)**

returns the product of the elements of the input vector `vec` ,

(idx-d1logsum `m`**)**

computes a "log-add" of the elements in `m` , i.e. -log(1/n Sum_i (exp(- m_i))) where n is the dimension of `m` . This is the wrong logsum advocated by Yann. Use idx-d1logadd instead.

(idx-d2logsum `m`**)**

same as idx-d1logsum, but in 2 dimensions -log(1/n Sum_ij (exp(- m_ij))) where n is the number of elements in `m` This is the wrong logsum advocated by Yann. Use idx-d2logadd instead.

(idx-d1logadd `m`**)**

computes a "log-add" of the elements in `m` , i.e. -log(Sum_i (exp(- m_i)))

(idx-d2logadd `m`**)**

same as idx-d1logadd, but in 2 dimensions -log(Sum_ij (exp(- m_ij)))

(idx-d1logaddb `m beta`**)**

computes a "log-add" with inverse temperature `b` of the elements in `m` , i.e. -1/beta log (Sum_i (exp(- beta*m_i)))

(idx-d2logaddb `m beta`**)**

same as idx-f1logaddb, but in 2 dimensions -1/beta log(Sum_ij (exp(- beta*m_ij)))

(idx-d1logdotd1 `m p`**)**

computes a "log-add" of term-by-termt products of elements in `m` and `p` , i.e. -log(p_i Sum_i (exp(- m_i))) `p` should be a normalized probability vector for this to be meaningful

(idx-d2logdotd2 `m`**)**

same as idx-d1logdotd1, but in 2 dimensions

(idx-d1blogdotd1 `m p out`**)**

This is the "bprop" corresponding to idx-d1logdotd1 `m` is the input (preferably positive costs), `p` is the prior vector, and on output `out` is the vector of partial derivatives of (idx1-d1logdotd1 `m p`) with respect to each element of `m` .

(idx-d1subextd1 `x y z`)

Computes the external subtraction z[i, j] = x[i] - y[j]

(idx-d1subd0 `x1 y0 z1`)

Substracts (`y0`) from elements of vector `x1` . Results in vector `z1` .

(idx-d1mulacc `x1 y1 z1`)

z1[i] += x1[i] * y1[i]. not efficient. Could be improved a lot.

(idx-d1entropy `m c`)

computes the entropy of a vector (- sum Pi log Pi) this does a normalization and adds a constant `c` to all the elements (so it does not crash of they are 0)

(idx-d1subsample-fast `in nlin`)

subsamples vector (considered as a signal) by integer factor `nlin` the returned output is a vector whose size is int(N/ `nlin`) where N is the size of `in` . there is a idx-d1subsample defined in libconvol.sn, it is a little slower and takes 3 parameters.

(idx-f4dotd3 `x y z`)

z[i] = sum_jkl x[i, j, k, l]*y[j, k, l]

(idx-d3extd1 `x y z`)

External (or cross-) product of 3d `x` with 1d `y` to yield 4d `z` . z[i, j, k, l] = x[i, j, k]*y[l]

(idx-d1extd3 `x y z`)

External (or cross-) product of 1d `x` with 3d `y` to yield 4d `z` . z[i, j, k, l] = x[i]*y[j, k, l]

(idx-d3dotd3 `a b c`)

`c` is an idx0 which will contain the sum of all the products of elements from `a` and `b` .

(idx-d3dotd2 `x y z`)

z[i] = sum_jk x[i, j, k]*y[j, k]

(idx-d4dotd1 `x y z`)

z[j, k, l] = sum_i x[i, j, k, l]*y[i]

(idx-d4dotd1acc `x y z`)

z[j, k, l] += sum_i x[i, j, k, l]*y[i]

(idx-d1extd2 `x y z`)

External (or cross-) product of 1d `x` with 2d `y` to yield 3d `z` . z[i, j, k] = x[i]*y[j, k]

(idx-d3dotd1 `x y z`)

z[j, k] = sum_i x[i, j, k]*y[i]

(idx-d3dotd1acc `x y z`)

z[j, k] += sum_i x[i, j, k]*y[i]

(make-d2-place-target `n`)

return an `n` x `n` matrix with 1s on the diagonal and -1s everywhere else.

(idx-d1squextd1 `m1 m2 m3`)

square outer product of `m1` and `m2` . M3ij = M1i * M2jˆ2

(idx-d2squextd2 `m1 m2 m3`)

square outer product of `m1` and `m2` . M3ijkl = M1ij * M2klˆ2

(idx-d1squextd1acc `m1 m2 m3`)

square outer product of `m1` and `m2` . M3ij += M1i * M2jˆ2

(idx-d2squextd2acc `m1 m2 m3`)

square outer product of `m1` and `m2` . M3ijkl += M1ij * M2klˆ2

(idx-d2squdotd1 `m1 m2 m3`)

multiply vector `m2` by matrix `m1` using square of `m1` elements M3i = sum_j M1ijˆ2 * M2j

(idx-d4squdotd2 `m1 m2 m3`)

multiply matrix `m2` by tensor `m1` using square of `m1` elements M3ij = sum_kl M1ijklˆ2 * M2kl

(idx-d2squdotd1acc `m1 m2 m3`**)**

multiply vector `m2` by matrix `m1` using square of `m1` elements M3i += sum_j M1ij^2 * M2j

(idx-d4squdotd2acc `m1 m2 m3`**)**

multiply matrix `m2` by tensor `m1` using square of `m1` elements M3ij += sum_kl M1ijkl^2 * M2kl

(idx-d1squdotd1acc `m1 m2 m3`**)**

dot product between `m1` and `m2` , except square of terms of `m1` are used: M3 += sum_i M1i^2 * M2i

(idx-d2squdotd2acc `m1 m2 m3`**)**

dot product between matrices `m1` and `m2` , except square of terms of `m1` are used: M3 += sum_ij M1ij^2 * M2ij

(idx-d1cumsum `v1 v2`**)**

Write cumulative sum of vector `v1` to `v2` .

(idx-d1cumsum! `v`**)**

Replace contents of vector `v` by its cumulative sum.

9.9.4 IDX of Integers

Basic operations on vectors, matrices, and tensors of integers.

(idx-i1addc `m1 c m2`**)**

Add constant to idx1 of int.

(idx-i2addc `m1 c m2`**)**

Add constant to idx2 of int.

(idx-i1concat `int-vec1 int-vec2` **...)**

Concatenate the `int-vectors` in argument list and return new big vector.

(random-permute `perm-vector`**)**

randomly permute the elements of the given `permutation-vector` (an integer vector)

(multinomial-sample `probabilities`**)**

Given a vector of `probabilities` of length N summing to one for events 0 to N-1, return an integer i associated to these events with probability `probabilities` [i]. The algorithm samples a uniform number between 0 and 1 and loops through the `probabilities` vector, so its running time is O(N).

N.B.: if the sum S of the elements is `1, then the value N-1 will be sampled more frequently with extra probability 1 - S. if the sum S of the elements is` 1, then only the first M elements will be sampled, with M being such that it is the largest integer such that the sum of the `probabilities` from 0 to M-1 is above 1.

(fast-multinomial-sample `cumulative-probabilities`**)**

Given a vector of `cumulative-probabilities` of length N, monotonically increasing between 0 and 1, return an integer between 0 and N-1 sampled from the corresponding distribution. The algorithm samples a uniform number between 0 and 1 and does a binary search in the cumulative probabilities, so its running time O(log(N)) whereas the running time of (multinomial-sample `probabilities`) is O(N). For this call to be really fast, the computation of `cumulative-probabilities` from `probabilities` must be done ahead of time (because it takes O(N) time). It can be done with (distribution2cumulative `probabilities cumulative-probabilities`).

See: (multinomial-sample `probabilities`)

See: (distribution2cumulative `probabilities cumulative-probabilities`)

(distribution2cumulative `probabilities cumulative-probabilities`**)**

Fill `cumulative-probabilities` by putting at the i-th position the sum the numbers in `probabilities` from the beginning to the (i-1)-th position. Therefore, cp[0]=0 and cp[N-1] may be `1. This is P(value < i).`

(element-in-set `int-element int-set`**)**

Return true iff `int-element` is in the `int-set` vector.

(idx-i1max `m`**)**

returns largest element in `m`

(idx-i1min `m`**)**

same as idx-m1min for vectors of integers

(idx-i1fill-with-index `m`**)**

fill the elements of idx1 `m` with their index.

(idx-i1fill `m c`**)**

```
((-idx1- (-int-)) m):
((-int-) c):
CREATED: P.Haffner, April 97
DESCRIPTION: Fills 1D int matrix with constant <c>
```

(idx-i2fill `m c`**)**

```
((-idx2- (-int-)) m)):
((-int-) c):
CREATED: P.Haffner, April 97
DESCRIPTION: Fills 2D int matrix with constant <c>
```

9.9.5 IDX of unsigned bytes

Utilities for matrices and tensors of unsigned bytes [partially documented]

(idx-ub1add `m c`**)**

```
((-idx1- (-ubyte-)) m):
((-int-) c):
CREATED: P.Haffner, April 97
DESCRIPTION: Adds 1D ubyte  matrix with constant <c>
```

(idx-ub2add `m c`**)**

```
((-idx2- (-ubyte-)) m):
((-int-) c):
CREATED: P.Haffner, August 97
DESCRIPTION: Adds 2D ubyte  matrix with constant <c>
```

(idx-ub2mul `m c`**)**

```
((-idx2- (-ubyte-)) m):
((-int-) c):
CREATED: P.Haffner, August 97
DESCRIPTION: Multipliess 2D ubyte  matrix with constant <c>
```

(idx-ub2-i2mul `i2 u2 c`**)**

```
((-idx2- (-ubyte-)) i2):
```

```
((-idx2- (-ubyte-)) u2):
((-int-) c):
CREATED: P.Haffner, Sept 97
DESCRIPTION:
```

(idx-ub3mat-add `m1 m2`**)**

```
((-idx3- (-ubyte-)) m1 m2):
CREATED: Oct 97
DESCRIPTION:
```

(idx-ub1fill `m c`**)**

```
((-idx1- (-ubyte-)) m):
((-int-) c):
CREATED: P.Haffner, April 97
DESCRIPTION: Fills 1D ubyte  matrix with constant <c>
```

(idx-ub2fill `m c`**)**

```
((-idx2- (-ubyte-)) m):
((-int-) c):
CREATED: P.Haffner, April 97
DESCRIPTION: Fills 2D ubyte matrix with constant <c>
```

(idx-ub-to-bits `ub`**)**

(idx-ub-from-bits `ub bits`**)**

(idx-i1indexmax `m`**)**

(idx-m2opp `m`**)**

```
((-idx2- (-flt-)) m):
CREATED: P.Haffner, June 97
TEMPORARIES: ?
DESCRIPTION: invert sign of elements
```

(idx-u2sub `ubim v`**)**

(idx-u2rev `ubim`**)**

(idx-copy-inv `u1 u2`**)**

```
((-idx2- (-ubyte-)) u1 u2):
CREATED: P.Haffner, August 97
DESCRIPTION: same as idx-copy, with byte inversion.
```

9.9.6 IDX reading and writing

Author(s): Pascal Vincent

Compilable routines for reading/writing vector/matrix/tensors from/to an open file.

These routines are based on the C-like Compilable file I/O routines. They use their own matrix file format, different from the traditional Lush matrix formats. Nevertheless, the traditional functions `load-matrix` and `save-matrix` are able to handle this format.

IDX file format for compilable I/O

Author(s): Pascal Vincent

the Lush interpreter has several function to read and write vectors/matrices/tensors in various portable ways, unfortunately, these functions cannot be called in compiled code at the moment. To alleviate the problem, a number of compilable functions are provided to read/write IDX. unfortunately again, the formats are not compatible with the those of the core read/write functions. This discrepancy exists for largely historical reasons and will eventually be fixed.

The format used by the compilable functions is as follows:

- A long word (4 bytes) coding the matrix type (see detail below)
- A long word (4 bytes) for EACH dimension of the matrix (thus for an idx2 it will be 2 long-words, the first one for the height, the second one for the width of the matrix)
- A (possibly reversed) memory-dump of the elements of the matrix (short: 2 bytes, int: 4 bytes, flt: 4 bytes real: 8 bytes)

The matrix types are encoded in the following way (hexadecimal representation):.PRE - idx1 of ubyte: 00 00 08 01 - idx2 of ubyte: 00 00 08 02 - idx3 of ubyte: 00 00 08 03 - idx1 of byte: 00 00 09 01 - idx2 of byte: 00 00 09 02 - idx3 of byte: 00 00 09 03 - idx1 of short: 00 00 0B 01 - idx2 of short: 00 00 0B 02 - idx3 of short: 00 00 0B 03 - idx1 of int: 00 00 0C 01 - idx2 of int: 00 00 0C 02 - idx3 of int: 00 00 0C 03 - idx1 of flt: 00 00 0D 01 - idx2 of flt: 00 00 0D 02 - idx3 of flt: 00 00 0D 03 - idx1 of real: 00 00 0E 01 - idx2 of real: 00 00 0E 02 - idx3 of real: 00 00 0E 03

The long-words and the elements of the matrix MUST correspond to the byte order and coding of the Sparc stations (classical big-endian, and IEEE standard floating point representation) If the system is not a Sparc, then the functions that write or load these matrixes reverse the bytes if necessary to keep the file format consistant between systems.

(little-endianp)

return t if the CPU is little endian (e.g. x86). and nil if it is not little endian (like sparcs).

(fread-idx1-ubyte `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-ubyte-)) m) ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx1-ubyte `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-ubyte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx2-ubyte `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-ubyte-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx2-ubyte `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-ubyte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx3-ubyte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-ubyte-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx3-ubyte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-ubyte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fwrite-idx4-ubyte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx4- (-ubyte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Yoshua Bengio 05 Feb 98
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx4-ubyte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx4- (-ubyte-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Yoshua Bengio 05 Feb 98
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fread-idx1-byte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-byte-)) m) ; the matrix to be read from the file
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx1-byte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-byte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx2-byte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-byte-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx2-byte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-byte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx3-byte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-byte-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx3-byte file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-byte-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx1-short file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-short-)) m) ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx1-short file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-short-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx2-short file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-short-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx2-short file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-short-)) m)  ; the matrix to be written to the file
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx3-short file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-short-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx3-short file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-short-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx1-int file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-int-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx1-int file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-int-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx2-int `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-int-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx2-int `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-int-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx3-int `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-int-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx3-int `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-int-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx1-flt `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-flt-)) m)  ; the matrix to be read from the file
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx1-flt file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-flt-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx2-flt file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-flt-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx2-flt file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-flt-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx3-flt file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-flt-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx3-flt `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-flt-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx1-real `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-real-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx1-real `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx1- (-real-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx2-real `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-real-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx2-real `file-pointer m`**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx2- (-real-)) m)  ; the matrix to be written to the file
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(fread-idx3-real file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-real-)) m)  ; the matrix to be read from the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: reads matrix <m> from file <file-pointer>
          <m> is "resized" to fit the size of the matrix in the file
```

(fwrite-idx3-real file-pointer m**)**

```
((-gptr-) file)   ; the file pointer returned by a fopen
((-idx3- (-real-)) m)  ; the matrix to be written to the file
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: writes matrix <m> to file <file-pointer>
```

(save-idx1-ubyte filename m**)**

```
((-str-) filename):
((-idx1- (-ubyte-)) m)
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx1-ubyte filename m**)**

```
((-str-) filename):
((-idx1- (-ubyte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx2-ubyte `filename m`**)**

```
((-str-) filename):
((-idx2- (-ubyte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx2-ubyte `filename m`**)**

```
((-str-) filename):
((-idx2- (-ubyte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx3-ubyte `filename m`**)**

```
((-str-) filename):
((-idx3- (-ubyte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx3-ubyte `filename m`**)**

```
((-str-) filename):
((-idx3- (-ubyte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx4-ubyte `filename m`**)**

```
((-str-) filename):
((-idx4- (-ubyte-)) m):
```

```
RETURNS: ()
CREATED: Yoshua Bengio,  5 Feb 98
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx4-ubyte `filename m`**)**

```
((-str-) filename):
((-idx4- (-ubyte-)) m):
RETURNS: ()
CREATED: Yoshua Bengio, 5 Feb 98
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx1-byte `filename m`**)**

```
((-str-) filename):
((-idx1- (-byte-)) m)
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx1-byte `filename m`**)**

```
((-str-) filename):
((-idx1- (-byte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx2-byte `filename m`**)**

```
((-str-) filename):
((-idx2- (-byte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx2-byte `filename m`**)**

```
((-str-) filename):
((-idx2- (-byte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx3-byte `filename m`**)**

```
((-str-) filename):
((-idx3- (-byte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx3-byte `filename m`**)**

```
((-str-) filename):
((-idx3- (-byte-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx1-short `filename m`**)**

```
((-str-) filename):
((-idx1- (-short-)) m)
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx1-short `filename m`**)**

```
((-str-) filename):
((-idx1- (-short-)) m):
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx2-short filename m**)**

```
((-str-) filename):
((-idx2- (-short-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx2-short filename m**)**

```
((-str-) filename):
((-idx2- (-short-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx3-short filename m**)**

```
((-str-) filename):
((-idx3- (-short-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx3-short filename m**)**

```
((-str-) filename):
((-idx3- (-short-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx1-int filename m**)**

```
((-str-) filename):
((-idx1- (-int-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx1-int filename m**)**

```
((-str-) filename):
((-idx1- (-int-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx2-int filename m**)**

```
((-str-) filename):
((-idx2- (-int-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx2-int filename m**)**

```
((-str-) filename):
((-idx2- (-int-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx3-int filename m**)**

```
((-str-) filename):
((-idx3- (-int-)) m):
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx3-int filename m**)**

```
((-str-) filename):
((-idx3- (-int-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx1-flt filename m**)**

```
((-str-) filename):
((-idx1- (-flt-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx1-flt filename m**)**

```
((-str-) filename):
((-idx1- (-flt-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx2-flt filename m**)**

```
((-str-) filename):
((-idx2- (-flt-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx2-flt `filename m`**)**

```
((-str-) filename):
((-idx2- (-flt-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx3-flt `filename m`**)**

```
((-str-) filename):
((-idx3- (-flt-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx3-flt `filename m`**)**

```
((-str-) filename):
((-idx3- (-flt-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx1-real `filename m`**)**

```
((-str-) filename):
((-idx1- (-real-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx1-real `filename m`**)**

```
((-str-) filename):
((-idx1- (-real-)) m):
```

```
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx2-real filename m**)**

```
((-str-) filename):
((-idx2- (-real-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx2-real filename m**)**

```
((-str-) filename):
((-idx2- (-real-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

(save-idx3-real filename m**)**

```
((-str-) filename):
((-idx3- (-real-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: saves matrix <m> to file <filename>
```

(load-idx3-real filename m**)**

```
((-str-) filename):
((-idx3- (-real-)) m):
RETURNS: ()
CREATED: Pascal Vincent 04/05/96
COMPILABLE: Yes
DESCRIPTION: loads matrix <m> from file <filename>
          <m> is "resized" to fit the size of the matrix in the file
```

9.9.7 IDX mapping from files

functions to map files into virtual memory and to access the data as an array

(mmap-idx `s type ndim magic`**)**

low-level function to map a file as an IDX.

(mmap-idx1-ubyte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx2-ubyte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx3-ubyte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx4-ubyte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx1-byte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx2-byte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx3-byte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx4-byte `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx1-short `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx2-short `s`**)**

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx3-short s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx4-short s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx1-int s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx2-int s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx3-int s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx4-int s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx1-float s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx2-float s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx3-float s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx4-float s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx1-double s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx2-double s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx3-double s)

map the IDX file `s` into virtual memory and returns an IDX to access it

(mmap-idx4-double s)

map the IDX file `s` into virtual memory and returns an IDX to access it

9.9.8 Convolution, Subsampling, Oversampling.

This library contains functions for performing 1D and 2D convolutions, backconvolutions, subsampling with local averaging, and oversampling by sample replication. Each functionality is provided as a type-independent splicing macro (DMD) with names of the form `midx-m1xxx` and `midx-m2xxx` , as well as type specific functions for floats, doubles, and bytes (with names of the form `idx-f1xxx` , `idx-d1xxx` , `idx-u1xxx` . Two kinds of convolution functions are provided: generic 1D and 2D convolutions for any size kernel, and optimized 1D convolutions with kernels of size 2, 3, 4 and 5. The optimized versions are faster than the regular ones by about a factor of 2 through the use of loop unrolling and circular register stacks.

Fast 1D Convolutions

a library of 1D convolutions that uses automatically generated code with unrolled loops and circular register stacks to limit memory accesses and speed up execution.

(midx-m1fastconvolacc `in kernel out ksize type`**)**

ultrafast 1D convolution with accumulation of result into the output. `in` must be a CONTIGUOUS idx1, `kernel` an idx1, `out` a CONTIGUOUS idx1, `ksize` must be the size of the kernel, and `type` a string containing a valid C type, e.g. "float" (preferably the type of the objects in the 3 vectors). This macro is expanded at compile time into a bunch of "cinline" instructions. It unrolls loops with a circular register stack to minimize index calculations and memory access. Though this code is particularly fast on processors that have lots of floating point registers (MIPS, etc), it manages to to beat the pedestrian version by about a factor of 2 on pentiums (reaching approx 200MFLOPS per GHz for kernel of size 5). THIS CODE DOES NOT DO ANY RUN-TIME CHECK FOR ARRAY SIZE, SO BE CAREFULL.

(idx-f1fastconvol2acc `input kernel output`**)**

super-fast 1D convolution with size 2 kernel (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing floats.

(idx-d1fastconvol2acc `input kernel output`**)**

super-fast 1D convolution with size 2 kernel (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing doubles.

(idx-u1fastconvol2acc `input kernel output`**)**

super-fast 1D convolution with size 2 kernel (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing ubytes.

(idx-f1fastconvol3acc `input kernel output`**)**

super-fast 1D convolution with size 3 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing floats.

(idx-d1fastconvol3acc `input kernel output`**)**

super-fast 1D convolution with size 3 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing doubles.

(idx-u1fastconvol3acc `input kernel output`**)**

super-fast 1D convolution with size 3 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing ubytes.

(idx-f1fastconvol4acc `input kernel output`**)**

super-fast 1D convolution with size 4 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing floats.

(idx-d1fastconvol4acc `input kernel output`**)**

super-fast 1D convolution with size 4 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing doubles.

(idx-u1fastconvol4acc `input kernel output`**)**

super-fast 1D convolution with size 4 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing ubytes.

(idx-f1fastconvol5acc `input kernel output`**)**

super-fast 1D convolution with size 5 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing floats.

(idx-d1fastconvol5acc `input kernel output`**)**

super-fast 1D convolution with size 5 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing doubles.

(idx-u1fastconvol5acc `input kernel output`**)**

super-fast 1D convolution with size 5 kernel. result is ACCUMULATED in the output (with unrolled loops and optimal register assignment) `input` and `output` must be contiguous idx1, `kernel` must be an idx1 all containing ubytes.

Generic Convolutions

Regular 1D and 2D convolution and backconvolutions for idx of doubles, floats, and ubytes. Each function take 3 arguments: input, kernel, output. The size of the output plus the size of the kernel minus 1 must equal to the size of the input in all dimensions (to prevent annoying boundary conditions). The backconvolution can be seen the transposed operator of the convolution.

(idx-unfold `m d ksize step`**)**

See: idx-unfold!

Return an idx on the same storage as `m` (pointing to the same data) with an added dimension at the end obtained by "unfolding" the `d` -th dimension. The size of the new dimension is `ksize` . This essentially manipulates the modulos to make convolutions look like matrix-vector multiplies. For example, a one-dimensional convolution between vector `v` and kernel `k` can be done as follows:

```
? (setq v [0 1 2 3 4 5 6])
= [ 0.00  1.00  2.00  3.00  4.00  5.00  6.00 ]
? (setq k [1 2 1])
= [ 1.00  2.00  1.00 ]
? (setq uv (idx-unfold v 0 3 1))
= [[ 0.00  1.00  2.00 ]
   [ 1.00  2.00  3.00 ]
   [ 2.00  3.00  4.00 ]
   [ 3.00  4.00  5.00 ]
   [ 4.00  5.00  6.00 ]]
? (setq r (double-array 5))
= [ 0.00  0.00  0.00  0.00  0.00 ]
? (idx-m2dotm1 uv k r)
= [ 4.00  8.00 12.00 16.00 20.00 ]
```

A subsampled convolution (where the kernel is stepped by more than one element) can be performed by setting the `step` parameter to a value other than 1:

```
? (setq uv (idx-unfold v 0 3 2))
= [[ 0.00  1.00  2.00 ]
   [ 2.00  3.00  4.00 ]
   [ 4.00  5.00  6.00 ]]
? (setq r (double-array 3))
= [ 0.00  0.00  0.00 ]
? (idx-m2dotm1 uv k r)
= [ 4.00 12.00 20.00 ]
```

Here are other amusing examples:

```
? (idx-unfold [3 4 5 6 7] 0 5 1)
= [[ 3.00  4.00  5.00  6.00  7.00 ]]
? (idx-unfold [3 4] 0 1 1)
= [[ 3.00 ]
  [ 4.00 ]]
```

There is no real need for most programmers to use the `idx-unfold` construct directly because the standard library contains efficient predefined 1D and 2D convolutions.

(midx-m1convol `input kernel output`**)**
macro for 1D convolution. all arguments are idx1 (of any numerical type).
(idx-f1convol `input kernel output`**)**
1D convolution on idx1 of floats
(idx-m1convol `input kernel output`**)**
1D convolution on idx1 of floats (alias of idx-f1convol)
(idx-d1convol `input kernel output`**)**
1D convolution on idx1 of double
(idx-u1convol `input kernel output`**)**
1D convolution on idx1 of ubytes
(midx-m1backconvol `input kernel output`**)**
macro for 1D back convolution
(idx-f1backconvol `input kernel output`**)**
1D back convolution on floats
(idx-m1backconvol `input kernel output`**)**
1D back convolution on floats (alias of idx-f1backconvol.
(idx-d1backconvol `input kernel output`**)**
1D back convolution on doubles
(idx-u1backconvol `input kernel output`**)**
1D back convolution on ubytes
(midx-m2convol `input kernel output`**)**
macro for 2D convolution. all arguments are idx2.
(idx-f2convol `input kernel output`**)**
2D convolution. all arguments are idx2 of floats
(idx-m2convol `input kernel output`**)**
2D convolution. all arguments are idx2 of floats. (alias for idx-f2convol)
(idx-d2convol `input kernel output`**)**
2D convolution. all arguments are idx2 of doubles.
(idx-u2convol `input kernel output`**)**
2D convolution. all arguments are idx2 of ubytes
(midx-m2convolacc `input kernel output`**)**
macro for 2D convolution with accumulation . all arguments are idx2.
(idx-f2convolacc `input kernel output`**)**
2D convolution with accumulation . all arguments are idx2 of floats
(idx-m2convolacc `input kernel output`**)**
2D convolution with accumulation . all arguments are idx2.

(idx-d2convolacc `input kernel output`**)**
2D convolution with accumulation . all arguments are idx2 of double.
(idx-u2convolacc `input kernel output`**)**
2D convolution with accumulation . all arguments are idx2 of ubyte.
(midx-m2backconvol `input kernel output`**)**
macro for 2D back-convolution. all arguments are idx2.
(idx-f2backconvol `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.
(idx-m2backconvol `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats (alias for idx-f2baconvol)
(idx-d2backconvol `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.
(idx-u2backconvol `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.
(midx-m2backconvolacc `input kernel output`**)**
macro for 2D back-convolution. all arguments are idx2.
(idx-f2backconvolacc `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.
(idx-m2backconvolacc `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.
(idx-d2backconvolacc `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.
(idx-u2backconvolacc `input kernel output`**)**
2D back-convolution. all arguments are idx2 of floats.

1D and 2D Subsampling and Oversampling

functions that subsample 1D and 2D idx where each output sample is the average of `n` adjacent input samples (with no overlap). The oversmapling functions simply replicate the samples `n` times.

(midx-m1subsample `input n output`**)**
macro for subsampling and summing of size `n`
(idx-f1subsample `input n output`**)**
subsampling and summing of size `n`
(idx-m1subsample `input n output`**)**
subsampling and summing of size `n`
(idx-d1subsample `input n output`**)**
subsampling and summing of size `n`
(idx-u1subsample `input n output`**)**
subsampling and summing of size `n`
(midx-m1oversample `input n output`**)**
macro for oversampling of size `n`
(idx-f1oversample `input n output`**)**
oversampling of size `n`
(idx-m1oversample `input n output`**)**
oversampling of size `n`

(idx-d1oversample `input n output`**)**
oversampling of size `n`
(idx-u1oversample `input n output`**)**
oversampling of size `n`
(midx-m2subsample `input nrows ncols output`**)**
macro for subsampling and summing of size `n`
(idx-f2subsample `input nrows ncols output`**)**
subsampling and summing of size `n`
(idx-m2subsample `input nrows ncols output`**)**
subsampling and summing of size `n`
(idx-d2subsample `input nrows ncols output`**)**
subsampling and summing of size `n`
(idx-u2subsample `input nrows ncols output`**)**
subsampling and summing of size `n`
(midx-m2oversample `input nrows ncols output`**)**
macro for oversampling of size `n`
(idx-f2oversample `input nrows ncols output`**)**
oversampling of size `n`
(idx-m2oversample `input nrows ncols output`**)**
oversampling of size `n`
(idx-d2oversample `input nrows ncols output`**)**
oversampling of size `n`
(idx-u2oversample `input nrows ncols output`**)**
oversampling of size `n`

9.9.9 IDX sorting and binary search functions

Sorting of and binary search in vectors of type double, float, or int. These functions require that the input vector is contiguous.

(array-bsearch `vec val`**)**

Return largest index of value `val` in sorted vector `vec` .

If `val` is not an element of `vec` , return the largest index of the smallest value in `vec` not greater than `val` . If all all elements in `vec` are greater than `val` return zero. The vector `vec` is assumed to be sorted in ascending order.

(array-sortup! `vec` [`ivec`]**)**

Sort elements in vector `vec` in ascending order and return `()` . If present, apply the same permutation to integer vector `ivec` .

(array-sortdown! `vec` [`ivec`]**)**

Sort elements in vector `vec` in descending order and return `()` . If present, apply the same permutation to integer vector `ivec` .

9.9.10 IDX operations on squares of elements

matrix and tensor operations that use the squares of the elements. This is used primarily for second derivative backpropagations in gradient-based learning algorithms.

(idx-m1squextm1 `m1 m2 m3`**)**

square outer product of `m1` and `m2` . M3ij = M1i * M2j^2

(idx-m2squextm2 `m1 m2 m3`**)**

square outer product of `m1` and `m2` . M3ijkl = M1ij * M2kl^2

(idx-m1squextm1acc `m1 m2 m3`**)**

square outer product of `m1` and `m2` . M3ij += M1i * M2j^2

(idx-m2squextm2acc `m1 m2 m3`**)**

square outer product of `m1` and `m2` . M3ijkl += M1ij * M2kl^2

(idx-m2squdotm1 `m1 m2 m3`**)**

multiply vector `m2` by matrix `m1` using square of `m1` elements M3i = sum_j M1ij^2 * M2j

(idx-m4squdotm2 `m1 m2 m3`**)**

multiply matrix `m2` by tensor `m1` using square of `m1` elements M3ij = sum_kl M1ijkl^2 * M2kl

(idx-m2squdotm1acc `m1 m2 m3`**)**

multiply vector `m2` by matrix `m1` using square of `m1` elements M3i += sum_j M1ij^2 * M2j

(idx-m4squdotm2acc `m1 m2 m3`**)**

multiply matrix `m2` by tensor `m1` using square of `m1` elements M3ij += sum_kl M1ijkl^2 * M2kl

(idx-m1squdotm1acc `m1 m2 m3`**)**

dot product between `m1` and `m2` , except square of terms of `m1` are used: M3 += sum_i M1i^2 * M2i

(idx-m2squdotm2acc `m1` `m2` `m3`**)**

dot product between matrices `m1` and `m2` , except square of terms of `m1` are used: M3 += sum_ij M1ij^2 * M2ij

9.9.11 Array binning

These functions create frequency vectors (histograms) of array values.

(vector-bin `v` `bs` [`fs`]**)**

See: vector-bin*

Bin values in vector `v` into bins `bs` and return a vector of frequencies. Optionally, write the output to `fs` accumulatively. Example:

```
? (let ((v (double-array 100)))
    (for* (i 0 100) (v i (rand)))
    (vector-bin v [0.0 0.1 0.3 0.5 0.7 1.0]) )
= [i     7    24    19    18    32]
```

(vector-bin* `v` `n` [`range`]**)**

See: vector-bin, histo>

Create histogram of values in vector `v` with `n` equally wide bins; return the bin centers and the histogram. With no `range` argument given the bins cover the range of values in `v` . Example:

```
? (let ((v (double-array 100)))
    (for* (i 0 100) (v i (rand)))
    (vector-bin* v 4) )
= ([d  0.13  0.38  0.63  0.88] [i    27    24    24    24])
```

Chapter 10

Packages

Author(s): Yann LeCun

Packages are libraries that are not considered part of the standard library because they may not be available on all the platforms, and they may not be useful to everybody. Packages are generally built around popular pre-existing C libraries that have been interfaced to Lush with a layer to facilitate their use in Lush applications. Current packages include

- An extensive library of classes for gradient-based machine learning algorithms, including neural nets, convolutional nets, and many others.
- an environment for writing simple video games (SDL)
- an interface to the popular 3D graphic API OpenGL/GLUT
- a partial interface to the LAPACK/BLAS Linear algebra libraries
- an interface to the Intel Computer Vision Library
- an interface to the Video4Linux video-grabbing API
- an interface to the Advanced Linux Sound Architecture API (Alsa)

10.1 Plotting with gnuplot

Gnuplot is a program for creating plots of functions or data that is preinstalled on many Unix systems. Plots are created by writing gnuplot programs that consist of commands for loading and selecting data from a file, controlling the appearance of points, curves, axes, legends and so forth.

Lush offers a high-level and a low-level interface to gnuplot. The high-level interface is easy to use and does not require much knowledge of the gnuplot command language. The low-level interface gives direct access to gnuplot. Using it requires good knowledge of the gnuplot program. To learn more about gnuplot go to [http://www.gnuplot.info/documentation].

10.1.1 The High-Level gnuplot Interface

In gnuplot terminology, a visualization of a data set is a "line". Gnuplot may visualize one or more data sets in a single plot command, and we say that a plot is composed of one ore more "lines".

Gnuplot may visualize two-dimensional data with command "plot", and three-dimensional data with command "splot". For either command, data may be visualized in a variety of ways which are called "plotting styles". The high-level interface offers the two special forms `plot` and `splot` for composing plots of two- and three- dimensional data, respectively.

(plot* `l1 .. ln`)

See: Plot-line generating functions

Compose a new graph from plot instructions `l1 .. ln` and return it.

When `plot*` appears within the scope of another `plot*` or `plot` then the plot instructions are added to the graph of the enclosing `plot*` -form.

(plot `l1 .. ln`)

See: Plot-line generating functions

Compose a new graph from plot instructions `l1 .. ln` , send it to a `Gnuplot` instance, and return the `Gnuplot` instance.

If symbol `plotter` is bound to a `Gnuplot` instance then `plot` sends the graph to `plotter` , otherwise it creates a new `Gnuplot` object. Note that, when providing `plotter` , `plot` does not send any extra commands to `plotter` to prepare gnuplot for a new plot (cf. help for gnuplot commands "reset" and "clear").

Plot-line Generators

The following functions each visualize a single data set in on of the plotting styles understood by gnuplot's plot command. The functions are to be used in a `plot*` - or `plot` -form.

Each line-generating function takes the data set (a vector or matrix), an optional legend key (a string), and other optional line-modifying arguments (see below).

(boxes `data` [...])

Plot data with boxes.

(circles `data` [...])

Plot data with circles.

(dots `data` [...])

Plot data with dots.

(image `data` [...])

Plot `data` as image. When `data` is a plain matrix, plot as grayscale image. When `data` is `m` x `n` x `3` -dimensional, plot as RGB image, when `m` x `n` x `4` -dimensional, as RGBA image.

Note that every data point is visualized as a colored rectangle. The coordinates of the rectangle's center points may be controlled by the `origin` , `dx` , and `dy` modifiers, respectively.

(lines `data` **[...])**
Plot data with lines.

(linespoints `data` **[...])**
Plot data with lines and points.

(xerrorbars `data` **[...])**
Plot data with dots and x-error bars.

(yerrorbars `data` **[...])**
Plot data with dots and y-error bars.

(xyerrorbars `data` **[...])**
Plot data with dots and x/y-error bars.

(points `data` **[...])**
Plot data with points.

Plot-line Modifiers

The functions in this category modify appearance of an individual line and may appear as optional argument in a plot-line generating function.

Example:

```
(linespoints xy (key "temperature") (axes 'x1y2) (ps p)))
```

Note: Not all plot-line modifiers are applicable with all plot styles. While all styles except the image styles accept `lc` , the `points` type, for instance, has no interior and no lines. Therefore `lw` , `lt` , `fill` and `bc` are not applicable with `points` .

(key `s`**)**
Add key `s` to legend for this plot-line.

(axes `a`**)**
Select axes for this plot-line. Possible values are `'x1y1` , `'x1y2` , `'x2y1` , and `'x2y2` .

(lc `arg`**)**
Choose line color for this plot-line.

(lt `n`**)**
Choose line type for this plot-line.

Line type is a terminal-specific property. Send "test" to gnuplot to see available line types.

(lw `w`**)**
Choose line width for this plot-line.

(ps `arg`**)**
Choose point size for this plot-line.

`Arg` may either be a number (multiple of the global point size) or a vector with same length as `data` (variable point size).

(pt `n`**)**
Choose point type for this plot-line.

Point type is a terminal-specific property. Send "test" to gnuplot to see available point types.

(fill `density`)

Set fill density (number between 0 and 1) for this plot-line.

(bc [`arg`])

Choose border color for this plot-line. With no argument, don't draw a border.

Example:

```
(bc 'black)
```

Modifiers for Images

The following modifiers apply to the `image` plot style only.

(origin <[x0 y0]>)

Set coordinates for the lower left data point or pixel (default is [0 0]).

(dx `d`)

Set sampling interval for grid-data.

(dy `d`)

See: dx

Set different sampling interval for Y-direction.

Objects

You may place objects in a plot at specified positions relative to the `x1y1` axes.

(add-arc `x y r a0 a1` [...])

Add arc with radius `r` from angle `a0` to `a1` at position `x` , `y` . Return the object id.

(add-arrow `x y a b` [...])

Add arrow pointing in direction `a` , `b` at position `x` , `y` and return the object id.

Note: Arrows have no fill and accept only the modifiers `lc lw` and `lt` .

(add-arrow* `x y x1 y1` [...])

Add arrow pointing from `x` , `y` to `x1` , `y1` and return the object id.

Note: Arrows have no fill and accept only the modifiers `lc lw` and `lt` .

(add-circle `x y r` [...])

Add circle with radius `r` to plot at position `x` , `y` and return the object id.

(add-label `x y s` [...])

Add label with text `s` at position `x,y` and return the object id.

(add-hline `y` [...])

See: add-vline, add-h2line

Add a horizontal line to plot at position `y` on axis y1 and return the object id.

(add-h2line `y` [...])

See: add-hline

Add a horizontal line to plot at position `y` on axis y2 and return the object id.

(add-vline `x` [**...**])
See: add-hline, add-v2line
Add a vertical line to plot at position `x` on axis x1 and return the object id.
(add-v2line `x` [**...**])
See: add-vline
Add a vertical line to plot at position `x` on axis x2 and return the object id.

Object Modifiers

Functions in this category modify appearance of individual objects. Note that, like plot-line modifiers, not all object modifiers are applicable with all objects.
Example:

```
(add-circle [5 3 2.2] (fill 0.2) (fc 'blue) (bc ()))
```

(fill `density`**)**
Set fill density (number between 0 and 1) for this object.
(fc `arg`**)**
Choose fill color for this object.
Example:

```
(fc 'purple)
```

(bc [`arg`]**)**
Choose border color for this object. With no argument, don't draw a border.
Example:

```
(bc 'black)
```

(zp `which`**)**
Choose z-position of object with respect to graph. Possible values are `'front` , `'back` , and `'behind` .
(bw `w`**)**
Choose border width for this object.

Other Graph Elements and Properties

The following functions add elements to or modify aspects of a plot. They may be used in `plot` or `splot` .
(title `str`**)**
Add title to the plot.
(xrange `from to`**)**
Set range of x-axis. Either `from` or `to` may be `*` , which indicates auto-scaling for the respective end.
(yrange `from to`**)**
Set range of y-axis.
(x2range `from to`**)**
Set range of alternate (upper) x-axis.

(y2range `from to`**)**
Set range of alternate (right-hand) y-axis.
(xy-aspect `r`**)**
Set the aspect ratio of the x- and y-axis scales to `r` .
(xlabel `str`**)**
Add label `str` to the x-axis.
(ylabel `str`**)**
Add an axis label to the y-axis.
(x2label `str`**)**
Add label `str` to the alternate (upper) x-axis.
(y2label `str`**)**
Add label `str` to the alternate (right-hand) y-axis.
(xtics `arg`**)**
Set tics for x-axis.
`Arg` may be a vector of tic coordinates or a `htable` mapping labels (strings) to tic coordinates.
(ytics `arg`**)**
See: xtics
Set tics for y-axis.
(ztics `arg`**)**
See: xtics
Set tics for z-axis.
notics
Turn off tics on all axes.
(x2tics `arg`**)**
See: xtics
Set tics for alternate (upper) x-axis.
(y2tics `arg`**)**
See: xtics
Set tics for alternate (right-hand) y-axis.
(grid [`which-tics`]**)**
See: xtics
Draw grid lines at tics-coordinates. With no argument, draw grid lines for all tics.
(margin `which f` [`which2 f2` **...**]**)**
Set margin to fraction `f` of canvas size.
Gnuplot chooses margins automatically be default. If this is not a good choice then each margin (`left` , `right` , `top` , `bottom`) may be set explicitly to `f` , which is a fraction of the horizontal or vertical extent of the plotting canvas, respectively.
Example:

```
(margin 'left 0.2  'right 0.9)
```

(linecolor [`color color2 color3` **...**]**)**
Set/query line color(s).

With no arguments return color for next plot-line. With arguments set line colors for the following lines.

(linewidth [`width width2 width3` **...**]**)**

Set/query line width(s).

With no arguments return line width for next plot-line. With arguments set line widths for the following lines.

(linetype [`type type2 type3` **...**]**)**

Set/query line type(s).

With no arguments return line type for next plot-line. With arguments set line types for the following lines.

(pointtype [`pt pt2 pt3` **...**]**)**

Set/query point type(s).

With no arguments return point type for next plot-line. With arguments set point types for the following lines.

(pointsize [`ps ps2 ps3` **...**]**)**

Set/query point size(s).

With no arguments return point size for next plot-line. With arguments set point size for the following lines.

10.1.2 The Low-Level gnuplot Interface

An object of class `Gnuplot` provides a simple interface to an external gnuplot process. A `Gnuplot` object behaves like a function. It takes one or more gnuplot commands (strings), sends them to the external process, receives the console output issued by gnuplot, prints this output to `stdout` , and yields `()` . In this "interactive mode" communicating to a gnuplot instance via a `Gnuplot` object is very similar to communicating with the gnuplot program directly in a system shell.

```
? (setq gp (new Gnuplot))
= ::gnuplot:a48e920
? (gp "show terminal")

   terminal type is x11

= ()
```

A `Gnuplot` object in "non-interactive mode" does not print to `stdout` but returns the output of gnuplot as a string.

```
? (setq gp (new Gnuplot 'interactive ()))
= ::gnuplot:a70d5b0
? (gp "show terminal")
= "\\n   terminal type is x11 \\n\\n"
```

Arguments to `Gnuplot` objects can be either strings or arrays. String arguments are passed to gnuplot verbatim, arrays are passed as inline data.

```
? (setq xs (arange -3 3 0.01))
= ::INDEX:<601>
? (setq ys (sin xs))
= ::INDEX:<601>
? (setq xys (array-combine* xs ys))        ; make a two-column table
= ::INDEX:<601x2>
? (gp "set xrange [-4:4]; plot '-'" xys)   ; send xy-data to gnuplot
= ()
```

(new-gnuplot [`x y`] `w h` [`title`]**)**

Create `Gnuplot` instance with X11 terminal of size `w` x `h` to appear at position (`x` , `y`).

Class Gnuplot

Instances of class `Gnuplot` serve as proxies for external gnuplot processes. Creating a new `Gnuplot` involves starting a new external gnuplot process. The external gnuplot process exists until the proxy object gets deleted (that is, until its destructor is called). A `Gnuplot` object may be "interactive" or "non-interactive", depending on the current state of the (boolean) instance variable `interactive` .

(new Gnuplot)

Create new gnuplot process and an associated `Gnuplot` object.

```
Optional keyword arguments:
logfile     : name of log file for debugging (default is nil)
interactive : true triggers interactive mode (default is t)
```

(==> `Gnuplot` variables)

Return `Htable` of currently defined gnuplot variables.

(==> `Gnuplot` colors)

Return `Htable` of named colors.
See: *gnuplot-colors*

(==> `Gnuplot` reset)

Reset gnuplot for new plot.

gnuplot-preamble

A gnuplot command string that is sent to each new gnuplot instance upon creation. You may customize this setting (e.g., in your lushrc.lsh file).

gnuplot-version

Version of gnuplot program.

gnuplot-build-options

String of gnuplot build options.

gnuplot-version-string

Fully qualified version of gnuplot program.

gnuplot-colors

`HTable` mapping color names to RGB triples.

gnuplot-shape-to-pointtype

`Htable` mapping shape names (symbols) to gnuplot pointtype codes.

10.2 GBLearn2: Machine Learning Library

Author(s): Yann LeCun

The GBLearn2 Library is an object-oriented framework for building, training and testing machine learning systems and algorithms. The "GB" stands for "gradient-based", but the library can handle learning algorithms that are not gradient-based.

The basic concepts of GBLearn2 are: modules, states, params, trainers, data sources, and meters.

10.2.1 GBLearn2: Basic Concepts

Modules

Modules (subclasses of `gb-module`) are components of learning machines. Modules generally accept at least the method `fprop` . which computes the output of the module from its inputs. Input and outputs are generally subclasses of `idx-state` . Many commonly used modules are pre-defined, including various types of neural nets layers, convolutional layers, RBFs, Softmax and many others. Many of these modules use inputs and outputs that are of class `idx3-ddstate` (more on this below).

Some modules provide two other methods: `bprop` and `bbprop` . Those methods are defined for modules that are used in gradient-based machine learning algorithms. Method `bprop` back-propagates gradients through the module, while `bbprop` backpropagate diagonal second derivatives.

An large collection of high-level "macro-modules" is provided which include various types of fully-connected neural nets, convolutional networks, time-delay neural nets, etc.

Machine learning systems are usually built by using one of these predefined classes, or by subclassing them. Specific machine are generally created by defining classes whose slots are `gb-modules` and `gb-states` , with appropriate `fprop` and `bprop` methods.

States

Modules exchange information through subclasses of `gb-state` .

Most predefined modules use `idx3-state` and its subclasses as input and output. The class `idx3-ddstate` stores numbers arranged in a tensor with 3 indices. Most of those modules (e.g. as convolutional layers) interpret the first index as a feature index, and the last two indexes as spatial dimensions. This allows to easily replicate the modules spatially (e.g. for applications that require scanning a classifier over all locations of an image, more on this later).

Machines and Cost Modules

Basic modules generally do not assume much about the kind of learning algorithm with which they will be trained. The most common form of training is gradient-based training. gradient-based training consists in finding the set of parameters that minimize a particular energy function (generally computed by averaging over a set of training examples).

Classes are provided to conveniently assemble trainable modules and cost modules (e.g. energy functions) into a complete learning machine. An example of such class is `idx3-supervised-module` . This class can be used for supervised training of classifiers. The `idx3-supervised-module` contains:

- a learning machine with one input and one output (both `idx3-ddstate`).
- a cost module whose first inputs is the output of the machine, whose second input is a desired category (stored in an idx0 of int), and whose output is an `idx0-ddstate` which contains the value of the energy.
- a classifier module which takes the machine's output and computes a `class-state` which stores the label of the recognized category together with confidence scores and runner-up categories.

Parameters

Modules may have trainable parameters that are computed or adjusted by the various learning algorithms. Learning algorithms are designed to operate on `gb-param` objects, so as to make the algorithm implementations independent of the details of the learning machine's architecture. Whenever a instance of a module is created, a `gb-param` object is passed to the constructor. The modules trainable parameters are added to the `param` object.

Trainers

Learning machines created as above are then inserted in a `gb-trainer` object. The `gb-trainer` class (or rather, its subclasses) define the learning algorithm being used to train the machine. The most commonly used `gb-trainer` is `supervised-gradient` , which implements learning algorithms based on stochastic gradient descent.

The `supervised-gradient` trainer class expects a machine whose `fprop` and `bprop` methods take input, an output, a desired output, and an energy. `supervised-gradient` is an generic interpreted class which makes no hard assumptions on the type of the objects it manipulates as long as those objects understand the required set of methods.

Training a machine in such a trainer is performed by calling one of the training methods (e.g. `train` in the case of `supervised-gradient`) with a data source and a `gb-meter` (and possibly other specific parameters, depending on the trainer and method considered).

Data Sources

A data source is a source of training and testing examples. Data sources are a bit like regular modules, but they generally don't have inputs. Most `gb-trainer` subclasses expect data sources to undertand the method `size` , `seek` , `tell` , and `next` , which they use to sweep through the set of examples.

Since many pre-defined machine expect an `idx3-state` as input, and an idx0 of int as desired output, the data source class `dsource-idx3l` exists for that purpose. Subclasses of `dsource-idx3l` are defined for databases of examples stored as vectors of floats or ubytes, or for databases of images of variable size.

Performance Meters

Meters are used by `gb-trainer` to measure the performance of learning machines. A commonly used one is the `classifier-meter` which can be used to measure the performance of supervised classifiers.

10.2.2 Examples and Demos

Learning the proverbial XOR with backprop

the content of this file is a complete example of how to build a single hidden layer backprop net to solve the XOR problem. Yeah, it's a lame example. Just have a look at the source code.

Lenet5 Convolutional Neural Network Demo

Author(s): Yann LeCun, December 2002

This is an example of how to train a convolutional network on the MNIST database (handwritten digits). The database can be obtained at http://yann.lecun.com A paper describing an experiment similar to the one demonstrated here is described in LeCun, Bottou, Bengio, Haffner: "gradient-based learning applied to document recognition", Proceedings of the IEEE, Nov 1998. This paper is also available at the above URL. This demo assumes that the MNIST data files are in LUSHDIR/local/mnist. If you installed them someplace else, simply do `(setq *mnist-dir* "your-mnist-directory")` before loading the present demo. The MNIST datafiles are:

- `train-images-idx3-ubyte` : training set, images, 60000 samples.
- `train-labels-idx1-ubyte` : training set, labels, 60000 samples.
- `t10k-images-idx3-ubyte` : testing set, images, 10000 samples.
- `t10k-labels-idx1-ubyte` : testing set, labels, 10000 samples.

10.2.3 Learning Algorithms

Author(s): Yann LeCun

Various learning algorithm classes are defined to train learning machines. Learning machines are generally subclasses of `gb-module` . Learning algorithm classes include gradient descent for supervised learning, and others.

eb-trainer

Abstract class for energy-based learning algorithms. The class contains an input, a (trainable) parameter, and an energy. this is an abstract class from which actual trainers can be derived.

supervised

A an abstract trainer class for supervised training with of a feed-forward classifier with discrete class labels. Actual supervised trainers can be derived from this. The machine's fprop method must have four arguments: input, output, energy, and desired output. A call to the machine's fprop must look like this:

```
(==> machine fprop input output desired energy)
```

By default, `output` must be a `class-state` , `desired` an idx0 of int (integer scalar), and `energy` and `idx0-ddstate` (or subclasses thereof). The meter passed to the training and testing methods should be a `classifier-meter` , or any meter whose update method looks like this:

```
(==> meter update output desired energy)
```

where `output` must be a `class-state` , `desired` an idx0 of int, and `energy` and `idx0-ddstate` .

(new supervised `m p` [`e in out des`]**)**
create a new `supervised` trainer. Arguments are as follow:

- `m` : machine to be trained.
- `p` : trainable parameter object of the machine.
- `e` : energy object (by default an idx0-ddstate).
- `in` : input object (by default an idx3-ddstate).
- `out` : output object (by default a class-state).
- `des` : desired output (by default an idx0 of int).

(==> supervised train `dsource mtr`**)**
train the machine with on the data source `dsource` and measure the performance with `mtr` . This is a dummy method that should be defined by subclasses.

(==> supervised test `dsource mtr`**)**
measures the performance over all the samples of data source `dsource` . `mtr` must be an appropriate meter.

(==> supervised test-sample `dsource mtr i`**)**
measures the performance over a single sample of data source `dsource` . This leaves the internal state of the meter unchanged, and can be used for a quick test of a whether a particular pattern is correctly recognized or not.

supervised-gradient

A basic trainer object for supervised stochastic gradient training of a classifier with discrete class labels. This is a subclass of `supervised` . The machine's fprop method must have four arguments: input, output, energy, and desired output. A call to the machine's fprop must look like this:

```
(==> machine fprop input output desired energy)
```

where `output` must be a `class-state` , `desired` an idx0 of int (integer scalar), and `energy` and `idx0-ddstate` . The meter passed to the training and testing methods should be a `classifier-meter` , or any meter whose update method looks like this:

```
(==> meter update output desired energy)
```

where `output` must be a `class-state` , `desired` an idx0 of int, and `energy` and `idx0-ddstate` . The trainable parameter object must understand the following methods:

- `(==> param clear-dx)` : clear the gradients.
- `(==> param update eta inertia)` : update the parameters with learning rate `eta` , and momentum term `inertia` .

If the diagonal hessian estimation is to be used, the param object must also understand:

- `(==> param clear-ddx)` : clear the second derivatives.
- `(==> param update-ddeltas knew kold)` : update average second derivatives.
- `(==> param compute-epsilons mu )` : set the per-parameter learning rates to the inverse of the sum of the second derivative estimates and `mu` .

(new supervised-gradient `m p [e in out des]`**)**
create a new `supervised-gradient` trainer. Arguments are as follow:

- `m` : machine to be trained.
- `p` : trainable parameter object of the machine.
- `e` : energy object (by default an idx0-ddstate).
- `in` : input object (by default an idx3-ddstate).
- `out` : output object (by default a class-state).
- `des` : desired output (by default an idx0 of int).

(==> supervised-gradient train-online `dsource mtr n eta [inertia] [kappa]`**)**
train with stochastic (online) gradient on the next `n` samples of data source `dsource` with global learning rate `eta` . and "momentum term" `inertia` . Optionally maintain a running average of the weights with positive rate `kappa` . A negative value for kappa sets a rate equal to - `kappa` / `age` . No such update is performed if `kappa` is 0.
Record performance in `mtr` . `mtr` must understand the following methods:

```
(==> mtr update age output desired energy)
(==> mtr info)
```

where `age` is the number of calls to parameter updates so far, `output` is the machine's output (most likely a `class-state`), `desired` is the desired output (most likely an idx0 of int), and `energy` is an `idx0-state` . The `info` should return a list of relevant measurements.

(==> supervised-gradient train `dsource mtr eta` [`inertia`] [`kappa`]**)**

train the machine on all the samples in data source `dsource` and measure the performance with `mtr` .

(==> supervised-gradient compute-diaghessian `dsource n mu`**)**

Compute per-parameter learning rates (epsilons) using the stochastic diaginal levenberg marquardt method (as described in LeCun et al. "efficient backprop", available at [http://yann.lecun.com]). This method computes positive estimates the second derivative of the objective function with respect to each parameter using the Gauss-Newton approximation. `dsource` is a data source, `n` is the number of patterns (starting at the current point in the data source) on which the estimate is to be performed. Each parameter-specific learning rate epsilon_i is computed as 1/(H_ii + mu), where H_ii are the diagonal Gauss-Newton estimates and `mu` is the blowup prevention fudge factor.

(==> `supervised-gradient` **saliencies** `ds n`**)**

Compute the parameters saliencies as defined in the Optimal Brain Damage algorithm of (LeCun, Denker, Solla, NIPS 1989), available at http://yann.lecun.com. This computes the first and second derivatives of the energy with respect to each parameter averaged over the next `n` patterns of data source `ds` . A vector of saliencies is returned. Component `i` of the vector contains `Si = -Gi * Wi + 1/2 Hii * Wi^2` , this is an estimate of how much the energy would increase if the parameter was eliminated (set to zero). Parameters with small saliencies can be eliminated by setting their value and epsilon to zero.

10.2.4 Data Sources

data sources are used in machine learning experiments as sources of training and testing examples. Data sources are pretty much like regular modules. Most data sources should implement the following methods to be compatible with most gradient-based learning algorithms

- `(==> ds fprop arg1 ... argn)` : write the current item into `arg1` ... `argn` .
- `(==> ds next)` : go to the next item in the data source.
- `(==> ds seek i)` : go to the i-th item in the data source.
- `(==> ds tell)` : return the index of the current item.
- `(==> ds size)` : return the total number of item.

Optionally, some data sources may implement the following methods:

- `(==> ds bprop arg1 ... argn)` : backpropagate gradients.
- `(==> ds bbprop arg1 ... argn)` : backpropagate diagonal second derivatives.

These methods can be used by algorithms that updates samples or preprocessings internal to the data source, based on gradients coming from the learning machine. However, since those methods are used by a very small number of very special learning algorithms, implementing them is generally not necessary.

dsource

Semi-abstract class for a data source. Various subclasses are defined for special cases. This root class defines the methods seek, tell, next, and size. Subclasses should define (or redefine) at least fprop and size.

(==> `dsource` seek `i`)
set pointer to `i` -th item

(==> `dsource` tell)
return index of current item.

(==> `dsource` next)
move pointer to next item.

(==> `dsource` size)
reutrns the size of the data source.

dsource-idx3l

An abstract data source class appropriate for most supervised learning algorithms that take input data in the form of an idx3-state and a label in the form of an idx0 of int. Most learning machines implemented in the library are designed to be compatible with this data source and its subclasses. This class just defines the fprop method prototypes. Subclasses actually implement the functionalities.

(==> `dsource-idx3l` fprop `out lbl`)
get the current item and copy the sample into `out` (an idx3-state) and the corresponding label into `lbl` (and idx0 of int).

dsource-idx3fl

a data source that stores samples as idx3 of floats. As a subclass of `dsource-idx3l` , this class is compatible with most learning machines defined in the gblearn2 library.

(new dsource-idx3fl `inp lbl`)
create a `dsource-idx3fl` where the input samples are slices of an PxDxYxX idx4 of floats passed as argument `inp` and labels are slices of an idx1 of int

passed as argument `lbl` . P is the number of samples, and D, Y, X the dimensions of each sample. If your data needs less that three dimensions simply setq Y and/or X to 1 (i.e. `inp` would be a PxDx1x1 matrix).

(==> `dsource-idx3fl` fprop `out lbl`)

get the current item and copy the sample into `out` (an idx3-state) and the corresponding label into `lbl` (and idx0 of int).

dsource-idx3ul

a data source that stores samples as idx3 of ubytes. As a subclass of `dsource-idx3l` , this class is compatible with most learning machines defined in the `gblearn2` library. The ubyte values (between 0 255) can be shifted and scaled by before being written to the output idx3-state. This data source is considerably more economical in term of memory than `dsource-idx3fl` (1 byte per sample per variable, versus 4).

(new dsource-idx3ul `inp lbl bias coeff`)

create a `dsource-idx3ul` where the input samples are idx3 slices of an PxDxYxX idx4 of ubytes passed as argument `inp` and labels are slices of an idx1 of int passed as argument `lbl` . P is the number of samples, and D, Y, X the dimensions of each sample. If your data needs less that three dimensions simply setq Y and/or X to 1 (i.e. `inp` would be a PxDx1x1 matrix). values in the item are shifted by `bias` and multiplied by `coeff` before being copied into the destination by `fprop` .

(==> `dsource-idx3ul` fprop `out lbl`)

get the current item and copy the sample into `out` (an idx3-state) and the corresponding label into `lbl` (and idx0 of int). Raw values are shifted by the `bias` parameter and multiplied by the `coeff` parameter before being copied into the x slot of `out` .

dsource-image

A data source that stores images of variable sizes, with any number of components per pixel, and one byte per component. As a subclass of `dsource-idx3l` , this class is compatible with most learning machines defined in the `gblearn2` library. The `fprop` method resizes the output argument to the size of the current image. Pixel values may be shifted and scaled before being written the the output `idx3-state` .

(new dsource-image `bias coeff`)

create an empty `dsource-image` . When accessing an item, the values are shifted by `bias` and multiplied by `coeff` .

(==> `dsource-image` size)

reutnrs the size of the database

(==> `dsource-image` fprop `dest lbl`)

writes the current item into the x slot of `dest` (an idx3-state) and the corresponding label to `lbl` (and idx0 of int). `dest` is automatically resized to the size of the current item.

(==> `dsource-image` load-ppms `flist clist`)

fills up the database with images from a bunch of PPM files `clist` is a list of labels (one integer for each image) or a single category, in which case all the loaded images will be in that category.

(==> `dsource-image` load-pgms `flist clist`)

fills up the database with images from a bunch of PGM files `clist` is a list of labels (one integer for each image) or a single category, in which case all the loaded images will be in that category.

(==> `dsource-image` save `basename`)

save database into pre-cooked IDX files. These files can be subsequently loaded quickly using the load or map methods. The database is saved in four files named `basename` images.idx, `basename` starts.idx, `basename` sizes.idx, and `basename` labels.idx

(==> `dsource-image` load `basename`)

load database from pre-cooked IDX files produced through the `save` method. The database will be loaded from four files named `basename` images.idx, `basename` starts.idx, `basename` sizes.idx, and `basename` labels.idx

(==> `dsource-image` map `basename`)

memory-map database from pre-cooked IDX files produced through the `save` method. This is MUCH faster than load, and consumes fewer memory/disk bandwidth. The database will be mapped from four files named `basename` images.idx, `basename` starts.idx, `basename` sizes.idx, and `basename` labels.idx

dsource-idx3l-narrow

a data source constructed by taking patterns in an existing data source whose indices are within a given range.

(new dsource-idx3l-narrow `base size offset`)

make a new data source by taking `size` items from the data source passed as argument, starting at item `offset` .

(==> `dsource-idx3l-permute` fprop `out lbl`)

copy current item and label into `out` and `lbl` .

dsource-idx3l-permute

a data source constructed by shuffling the items of a base data source.

(new dsource-idx3l-permute `base`)

make a new data source by taking the `base` data source and building a permutation map of its items. Initially, the permutation map is equal to the identity. It can be shuffled into a random order by calling the `shuffle` method.

(==> `dsource-idx3l-permute` fprop `out lbl`)

copy current item and label into `out` and `lbl` .

(==> `dsource-idx3l-permute` shuffle)

randomly shuffles the samples of the db.

dsource-idx3l-concat

a data source constructed by concatenating two base data sources.

(new dsource-idx3l-concat `base1 base2`**)**

make a new data source by taking concatenating two base data sources `base1` and `base2` . the Two data sources must be `dsource-idx3l` or subclasses thereof.

(==> `dsource-idx3l-concat` **fprop** `out lbl`**)**

copy item and label into `out` and `lbl` .

10.2.5 Meters

Author(s): Yann LeCun

Meters are classes used to measure the performance of learning machines. There are several types of meters for each specific situation. meters are generally assumed to have at least the following methods:

- update: updates the meter with the objects and values passed as argument.
- clear: resets the meter, so it can be used for a new series of measurements.
- test: simply prints performance information for the data passed as argument. This does not update any internal state.

Methods are provided to compute and display the information measured by a meter.

- display: display performance information on the terminal
- info: returns a list of the informations printed by display

(same-class? `actual desired` [`dummies`]**)**

return 0 if `actual` equals -1, otherwise, return 1 if `actual` and `desired` are equal, -1 otherwise.

classifier-meter

a class that can be used to measure the performance of classifiers. This is a simple version that does not record anything but simply computes performance measures.

(new classifier-meter [`comparison-function`]**)**

Create a new `classifier-meter` using `comparison-function` to compare actual and desired answers. By default the `same-class?` function is used for that purpose. It takes two integer arguments, and returns 1 if they are equal, -1 if they are different, and 0 if the first argument is -1 (which means reject).

(==> `classifier-meter` **clear)**

reset the meter. This must be called before a measurement campaign is started.

(==> `classifier-meter` update `age actual desired energy`)

update the meter with results from a new sample. `age` is the number of training iterations so far, `actual` (a `class-state`) the actual output of the machine, `desired` (an idx0 of int) the desired category, and `energy` (an idx0-state) the energy.

(==> `classifier-meter` info)

return a list with the age, the number of samples (number of calls to update since the last clear), the average energy, the percentage of correctly recognize samples, the percentage of erroneously recognized samples, and the percentage of rejected samples.

(==> `classifier-meter` display)

Display the meter's information on the terminal. namely, the age, the number of samples (number of calls to update since the last clear), the average energy, the percentage of correctly recognize samples, the percentage of erroneously recognized samples, and the percentage of rejected samples.

10.2.6 gb-states

Author(s): Yann LeCun

states are the main objects used to store state variables in or between modules.

(each-idx-slots `o s f1` [`f2...[fn]`])

evaluates lists in `f` for all active idx slots of object `o` , (as returned by (==> `o` idx-slots)) with `s` successively taking the values of the slots in question.

idx-state

an idx-state is a state that contains vector or matrix variables.

(==> `idx-state` idx-slots)

return the complete list of idx slots.

(new idx-state [`params`])

create a new simple-state. if `params` is absent, the simple-state is left unsized. Otherwise `params` must be a list of integer (possibly empty), which will be used to determine the size of the object. `params` has the same meaning as the 2nd parameter of the functions new-index.

(==> `idx-state` resize [`list-of-dims`])

resize all the idx slots (as returned by idx-slots) `list-of-dims` is a list with p numbers specifying the size of the last p dimensions of each slot. If `list-of-dim` is absent, all the active slots are undimed.

See: (==> `idx-state` idx-slots)

(==> `idx-state` undim)

undim all the active idx slots of `idx-state` .

(==> idx-state clear)
fill all the active idx slots with zeroes
See: (==> `idx-state` idx-slots)
(==> `idx-state` load `s`)
fill slot x with content of file `s`
(==> `idx-state` save `s`
save content of slot x into file `s` .
(==> `idx-state` dump `s`
save the entire object into file `s`

idx0-state

idx0-dstate

idx0-ddstate

idx1-state

idx1-dstate

idx1-ddstate

idx2-state

idx2-dstate

idx2-ddstate

idx3-state

idx3-dstate

idx3-ddstate

idx4-state

idx4-dstate

idx4-ddstate

class-state

a special kind of state used to store the output of a classifier. class-state are generated by modules such as class-max, and used by meters such as classifier-meter. No backprop is possible through a class-state.

lclass-state

a special kind of state used to store the output of a spatial classifier such as ledist-classer.

10.2.7 gb-param

Author(s): Yann LeCun

a hierarchy of classes that implements gradient-based learning algorithms. Various subclasses of gb-param can be used for various learning algorithms (stochastic gradient, with separate epsilons, with second derivatives.....) The standard gb-param type contains idx1 slots.

(new-index-offset `s dlist o`**)**

like new index, but puts the index at a specific offset in the storage `s` is a storage, `dlist` a list of dimensions, `o` an offset. Compilable macro.

idx1-param

a gb-param whose slots are idx1. This is an abstract class (useful classes are subclasses thereof). no learning algorithm is defined. Only the x slot is present. This class is useful for fixed (non adaptive) parameters.

(==> `idx1-param` **resize** `s`**)**

resize idx1-param to `s` elements.

(new idx1-param `s sts`**)**

create a new idx1-param. `s` is the size, and `sts` is the initially allocated size.

(==> `idx1-param` **load** `s`**)**

load the values of `idx1-param` from a the matrix file `s` , which must be an idx1 of floats. The number of elements in the file `s` must match the size of `idx1-param` .

(==> `idx1-param` **save** `s`**)**

save the slot `x` of the `idx1-param` in file `s` . This can be subsequently recovered with the load method.

(==> `idx1-param` **size)**

return the number of elements of `idx1-param` .

idx1-dparam

gb-param class for regular stochastic gradient descent algorithm

(new idx1-dparam `s sts`**)**

`s` is a size (possibly zero), and `sts` is the initial storage size which must be larger than zero, and (can be larger than `s` to avoid unnecessary reallocs when the size of the param is increased.

(==> `idx1-dparam` **update-deltas** `knew kold`**)**

update the average derivatives `deltas` as follow: `deltas = knew*dx + kold*deltas` . The `update` method calls this whenever it is called with a non-zero `inertia` parameter.

(==> `idx1-dparam` **update** `eta inertia`**)**

simple gradient descent update. This will use momentum if the `inertia` parameter is non zero. CAUTION: the `deltas` slot is not updated if the `inertia`

is zero. When `inertia` is non zero, the `deltas` are updated as follows: `deltas = (1-inertia)*dx + inertia*deltas` (where `dx` is the gradient) and the parameters are subsequently updated as follows: `x = eta*deltas` .

idx1-dparam-eps

gb-param class for gradient descent with separate epsilons for each parameter

(new idx1-dparam-eps `s sts`**)**

`s` is a size (possibly zero), and `sts` is the initial storage size which must be larger than zero, and (can be larger than `s` to avoid unnecessary reallocs when the size of the param is increased.

(==> `idx1-dparam-eps` **update** `eta inertia`**)**

simple gradient descent update with one individual learning rate per parameter. `eta` is a global learning rate by which each individual parameter learning rate will be multiplied. This will perform an update "with momentum" if the `inertia` parameter is non zero. CAUTION: the `deltas` slot is not updated if `inertia` is zero. When `inertia` is non zero, the `deltas` are updated as follows: `deltas = (1-inertia)*dx + inertia*deltas` (where `dx` is the gradient) and the parameters are subsequently updated as follows: `x = eta*deltas` .

(==> `idx1-dparam-eps` **set-epsilons** `m`**)**

copy the values in vector `m` to epsilons

(==> `idx1-dparam-eps` **set-epsilons** `m`**)**

copy the values in vector `m` to epsilons

idx1-ddparam

a gb-param class for the stochastic diagonal levenberg-marquardt algorithm. In addition to the usual update method, it has an update-bbprop method for computing the second derivatives, and a set-epsilons method to set the epsilons using the second derivatives.

(new idx1-ddparam `s alloc`**)**

`s` is the size (can be 0) `alloc` is the size of storages to be preallocated. This will prevent memory fragmentation when the size of the gb-param is subsequently increased.

(==> `idx1-ddparam` **clear-ddx)**

set all the `ddx` vector slot to zero.

(==> `idx1-ddparam` **clear-ddeltas)**

set all the `ddeltas` vector slot to zero.

(==> `idx1-ddparam` **update-ddeltas** `knew kold`**)**

update average second derivative `ddeltas` as follows: `ddeltas = knew*ddx + kold*ddletas` . where `ddx` is the instantaneous second derivative.

(==> `idx1-ddparam` **update-xaverage** `kappa`**)**

Update running average of x xaverage += kappa (x - xaverage)

(==> `idx1-ddparam` **copy-xaverage)**

Copy contents of xaverage into x

(==> `idx1-ddparam` swap-xaverage)

Swap contents of x and xaverage

(==> `idx1-ddparam` saliencies)

Compute the parameters saliencies as defined by the Optimal Brain Damage algorithm of (LeCun, Denker, Solla, NIPS 1989). This uses the average first and second derivatives of the energy with respect to each parameter to compute a saliency for each weight. The saliency is an estimate of how much the energy would increase by if the parameter was set to zero. It is computed as: `Si = -Gi * Wi + 1/2 Hii * Wi^2` . A vector of saliencies is returned. The `deltas` and `ddeltas` field must have relevant values before this function is called.

(==> `idx1-ddparam` compute-epsilons `mu`)

compute and set the epsilons using the second derivative. this method should be called after a few iteration of update-bbprop

Allocating an idx-state within an idx1-ddparam

It is often useful to have access to the parameters of a trainable module in two different ways. The first access method is through a slot in the module object (e.g. the slot kernel in a convolutional layer [a.k.a. c-layer]). This slot is generally an idxN-ddstate with an x slot (value), dx slot (gradient), and ddx slot (second derivatives). The second access method is through an idx1-ddparam that collects all the trainable parameters of a learning machine. The functions described here provide a way of allocating multiple idxN-ddstate instances within a single idx1-ddparam. As the modules of a learning machine are created, their paramter states are allocated within a single idx1-ddparam which collects all the parameters.

(==> `idx1-ddparam` alloc-idx0-ddstate)

Allocate an idx0-ddstate in `idx1-ddparam`

(==> `idx1-ddparam` alloc-idx1-ddstate `d0`)

Allocate an idx1-ddstate of size `d0 idx1-ddparam`

(==> `idx1-ddparam` alloc-idx2-ddstate `d0 d1`)

Allocate an idx2-ddstate of size `d0 , d1 idx1-ddparam`

(==> `idx1-ddparam` alloc-idx3-ddstate `d0 d1 d2`)

Allocate an idx3-ddstate of size `d0 , d1 , d2 idx1-ddparam`

(==> `idx1-ddparam` alloc-idx4-ddstate `d0 d1 d2 d3`)

Allocate an idx4-ddstate of size `d0 , d1 , d2 , d3 idx1-ddparam`

10.2.8 Module Libraries

modules

Author(s): Yann LeCun

In Lush, building and training a complex system is done by assembling basic blocks, called modules. Modules are subclasses of the class gb-module. Though there are several predefined module classes, you can define your own pretty easily. modules must understand 2 basic methods, fprop and bprop, whose arguments are the inputs and outputs of the module. Optionally most module should understand a bbprop method for computing diagonal second derivatives.

Modules can have as many input/output "ports" as desired. these "ports" are passed as arguments to the methods that require them, such as fprop and bprop. In most cases, these arguments belong to the class idx-state, or one of its subclasses. Some modules may have internal trainable parameters. When this is the case, an idx3-ddparam object must be passed to the constructor. Internal parameters will then be allocated in that param. the bprop and bbprop methods ACCUMULATE gradients in these parameters so multiple modules can share a single parameter and automatically compute the correct gradient. Gradients on input ports are NOT accumulated.

A special class called trainer provides a convenient way to train and test a module combined with pre- and post-processors. Once a module has been created, inserting it in an instance of the trainer class is the easiest and fastest way to train it on a database and to measure its performance. the trainer class understands methods such as train, test etc... Most of these methods take instances of database as argument. They also take another argument called a meter. A meter is an object whose role is to keep track of the performance of the machine during a training or test session. trainers, meters, and databases can be put in an instance of workbench that handles standard learning sequences (estimate second derivatives, train, test....)

A certain number of predefined basic modules are provided in modules.sn. This includes idx3-module, a "root" class of moduless with one input and one output, both of type idx3-state. Many predefined modules are subclasses of idx3-module. Also included are idx3-squasher (sigmoid layer), logadd-layer (transforms a idx3-state into an idx1-state by log-adding over the 2 spatial dimensions), mle-cost (cost module for minimizing the cost of the desired output).

gb-module

The class gb-module is the basic class for objects that can be used with the library of training routines. Specific trainable modules and cost functions etc... are subclasses of `gb-module` and can be combined to build complex adaptive machines. `gb-module` are expected to accept at least the methods `fprop` , `bprop` , and optionally the following methods: `bbprop` , `load` , and `save` . the external "plugs" of a `gb-module` are passed as argument to the methods. For example, the `fprop` method of a module with one input vector and one output vector, and one parameter vector can be called with

```
(==> <gb-module> fprop <input> <parameter> <output>)
```

where `input` , `parameter` and `output` are instances of the `gb-state` or one of its subclasses. As a convention, the methods `fprop` , `bprop` , and `bbprop` take the same arguments in the same order. Results of these methods are accumulated in the appropriate slot of the objects passed as paramters. This allows modules to share inputs and outputs while preserving the correctness of forward and backward propagations

a few convenient subclasses of `gb-module` are predefined in the `gblearn2` library. This includes cost functions, classifiers, and others.

(==> `gb-module` fprop [`args`])

performs a forward propagation on the `gb-module` . `args` are optional arguments which represent the external "plugs" of the module. When possible, modules with variable size outputs resize their output ports automatically.

See: (==> `gb-module` bprop [`args`])

See: (==> `gb-module` bbprop [`args`])

(==> `gb-module` bprop [`args`])

performs a backward propagation on the `gb-module` (propagates derivatives). `args` are optional arguments which represent the external "plugs" of the module. By convention, the list of `args` is the same as for the fprop method. bprop assumes fprop has been called beforehand. If the module has internal parameters, the bprop method will ACCUMULATE the gradients in it, so that multiple modules can share the same parameters

See: (==> `gb-module` fprop [`args`])

See: (==> `gb-module` bbprop [`args`])

(==> `gb-module` bbprop [`args`])

performs a backward propagation of second derivatives on the `gb-module` `args` are optional arguments which represent the external "plugs" of the module. By convention, the list of `args` is the same as for the fprop method. bbprop assumes fprop and bprop have been run beforehand. If the module has internal parameters, the bbprop method will ACCUMULATE second derivatives in it, so that multiple modules can share the same parameters

See: (==> `gb-module` fprop [`args`])

See: (==> `gb-module` bprop [`args`])

noop-module

a module that does not do anything (a place-holder). This is NOT an identity-function module not compilable

See: id-module

See: gb-module

id-module

identity function module. It's a straight pass-through forward and backward. arguments must be idx-ddstates non compilable.

idx4-module

a basic "root" class for modules that have one single idx-state input and one single idx4-state output. the fprop, bprop and bbprop methods of this root class merely act as identity functions

idx3-module

a basic "root" class for modules that have one single idx-state input and one single idx3-state output. the fprop, bprop and bbprop methods of this root class merely act as identity functions

idx2-module

a basic "root" class for modules that have one single idx-state input and one single idx2-state output. the fprop, bprop and bbprop methods of this root class merely act as identity functions

idx1-module

a basic "root" class for modules that have one single idx-state input and one single idx1-state output. the fprop, bprop and bbprop methods of this root class merely act as identity functions

idx4-squasher

a basic squashing function layer for idx4-state you can udefine subclasses of this to change the squashing function

idx3-squasher

a basic squashing function layer for idx3-state you can udefine subclasses of this to change the squashing function

idx4-sqsquasher

square of hyperbolic tangent (or a rational approximation to it).

idx3-sqsquasher

square of hyperbolic tangent (or a rational approximation to it).

idx4-halfsquare

takes half square of each component.

idx3-halfsquare

takes half square of each component.

logadd-layer

performs a log-add over spatial dimensions of an idx3-state output is an idx1-state

cost

costs are a special type of modules (although there is no definite subclass for them) with two inputs and one output. the output is an idx0-ddstate which stores a cost or energy. one of the inputs is meant to be the output of another module (e.g. a network), and the other input a desired output (or any kind of supervisor signal like a reinforcement). the gradient slot (dx) of the output state is generally filled with +1. That way, the bprop method of the cost module automatically computes the gradient.

idx3-cost

abstract class for a cost function that takes an idx3-state as input, an int as desired output, and an idx0-state as energy.

mle-cost

a cost module that propagates the output corresponding to the desired label. If the output is interpreted as a negative log likelihood, minimizing this output is equivalent to maximizing the likelihood. outputs are log-added over spatial dimensions in case of spatial replication.

(new mle-cost `classes si sj`**)**

make a new mle-cost. `classes` is an integer vector which contains the labels associated with each output. From that vector, the reverse table is constructed to map labels to class indices. Elements in `classes` must be positive or 0, and not be too large, as a table as large as the maximum value is allocated. `si` and `sj` are the expected maximum sizes in the spatial dimensions (used for preallocation to prevent memory fragmentation).

mmi-cost

a cost function that maximizes the mutual information between the actual output and the desired output. This assumes that the outputs are costs, or negative log likelihoods. this modules accepts spatially replicated inputs.

(new mmi-cost `classes priors si sj prm`**)**

make a new mmi-cost. `classes` is an integer vector which contains the labels associated with each output. From that vector, the reverse table is constructed to map labels to class indices. Elements in `classes` must be positive or 0, and not be too large, as a table as large as the maximum value is allocated. `priors` : an idx1 of gbtypes, whose size must be the size of `classes` +1. It specifies the prior probability for each class, and for the junk class. The prior for the junk class must be in the last element. In absence of a better guess, the prior vector should be filled with 1/n, where n is its size. `si` and `sj` are the expected maximum sizes in the spatial dimensions (used for preallocation to prevent memory fragmentation). `prm` is an idx1-ddparam in which the value that determines the constant cost of the junk class will be stored. If the system is to be trained without junk examples, this parameter can be set to a very large value, and not be trained. The effect of setting this parameter to a fixed value is to softly saturate the costs of all the class to the half-square of that value (the overall energy will never be significantly larger then the half-square of the set value), and to softly clip the gradients, i.e. the units whose cost is higher than the half-square of the set value will receive negligible gradients. The parameter can be learned ONLY IF junk examples (with label -1) are present in the training set. There is a method, called set-junk-cost that allows to directly set the value of the junk without having to figure out the half-square business.

(==> `mmi-cost` **set-junk-cost** `c`**)**

set the constant cost of the junk class to `c` . the underlying parameter is given the value (sqrt (* 2 `c`)), so `c` must be positive.

fed-cost

a replicable cost module that computes difference between the desired output (interpreted as a cost, log-summed over space) and the free energy of the set of outputs (i.e. the logsum of all the outputs over all locations). A label of -1 indicates that the sample is "junk" (none of the above). This cost module makes sense if it follows a an e-layer. FED stands for "free energy difference".

crossentropy-cost

a replicable cross-entropy cost function. computes the log-sum over the 2D spatial output of the log cross-entropy between the desired distribution over output classes and the actual distribution over output classes produced by the network. This is designed to

edist-cost

a replicable Euclidean distance cost function. computes the log-sum over the 2D spatial output of the half squared error between the output and the prototype with the desired label. this does not generate gradients on the prototypes

(new edist-cost `classes si sj p`**)**

make a new edist-cost. `classes` is an integer vector which contains the labels associated with each output. From that vector, the reverse table is constructed to map labels to class indices. Elements in `classes` must be positive

or 0, and not be too large, as a table as large as the maximum value is allocated. `si` and `sj` are the expected maximum sizes in the spatial dimensions (used for preallocation to prevent memory fragmentation). `p` is an idx2 containing the prototype for each class label. The first dimension of `p` should be equal to the dimension of `classes` . the second dimension of `p` should be equal to the number of outputs of the previous module. The costs are "log-summed" over spatial dimensions

wedist-cost

a replicable weighted Euclidean distance cost function. computes the log-sum over the 2D spatial output of the weighted half squared error between the output and the prototype with the desired label. this does not generate gradients on the prototypes.

(new wedist-cost `classes si sj p w`**)**

make a new wedist-cost. `classes` is an integer vector which contains the labels associated with each output. From that vector, the reverse table is constructed to map labels to class indices. Elements in `classes` must be positive or 0, and not be too large, as a table as large as the maximum value is allocated. `si` and `sj` are the expected maximum sizes in the spatial dimensions (used for preallocation to prevent memory fragmentation). `p` is an idx2 containing the prototype for each class label, and `w` is an idx2 with a single weight for each of these prototype and each of its elements. The first dimension of `p` (and `w`) should be equal to the dimension of `classes` . the second dimension of `p` (and `w`) should be equal to the number of outputs of the previous module. The costs are "log-summed" over spatial dimensions

weighted-mse-cost

This is similar to wedist-cost but the weights matrix may run over patterns. The desired output vector has size two: the first element gives the class label, and the second element gives the position (row) in the weights matrix to use for the weighted euclidean distance. It is a replicable weighted Euclidean distance cost function. computes the log-sum over the 2D spatial output of the weighted half squared error between the output and the prototype with the desired label. this does not generate gradients on the prototypes

(new weighted-mse-cost `classes si sj p w`**)**

make a new weighted-mse-cost. `classes` is an integer vector which contains the labels associated with each output. From that vector, the reverse table is constructed to map labels to class indices. Elements in `classes` must be positive or 0, and not be too large, as a table as large as the maximum value is allocated. `si` and `sj` are the expected maximum sizes in the spatial dimensions (used for preallocation to prevent memory fragmentation). `p` is an idx2 containing the prototype for each class label, and `w` is an idx2 with a single weight for each of these prototype and each of its elements. The first dimension of `p` (and `w`) should be equal to the dimension of `classes` . the second dimension of `p` (and `w`) should be equal to the number of outputs of the previous module. The costs are "log-summed" over spatial dimensions

ledist-cost

a replicable Euclidean distance cost function with LOCAL TARGETS at

each position. Target prototypes are associated to classes. The cost is the sum over the 2D output of the half squared error between the local output and the prototype with the desired label at that position. This does not generate gradients on the prototypes

(new ledist-cost `classes p`**)**

make a new ledist-cost. `classes` is an integer vector which contains the labels associated with each output. From that vector, the reverse table is constructed to map labels to class indices. Elements in `classes` must be positive or 0, and not be too large, as a table as large as the maximum value is allocated. `p` is an idx2 containing the prototype for each class label. The first dimension of `p` should be equal to the dimension of `classes` . the second dimension of `p` should be equal to the number of outputs of the previous module. The costs are summed over spatial dimensions.

Classifiers

idx3-classifier

The `idx3-classifier` module take an `idx3-state` as input and produce a `class-state` on output. A `class-state` is used to represent the output of classifiers with a discrete set of class labels.

min-classer

a module that takes an idx3-state, finds the lowest value and output the label associated with the index (in the first dimension of the state) of this lowest value. It actually sorts the labels according to their score (or costs) and outputs the sorted list.

(new min-classer `classes`**)**

makes a new min-classer. `classes` is an integer vector which contains the labels associated with each output.

max-classer

a module that takes an idx3-state, finds the lowest value and output the label associated with the index (in the first dimension of the state) of this lowest value. It actually sorts the labels according to their score (or costs) and outputs the sorted list.

(new max-classer `classes`**)**

makes a new max-classer. `classes` is an integer vector which contains the labels associated with each output.

edist-classer

a replicable Euclidean distance pattern matcher, which finds the class prototype "closest" to the output, where "close" is based on the log-added euclidean distances between the prototype and the output at various positions. This corresponds to finding the class whose a-posteriori probability is largest, when P(c—data) = sum_[position=x] P(c at x —data at x) / n_positions and the priors over classes are uniform, and the local class likelihoods P(data at x — c at x) are Gaussian with unit variance and mean = prototype(c).

ledist-classer

a replicable Euclidean distance pattern matcher, which finds the class prototype closest to the output for the vectors at each position in the output image.

(new ledist-classer `classes p`**)**

make a new ledist-classer. `classes` is an integer vector which contains the labels associated with each prototype. `p` is an idx2 containing the prototype for each class label. The first dimension of `p` should be equal to the dimension of `classes` . the second dimension of `p` should be equal to the number of outputs of the previous module.

mmi-classer

a classifier that computes class scores based on an MMI type criterion (a kind of softmax in log) It gives scores (cost) for all classes including junk. It should be used in conjunction with mmi-cost. This assumes that the output of the previous module are costs, or negative log likelihoods. this modules accepts spatially replicated inputs.

(new mmi-classer `classes priors si sj prm`**)**

makes a new mmi-classer. The arguments are identical to that of mmi-cost. In fact if an mmi-classer is to used in conjunction with an mmi-cost, they should share the prior vector and the parameter. sharing the parameter can be done by first building the classer, then reducing the size of the parameter by one, then creating the cost.

(==> `mmi-classer` **set-junk-cost** `c`**)**

set the constant cost of the junk class to `c` . the underlying parameter is given the value (sqrt (* 2 `c`)), so `c` must be positive. BE CAREFUL that the junk parameter of an mmi-classer is usually shared by an mmi-cost, changing one will change the other.

(build-ascii-proto `targets charset`**)**

idx3-supervised-module

a module that takes an idx3 as input, runs it through a machine, and runs the output of the machine through a cost function whose second output is the desired label stored in an idx0 of int.

Basic Neural Net Modules

Author(s): Yann LeCun

a bunch of standard compiled modules for building neural net architectures: sigmoids, RBF, fully connected, convolutions, subsampling layers. Each of these classes has methods for fprop, bprop, bbprop, load, save, forget, and setting various parameters.

f-layer

full connection between replicable 3D layers the full connection is only across the first dimension of the input and output layer. the other two dimensions are spatial dimensions accross which the weight matrix is shared. This is much more efficient than using a c-layer with a 1x1 convolution and a full-table.

(new f-layer `tin tout si sj sqsh`**)**

makes a new replicable fully-connected layer. `tin` and `tout` are the number of units in the input and output. `si` and `sj` are the vertical and horizontal sizes used for pre-allocating the internal state variables (they, as well as the output. are automatically resized, but the resizing is more efficient of it is a downsizing).

(==> `f-layer` save `file`)

save weights and biases

(==> `f-layer` load `file`)

load weights and biases

(==> `f-layer` set-squash `squash`)

sets the squashing function to `squash` . `squash` must be an instance of the the class `idx3-module` .

(==> `f-layer` forget `v p`)

initialize weights to random values uniformly distributed between -c and c, where c is v/(fanin^(1/p)). reasonnable values for v and p are 2.5 and 2.

(==> `f-layer` fprop `in out`)

forward prop with `in` as input and `out` as output. `in` and `out` must both be `idx3-state` .

(==> `f-layer` bprop `in out`)

backward prop with `in` as input and `out` as output. `in` and `out` must both be `idx3-state` .

(==> `f-layer` bbprop `in out`)

backward prop of second derivatives with `in` as input and `out` as output. `in` and `out` must both be `idx3-state` .

e-layer

full connection with Euclidean distance between replicable 3D layers. This is like a layer of RBF WITHOUT NON-LINEAR FUNCTIONS. the output is the square Euclidean distance between the input and the weight the full connection is only across the first dimension of the input and output layer. the other two dimensions are spatial dimensions accross which the weight matrix is shared.

(new e-layer `tin tout prm`)

new e-layer (Eucliden distance RBF). `tin` is the thickness of the input layer, `tout` is the thickness of the output layer, `prm` is the parameter.

e-layer-sparse

like e-layer, but each RBF is only connected to a subset of the inputs this should be used for "multiple bitmap" outputs.

(new e-layer-sparse `tin off prm`)

new e-layer (Eucliden distance RBF). `tin` is the thickness of the input layer (dimension of prototypes), `off` is an idx1 with as many elements as there are output. each element contains the offset in the input for the input vector of each output. `prm` is the parameter.

c-layer

convolutional layer module. Performs multiple convolutions between an idx3-state input and an idx3-state output. includes bias and sigmoid.

(new c-layer `ki kj ri rj tbl thick si sj sqsh`)

Creates a new convolution layer. `ki` (int) vertical kernel size. `kj` (int) horizontal kernel size. `ri int) vertical stride (number of pixels by which`

`the kernels are stepped) rj <int) horizontal stride tbl (idx2) N by 2 matrix containing source and destination feature maps for coresponding kernel thick (int) thickness of output layer (number of feature maps) si (int) vertical size for preallocation of internal state sj (int) horizontal size for preallocation of internal state sqsh (idx3-module) a squashing function module that operates on idx3-state. prm and idx1-ddparam from which the parameters will be allocated`

s-layer

subsampling layer class

(new s-layer `ki kj thick si sj sqsh prm`**)**

Creates a new subsampling layer for convolutional nets `ki` (int) vertical subsampling ratio. `kj` (int) horizontal subsampling ratio. `thick` (int) thickness of output layer (number of feature maps) `si` (int) vertical size for preallocation of internal state `sj` (int) horizontal size for preallocation of internal state `sqsh` (idx3-module) a squashing function module that operates on idx3-state. `prm` and idx1-ddparam from which the parameters will be allocated

net-ff

Author(s): Yann LeCun

standard one-hidden layer, fully-connected neural net. this network is replicable.

(new net-ff `thickin thick0 thick1 ini inj prm`**)**

create a new net-ff network.

```
<thickin>: number of inputs.
<thick0>: size of hidden layer
<thick1>: number of outputs
<ini> <inj>: expected max size of input over which the network
             shbould be replicated.
<prm> an idx1-ddparam in which the parameters will be allocated.
```

net-c

Author(s): Yann LeCun

a neural net class with one convolutional layer and one fully connected layer. The main purpose of this class is to make replicable fully connected networks. Unlike with net-ff, the input spatial dimensions are treated correctly here. So if you want to recognize characters, use this class rather than net-ff.

(new net-c `ini inj ki0 kj0 inthick outthick prm`**)**

makes a new net-cscscf module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer a standard fully-connected network can be obtained when `ini` = `ki0` and `inj` = `kj0` . `inthick` : number of input slices `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

net-cf

a neural net class with one convolutional layer and one fully connected layer. The main purpose of this class is to make replicable fully connected networks. Unlike with net-ff, the input spatial dimensions are treated correctly here. So if you want to recognize characters, use this class rather than net-ff.

(new net-cf `ini inj ki0 kj0 tbl0 outthick prm`**)**

makes a new net-cscscf module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer a standard fully-connected network can be obtained when `ini` = `ki0` and `inj` = `kj0` . `tbl0` : table of connections between input anf feature maps for first layer `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

net-cfe

Author(s): Yann LeCun

neural net with one convolutional layer, one (replicable) fully-connected layer, and one RBF (euclidiean distance) layer.

(new net-cfe `n e thick si sj`**)**

create a new net-cfe. `n` is a net-cf, `e` is a e-layer, `thick` is the number of outputs, and `si` , `sj` are the number of output replications.

net-cff

Author(s): Yann LeCun

a neural net class with one convolutional layer and two fully connected layers. The main purpose of this class is to make replicable 2-hidden layer fully connected networks.

(new net-cff `ini inj ki0 kj0 tbl0 f1thick outthick prm`**)**

makes a new net-cff module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer a standard fully-connected network can be obtained when `ini` = `ki0` and `inj` = `kj0` . `tbl0` : table of connections between input anf feature maps for first layer `f1thickk` : number of hidden units in second hidden layer. `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

net-ccc

Author(s): Yann LeCun

convolutional net with 3 convolutional layers.

(new net-ccc `ini inj ki0 kj0 tbl0 ki1 kj1 tbl1 ki2 kj2 tbl2 prm`**)**

makes a new net-ccc module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer `tbl0` : table of connections between input anf feature maps for first layer `ki1`

`kj1 tbl1` : kernel and table for next layer `ki2 kj2 tbl2` : same for last convolution layer `prm` an idx1-ddparam in which the parameters will be allocated.

net-cscf

Author(s): Yann LeCun

convolutional net where the layers are convolution, subsampling, convolution, full connect.

(new net-cscf `ini inj ki0 kj0 tbl0 si0 sj0 ki2 kj2 tbl2 outthick prm`**)**

makes a new net-cscf module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer `tbl0` : table of connections between input anf feature maps for first layer `si0 sj0` : subsampling for first layer `ki2 kj2 tbl2` : same for last convolution layer `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

net-cscfe

Author(s): Yann LeCun

convolutional net where the layers are convolution, subsampling, convolution, full connect, and euclidean distance RBF.

(new net-cscfe `n e thick si sj`**)**

`n` is a net-cscf, `e` is an e-dist layer, `thick` is the number of outputs. `si` , `sj` are the initial replication factors.

net-csccf

Author(s): Yann LeCun

a standard LeNet2/LeNet5 architecture without the last RBF layer (look into net-cscscfe for that).

(new net-csccf `ini inj ki0 kj0 tbl0 si0 sj0 ki1 kj1 tbl1 ki2 kj2 tbl2 outthick prm`**)**

makes a new net-csccf module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer `tbl0` : table of connections between input anf feature maps for first layer `si0 sj0` : subsampling for first layer `ki1 kj1 tbl1` : kernel and table for next layer `ki2 kj2 tbl2` : same for last convolution layer `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

net-cscscf

Author(s): Yann LeCun

Standard LeNet5-type architecture without the final e-dist RBF layer.

(new net-cscscf `ini inj ki0 kj0 tbl0 si0 sj0 ki1 kj1 tbl1 si1 sj1 ki2 kj2 tbl2 outthick prm`**)**

makes a new net-cscscf module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer `tbl0` : table of connections between input anf feature maps for first layer `si0 sj0` : subsampling for first layer `ki1 kj1 tbl1 si1 sj1` : same for next 2 layers `ki2 kj2 tbl2` : same for last convolution layer `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

net-cscscfe

Author(s): Yann LeCun

Standard LeNet5-type architecture.

(new net-cscscfe `n e thick si sj`**)**

`n` is a net-cscscf, `e` is an e-dist layer, `thick` is the number of outputs. `si , sj` are the initial replication factors.

net-cscscscf

Author(s): Yann LeCun

A deep convolutional net, with one convo/subsamp set than LeNet5.

(new net-cscscscf `ini inj ki0 kj0 tbl0 si0 sj0 ki1 kj1 tbl1 si1 sj1 ki2 kj2 tbl2 si2 sj2 ki3 kj3 tbl3 outthick prm`**)**

makes a new net-cscscf module. `ini inj` : expected max size of input for preallocation of internal states `ki0 kj0` : kernel size for first convolutional layer `tbl0` : table of connections between input and feature maps for first layer `si0 sj0` : subsampling for first layer `ki1 kj1 tbl1 si1 sj1` : same for next 2 layers... `ki2 kj2 tbl2 si2 sj2` : ...and the next 2 `ki3 kj3 tbl3` : same yet again for last convolution layer `outthick` : number of outputs. `prm` an idx1-ddparam in which the parameters will be allocated.

lenet5

LeNet5 is a convolutional network architecture described in several publications, notably in [LeCun, Bottou, Bengio and Haffner 1998]: "gradient-based learning applied to document recognition", Proc IEEE, Nov 1998. The paper is also available at [http://yann.lecun.com].

(new-lenet5 `image-height image-width ki0 kj0 si0 sj0 ki1 kj1 si1 sj1 hid output-size net-param`**)**

create a new instance of net-cscscf implementing a LeNet-5 type convolutional neural net. This network has regular sigmoid units on the output, not an extra RBF layer as described in the Proc. IEEE paper. The network has 6 feature maps at the first layer and 16 feature maps at the second layer with a connection matrix between feature maps as described in the paper. Arguments:

```
<image-height> <image-width>: height and width of input image
<ki0> <kj0>: height and with of convolutional kernel, first layer.
<si0> <sj0>: subsampling ratio of subsampling layer, second layer.
```

```
<ki1> <kj1>: height and with of convolutional kernel, third layer.
<si1> <sj1>: subsampling ratio of subsampling layer, fourth layer.
<hid>: number of hidden units, fifth layer
<output-size>: number of output units
<net-param>: idx1-ddparam that will hold the trainable parameters
             of the network
```

example

```
(setq p (new idx1-ddparam 0 0.1 0.02 0.02 80000))
(setq z (new-lenet5 32 32 5 5 2 2 5 5 2 2 120 10 p))
```

ccc-tdnn

Author(s): Yann LeCun

Class for a Time Delay Neural Network with 3 convolutional layers (with local connectivity). This type of network is appropriate for classifying sequences of time/frequency represnetations such as cepstrum, MEL Scale spectrum, and other.

(build-ccc-tdnn `params freqs max-seq-len fk1 fs1 nh1 tk1 fk2 fs2 nh2 tk2 fk3 fs3 nh3 tk3`**)**

Build a Time-Delay Neural Network for data such as spectral sequences in which the features have a topology, choosing the connection tables to obtain local connections in feature space (as specified by kernel size and stride for the frequency axis, for each of these layers). The input to the net is assumed to be Fx1xT where F is for example the number of frequency channels (spectral representation) and T is the length of the sequence. Here, the network has 3 convolutional layers. The arguments are the following:

```
- <params> is a idx1-ddparam on which to allocate parameters for the layers.
- <freqs> is the number of input frequency channels.
- <max-seq-len> = maximum sequence length.
- <fk1>,<fk2>,<fk3> = sizes of frequency kernels (= width of local freq. windows)
- <fs1>,<fs2>,<fs3> = step sizes which separate the successive frequency windows
- <nh1>,<nh2>,<nh3> = number of hidden units per frequency channel, for each layer
- <tk1>,<tk2>,<tk3> = sizes of the temporal kernels, for each layer.
```

(tdnn-present-pattern `from to mean idev temporal-window-size`**)**

`from` is a Txf source matrix, `to` is a fx1xT' destination state (T' = T + temporal-window-size - 1), `mean` is a f-vector to substract from source, `idev` is a f-vector to multiply by source. `temporal-window-size` is the length of the input of the network yielding an output of length 1.

norm-ftdnn

Wrapper around ccc-tdnn which does input normalization and pads the input according to the network architecture.

(new norm-ftdnn `n-inputs nh1 nh2 n-outputs weight-file norm-file` **[(-tk1 5)(-tk2 8)(-tk3 12) (-fk1 6)(-fk2 3)(-fk3 3) (-fs1 3)(-fs2 2)(-fs3 1) (max-seq-len 2000)])**

net-xx

generic `idx3-module` with two sub-modules in series.

(new net-xx `mod1 thick1 si1 sj1 mod2`**)**

create new `net-xx` with `mod1` as first module and `mod2` as second module. `thick1` , `si1` , sj1> are the initial sizes of the intermediate `idx3-ddstate` which is the output of `mod1` and the input of `mod2` .

net-xxx

generic `idx3-module` with three sub-modules in series.

(new net-xxx `mod1 thick1 si1 sj1 mod2 thick2 si2 sj2 mod3`**)**

create new `net-xx` with `mod1` as first module `mod2` as second module, and `mod3` as the third module. `thick1` , `si1` , sj1> are the initial sizes of the intermediate `idx3-ddstate` which is the output of `mod1` and the input of `mod2` . `thick2` , `si2` , sj2> are the initial sizes of the intermediate `idx3-ddstate` which is the output of `mod2` and the input of `mod3` .

gb-conx

Author(s): Yann LeCun

A module architecture that connects inputs to outputs (with exactly one input per output but maybe more than one output per input, i.e. a one to many map). The inputs and outputs are bidimensional (i.e. sequences of vectors). It has no adjustable parameters. Instead it has internal non-adjustable parameters specifying the input/output connectivity.

(new gb-conx `n-outputs n-inputs indices`**)**

Construct a gb-conx architecture. The parameters are given when constructing the corresponding machine.

See: machine

See: gb-module

See: gb-conx

(==> `gb-conx` fprop `parameters input output`**)**

Fprop method for a gb-conx architecture. Just gather the inputs into the output vectors. Both the input and output are assumed to be sequences of vectors [i,t] (2D matrices).

See: gb-conx

(==> `gb-conx` bprop `input output`**)**

Bprop method for a gb-conx architecture. Just scatter with accumulate the derivatives of the outputs into the derivative of the inputs. Both the input and output are assumed to be sequences of vectors [i,t] (2D matrices).

See: gb-conx

(==> gb-conx generate input output)

:output:x is a matrix whose columns have zeros and a single one. According to the permutation matrix `parameters` , a similar matrix :input:x is produced.

sum-arch

An architecture that takes the sum of its inputs. The input is a 2D matrix whose first index is `n-inputs` and second index is variable (e.g. time). The output is zero-dimensional (an idx0). There are no parameters.

See: gb-module

See: (new sum-arch)

See: (==> `sum-arch` fprop `parameters input output`)

See: (==> `sum-arch` bprop `parameters input output`)

See: (==> `sum-arch` output-dim `inputx`)

See: (==> `sum-arch` generate `parameters input output`)

(new sum-arch)

Construct a sum-arch gb-module.

See: machine

See: gb-module

See: sum-arch

(==> `sum-arch` output-dim `inputx`)

(==> `sum-arch` fprop `parameters input output`)

Fprop method for a sum-arch architecture. Just add all the inputs. The input is assumed to be any matrix (except an idx0). The output is an idx0.

See: sum-arch

(==> `sum-arch` bprop `parameters input output`)

Bprop method for a sum-arch architecture.

See: prod

(==> `sum-arch` generate `parameters input output`)

Generate a 2-d sequence of inputs. A prior distribution on sequence length is chosen here to be uniform in [5, 6, ..10]. Ignore the output. For each time step, the input vector is a vector of 0's and a single 1, chosen uniformly.

10.3 SDL: Simple DirectMedia Layer

Author(s): Yann LeCun

Lush's interface to the popular SDL library provides access to every function of the SDL API. It also provides a set of higher-level classes to make it easy to write 2D video games and other interactive multimedia applications. The high-level interface provides classes for double-buffered SDL screens, event grabbing, and sprites with pixel-accurate collision detection.

The help text associated with each low-level function is from the corresponding .h header file that comes with SDL. If you installed libSDL1.2-devel with RPM, you may find the API documentation in [file:/usr/share/doc/libSDL1.2-devel-1.2.3/index.html]Otherwise, the documentation is available at [http://www.libsdl.org]

The Tutorial section of the Lush manual contains a tutorial on how to write simple video games with Lush and SDL. [Tutorial: Writing Games with Lush and SDL.
]

10.3.1 Installing SDL

To make the SDL interface work with Lush, you need:

- The SDL libraries (libSDL1.2)
- The SDL development libraries and include files (libSDL1.2-devel)
- The SDL_image library (libSDL_image1.2-devel)
- The SDL_image development libraries and include files (libSDL_image1.2_devel)

Those are generally available as RPMs or APTs for most Linux distributions. Otherwise, you can download the latest version from [http://www.libsdl.org]

By default, Lush looks for the SDL .so libraries in "/usr/lib". It expects to find "libSDL-1.2.so.0" and "libSDL_image-1.2.so.0". If these libraries are installed someplace else on your system, you must edit packages/sdl/sdl-config.lsh.

10.3.2 Introduction to Lush's "SDL screen classes"

The present section is a quick introduction to the `libsdl` library. Loading and initializing the `libsdl` library is performed with:

```
(libload "sdl/libsdl")
(sdl-initialize)
```

Next, we can open an SDL screen. `libsdl` provides three kinds of SDL screen classes.

- The first kind, `sdl-screen` is prefered if you intend to draw on the screen using `libsdl` 's sprite class, or using SDL's low-level drawing functions.
- The second kind, `sdlgl-screen` is prefered if you intend draw 3D graphics using OpenGL.
- The third kind, `sdlidx-screen` is prefered if you want to draw your images into a regular Lush matrix, and quickly update an SDL screen with the content of that matrix.

sdlidx-screen class: Mapping the Contents of an IDX to an SDL Screen

Let's start by describing the `sdlidx-screen` class. Opening an `sdlidx-screen` is done by first allocating a `ubyte-array` of size HxWx4, and then creating an `sdlidx-screen` with this matrix as argument. This will open an SDL screen of width W and height H:

```
(setq depth 3)
(setq m (ubyte-array 480 640 depth))
(setq screen (new sdlidx-screen m "my title"))
```

If the `depth` parameter is 3, each element of matrix `m` is a component of an RGB pixel. So for example `(m y x 0)` is the R component of pixel (x, y) (from the top left of the screen). If the `depth` parameter is 4, each element of `m` is a component of an RGBA pixel. The A component is the Alpha channel (transparency). An alpha value of 255 is opaque, while 128 is half transparent. The matrix `m` can be seen as the "back buffer" of a double-buffered screen. Changing values in `m` will not immediately change the corresponding pixel on the screen. Once the desired RGBA image has been written in `m` , slapping it on the screen can be done with `(==> screen flip)` . If the screen was opened with `depth` =4, the image displayed will be an alpha-blended mixture of the previously displayed image and the content of `m` . Here is a complete example:

```
(libload "sdl/libsdl")
(libload "libimage/image-io")
(libload "libimage/rgbaimage")
(sdl-initialize)
(setq m (ubyte-array 480 640 3))
(setq screen (new sdlidx-screen m "my picture"))
(setq photo (rgbaim-resize (image-read-rgba "mypicture.jpg") 640 480 0))
(array-copy (idx-trim photo 2 0 3) m)
(==> screen flip)
```

Subsequent calls to `(==> screen flip)` will refresh the screen with the content of `m` . Note that SDL screens (unlike regular Lush graphic windows) are not automatically refreshed when uncovered from behind other windows. That is why SDL screens should be used only for animated graphics and games. Refresh rates on SDL screens are likely to be faster than with regular Lush graphic windows.

sdl-screen class: Doubled Buffered Animations

The `sdl-screen` class is intended for double-buffered animations using sprites (movable screen objects). This situation is primarily encountered when writing video games and other interactive multimedia applications.

Opening an `sdl-screen` is simply done with `(setq screen (new sdl-screen 800 600 "the title"))` . Drawing into an `sdl-screen` is made very simple by the `sdl-sprite` class (as explained below), but low-level SDL drawing can be performed by creating `SDL_surface` objects and calling `SDL_BlitSurface` on the `p` slot of the `sdl-screen` .

sdl-sprite: Movable Animated Objects

Interactive multimedia applications often involve drawing movable objects on the screen. This can be performed using Lush's sdl-sprite class, defined also in `sdl/libsdl` . Creating a sprite for drawing on the sdl-screen `screen` , and loading an image into it is done as follows:

```
(setq sprite (new sdl-sprite screen 1))
(==> sprite load-frame "mysprite.png" 0 0 0)
```

This creates a new `sdl-sprite` with id number 1 on screen `screen` , and loads frame 0 of this sprite with the image in file "mysprite.png" (the image can be in any format accepted by the SDL-image library dstributed with SDL). The three numerical arguments (all zeros in the example) are the frame number to be loaded, and the coordinates of the "handle" or "hot-point" of the sprite relative to the top-left corner.

Sprites can be moved around and drawn on the screen with:

```
(==> sprite move x y)
(==> sprite draw)
(==> screen flip)
```

All drawing operations occur in the back buffer (not on the visible screen) until the call to `(==> screen flip)` .

The main loop of a typical application will look something like this:

```
(while (not stop)
  [get user input, update coordinates, and such]
  (==> screen clear)
  (==> sprite1 move x1 y1)
  (==> sprite2 move x2 y2)
  (==> sprite1 draw)
  (==> sprite2 draw)
  (==> screen flip))
```

Sprites can have multiple frames. Switching between frames provides a mechanism for animating sprites. More details on how to write such applications (including reading keyboard events, animation and such) are given in the tutorial ["Tutorial: Writing Games with Lush and SDL."]. The code for that tutorial is available at .

sdlgl-screen class: OpenGL 3D Graphics in SDL Screens

The `sdlgl-screen` class is designed to allow OpenGL 3D graphics in an SDL screen. OpenGL graphics can be done in such an SDL screen, or can be done directly in GLUT windows. The `sdlgl-screen` should be used when SDL functionalities are used. Opening an `sdlgl-screen` can be done just like opening an `sdl-screen` :

```
(setq screen (new sdlgl-screen 640 480 "my title"))
```

An example of use of `sdlgl-screen` is available at .

10.3.3 SDL Demos

(sdl-lander-simple)

a simple lunar lander game in SDL that uses the high-level interface library (sdl-screen, sdl-sprite, sdl-event). Use the arrows keys to activate the engines. Hit "q" to quit.

Simple Demos of the SDL Lush interface

(sdl-showimage <filename)
load an image and display it in an SDL window example:

```
(sdl-showimage (concat lushdir "/packages/sdl/demos/moon.png"))
```

(sdl-bouncy `f b n`)
bounce a bunch of objects on a background `f` is an image filename for the object `b` is a backgroun image `n` is the number of objects. This uses the low-level SDL interface. example:

```
(bounce (concat lushdir "/packages/sdl/demos/lem.png") (concat lushdir "/packages/sdl/demos/moon.
```

10.3.4 LibSDL: High-level Interface to SDL

libsdl is a library of "high level" functions and classes built on top of the SDL (Simple Directmedia Layer) library to facilitate its use from Lush. It includes objects such as screens and sprites with pixel-accurate collision detection. It also includes access functions to keyboard and mouse events, as well as to several common SDL data structures.

(sdl-initialize)

initializes the SDL engine with all its subsystems (timer, audio, video, cdrom, joystick). This function MUST BE CALLED before any other SDL function or the interpreter will probably crash.

this function sets the global variable *sdl-initialized* to true, and only executes the initialization code if this variable is nil. Returns true on success. If you want to initialize only certain subsystems, call the lower-level function SDL_Init with the appropriate flags and set *sdl-initialized* to true before opening an sdl-screen.

(sdl-terminate)

This shuts down the SDL subsystem. `sdl-initialize` must be called before any new SDL call can be performed again. A call to `sdl-terminate` will close any open SDL screen. In fact, calling `sdl-terminate` is the only way to close an SDL screen.

sdl-screen

a class that creates and manipulates SDL Screens. This class can handle pixel-accurate collision detection. It maintains a timer that measures the time between screen flips. This allows to do time-accurate real-time physical simulation. An SDL Screens is not closed when the corresponding `sdl-screen` object is freed. Closing a screen can be done with a call to `(sdl-terminate)` .

(new sdl-screen `w h cap`**)**

open a new SDL Screen of size `w` , `h` with title `cap` . This attempts to open a double-buffered 32 bit-per-pixel surface in the video hardware. It reverts to 24 or 16 bits (with 32 bit emulation) if a 32 bit screen cannot be allocated. The function sdl-intialize MUST be called before opening an sdl-screen, or the interpreter will crash!

(==> `sdl-screen` **flip)**

flip the buffers. The content of the visible screen is replaced by the content of the back buffer (in which drawing operations are performed). This returns the time since the last call to flip in seconds (as a floating point number). The collision detection array is cleared by this call.

(==> `sdl-screen` **toggle-fullscreen)**

Toggle full screen mode. Returns 1 on success, 0 on failure.

(==> `sdl-screen` **clear)**

fill the screen with back. Nothing actually happens on the screen until the next call to flip.

(new sdlgl-screen `w h cap`**)**

just like sdl-screen, except that OpenGL drawing is enabled. Always opens a 32 bit window.

sdlidx-screen

A subclass of sdl-screen that associates an SDL screen with an IDX of the same size. Whatever is contained in the IDX can drawn on the screen by calling the "flip" method. Unlike sdl-screen, this class does not use SDL's double buffering mechanism, but uses its own back buffer in which all drawing operations take place. sdlidx-screen can be used with sprites. Sprites drawn into the screen will appear in the IDX. No collision detection between sprites and the background directly drawn into the IDX can be performed (but it works between sprites).

(new sdlidx-screen `m caption`**)**

opens a new SDL screen where IDX `m` (and idx3 of ubytes) is used as a back buffer in which images can be drawn. The last dimension of `m` should be 3 or 4. If it is 3, the RGB image in `m` will be drawn opaque, if it is 4, the RGBA image in `m` will be drawn on top of the current image using the A channel as transparency. The screen is updated by a call to the method flip.

(==> `sdlidx-screen` **flip)**

Update screen with content of the IDX back buffer. return number of milliseconds since the last call to flip.

sdl-event

a high-level class to manipulate SDL events.

(new sdl-event)

create a new sdl-event object.

(==> `sdle` **type)**

return type of sdl-even `sdle` .

(==> `sdle` **key-keysym-sym)**

return symbol of keyboard key symbol of an sdl-event.

(==> `sdle` **button-x)**

return mouse button 1 value of an sdl-event.

(==> `sdle` **button-y)**

return mouse button 2 value of an sdl-event.

(==> `sdle` **motion-x)**

return horizontal mouse motion of an sdl-event.

(==> `sdle` **motion-y)**

return vertical mouse motion of an sdl-event.

(==> `sdle` **free)**

sdl-event destructor

(==> `sdl-event` **get-arrows** `xyk`**)**

xyk is an idx1 of ints of size 3. (xyk 0) is set to +1 if right-arrow is pressed and -1 if left-arrow is pressed (xyk 1) is set to +1 if up-arrow is pressed and -1 if down-arrow is pressed (xyk 2) is set to the keysym of any other key that's pressed

(==> `sdl-event` **get-keys** `keymap keystate`**)**

`keymap` is an idx1 of -int- containing a list keys to be polled. The keys are identified by their SDLK code. `keystate` is an -idx1- of -int- of the same size.

Upon exit, each element of `keystate` whose corresponding key (as specified in keymap) is pressed down is set to 1. If the corresponding key is not pressed, it is set to 0. This method can be used for games when simultaneous keypresses of all the keys specified in `keymap` must be detected.

```
(libload "sdl/libsdl")
(sdl-initialize)
(setq scr (new sdl-screen 640 480 "Key Test"))
(setq ev (new sdl-event))
(setq kmap (int-matrix 4))
(kmap () '(SDLK_a SDLK_d SDLK_w SDLK_s))
(setq ks (int-array 4))
(while t
  (==> ev get-keys kmap ks)
  (when (<> 0 ((idx-sum ks)))
   (print ks)
   (sleep 0.25)))
```

sdl-sprite

a sdl-sprite class for drawing movable objects in an SDL screen with pixel-accurate collision detection. Sprites can contain multiple frames that can be flipped through to make animations. or rotations. Each sprite object can be given an ID (which is preferably between 1 and 32, but can be larger), which identifies it during collision detection. For example, all background sprites can be given ID 1, spaceship 1 ID 2, spaceship 2 ID 3, and missiles, ID 4. IDs do not need to be unique (several sprites can share the same ID), but then the collision detection mechanism won't be able to distinguish them.

(==> `sdl-sprite` set-hotpoint `hx hy`)
sets the "hot point" (the handle) of the sprite to coordinates `hx` , `hy` .

(==> `sdl-sprite` get-hotpoint)
get the "hot point" (the handle) of the sprite as a list (`hx` , `hy`).

(==> `sdl-sprite` get-width)
return width of current frame

(==> `sdl-sprite` get-height)
return height of current frame

(new sdl-sprite `screen id`)
create a new sprite on screen `screen` with ID `id` .

(==> `sdl-sprite` alloc-frame)
private method for allocating new frames in a sprite

(==> `sdl-sprite` make-frame `img i hx hy`)
private method to set frame `i` to SDL_Surface `img` . the screen must be initialized before calling this.

(==> `sdl-sprite` load-frame `file i hx hy`)

load image file `file` into frame `i` , and set the hot point (handle) to `hx hy` . The images file can be in any format that sdlimage can handle (see IMG_Load). The screen must be initialized before calling this

(==> `sdl-sprite` rotscale-frame `src dst angle coeff`)

take frame `src` , rotate it by `angle` degreed, and scale it by coefficient `coeff` , then sets the `dst` -th frame to the resulting image. This function is convenient for generating views of a sprite at all possible angles automatically. This must be done in advance as this functions is too slow to generate frames on-the-fly while the game is running.

(==> `sdl-sprite` make-frame-idx `m i hx hy`)

set frame `i` to the image contained in idx `m` . The idx will be interpreted as an heightxwidthx4 idx of ubytes containing an RGBA images. NOTE: the alpha channel of `m` must be set to 255 for opaque pixels. If `m` was filled up with image-read-rgb the alpha channel is likely to be all zero. The screen must be initialized before calling this

(==> `sdl-sprite` get-frame-idx `i`)

Return an idx containing the image of the `i` -th frame.

(==> `sdl-sprite` set-frame `i`)

make frame `i` the current frame (i.e. the one that will be drawn next).

(==> `sdl-sprite` get-frame)

return index of current frame.

(==> `sdl-sprite` move `lx ly`)

move sprite to `lx` , `ly` . This does not actually draw anything.

(==> `sdl-sprite` moverel `dx dy`)

move sprite by `dx` , `dy` relative to the current position. This does not actually draw anything.

(==> `sdl-sprite` draw)

draw the sprite on the screen using the current frame and position. This does not do any collision detection.

(==> `sdl-sprite` drawc)

draw the sprite on the screen using the current frame and position. This performs all the operation necessary to do pixel-accurate collision detection. Any pixel with a non-zero alpha value drawn on top of another previously drawn object triggers the collision detection.

(==> `sdl-sprite` test-collision)

Prepare to test if the sprite would collide with any sprite already drawn on the screen. Unlike `drawc` , this method does not draw the sprite and does not modify the screen's collision map. It provides a way to test for collision and take actions before the collisio actually happens. The function `collided` must be subsequently called to actually test if the current sprite would have collided with any other already drawn sprites.

(collided `sp1 sp2`)

returns true any sprite with the same id as `sp1` has collided with any sprite with the same id as `sp2` . For this to return a meaningful result, both sprites must have been drawn with the `drawc` , or the one must have been drawn with `drawc` , and the other one must have been sent the message `test-collision`

. This must be called before the screen flip (but after the calls to `drawc` or `test-collision`) since the screen flip clears the collision detection array.

mover

a class that can move sprites according to 2D Newtonian dynamics. The simplest use consists in creating a mover, then go around a loop that:

```
- gets input from the keyboard, mouse or joystick.
- calls the mover methods push, and update
- calls the mover method move with a sprite as argument
- draws the sprite
- flips the screen
- sets the deltat of the mover to the value
  returned by the screen flip method.
```

(==> `mover` set-mass `m`)
set the mass of the mover.
(==> `mover` get-mass)
return mass of mover
(==> `mover` set-deltat `dt`)
sets the time increment between updates (in seconds) to `dt` .
(==> `mover` get-deltat)
gets the time increment between updates.
(==> `mover` set-state `lx ly lvx lvy lax lay`)
(==> `mover` get-state)
(==> `mover` set-state `x y vx vy ax ay mass deltat`)
create a new sprite mover with initial position `x` , `y` , initial velocity `vx` , `vy` , initial acceleration `ax` , `ay` , mass `mass` , and time increment `deltat` .
(==> `mover` push `fx fy`)
apply a force vector `fx fy` to the mover. the unit is mass untis times pixels per second per second. In other words, a mover of mass 1, with a force vector of length 1, will accelerate 1 pixel per second per second (assuming the time increment is correct).
(==> `mover` update)
updates the positions and velocity according to 2D Newtonian mechanics.
(==> `mover` move-sprite `sp`)
move sprite `sp` to the position of the mover.

SDL_Rect function

functions to manipulate SDL_Rect structures Most users will prefer to use the higher level sdl-rect class rather than these functions
low-level functions on SDL-Rect
(new-sdl-rect `x y w h`)

allocate a new SDL_Rect structure, fill it up with `x` , `y` , `w` , `h` (position, width, height), and return a gptr to it. The result must be freed with free-sdl-rect.

(set-sdl-rect `r x y w h`)

sets the position width and height of SDL_Rect pointed to by `r` to the arguments.

(free-sdl-rect `r`)

deallocate SDL_Rect structure pointed to by `r` .

sdl-rect

a high-level class to manipulate SDL_Rect with automatic garbage collection and the like.

(new sdl-rect `x y w h`)

create a new sdl-rect.

SDL_Surface

manipulating SDL_Surface structures

low level functions on SDL_Surface

(sdl-surface-w `s`)

return width of SDL_Surface pointed to by `s` .

(sdl-surface-h `s`)

return height of SDL_Surface pointed to by `s` .

(sdl-surface-pitch `s`)

return pitch of SDL_Surface pointed to by `s` .

(sdl-surface-pixels `s`)

return a pointer to the pixel data of the SDL_Surface pointed to by `s` .

(sdl-surface-offset `s`)

(sdl-surface-ptr-idx `m`)

create an SDL_Surface from an IDX. the pixel area of the SDL_Surface points to the data of the IDX. CAUTION: this function is a bit dangerous because the idx can be deallocated without the sdl_surface knowing about it.

(idx-to-sdl-surface `m`)

Return an RGB SDL_Surface whose pixels are filled up with the content of IDX `m` . `m` must be an idx3 of ubyte whose last dimension must be 4. It will be interpreted as an RGBA image.

(sdl-surface-to-idx `surface`)

returns an idx3 of ubytes filled up with the RGBA pixel values of SDL_surface `surface` .

SDL Drawing Functions

to draw into SDL_Surfaces.

(sdl-fill-rect `dst x y w h r g` <b)

fills a rectangle defined by `x y w h` of an SDL_Surface `dst` with RGB values `r g b)`.

Collision Detection (low level functions)

a set of function to manage pixel-accurate collision detection between object drawn on an SDL_Surface.

(sdl-collide `id src srcrect dst dstrect coll`**)**

this is a low-level function for pixel-accurate collision detection. The arguments `src srcrect dst dstrect` play the samee role as the corresponding arguments of SDL_BlitSurface. `dst` is an idx3 of ints. The vector of ints obtained by fixing the first two indexes and varying the last is interpreted as a bitmap. The `id` -th bit of the bitmap at location `i` , `j` is set when a non-transparent pixel of an object whose ID is `id` has been painted at the corresponding location. `src` is an RLE-encoded mask of an object (a list of runs) as returned by rle-encode-alpha which indicates which pixels trigger collision. `srcrect` is a subrectangle in `src` that is actually painted (must be NULL in current implementation). `dstrect` is the clipping rectangle in `dst` . `coll` is an idx1 of int interpreted as a bitmap, which on exit contains a set bit for every already painted object that collided with the current object.

(sdl-test-collision `id src srcrect dst dstrect coll`**)**

this is a low-level function for pixel-accurate collision detection. This function is essentially identical to `sdl-collide` , except it doesn't modify the destination bitmap `dst` . This function can be used to get a list of which sprites would be collided if a sprite were drawn. On output, the i-th bit of `coll` (idx1 of ubyte) is set if the any pixel of `src` would collide with any non-transparent pixel of `dst` . The arguments `src srcrect dst dstrect` play the samee role as the corresponding arguments of SDL_BlitSurface. `dst` is an idx3 of ints. The vector of ints obtained by fixing the first two indexes and varying the last is interpreted as a bitmap. `src` is an RLE-encoded mask of an object (a list of runs) as returned by rle-encode-alpha which indicates which pixels trigger collision. `srcrect` is a subrectangle in `src` that is actually painted (must be NULL in current implementation). `dstrect` is the clipping rectangle in `dst` . `coll` is an idx1 of int interpreted as a bitmap, which on exit contains a set bit for every already painted object that collided with the current object.

SDL_Event manipulation

functions to create and manipulate SDL_Event structures for keyboard/mouse/joystick event grabbing.

low level access to SDL_event structures

(new-sdl-event)

allocate and return a pointer to a new SDL_Event structure.

(sdl-event-type `e`**)**

return the type field of an SDL_Event pointed to by `e` .

(sdl-event-key-keysym-sym `e`**)**

return the field key.keysym.sym of an SDL_Event pointed to by `e` .

(sdl-event-button-x `e`**)**

return the button.x field of an SDL_Event pointed to by `e` .

(sdl-event-button-y `e`**)**
return the button.y field of an SDL_Event pointed to by `e` .
(sdl-event-motion-x `e`**)**
return the motion.x field of an SDL_Event pointed to by `e` .
(sdl-event-motion-y `e`**)**
return the motion.y field of an SDL_Event pointed to by `e` .

Miscellaneous/internal functions

these are not meant to be called directly.
(rle-encode-alpha `image thres`**)**
private returns a gptr on an RLE-encoded binary mask (used for collision detection) where any pixel of SDL_Surface `image` whose alpha value is less than `thres` is considered transparent

10.3.5 Low-Level Interface to SDL

sdl/SDL.lsh

(SDL_Init `flags`**)**

```
/* This function loads the SDL dynamically linked library and initializes
 * the subsystems specified by 'flags' (and those satisfying dependencies)
 * Unless the SDL_INIT_NOPARACHUTE flag is set, it will install cleanup
 * signal handlers for some commonly ignored fatal signals (like SIGSEGV)
 */
extern DECLSPEC int SDL_Init(Uint32 flags);
Flags are:
SDL_INIT_TIMER        0x00000001
SDL_INIT_AUDIO        0x00000010
SDL_INIT_VIDEO        0x00000020
SDL_INIT_CDROM        0x00000100
SDL_INIT_JOYSTICK        0x00000200
SDL_INIT_NOPARACHUTE        0x00100000  /* don't catch fatal signals */
SDL_INIT_EVENTTHREAD  0x01000000) /* Not supported on all OS's */)
SDL_INIT_EVERYTHING   0x0000FFFF
```

(SDL_InitSubSystem `flags`**)**

```
/* This function initializes specific SDL subsystems */
extern DECLSPEC int SDL_InitSubSystem(Uint32 flags);
```

(SDL_QuitSubSystem `flags`**)**

```
/* This function cleans up specific SDL subsystems */
extern DECLSPEC void SDL_QuitSubSystem(Uint32 flags);
```

(SDL_WasInit flags**)**

```
/* This function returns mask of the specified subsystems which have
   been initialized.
   If 'flags' is 0, it returns a mask of all initialized subsystems.
*/
extern DECLSPEC Uint32 SDL_WasInit(Uint32 flags);
```

(SDL_Quit)

```
/* This function cleans up all initialized subsystems and unloads the
 * dynamically linked library.  You should call it upon all exit conditions.
 */
extern DECLSPEC void SDL_Quit(void);
```

sdl/SDL_active.lsh

(SDL_GetAppState)

```
/* Function prototypes */
/*
 * This function returns the current state of the application, which is a
 * bitwise combination of SDL_APPMOUSEFOCUS, SDL_APPINPUTFOCUS, and
 * SDL_APPACTIVE.  If SDL_APPACTIVE is set, then the user is able to
 * see your application, otherwise it has been iconified or disabled.
 */
extern DECLSPEC Uint8 SDL_GetAppState(void);
```

sdl/SDL_audio.lsh

(SDL_AudioInit driver_name**)**

```
/* These functions are used internally, and should not be used unless you
 * have a specific need to specify the audio driver you want to use.
 * You should normally use SDL_Init() or SDL_InitSubSystem().
 */
extern DECLSPEC int SDL_AudioInit(const char *driver_name);
```

(SDL_AudioQuit)

```
extern DECLSPEC void SDL_AudioQuit(void);
```

(SDL_AudioDriverName namebuf maxlen**)**

```
/* This function fills the given character buffer with the name of the
 * current audio driver, and returns a pointer to it if the audio driver has
 * been initialized.  It returns NULL if no driver has been initialized.
 */
extern DECLSPEC char *SDL_AudioDriverName(char *namebuf, int maxlen);
```

(**SDL_OpenAudio** desired obtained)

```
/*
 * This function opens the audio device with the desired parameters, and
 * returns 0 if successful, placing the actual hardware parameters in the
 * structure pointed to by 'obtained'.  If 'obtained' is NULL, the audio
 * data passed to the callback function will be guaranteed to be in the
 * requested format, and will be automatically converted to the hardware
 * audio format if necessary.  This function returns -1 if it failed
 * to open the audio device, or couldn't set up the audio thread.
 *
 * When filling in the desired audio spec structure,
 *  'desired->freq' should be the desired audio frequency in samples-per-second.
 *  'desired->format' should be the desired audio format.
 *  'desired->samples' is the desired size of the audio buffer, in samples.
 *     This number should be a power of two, and may be adjusted by the audio
 *     driver to a value more suitable for the hardware.  Good values seem to
 *     range between 512 and 8096 inclusive, depending on the application and
 *     CPU speed.  Smaller values yield faster response time, but can lead
 *     to underflow if the application is doing heavy processing and cannot
 *     fill the audio buffer in time.  A stereo sample consists of both right
 *     and left channels in LR ordering.
 *     Note that the number of samples is directly related to time by the
 *     following formula:  ms = (samples*1000)/freq
 *  'desired->size' is the size in bytes of the audio buffer, and is
 *     calculated by SDL_OpenAudio().
 *  'desired->silence' is the value used to set the buffer to silence,
 *     and is calculated by SDL_OpenAudio().
 *  'desired->callback' should be set to a function that will be called
 *     when the audio device is ready for more data.  It is passed a pointer
 *     to the audio buffer, and the length in bytes of the audio buffer.
 *     This function usually runs in a separate thread, and so you should
 *     protect data structures that it accesses by calling SDL_LockAudio()
 *     and SDL_UnlockAudio() in your code.
 *  'desired->userdata' is passed as the first parameter to your callback
 *     function.
 *
 * The audio device starts out playing silence when it's opened, and should
```

```
 * be enabled for playing by calling SDL_PauseAudio(0) when you are ready
 * for your audio callback function to be called.  Since the audio driver
 * may modify the requested size of the audio buffer, you should allocate
 * any local mixing buffers after you open the audio device.
 */
extern DECLSPEC int SDL_OpenAudio(SDL_AudioSpec *desired, SDL_AudioSpec *obtained);
```

(**SDL_GetAudioStatus**)

```
/*
 * Get the current audio state:
 */
typedef enum {
       SDL_AUDIO_STOPPED = 0,
       SDL_AUDIO_PLAYING,
       SDL_AUDIO_PAUSED
} SDL_audiostatus;
extern DECLSPEC SDL_audiostatus SDL_GetAudioStatus(void);
```

(**SDL_PauseAudio** pause_on)

```
/*
 * This function pauses and unpauses the audio callback processing.
 * It should be called with a parameter of 0 after opening the audio
 * device to start playing sound.  This is so you can safely initialize
 * data for your callback function after opening the audio device.
 * Silence will be written to the audio device during the pause.
 */
extern DECLSPEC void SDL_PauseAudio(int pause_on);
```

(**SDL_LoadWAV_RW** src freesrc spec audio_buf audio_len)

```
/*
 * This function loads a WAVE from the data source, automatically freeing
 * that source if 'freesrc' is non-zero.  For example, to load a WAVE file,
 * you could do:
 *       SDL_LoadWAV_RW(SDL_RWFromFile("sample.wav", "rb"), 1, ...);
 *
 * If this function succeeds, it returns the given SDL_AudioSpec,
 * filled with the audio data format of the wave data, and sets
 * 'audio_buf' to a malloc()'d buffer containing the audio data,
 * and sets 'audio_len' to the length of that audio buffer, in bytes.
 * You need to free the audio buffer with SDL_FreeWAV() when you are
 * done with it.
```

```
 *
 * This function returns NULL and sets the SDL error message if the
 * wave file cannot be opened, uses an unknown data format, or is
 * corrupt.  Currently raw and MS-ADPCM WAVE files are supported.
 */
extern DECLSPEC SDL_AudioSpec *SDL_LoadWAV_RW(SDL_RWops *src, int freesrc,
               SDL_AudioSpec *spec, Uint8 **audio_buf, Uint32 *audio_len);
```

(**SDL_LoadWAV** file spec audio_buf audio_len)

```
/* Compatibility convenience function -- loads a WAV from a file */
```

(**SDL_FreeWAV** <**audio_buf?**)

```
/*
 * This function frees data previously allocated with SDL_LoadWAV_RW()
 */
extern DECLSPEC void SDL_FreeWAV(Uint8 *audio_buf);
```

(**SDL_BuildAudioCVT** cvt src_format src_channels src_rate dst_format dst_channels dst_rate)

```
/*
 * This function takes a source format and rate and a destination format
 * and rate, and initializes the 'cvt' structure with information needed
 * by SDL_ConvertAudio() to convert a buffer of audio data from one format
 * to the other.
 * This function returns 0, or -1 if there was an error.
 */
extern DECLSPEC int SDL_BuildAudioCVT(SDL_AudioCVT *cvt,
              Uint16 src_format, Uint8 src_channels, int src_rate,
              Uint16 dst_format, Uint8 dst_channels, int dst_rate);
```

(**SDL_ConvertAudio** cvt)

```
/* Once you have initialized the 'cvt' structure using SDL_BuildAudioCVT(),
 * created an audio buffer cvt->buf, and filled it with cvt->len bytes of
 * audio data in the source format, this function will convert it in-place
 * to the desired format.
 * The data conversion may expand the size of the audio data, so the buffer
 * cvt->buf should be allocated after the cvt structure is initialized by
 * SDL_BuildAudioCVT(), and should be cvt->len*cvt->len_mult bytes long.
 */
extern DECLSPEC int SDL_ConvertAudio(SDL_AudioCVT *cvt);
```

(SDL_MixAudio `dst src len volume`**)**

```
/*
 * This takes two audio buffers of the playing audio format and mixes
 * them, performing addition, volume adjustment, and overflow clipping.
 * The volume ranges from 0 - 128, and should be set to SDL_MIX_MAXVOLUME
 * for full audio volume.  Note this does not change hardware volume.
 * This is provided for convenience -- you can mix your own audio data.
 */
```

(SDL_LockAudio)

```
/*
 * The lock manipulated by these functions protects the callback function.
 * During a LockAudio/UnlockAudio pair, you can be guaranteed that the
 * callback function is not running.  Do not call these from the callback
 * function or you will cause deadlock.
 */
extern DECLSPEC void SDL_LockAudio(void);
extern DECLSPEC void SDL_UnlockAudio(void);
```

(SDL_UnlockAudio)

```
unlock audio.
```

(SDL_CloseAudio)

```
/*
 * This function shuts down audio processing and closes the audio device.
 */
extern DECLSPEC void SDL_CloseAudio(void);
```

sdl/SDL_byteorder.lsh

(SDL_byteorder)

```
/* Pardon the mess, I'm trying to determine the endianness of this host.
   I'm doing it by preprocessor defines rather than some sort of configure
   script so that application code can use this too.  The "right" way would
   be to dynamically generate this file on install, but that's a lot of work.
 */
#if  defined(__i386__) || defined(__ia64__) || defined(WIN32) || \\
    (defined(__alpha__) || defined(__alpha)) || \\
```

```
     defined(__arm__) || \\
    (defined(__mips__) && defined(__MIPSEL__)) || \\
     defined(__LITTLE_ENDIAN__)
#define SDL_BYTEORDER       SDL_LIL_ENDIAN
#else
#define SDL_BYTEORDER       SDL_BIG_ENDIAN
#endif
```

sdl/SDL_cdrom.lsh

(CD_INDRIVE status**)**

```
/* The possible states which a CD-ROM drive can be in. */
typedef enum {
       CD_TRAYEMPTY,
       CD_STOPPED,
       CD_PLAYING,
       CD_PAUSED,
       CD_ERROR = -1
} CDstatus;
/* Given a status, returns true if there's a disk in the drive */
```

(SDL_CDNumDrives)

```
/* Returns the number of CD-ROM drives on the system, or -1 if
   SDL_Init() has not been called with the SDL_INIT_CDROM flag.
 */
extern DECLSPEC int SDL_CDNumDrives(void);
```

(SDL_CDName drive**)**

```
/* Returns a human-readable, system-dependent identifier for the CD-ROM.
   Example:
       "/dev/cdrom"
       "E:"
       "/dev/disk/ide/1/master"
*/
extern DECLSPEC const char * SDL_CDName(int drive);
```

(SDL_CDOpen drive**)**

```
/* Opens a CD-ROM drive for access.  It returns a drive handle on success,
   or NULL if the drive was invalid or busy.  This newly opened CD-ROM
```

```
   becomes the default CD used when other CD functions are passed a NULL
   CD-ROM handle.
   Drives are numbered starting with 0.  Drive 0 is the system default CD-ROM.
*/
extern DECLSPEC SDL_CD * SDL_CDOpen(int drive);
```

(**SDL_CDStatus** cdrom)

```
/* This function returns the current status of the given drive.
   If the drive has a CD in it, the table of contents of the CD and current
   play position of the CD will be stored in the SDL_CD structure.
*/
extern DECLSPEC CDstatus SDL_CDStatus(SDL_CD *cdrom);
```

(**SDL_CDPlayTracks** cdrom start_track start_frame ntracks nframes)

```
/* Play the given CD starting at 'start_track' and 'start_frame' for 'ntracks'
   tracks and 'nframes' frames.  If both 'ntrack' and 'nframe' are 0, play
   until the end of the CD.  This function will skip data tracks.
   This function should only be called after calling SDL_CDStatus() to
   get track information about the CD.
   For example:
       // Play entire CD:
       if ( CD_INDRIVE(SDL_CDStatus(cdrom)) )
              SDL_CDPlayTracks(cdrom, 0, 0, 0, 0);
       // Play last track:
       if ( CD_INDRIVE(SDL_CDStatus(cdrom)) ) {
              SDL_CDPlayTracks(cdrom, cdrom->numtracks-1, 0, 0, 0);
       }
       // Play first and second track and 10 seconds of third track:
       if ( CD_INDRIVE(SDL_CDStatus(cdrom)) )
              SDL_CDPlayTracks(cdrom, 0, 0, 2, 10);
   This function returns 0, or -1 if there was an error.
*/
extern DECLSPEC int SDL_CDPlayTracks(SDL_CD *cdrom,
              int start_track, int start_frame, int ntracks, int nframes);
```

(**SDL_CDPlay** cdrom start length)

```
/* Play the given CD starting at 'start' frame for 'length' frames.
   It returns 0, or -1 if there was an error.
*/
extern DECLSPEC int SDL_CDPlay(SDL_CD *cdrom, int start, int length);
```

(**SDL_CDPause** cdrom)

```
/* Pause play -- returns 0, or -1 on error */
extern DECLSPEC int SDL_CDPause(SDL_CD *cdrom);
```

(**SDL_CDResume** cdrom)

```
/* Resume play -- returns 0, or -1 on error */
extern DECLSPEC int SDL_CDResume(SDL_CD *cdrom);
```

(**SDL_CDStop** cdrom)

```
/* Stop play -- returns 0, or -1 on error */
extern DECLSPEC int SDL_CDStop(SDL_CD *cdrom);
```

(**SDL_CDEject** cdrom)

```
/* Eject CD-ROM -- returns 0, or -1 on error */
extern DECLSPEC int SDL_CDEject(SDL_CD *cdrom);
```

(**SDL_CDClose** cdrom)

```
/* Closes the handle for the CD-ROM drive */
extern DECLSPEC void SDL_CDClose(SDL_CD *cdrom);
```

sdl/SDL_endian.lsh

(**SDL_ReadLE16** src)

```
/* Read an item of the specified endianness and return in native format */
extern DECLSPEC Uint16 SDL_ReadLE16(SDL_RWops *src);
```

(**SDL_ReadBE16** src)

```
extern DECLSPEC Uint16 SDL_ReadBE16(SDL_RWops *src);
```

(**SDL_ReadLE32** src)

```
extern DECLSPEC Uint32 SDL_ReadLE32(SDL_RWops *src);
```

(**SDL_ReadBE32** src)

```
extern DECLSPEC Uint32 SDL_ReadBE32(SDL_RWops *src);
```

(**SDL_WriteLE16** dst value)

```
/* Write an item of native format to the specified endianness */
extern DECLSPEC int SDL_WriteLE16(SDL_RWops *dst, Uint16 value);
```

(**SDL_WriteBE16** dst value)

```
extern DECLSPEC int SDL_WriteBE16(SDL_RWops *dst, Uint16 value);
```

(**SDL_WriteLE32** dst value)

```
extern DECLSPEC int SDL_WriteLE32(SDL_RWops *dst, Uint32 value);
```

(**SDL_WriteBE32** dst value)

```
extern DECLSPEC int SDL_WriteBE32(SDL_RWops *dst, Uint32 value);
```

sdl/SDL_error.lsh

(SDL_GetError)

```
extern DECLSPEC char * SDL_GetError(void);
```

(**SDL_ClearError**)

```
extern DECLSPEC void SDL_ClearError(void);
```

(**SDL_Error** code)

```
/* Private error message function - used internally */
#define SDL_OutOfMemory()       SDL_Error(SDL_ENOMEM)
typedef enum {
       SDL_ENOMEM,
       SDL_EFREAD,
       SDL_EFWRITE,
       SDL_EFSEEK,
       SDL_LASTERROR
} SDL_errorcode;
extern void SDL_Error(SDL_errorcode code);
```

sdl/SDL_events.lsh

(SDL_PumpEvents)

```
/* Pumps the event loop, gathering events from the input devices.
   This function updates the event queue and internal input device state.
   This should only be run in the thread that sets the video mode.
*/
extern DECLSPEC void SDL_PumpEvents(void);
```

(SDL_PeepEvents events numevents action mask**)**

```
/* Checks the event queue for messages and optionally returns them.
   If 'action' is SDL_ADDEVENT, up to 'numevents' events will be added to
   the back of the event queue.
   If 'action' is SDL_PEEKEVENT, up to 'numevents' events at the front
   of the event queue, matching 'mask', will be returned and will not
   be removed from the queue.
   If 'action' is SDL_GETEVENT, up to 'numevents' events at the front
   of the event queue, matching 'mask', will be returned and will be
   removed from the queue.
   This function returns the number of events actually stored, or -1
   if there was an error.  This function is thread-safe.
*/
typedef enum {
       SDL_ADDEVENT,
       SDL_PEEKEVENT,
       SDL_GETEVENT
} SDL_eventaction;
/* */
extern DECLSPEC int SDL_PeepEvents(SDL_Event *events, int numevents,
                           SDL_eventaction action, Uint32 mask);
```

(SDL_PollEvent event**)**

```
/* Polls for currently pending events, and returns 1 if there are any pending
   events, or 0 if there are none available.  If 'event' is not NULL, the next
   event is removed from the queue and stored in that area.
 */
extern DECLSPEC int SDL_PollEvent(SDL_Event *event);
```

(SDL_WaitEvent event**)**

```
/* Waits indefinitely for the next available event, returning 1, or 0 if there
```

```
   was an error while waiting for events.  If 'event' is not NULL, the next
   event is removed from the queue and stored in that area.
 */
extern DECLSPEC int SDL_WaitEvent(SDL_Event *event);
```

(**SDL_PushEvent** event)

```
/* Add an event to the event queue.
   This function returns 0, or -1 if the event couldn't be added to
   the event queue.  If the event queue is full, this function fails.
 */
extern DECLSPEC int SDL_PushEvent(SDL_Event *event);
```

(**SDL_SetEventFilter** filter)

```
/*
  This function sets up a filter to process all events before they
  change internal state and are posted to the internal event queue.
  The filter is protypted as:
*/
typedef int (*SDL_EventFilter)(const SDL_Event *event);
/*
  If the filter returns 1, then the event will be added to the internal queue.
  If it returns 0, then the event will be dropped from the queue, but the
  internal state will still be updated.  This allows selective filtering of
  dynamically arriving events.
  WARNING:  Be very careful of what you do in the event filter function, as
            it may run in a different thread!
  There is one caveat when dealing with the SDL_QUITEVENT event type.  The
  event filter is only called when the window manager desires to close the
  application window.  If the event filter returns 1, then the window will
  be closed, otherwise the window will remain open if possible.
  If the quit event is generated by an interrupt signal, it will bypass the
  internal queue and be delivered to the application at the next event poll.
*/
```

(**SDL_GetEventFilter**)

```
/*
  Return the current event filter - can be used to "chain" filters.
  If there is no event filter set, this function returns NULL.
*/
extern DECLSPEC SDL_EventFilter SDL_GetEventFilter(void);
```

(**SDL_EventState** type state)

```
/*
  This function allows you to set the state of processing certain events.
  If 'state' is set to SDL_IGNORE, that event will be automatically dropped
  from the event queue and will not event be filtered.
  If 'state' is set to SDL_ENABLE, that event will be processed normally.
  If 'state' is set to SDL_QUERY, SDL_EventState() will return the
  current processing state of the specified event.
*/
```

sdl/SDL_getenv.lsh

(SDL_putenv variable**)**

```
/* Put a variable of the form "name=value" into the environment */
extern DECLSPEC int SDL_putenv(const char *variable);
```

(SDL_getenv name**)**

```
/* Retrieve a variable named "name" from the environment */
extern DECLSPEC char *SDL_getenv(const char *name);
```

sdl/SDL_image.lsh

(IMG_LoadTyped_RW src freesrc type**)**

```
/* Load an image from an SDL data source.
   The 'type' may be one of: "BMP", "GIF", "PNG", etc.
   If the image format supports a transparent pixel, SDL will set the
   colorkey for the surface.  You can enable RLE acceleration on the
   surface afterwards by calling:
       SDL_SetColorKey(image, SDL_RLEACCEL, image->format->colorkey);
 */
extern DECLSPEC SDL_Surface *IMG_LoadTyped_RW(SDL_RWops *src, int freesrc,
                                              char *type);
```

(IMG_Load file**)**

```
/* Convenience functions */
extern DECLSPEC SDL_Surface *IMG_Load(const char *file);
```

(IMG_Load_RW src freesrc**)**

```
extern DECLSPEC SDL_Surface *IMG_Load_RW(SDL_RWops *src, int freesrc);
```

(IMG_InvertAlpha on**)**

```
/* Invert the alpha of a surface for use with OpenGL
   This function is now a no-op, and only provided for backwards compatibility.
*/
extern DECLSPEC int IMG_InvertAlpha(int on);
```

sdl/SDL_joystick.lsh

(SDL_NumJoysticks)

```
/* Function prototypes */
/*
 * Count the number of joysticks attached to the system
 */
extern DECLSPEC int SDL_NumJoysticks(void);
```

(SDL_JoystickName device_index**)**

```
/*
 * Get the implementation dependent name of a joystick.
 * This can be called before any joysticks are opened.
 * If no name can be found, this function returns NULL.
 */
extern DECLSPEC const char *SDL_JoystickName(int device_index);
```

(SDL_JoystickOpen device_index**)**

```
/*
 * Open a joystick for use - the index passed as an argument refers to
 * the N'th joystick on the system.  This index is the value which will
 * identify this joystick in future joystick events.
 *
 * This function returns a joystick identifier, or NULL if an error occurred.
 */
extern DECLSPEC SDL_Joystick *SDL_JoystickOpen(int device_index);
```

(SDL_JoystickOpened device_index**)**

```
/*
 * Returns 1 if the joystick has been opened, or 0 if it has not.
 */
extern DECLSPEC int SDL_JoystickOpened(int device_index);
```

(SDL_JoystickIndex joystick**)**

```
/*
 * Get the device index of an opened joystick.
 */
extern DECLSPEC int SDL_JoystickIndex(SDL_Joystick *joystick);
```

(SDL_JoystickNumAxes joystick**)**

```
/*
 * Get the number of general axis controls on a joystick
 */
extern DECLSPEC int SDL_JoystickNumAxes(SDL_Joystick *joystick);
```

(SDL_JoystickNumBalls joystick**)**

```
/*
 * Get the number of trackballs on a joystick
 * Joystick trackballs have only relative motion events associated
 * with them and their state cannot be polled.
 */
extern DECLSPEC int SDL_JoystickNumBalls(SDL_Joystick *joystick);
```

(SDL_JoystickNumHats joystick**)**

```
/*
 * Get the number of POV hats on a joystick
 */
extern DECLSPEC int SDL_JoystickNumHats(SDL_Joystick *joystick);
```

(SDL_JoystickNumButtons joystick**)**

```
/*
 * Get the number of buttons on a joystick
 */
extern DECLSPEC int SDL_JoystickNumButtons(SDL_Joystick *joystick);
```

(SDL_JoystickUpdate)

```
/*
 * Update the current state of the open joysticks.
 * This is called automatically by the event loop if any joystick
 * events are enabled.
 */
extern DECLSPEC void SDL_JoystickUpdate(void);
```

(**SDL_JoystickEventState** state)

```
/*
 * Enable/disable joystick event polling.
 * If joystick events are disabled, you must call SDL_JoystickUpdate()
 * yourself and check the state of the joystick when you want joystick
 * information.
 * The state can be one of SDL_QUERY, SDL_ENABLE or SDL_IGNORE.
 */
extern DECLSPEC int SDL_JoystickEventState(int state);
```

(**SDL_JoystickGetAxis** joystick axis)

```
/*
 * Get the current state of an axis control on a joystick
 * The state is a value ranging from -32768 to 32767.
 * The axis indices start at index 0.
 */
extern DECLSPEC Sint16 SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
```

(**SDL_JoystickGetHat** joystick hat)

```
/*
 * Get the current state of a POV hat on a joystick
 * The return value is one of the following positions:
 */
```

(**SDL_JoystickGetBall** joystick ball dx dy)

```
/*
 * Get the ball axis change since the last poll
 * This returns 0, or -1 if you passed it invalid parameters.
 * The ball indices start at index 0.
 */
extern DECLSPEC int SDL_JoystickGetBall(SDL_Joystick *joystick, int ball, int *dx, int
```

(**SDL_JoystickGetButton** joystick button)

```
/*
 * Get the current state of a button on a joystick
 * The button indices start at index 0.
 */
extern DECLSPEC Uint8 SDL_JoystickGetButton(SDL_Joystick *joystick, int button);
```

(SDL_JoystickClose joystick**)**

```
/*
 * Close a joystick previously opened with SDL_JoystickOpen()
 */
extern DECLSPEC void SDL_JoystickClose(SDL_Joystick *joystick);
```

sdl/SDL_keyboard.lsh

(SDL_EnableUNICODE enable**)**

```
 * Enable/Disable UNICODE translation of keyboard input.
 * This translation has some overhead, so translation defaults off.
 * If 'enable' is 1, translation is enabled.
 * If 'enable' is 0, translation is disabled.
 * If 'enable' is -1, the translation state is not changed.
 * It returns the previous state of keyboard translation.
 */
extern DECLSPEC int SDL_EnableUNICODE(int enable);
```

(SDL_EnableKeyRepeat delay interval**)**

```
/*
 * Enable/Disable keyboard repeat.  Keyboard repeat defaults to off.
 * 'delay' is the initial delay in ms between the time when a key is
 * pressed, and keyboard repeat begins.
 * 'interval' is the time in ms between keyboard repeat events.
 */
/*
 * If 'delay' is set to 0, keyboard repeat is disabled.
 */
extern DECLSPEC int SDL_EnableKeyRepeat(int delay, int interval);
```

(SDL_GetKeyState numkeys**)**

```
/*
 * Get a snapshot of the current state of the keyboard.
 * Returns an array of keystates, indexed by the SDLK_* syms.
 * Used:
 *        Uint8 *keystate = SDL_GetKeyState(NULL);
 *       if ( keystate[SDLK_RETURN] ) ... <RETURN> is pressed.
 */
extern DECLSPEC Uint8 * SDL_GetKeyState(int *numkeys);
```

(SDL_GetModState)

```
/*
 * Get the current key modifier state
 */
extern DECLSPEC SDLMod SDL_GetModState(void);
```

(SDL_SetModState `modstate`**)**

```
/*
 * Set the current key modifier state
 * This does not change the keyboard state, only the key modifier flags.
 */
extern DECLSPEC void SDL_SetModState(SDLMod modstate);
```

(SDL_GetKeyName `key`**)**

```
/*
 * Get the name of an SDL virtual keysym
 */
extern DECLSPEC char * SDL_GetKeyName(SDLKey key);
```

sdl/SDL_keysym.lsh

SDL_Keysym

each keyboard symbol is associated with a constant whose symbolic name represents the key, and value represents the key code (generally the ASCII code of the character). Key symbols are of the form SDLK_0, SDLK_A, SDLK_b, SDLK_SPACE, etc.... Look into sdl/SDL_keysym.lsh for a complete list.

sdl/SDL_mouse.lsh

(SDL_GetMouseState `x y`**)**

```
/*
 * Retrieve the current state of the mouse.
 * The current button state is returned as a button bitmask, which can
 * be tested using the SDL_BUTTON(X) macros, and x and y are set to the
 * current mouse cursor position.  You can pass NULL for either x or y.
 */
extern DECLSPEC Uint8 SDL_GetMouseState(int *x, int *y);
```

(SDL_GetRelativeMouseState `x y`**)**

```
/*
 * Retrieve the current state of the mouse.
 * The current button state is returned as a button bitmask, which can
 * be tested using the SDL_BUTTON(X) macros, and x and y are set to the
 * mouse deltas since the last call to SDL_GetRelativeMouseState().
 */
extern DECLSPEC Uint8 SDL_GetRelativeMouseState(int *x, int *y);
```

(SDL_WarpMouse x y**)**

```
/*
 * Set the position of the mouse cursor (generates a mouse motion event)
 */
extern DECLSPEC void SDL_WarpMouse(Uint16 x, Uint16 y);
```

(SDL_CreateCursor data mask w h hot_x hot_y**)**

```
/*
 * Create a cursor using the specified data and mask (in MSB format).
 * The cursor width must be a multiple of 8 bits.
 *
 * The cursor is created in black and white according to the following:
 * data  mask    resulting pixel on screen
 *  0     1       White
 *  1     1       Black
 *  0     0       Transparent
 *  1     0       Inverted color if possible, black if not.
 *
 * Cursors created with this function must be freed with SDL_FreeCursor().
 */
extern DECLSPEC SDL_Cursor *SDL_CreateCursor
              (Uint8 *data, Uint8 *mask, int w, int h, int hot_x, int hot_y);
```

(SDL_SetCursor cursor**)**

```
/*
 * Set the currently active cursor to the specified one.
 * If the cursor is currently visible, the change will be immediately
 * represented on the display.
 */
extern DECLSPEC void SDL_SetCursor(SDL_Cursor *cursor);
```

(SDL_GetCursor)

```
/*
 * Returns the currently active cursor.
 */
extern DECLSPEC SDL_Cursor * SDL_GetCursor(void);
```

(SDL_FreeCursor cursor**)**

```
/*
 * Deallocates a cursor created with SDL_CreateCursor().
 */
extern DECLSPEC void SDL_FreeCursor(SDL_Cursor *cursor);
```

(SDL_ShowCursor toggle**)**

```
/*
 * Toggle whether or not the cursor is shown on the screen.
 * The cursor start off displayed, but can be turned off.
 * SDL_ShowCursor() returns 1 if the cursor was being displayed
 * before the call, or 0 if it was not.  You can query the current
 * state by passing a 'toggle' value of -1.
 */
extern DECLSPEC int SDL_ShowCursor(int toggle);
```

sdl/SDL_mutex.lsh

(SDL_CreateMutex)

```
/* Create a mutex, initialized unlocked */
extern DECLSPEC SDL_mutex * SDL_CreateMutex(void);
```

(SDL_mutexP mutex**)**

```
/* Lock the mutex  (Returns 0, or -1 on error) */
#define SDL_LockMutex(m)       SDL_mutexP(m)
extern DECLSPEC int SDL_mutexP(SDL_mutex *mutex);
```

(SDL_mutexV mutex**)**

```
/* Unlock the mutex  (Returns 0, or -1 on error) */
#define SDL_UnlockMutex(m)       SDL_mutexV(m)
extern DECLSPEC int SDL_mutexV(SDL_mutex *mutex);
```

(SDL_DestroyMutex mutex**)**

```
/* Destroy a mutex */
extern DECLSPEC void SDL_DestroyMutex(SDL_mutex *mutex);
```

(SDL_CreateSemaphore initial_value**)**

```
/* Create a semaphore, initialized with value, returns NULL on failure. */
extern DECLSPEC SDL_sem * SDL_CreateSemaphore(Uint32 initial_value);
```

(SDL_DestroySemaphore sem**)**

```
/* Destroy a semaphore */
extern DECLSPEC void SDL_DestroySemaphore(SDL_sem *sem);
```

(SDL_SemWait sem**)**

```
/* This function suspends the calling thread until the semaphore pointed
 * to by sem has a positive count. It then atomically decreases the semaphore
 * count.
 */
extern DECLSPEC int SDL_SemWait(SDL_sem *sem);
```

(SDL_SemTryWait sem**)**

```
/* Non-blocking variant of SDL_SemWait(), returns 0 if the wait succeeds,
   SDL_MUTEX_TIMEDOUT if the wait would block, and -1 on error.
*/
extern DECLSPEC int SDL_SemTryWait(SDL_sem *sem);
```

(SDL_SemWaitTimeout sem ms**)**

```
/* Variant of SDL_SemWait() with a timeout in milliseconds, returns 0 if
   the wait succeeds, SDL_MUTEX_TIMEDOUT if the wait does not succeed in
   the allotted time, and -1 on error.
   On some platforms this function is implemented by looping with a delay
   of 1 ms, and so should be avoided if possible.
*/
extern DECLSPEC int SDL_SemWaitTimeout(SDL_sem *sem, Uint32 ms);
```

(SDL_SemPost sem**)**

```
/* Atomically increases the semaphore's count (not blocking), returns 0,
   or -1 on error.
 */
extern DECLSPEC int SDL_SemPost(SDL_sem *sem);
```

(SDL_CreateCond)

```
/* Create a condition variable */
extern DECLSPEC SDL_cond * SDL_CreateCond(void);
```

(SDL_DestroyCond cnd**)**

```
/* Destroy a condition variable */
extern DECLSPEC void SDL_DestroyCond(SDL_cond *cond);
```

(SDL_CondSignal cnd**)**

```
/* Restart one of the threads that are waiting on the condition variable,
   returns 0 or -1 on error.
 */
extern DECLSPEC int SDL_CondSignal(SDL_cond *cond);
```

(SDL_CondBroadcast cnd**)**

```
/* Restart all threads that are waiting on the condition variable,
   returns 0 or -1 on error.
 */
extern DECLSPEC int SDL_CondBroadcast(SDL_cond *cond);
```

(SDL_CondWait cnd mut**)**

```
/* Wait on the condition variable, unlocking the provided mutex.
   The mutex must be locked before entering this function!
   Returns 0 when it is signaled, or -1 on error.
 */
extern DECLSPEC int SDL_CondWait(SDL_cond *cond, SDL_mutex *mut);
```

(SDL_CondWaitTimeout cnd mutex ms**)**

```
/* Waits for at most 'ms' milliseconds, and returns 0 if the condition
   variable is signaled, SDL_MUTEX_TIMEDOUT if the condition is not
   signaled in the allotted time, and -1 on error.
   On some platforms this function is implemented by looping with a delay
   of 1 ms, and so should be avoided if possible.
*/
extern DECLSPEC int SDL_CondWaitTimeout(SDL_cond *cond, SDL_mutex *mutex, Uint32 ms);
```

sdl/SDL_rwops.lsh

SDL_RWops
Functions to create SDL_RWops structures from various data sources
(SDL_RWFromFile `file mode`**)**

```
extern DECLSPEC SDL_RWops * SDL_RWFromFile(const char *file, const char *mode);
```

(SDL_RWFromFP `fp autoclose`**)**

```
extern DECLSPEC SDL_RWops * SDL_RWFromFP(FILE *fp, int autoclose);
```

(SDL_RWFromMem `mem size`**)**

```
extern DECLSPEC SDL_RWops * SDL_RWFromMem(void *mem, int size);
```

(SDL_AllocRW)

```
extern DECLSPEC SDL_RWops * SDL_AllocRW(void);
```

(SDL_FreeRW `area`**)**

```
extern DECLSPEC void SDL_FreeRW(SDL_RWops *area);
```

sdl/SDL_syswm.lsh

(SDL_GetWMInfo `info`**)**

```
/*
 * This function gives you custom hooks into the window manager information.
 * It fills the structure pointed to by 'info' with custom information and
 * returns 1 if the function is implemented.  If it's not implemented, or
 * the version member of the 'info' structure is invalid, it returns 0.
 */
extern DECLSPEC int SDL_GetWMInfo(SDL_SysWMinfo *info);
```

sdl/SDL_thread.lsh

(SDL_CreateThread `fn data`**)**

```
/* Create a thread */
extern DECLSPEC SDL_Thread * SDL_CreateThread(int (*fn)(void *), void *data);
```

(SDL_ThreadID)

```
/* Get the 32-bit thread identifier for the current thread */
extern DECLSPEC Uint32 SDL_ThreadID(void);
```

(SDL_GetThreadID `thread`**)**

```
/* Get the 32-bit thread identifier for the specified thread,
   equivalent to SDL_ThreadID() if the specified thread is NULL.
 */
extern DECLSPEC Uint32 SDL_GetThreadID(SDL_Thread *thread);
```

(SDL_WaitThread `thread status`**)**

```
/* Wait for a thread to finish.
   The return code for the thread function is placed in the area
   pointed to by 'status', if 'status' is not NULL.
 */
extern DECLSPEC void SDL_WaitThread(SDL_Thread *thread, int *status);
```

(SDL_KillThread `thread`**)**

```
/* Forcefully kill a thread without worrying about its state */
extern DECLSPEC void SDL_KillThread(SDL_Thread *thread);
```

sdl/SDL_timer.lsh

(SDL_GetTicks)

```
/* Get the number of milliseconds since the SDL library initialization.
 * Note that this value wraps if the program runs for more than ~49 days.
 */
extern DECLSPEC Uint32 SDL_GetTicks(void);
```

(SDL_Delay `ms`**)**

```
/* Wait a specified number of milliseconds before returning */
extern DECLSPEC void SDL_Delay(Uint32 ms);
```

(SDL_SetTimer `interval callback`**)**

```
/* Function prototype for the timer callback function */
typedef Uint32 (*SDL_TimerCallback)(Uint32 interval);
/* Set a callback to run after the specified number of milliseconds has
 * elapsed. The callback function is passed the current timer interval
 * and returns the next timer interval.  If the returned value is the
 * same as the one passed in, the periodic alarm continues, otherwise a
 * new alarm is scheduled.  If the callback returns 0, the periodic alarm
 * is cancelled.
 *
 * To cancel a currently running timer, call SDL_SetTimer(0, NULL);
 *
 * The timer callback function may run in a different thread than your
 * main code, and so shouldn't call any functions from within itself.
 *
 * The maximum resolution of this timer is 10 ms, which means that if
 * you request a 16 ms timer, your callback will run approximately 20 ms
 * later on an unloaded system.  If you wanted to set a flag signaling
 * a frame update at 30 frames per second (every 33 ms), you might set a
 * timer for 30 ms:
 *   SDL_SetTimer((33/10)*10, flag_update);
 *
 * If you use this function, you need to pass SDL_INIT_TIMER to SDL_Init().
 *
 * Under UNIX, you should not use raise or use SIGALRM and this function
 * in the same program, as it is implemented using setitimer().  You also
 * should not use this function in multi-threaded applications as signals
 * to multi-threaded apps have undefined behavior in some implementations.
 */
extern DECLSPEC int SDL_SetTimer(Uint32 interval, SDL_TimerCallback callback);
```

(SDL_AddTimer `interval callback param`**)**

```
/* Add a new timer to the pool of timers already running.
   Returns a timer ID, or NULL when an error occurs.
 */
extern DECLSPEC SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback, void *pa
```

(SDL_RemoveTimer `tim`**)**

```
/* Remove one of the multiple timers knowing its ID.
 * Returns a boolean value indicating success.
 */
extern DECLSPEC SDL_bool SDL_RemoveTimer(SDL_TimerID t);
```

sdl/SDL_version.lsh

(SDL_Linked_Version)

```
/* This function gets the version of the dynamically linked SDL library.
   it should NOT be used to fill a version structure, instead you should
   use the SDL_Version() macro.
 */
extern DECLSPEC const SDL_version * SDL_Linked_Version(void);
```

sdl/SDL_video.lsh

(SDL_MUSTLOCK surface**)**

```
/* Evaluates to true if the surface needs to be locked before access */
#define SDL_MUSTLOCK(surface)       \\
  (surface->offset ||               \\
  ((surface->flags & (SDL_HWSURFACE|SDL_ASYNCBLIT|SDL_RLEACCEL)) != 0))
```

(SDL_VideoInit driver_name flags**)**

```
/* These functions are used internally, and should not be used unless you
 * have a specific need to specify the video driver you want to use.
 * You should normally use SDL_Init() or SDL_InitSubSystem().
 *
 * SDL_VideoInit() initializes the video subsystem -- sets up a connection
 * to the window manager, etc, and determines the current video mode and
 * pixel format, but does not initialize a window or graphics mode.
 * Note that event handling is activated by this routine.
 *
 * If you use both sound and video in your application, you need to call
 * SDL_Init() before opening the sound device, otherwise under Win32 DirectX,
 * you won't be able to set full-screen display modes.
 */
extern DECLSPEC int SDL_VideoInit(const char *driver_name, Uint32 flags);
```

(SDL_VideoQuit)

```
extern DECLSPEC void SDL_VideoQuit(void);
```

(SDL_VideoDriverName namebuf maxlen**)**

```
/* This function fills the given character buffer with the name of the
```

```
 * video driver, and returns a pointer to it if the video driver has
 * been initialized.  It returns NULL if no driver has been initialized.
 */
extern DECLSPEC char *SDL_VideoDriverName(char *namebuf, int maxlen);
```

(SDL_GetVideoSurface)

```
/*
 * This function returns a pointer to the current display surface.
 * If SDL is doing format conversion on the display surface, this
 * function returns the publicly visible surface, not the real video
 * surface.
 */
extern DECLSPEC SDL_Surface * SDL_GetVideoSurface(void);
```

(SDL_GetVideoInfo)

```
/*
 * This function returns a read-only pointer to information about the
 * video hardware.  If this is called before SDL_SetVideoMode(), the 'vfmt'
 * member of the returned structure will contain the pixel format of the
 * "best" video mode.
 */
extern DECLSPEC const SDL_VideoInfo * SDL_GetVideoInfo(void);
```

(SDL_VideoModeOK `width height bpp flags`**)**

```
/*
 * Check to see if a particular video mode is supported.
 * It returns 0 if the requested mode is not supported under any bit depth,
 * or returns the bits-per-pixel of the closest available mode with the
 * given width and height.  If this bits-per-pixel is different from the
 * one used when setting the video mode, SDL_SetVideoMode() will succeed,
 * but will emulate the requested bits-per-pixel with a shadow surface.
 *
 * The arguments to SDL_VideoModeOK() are the same ones you would pass to
 * SDL_SetVideoMode()
 */
extern DECLSPEC int SDL_VideoModeOK(int width, int height, int bpp, Uint32 flags);
```

(SDL_ListModes `format flags`**)**

```
/*
```

```
 * Return a pointer to an array of available screen dimensions for the
 * given format and video flags, sorted largest to smallest.  Returns
 * NULL if there are no dimensions available for a particular format,
 * or (SDL_Rect **)-1 if any dimension is okay for the given format.
 *
 * If 'format' is NULL, the mode list will be for the format given
 * by SDL_GetVideoInfo()->vfmt
 */
extern DECLSPEC SDL_Rect ** SDL_ListModes(SDL_PixelFormat *format, Uint32 flags);
```

(SDL_SetVideoMode `width height bpp flags`**)**

```
/*
 * Set up a video mode with the specified width, height and bits-per-pixel.
 *
 * If 'bpp' is 0, it is treated as the current display bits per pixel.
 *
 * If SDL_ANYFORMAT is set in 'flags', the SDL library will try to set the
 * requested bits-per-pixel, but will return whatever video pixel format is
 * available.  The default is to emulate the requested pixel format if it
 * is not natively available.
 *
 * If SDL_HWSURFACE is set in 'flags', the video surface will be placed in
 * video memory, if possible, and you may have to call SDL_LockSurface()
 * in order to access the raw framebuffer.  Otherwise, the video surface
 * will be created in system memory.
 *
 * If SDL_ASYNCBLIT is set in 'flags', SDL will try to perform rectangle
 * updates asynchronously, but you must always lock before accessing pixels.
 * SDL will wait for updates to complete before returning from the lock.
 *
 * If SDL_HWPALETTE is set in 'flags', the SDL library will guarantee
 * that the colors set by SDL_SetColors() will be the colors you get.
 * Otherwise, in 8-bit mode, SDL_SetColors() may not be able to set all
 * of the colors exactly the way they are requested, and you should look
 * at the video surface structure to determine the actual palette.
 * If SDL cannot guarantee that the colors you request can be set,
 * i.e. if the colormap is shared, then the video surface may be created
 * under emulation in system memory, overriding the SDL_HWSURFACE flag.
 *
 * If SDL_FULLSCREEN is set in 'flags', the SDL library will try to set
 * a fullscreen video mode.  The default is to create a windowed mode
 * if the current graphics system has a window manager.
 * If the SDL library is able to set a fullscreen video mode, this flag
 * will be set in the surface that is returned.
```

```
 *
 * If SDL_DOUBLEBUF is set in 'flags', the SDL library will try to set up
 * two surfaces in video memory and swap between them when you call
 * SDL_Flip().  This is usually slower than the normal single-buffering
 * scheme, but prevents "tearing" artifacts caused by modifying video
 * memory while the monitor is refreshing.  It should only be used by
 * applications that redraw the entire screen on every update.
 *
 * If SDL_RESIZABLE is set in 'flags', the SDL library will allow the
 * window manager, if any, to resize the window at runtime.  When this
 * occurs, SDL will send a SDL_VIDEORESIZE event to you application,
 * and you must respond to the event by re-calling SDL_SetVideoMode()
 * with the requested size (or another size that suits the application).
 *
 * If SDL_NOFRAME is set in 'flags', the SDL library will create a window
 * without any title bar or frame decoration.  Fullscreen video modes have
 * this flag set automatically.
 *
 * This function returns the video framebuffer surface, or NULL if it fails.
 *
 * If you rely on functionality provided by certain video flags, check the
 * flags of the returned surface to make sure that functionality is available.
 * SDL will fall back to reduced functionality if the exact flags you wanted
 * are not available.
 */
extern DECLSPEC SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
```

(SDL_UpdateRects screen numrects rects**)**

```
/*
 * Makes sure the given list of rectangles is updated on the given screen.
 * If 'x', 'y', 'w' and 'h' are all 0, SDL_UpdateRect will update the entire
 * screen.
 * These functions should not be called while 'screen' is locked.
 */
extern DECLSPEC void SDL_UpdateRects(SDL_Surface *screen, int numrects, SDL_Rect *rects);
```

(SDL_UpdateRect screen x y w h**)**

```
extern DECLSPEC void SDL_UpdateRect(SDL_Surface *screen, Sint32 x, Sint32 y, Uint32 w, Uint32 h);
```

(SDL_Flip screen**)**

```
/*
```

```
 * On hardware that supports double-buffering, this function sets up a flip
 * and returns.  The hardware will wait for vertical retrace, and then swap
 * video buffers before the next video surface blit or lock will return.
 * On hardware that doesn not support double-buffering, this is equivalent
 * to calling SDL_UpdateRect(screen, 0, 0, 0, 0);
 * The SDL_DOUBLEBUF flag must have been passed to SDL_SetVideoMode() when
 * setting the video mode for this function to perform hardware flipping.
 * This function returns 0 if successful, or -1 if there was an error.
 */
extern DECLSPEC int SDL_Flip(SDL_Surface *screen);
```

(SDL_SetGamma red green blue**)**

```
/*
 * Set the gamma correction for each of the color channels.
 * The gamma values range (approximately) between 0.1 and 10.0
 *
 * If this function isn't supported directly by the hardware, it will
 * be emulated using gamma ramps, if available.  If successful, this
 * function returns 0, otherwise it returns -1.
 */
extern DECLSPEC int SDL_SetGamma(float red, float green, float blue);
```

(SDL_SetGammaRamp red green blue**)**

```
/*
 * Set the gamma translation table for the red, green, and blue channels
 * of the video hardware.  Each table is an array of 256 16-bit quantities,
 * representing a mapping between the input and output for that channel.
 * The input is the index into the array, and the output is the 16-bit
 * gamma value at that index, scaled to the output color precision.
 *
 * You may pass NULL for any of the channels to leave it unchanged.
 * If the call succeeds, it will return 0.  If the display driver or
 * hardware does not support gamma translation, or otherwise fails,
 * this function will return -1.
 */
extern DECLSPEC int SDL_SetGammaRamp(Uint16 *red, Uint16 *green, Uint16 *blue);
```

(SDL_GetGammaRamp red green blue**)**

```
/*
 * Retrieve the current values of the gamma translation tables.
 *
```

```
 * You must pass in valid pointers to arrays of 256 16-bit quantities.
 * Any of the pointers may be NULL to ignore that channel.
 * If the call succeeds, it will return 0.  If the display driver or
 * hardware does not support gamma translation, or otherwise fails,
 * this function will return -1.
 */
extern DECLSPEC int SDL_GetGammaRamp(Uint16 *red, Uint16 *green, Uint16 *blue);
```

(**SDL_SetColors** surface colors firstcolor ncolors)

```
/*
 * Sets a portion of the colormap for the given 8-bit surface.  If 'surface'
 * is not a palettized surface, this function does nothing, returning 0.
 * If all of the colors were set as passed to SDL_SetColors(), it will
 * return 1.  If not all the color entries were set exactly as given,
 * it will return 0, and you should look at the surface palette to
 * determine the actual color palette.
 *
 * When 'surface' is the surface associated with the current display, the
 * display colormap will be updated with the requested colors.  If
 * SDL_HWPALETTE was set in SDL_SetVideoMode() flags, SDL_SetColors()
 * will always return 1, and the palette is guaranteed to be set the way
 * you desire, even if the window colormap has to be warped or run under
 * emulation.
 */
extern DECLSPEC int SDL_SetColors(SDL_Surface *surface, SDL_Color *colors, int firstcolor, int nc
```

(**SDL_SetPalette** surface flags colors firstcolor ncolors)

```
/*
 * Sets a portion of the colormap for a given 8-bit surface.
 * 'flags' is one or both of:
 * SDL_LOGPAL  -- set logical palette, which controls how blits are mapped
 *                to/from the surface,
 * SDL_PHYSPAL -- set physical palette, which controls how pixels look on
 *                the screen
 * Only screens have physical palettes. Separate change of physical/logical
 * palettes is only possible if the screen has SDL_HWPALETTE set.
 *
 * The return value is 1 if all colours could be set as requested, and 0
 * otherwise.
 *
 * SDL_SetColors() is equivalent to calling this function with
 *     flags = (SDL_LOGPAL|SDL_PHYSPAL).
 */
```

```
extern DECLSPEC int SDL_SetPalette(SDL_Surface *surface, int flags,
                                   SDL_Color *colors, int firstcolor,
                                   int ncolors);
```

(**SDL_MapRGB** format r g b)

```
/*
 * Maps an RGB triple to an opaque pixel value for a given pixel format
 */
extern DECLSPEC Uint32 SDL_MapRGB(SDL_PixelFormat *format, Uint8 r, Uint8 g, Uint8 b);
```

(**SDL_MapRGBA** format r g b a)

```
/*
 * Maps an RGBA quadruple to a pixel value for a given pixel format
 */
extern DECLSPEC Uint32 SDL_MapRGBA(SDL_PixelFormat *format, Uint8 r, Uint8 g, Uint8 b,
```

(**SDL_GetRGB** format r g b a)

```
/*
 * Maps a pixel value into the RGB components for a given pixel format
 */
extern DECLSPEC void SDL_GetRGB(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8 *g
```

(**SDL_GetRGBA** format r g b a)

```
/*
 * Maps a pixel value into the RGBA components for a given pixel format
 */
extern DECLSPEC void SDL_GetRGBA(Uint32 pixel, SDL_PixelFormat *fmt,Uint8 *r, Uint8 *g
```

(**SDL_CreateRGBSurface** flags width height depth rmask gmask bmask
amask)

```
/*
 * Allocate and free an RGB surface (must be called after SDL_SetVideoMode)
 * If the depth is 4 or 8 bits, an empty palette is allocated for the surface.
 * If the depth is greater than 8 bits, the pixel format is set using the
 * flags '[RGB]mask'.
 * If the function runs out of memory, it will return NULL.
 *
 * The 'flags' tell what kind of surface to create.
```

```
 * SDL_SWSURFACE means that the surface should be created in system memory.
 * SDL_HWSURFACE means that the surface should be created in video memory,
 * with the same format as the display surface.  This is useful for surfaces
 * that will not change much, to take advantage of hardware acceleration
 * when being blitted to the display surface.
 * SDL_ASYNCBLIT means that SDL will try to perform asynchronous blits with
 * this surface, but you must always lock it before accessing the pixels.
 * SDL will wait for current blits to finish before returning from the lock.
 * SDL_SRCCOLORKEY indicates that the surface will be used for colorkey blits.
 * If the hardware supports acceleration of colorkey blits between
 * two surfaces in video memory, SDL will try to place the surface in
 * video memory. If this isn't possible or if there is no hardware
 * acceleration available, the surface will be placed in system memory.
 * SDL_SRCALPHA means that the surface will be used for alpha blits and
 * if the hardware supports hardware acceleration of alpha blits between
 * two surfaces in video memory, to place the surface in video memory
 * if possible, otherwise it will be placed in system memory.
 * If the surface is created in video memory, blits will be _much_ faster,
 * but the surface format must be identical to the video surface format,
 * and the only way to access the pixels member of the surface is to use
 * the SDL_LockSurface() and SDL_UnlockSurface() calls.
 * If the requested surface actually resides in video memory, SDL_HWSURFACE
 * will be set in the flags member of the returned surface.  If for some
 * reason the surface could not be placed in video memory, it will not have
 * the SDL_HWSURFACE flag set, and will be created in system memory instead.
 */
#define SDL_AllocSurface    SDL_CreateRGBSurface
extern DECLSPEC SDL_Surface *SDL_CreateRGBSurface
                     (Uint32 flags, int width, int height, int depth,
                     Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
extern DECLSPEC SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height, int depth,
```

(**SDL_CreateRGBSurfaceFrom** pixels width height depth pitch rmask gmask bmask amask)

```
extern DECLSPEC SDL_Surface *SDL_CreateRGBSurfaceFrom
                 (void *pixels, int width, int height, int depth, int pitch,
                  Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
```

(**SDL_FreeSurface** surface)

```
extern DECLSPEC void SDL_FreeSurface(SDL_Surface *surface);
```

(**SDL_LockSurface** surface)

```
/*
 * SDL_LockSurface() sets up a surface for directly accessing the pixels.
 * Between calls to SDL_LockSurface()/SDL_UnlockSurface(), you can write
 * to and read from 'surface->pixels', using the pixel format stored in
 * 'surface->format'.  Once you are done accessing the surface, you should
 * use SDL_UnlockSurface() to release it.
 *
 * Not all surfaces require locking.  If SDL_MUSTLOCK(surface) evaluates
 * to 0, then you can read and write to the surface at any time, and the
 * pixel format of the surface will not change.  In particular, if the
 * SDL_HWSURFACE flag is not given when calling SDL_SetVideoMode(), you
 * will not need to lock the display surface before accessing it.
 *
 * No operating system or library calls should be made between lock/unlock
 * pairs, as critical system locks may be held during this time.
 *
 * SDL_LockSurface() returns 0, or -1 if the surface couldn't be locked.
 */
extern DECLSPEC int SDL_LockSurface(SDL_Surface *surface);
```

(**SDL_UnlockSurface** surface)

```
extern DECLSPEC void SDL_UnlockSurface(SDL_Surface *surface);
```

(**SDL_LoadBMP_RW** src freesrc)

```
/*
 * Load a surface from a seekable SDL data source (memory or file.)
 * If 'freesrc' is non-zero, the source will be closed after being read.
 * Returns the new surface, or NULL if there was an error.
 * The new surface should be freed with SDL_FreeSurface().
 */
extern DECLSPEC SDL_Surface * SDL_LoadBMP_RW(SDL_RWops *src, int freesrc);
```

(**SDL_SaveBMP_RW** surface dst freedst)

```
/*
 * Save a surface to a seekable SDL data source (memory or file.)
 * If 'freedst' is non-zero, the source will be closed after being written.
 * Returns 0 if successful or -1 if there was an error.
 */
extern DECLSPEC int SDL_SaveBMP_RW
             (SDL_Surface *surface, SDL_RWops *dst, int freedst);
```

(SDL_SetColorKey surface flag key**)**

```
/*
 * Sets the color key (transparent pixel) in a blittable surface.
 * If 'flag' is SDL_SRCCOLORKEY (optionally OR'd with SDL_RLEACCEL),
 * 'key' will be the transparent pixel in the source image of a blit.
 * SDL_RLEACCEL requests RLE acceleration for the surface if present,
 * and removes RLE acceleration if absent.
 * If 'flag' is 0, this function clears any current color key.
 * This function returns 0, or -1 if there was an error.
 */
extern DECLSPEC int SDL_SetColorKey
                     (SDL_Surface *surface, Uint32 flag, Uint32 key);
extern DECLSPEC int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

(SDL_SetAlpha surface flag alpha**)**

```
/*
 * This function sets the alpha value for the entire surface, as opposed to
 * using the alpha component of each pixel. This value measures the range
 * of transparency of the surface, 0 being completely transparent to 255
 * being completely opaque. An 'alpha' value of 255 causes blits to be
 * opaque, the source pixels copied to the destination (the default). Note
 * that per-surface alpha can be combined with colorkey transparency.
 *
 * If 'flag' is 0, alpha blending is disabled for the surface.
 * If 'flag' is SDL_SRCALPHA, alpha blending is enabled for the surface.
 * OR:ing the flag with SDL_RLEACCEL requests RLE acceleration for the
 * surface; if SDL_RLEACCEL is not specified, the RLE accel will be removed.
 */
extern DECLSPEC int SDL_SetAlpha(SDL_Surface *surface, Uint32 flag, Uint8 alpha);
```

(SDL_SetClipRect surface rect**)**

```
/*
 * Sets the clipping rectangle for the destination surface in a blit.
 *
 * If the clip rectangle is NULL, clipping will be disabled.
 * If the clip rectangle doesn't intersect the surface, the function will
 * return SDL_FALSE and blits will be completely clipped.  Otherwise the
 * function returns SDL_TRUE and blits to the surface will be clipped to
 * the intersection of the surface area and the clipping rectangle.
 *
 * Note that blits are automatically clipped to the edges of the source
```

```
 * and destination surfaces.
 */
extern DECLSPEC SDL_bool SDL_SetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

(SDL_GetClipRect surface rect**)**

```
/*
 * Gets the clipping rectangle for the destination surface in a blit.
 * 'rect' must be a pointer to a valid rectangle which will be filled
 * with the correct values.
 */
extern DECLSPEC void SDL_GetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

(SDL_ConvertSurface src fmt flags**)**

```
/*
 * Creates a new surface of the specified format, and then copies and maps
 * the given surface to it so the blit of the converted surface will be as
 * fast as possible.  If this function fails, it returns NULL.
 *
 * The 'flags' parameter is passed to SDL_CreateRGBSurface() and has those
 * semantics.  You can also pass SDL_RLEACCEL in the flags parameter and
 * SDL will try to RLE accelerate colorkey and alpha blits in the resulting
 * surface.
 *
 * This function is used internally by SDL_DisplayFormat().
 */
extern DECLSPEC SDL_Surface *SDL_ConvertSurface
                    (SDL_Surface *src, SDL_PixelFormat *fmt, Uint32 flags);
```

(SDL_BlitSurface src srcrect dst dstrect**)**

```
/*
 * This performs a fast blit from the source surface to the destination
 * surface.  It assumes that the source and destination rectangles are
 * the same size.  If either 'srcrect' or 'dstrect' are NULL, the entire
 * surface (src or dst) is copied.  The final blit rectangles are saved
 * in 'srcrect' and 'dstrect' after all clipping is performed.
 * If the blit is successful, it returns 0, otherwise it returns -1.
 *
 * The blit function should not be called on a locked surface.
 *
 * The blit semantics for surfaces with and without alpha and colorkey
 * are defined as follows:
```

```
*
* RGBA->RGB:
*     SDL_SRCALPHA set:
*        alpha-blend (using alpha-channel).
*        SDL_SRCCOLORKEY ignored.
*     SDL_SRCALPHA not set:
*        copy RGB.
*        if SDL_SRCCOLORKEY set, only copy the pixels matching the
*        RGB values of the source colour key, ignoring alpha in the
*        comparison.
*
* RGB->RGBA:
*     SDL_SRCALPHA set:
*        alpha-blend (using the source per-surface alpha value);
*        set destination alpha to opaque.
*     SDL_SRCALPHA not set:
*        copy RGB, set destination alpha to opaque.
*     both:
*        if SDL_SRCCOLORKEY set, only copy the pixels matching the
*        source colour key.
*
* RGBA->RGBA:
*     SDL_SRCALPHA set:
*        alpha-blend (using the source alpha channel) the RGB values;
*        leave destination alpha untouched. [Note: is this correct?]
*        SDL_SRCCOLORKEY ignored.
*     SDL_SRCALPHA not set:
*        copy all of RGBA to the destination.
*        if SDL_SRCCOLORKEY set, only copy the pixels matching the
*        RGB values of the source colour key, ignoring alpha in the
*        comparison.
*
* RGB->RGB:
*     SDL_SRCALPHA set:
*        alpha-blend (using the source per-surface alpha value).
*     SDL_SRCALPHA not set:
*        copy RGB.
*     both:
*        if SDL_SRCCOLORKEY set, only copy the pixels matching the
*        source colour key.
*
* If either of the surfaces were in video memory, and the blit returns -2,
* the video memory was lost, so it should be reloaded with artwork and
* re-blitted:
      while ( SDL_BlitSurface(image, imgrect, screen, dstrect) == -2 ) {
             while ( SDL_LockSurface(image) < 0 )
```

```
                Sleep(10);
            -- Write image pixels to image->pixels --
            SDL_UnlockSurface(image);
        }
 * This happens under DirectX 5.0 when the system switches away from your
 * fullscreen application.  The lock will also fail until you have access
 * to the video memory again.
 */
/* You should call SDL_BlitSurface() unless you know exactly how SDL
   blitting works internally and how to use the other blit functions.
*/
#define SDL_BlitSurface SDL_UpperBlit
```

(**SDL_UpperBlit** src srcrect dst dstrect)

```
/* This is the public blit function, SDL_BlitSurface(), and it performs
   rectangle validation and clipping before passing it to SDL_LowerBlit()
*/
extern DECLSPEC int SDL_UpperBlit
                    (SDL_Surface *src, SDL_Rect *srcrect,
                     SDL_Surface *dst, SDL_Rect *dstrect);
```

(**SDL_LowerBlit** src srcrect dst dstrect)

```
/* This is a semi-private blit function and it performs low-level surface
   blitting only.
*/
extern DECLSPEC int SDL_LowerBlit
                    (SDL_Surface *src, SDL_Rect *srcrect,
                     SDL_Surface *dst, SDL_Rect *dstrect);
```

(**SDL_FillRect** dst dstrect color)

```
/*
 * This function performs a fast fill of the given rectangle with 'color'
 * The given rectangle is clipped to the destination surface clip area
 * and the final fill rectangle is saved in the passed in pointer.
 * If 'dstrect' is NULL, the whole surface will be filled with 'color'
 * The color should be a pixel of the format used by the surface, and
 * can be generated by the SDL_MapRGB() function.
 * This function returns 0 on success, or -1 on error.
 */
extern DECLSPEC int SDL_FillRect
             (SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);
```

(**SDL_DisplayFormat** surface)

```
/*
 * This function takes a surface and copies it to a new surface of the
 * pixel format and colors of the video framebuffer, suitable for fast
 * blitting onto the display surface.  It calls SDL_ConvertSurface()
 *
 * If you want to take advantage of hardware colorkey or alpha blit
 * acceleration, you should set the colorkey and alpha value before
 * calling this function.
 *
 * If the conversion fails or runs out of memory, it returns NULL
 */
extern DECLSPEC SDL_Surface * SDL_DisplayFormat(SDL_Surface *surface);
```

(**SDL_DisplayFormatAlpha** surface)

```
/*
 * This function takes a surface and copies it to a new surface of the
 * pixel format and colors of the video framebuffer (if possible),
 * suitable for fast alpha blitting onto the display surface.
 * The new surface will always have an alpha channel.
 *
 * If you want to take advantage of hardware colorkey or alpha blit
 * acceleration, you should set the colorkey and alpha value before
 * calling this function.
 *
 * If the conversion fails or runs out of memory, it returns NULL
 */
extern DECLSPEC SDL_Surface * SDL_DisplayFormatAlpha(SDL_Surface *surface);
```

(**SDL_CreateYUVOverlay** width height format display)

```
/* This function creates a video output overlay
   Calling the returned surface an overlay is something of a misnomer because
   the contents of the display surface underneath the area where the overlay
   is shown is undefined - it may be overwritten with the converted YUV data.
*/
extern DECLSPEC SDL_Overlay *SDL_CreateYUVOverlay(int width, int height,
                             Uint32 format, SDL_Surface *display);
```

(**SDL_LockYUVOverlay** overlay)

```
/* Lock an overlay for direct access, and unlock it when you are done */
extern DECLSPEC int SDL_LockYUVOverlay(SDL_Overlay *overlay);
```

(**SDL_UnlockYUVOverlay** overlay)

```
extern DECLSPEC void SDL_UnlockYUVOverlay(SDL_Overlay *overlay);
```

(**SDL_DisplayYUVOverlay** overlay dstrect)

```
/* Blit a video overlay to the display surface.
   The contents of the video surface underneath the blit destination are
   not defined.
   The width and height of the destination rectangle may be different from
   that of the overlay, but currently only 2x scaling is supported.
*/
extern DECLSPEC int SDL_DisplayYUVOverlay(SDL_Overlay *overlay, SDL_Rect *dstrect);
```

(**SDL_FreeYUVOverlay** overlay)

```
/* Free a video overlay */
extern DECLSPEC void SDL_FreeYUVOverlay(SDL_Overlay *overlay);
```

(**SDL_GL_LoadLibrary** path)

```
/*
 * Dynamically load a GL driver, if SDL is built with dynamic GL.
 *
 * SDL links normally with the OpenGL library on your system by default,
 * but you can compile it to dynamically load the GL driver at runtime.
 * If you do this, you need to retrieve all of the GL functions used in
 * your program from the dynamic library using SDL_GL_GetProcAddress().
 *
 * This is disabled in default builds of SDL.
 */
extern DECLSPEC int SDL_GL_LoadLibrary(const char *path);
```

(**SDL_GL_GetProcAddress** proc)

```
/*
 * Get the address of a GL function (for extension functions)
 */
extern DECLSPEC void *SDL_GL_GetProcAddress(const char* proc);
```

(**SDL_GL_SetAttribute** attr value)

```
/*
 * Set an attribute of the OpenGL subsystem before intialization.
 */
extern DECLSPEC int SDL_GL_SetAttribute(SDL_GLattr attr, int value);
```

(SDL_GL_GetAttribute attr value**)**

```
/*
 * Get an attribute of the OpenGL subsystem from the windowing
 * interface, such as glX. This is of course different from getting
 * the values from SDL's internal OpenGL subsystem, which only
 * stores the values you request before initialization.
 *
 * Developers should track the values they pass into SDL_GL_SetAttribute
 * themselves if they want to retrieve these values.
 */
extern DECLSPEC int SDL_GL_GetAttribute(SDL_GLattr attr, int* value);
```

(SDL_GL_SwapBuffers)

```
/*
 * Swap the OpenGL buffers, if double-buffering is supported.
 */
extern DECLSPEC void SDL_GL_SwapBuffers(void);
```

(SDL_WM_SetCaption title icon**)**

```
/*
 * Sets/Gets the title and icon text of the display window
 */
extern DECLSPEC void SDL_WM_SetCaption(const char *title, const char *icon);
```

(SDL_WM_GetCaption title icon**)**

```
extern DECLSPEC void SDL_WM_GetCaption(char **title, char **icon);
```

(SDL_WM_SetIcon icon mask**)**

```
/*
 * Sets the icon for the display window.
 * This function must be called before the first call to SDL_SetVideoMode().
 * It takes an icon surface, and a mask in MSB format.
 * If 'mask' is NULL, the entire icon surface will be used as the icon.
 */
extern DECLSPEC void SDL_WM_SetIcon(SDL_Surface *icon, Uint8 *mask);
```

(SDL_WM_IconifyWindow)

```
/*
 * This function iconifies the window, and returns 1 if it succeeded.
 * If the function succeeds, it generates an SDL_APPACTIVE loss event.
 * This function is a noop and returns 0 in non-windowed environments.
 */
extern DECLSPEC int SDL_WM_IconifyWindow(void);
```

(SDL_WM_ToggleFullScreen `surface`**)**

```
/*
 * Toggle fullscreen mode without changing the contents of the screen.
 * If the display surface does not require locking before accessing
 * the pixel information, then the memory pointers will not change.
 *
 * If this function was able to toggle fullscreen mode (change from
 * running in a window to fullscreen, or vice-versa), it will return 1.
 * If it is not implemented, or fails, it returns 0.
 *
 * The next call to SDL_SetVideoMode() will set the mode fullscreen
 * attribute based on the flags parameter - if SDL_FULLSCREEN is not
 * set, then the display will be windowed by default where supported.
 *
 * This is currently only implemented in the X11 video driver.
 */
extern DECLSPEC int SDL_WM_ToggleFullScreen(SDL_Surface *surface);
```

(SDL_WM_GrabInput `mode`**)**

```
/*
 * This function allows you to set and query the input grab state of
 * the application.  It returns the new input grab state.
 */
typedef enum {
       SDL_GRAB_QUERY = -1,
       SDL_GRAB_OFF = 0,
       SDL_GRAB_ON = 1,
       SDL_GRAB_FULLSCREEN       /* Used internally */
} SDL_GrabMode;
```

10.4 Video4Linux: video grabbing

Author(s): Yann LeCun

The video4linux package allows video frame grabbing under Linux using the video4linux API of the Linux kernel. This interface is compatible with most video grabbing hardware supported by v4l and v4l2.

IMPORTANT: two separate classes are provided for v4l and v4l2 devices. Some webcams and frame grabbers are only compatible with v4l2.

Notable examples include (among many others) TV/Tuner cards based on the BrookTree BT8x8 chips (such as the WinTV cards from Hauppaugge), v4l-supported webcams such the Logitech Quickam Pro 3000 and 4000, and v4l2-supported webcams, such as the 250 or so listed at `http://mxhaard.free.fr/spca5xx.html` .

NOTE: most of the webcams listed at the above page are only compatible with the v4l2 interface, NOT the v4l interface.

Personnally, I use a Logitech Quickcam for Notebooks Pro, which can go up to 960x720 resolution.

10.4.1 Requirements and Installation video4linux and appropriate device

drivers are usually provided with most recent Linux distributions (e.g. Ubuntu 7.x). No special installation is required.

10.4.2 Video4Linux-v2 API (v4l2)

v4l2device

the v4l2 device is the main class through which video frames can be grabbed using the Video4Linux v2 API. This requires a v4l2 compatible video grabbing card or webcam.

Although the v4l2 API supports a variety of video formats for grabbing, this interface only grabs images in the YUV422 format (also known as YUYV). The image can be converted to RGB is required using the `convert-yuv422-to-rgb` found in `libimage/img-util`

Here is an example of code that repeatedly grabs an image from the composite video input and displays it on the screen:

```
(de v4l2-test (d width height fps)
  (libload "libimage/img-util")
  (let  ((z (new v4l2device d width height fps 4))
        (frame-rgb (ubyte-matrix height width 3)))
    (when (not window) (new-window 0 0 (+ width 20) (+ height 20) "v4l2"))
    (cls)
    (repeat 1000
      (==> z get-frame-rgb frame-rgb)
      (rgb-draw-matrix 10 10 frame-rgb)) ))
(v4l2-test "/dev/video0" 640 480 30)
```

(new v4l2device devname w h fps nb**)**

Open video device (webcam etc) devname (e.g. "/dev/video0") preparing to grab images of resoltion w by h . fps is the number of frames per seconds that the v4l2 subsystem will grab. The period between two frames obtained from the device may be longer the 1/fps if Lush spends too much time between calls to one of the get-frame-XXX methods. nb is the number of v4l2 buffers requested. Typical numbers are 2 or 4.

(==> v4l2device **set-input** n**)**

Set active input to n . return 0 on success, -1 on failure.

(==> v4l2device **get-frame-rgb** frame**)**

Get the next available frame from the v4l2 device and place it in frame in RGB format. frame must be an idx3 of ubytes of size height by width by bpp , where bpp must be at least 3. The underlying frame is actually grabbed in YUYV and converted to RGB.

(==> v4l2device **get-frame-grey** frame**)**

Get the next available frame from the v4l2 device and place it in frame in greyscale format. frame must be an idx2 of ubytes of size height by width .

(==> v4l2device **get-frame-yuv** frame**)**

Get the next available frame from the v4l2 device and place it in frame in YUV format. Unlike with YUV422, the U and V components are not subsampled. frame must be an idx3 of ubytes of size height by width by bpp , where bpp must be at least 3. The underlying frame is actually grabbed in YUYV (aka YUV422) and converted to YUV.

(==> v4l2device **get-frame-yuv422** frame**)**

Get the next available frame from the v4l2 device and place it in frame in yuv422 format (also known as YUYV). frame must be an idx3 of ubytes of size height by width by 2 . This is the most efficient of the get-frame-XXX functions as YUYV is the native format of many devices.

(==> v4l2device start**)**

Tells the v4l2 device to start grabbing frames. There is no need to call this method explicitely as it is called automatically when the first call to one of the get-frame-XXX" methods is performed.

(==> v4l2device start**)**

Tells the v4l2 device to start grabbing frames. There is no need to call this method explicitely as it is called automatically when the first call to one of the get-frame-XXX" methods is performed.

(==> v4l2device **cleanup)**

private method to cleanup allocated objects in a v4l2 device.

v4l2 demos

(v4l2-demo [device] [width] [height] [input] [nbuf] [fps] [mode]**)**

Testing/demo function for v4l2device : grabs 1000 video frames and simultaneously displays them in a Lush graphic window. All arguments are optional: device : video device (default: "/dev/video0"). width , height : dimensions of the images to be grabbed (default: 320 by 240). input : input number from which frames will be grabbed (generally 0, 1, or 2). Default is -1, which leaves the input unchanged from the last time the device was used. nbuf :

number of frame buffers (default: 2). `mode` : 0 for RGB, 1 for grayscale, 2 for YUYV with "manual" conversion to RGB before display (default: 0). `fps` : frame rate at which v4l2 will grab frames. The actual frame rate is displayed on the screen. It may be lower than the one requested in `fps` (generally half). Example:

```
(v4l2-demo "/dev/video0" 960 720 0 2 30 0)
```

10.4.3 Video4Linux API (v4l)

The Video4Linux interface to Lush is implemented through the class `v4ldevice` . Creating a new `v4ldevice` will open and initialize one of the Linux video devices. A number of methods are provided to set the parameters (channel, image size, video mode, etc...) and to grab video frames.

Video Mode Constants

```
VIDEO_MODE_PAL       0
VIDEO_MODE_NTSC       1
VIDEO_MODE_SECAM  2
VIDEO_MODE_AUTO       3
```

Video Palette Constants

```
VIDEO_PALETTE_GREY       1
VIDEO_PALETTE_HI240       2
VIDEO_PALETTE_RGB565       3
VIDEO_PALETTE_RGB24       4
VIDEO_PALETTE_RGB32       5
VIDEO_PALETTE_RGB555       6
VIDEO_PALETTE_YUV422       7
VIDEO_PALETTE_YUYV       8
VIDEO_PALETTE_UYVY       9
VIDEO_PALETTE_YUV420       10
VIDEO_PALETTE_YUV411       11
VIDEO_PALETTE_RAW       12
VIDEO_PALETTE_YUV422P 13
VIDEO_PALETTE_YUV411P 14
VIDEO_PALETTE_YUV420P 15
VIDEO_PALETTE_YUV410P 16
VIDEO_PALETTE_PLANAR       13
VIDEO_PALETTE_COMPONENT 7
```

v4ldevice

the v4l device is the main class through which video frames can be grabbed using the Video4Linux device. This requires a v4l compatible video grabbing card or webcam.

Here is an example of code that repeatedly grabs an image from the composite video input and displays it on the screen:

```
(de v4l-demo (swidth sheight)
  (let* ((bg (ubyte-matrix sheight swidth 3)) ; make image buffer
         ;; open video device NTSC standard, composite input (channel 1).
           (vdev (new v4ldevice "/dev/video0" "NTSC" 1 swidth sheight)))
    ;; open window
    (setq screen (new-window))
    (while t
      ;; grab frame
      (==> vdev grab-into-rgbx bg)
      ;; display frame
      (rgb-draw-matrix 10 10 bg))))
```

(==> `v4ldevice` cleanup)

deallocate all malloced slots and close device. This is a private method called by the destructor and various error handlers, but rarely called directly.

(==> `v4ldevice` set-format `f w h`)

set the video format/palette (VIDEO_PALETTE_RGB24, VIDEO_PALETTE_RGB32, VIDEO_PALETTE_YUV422 etc...), and the width and the height of the image to be grabbed (see ["Video Palette Constants"]for a list of formats). Returns 0 on success and -1 if the format requested is not supported by the device. This method rarely called directly, unless the low-level `grab` method is to be used. The various `grab-into-rgb` and `grab-into-rgba` methods determine the appropriate format automatically, and all the `grab-xxx-into-xxx` methods set the desired format before grabbing, therefore a prior call to `set-format` is unnecessary if these high-level methods are used.

(new v4ldevice `devname n c w h`)

opens a new v4ldevice for video grabbing. Arguments are: device name (e.g. "/dev/video0"), norm, channel, width, and height. norm can be "NTSC", "PAL", "SECAM", "AUTO". The "channel" `c` determines which video source the card will listen to. On most bttv cards, 0 means TV tuner, 1 means composite video input, and 2 means S-video input (if present). Tuning to a particular broadcast or cable channel is done with the "tune" method.

example: open device to grab 320x240 NTSC video from S-video on BTTV/avermedia card (channel 2):

```
(setq a (new v4ldevice "/dev/video0" "NTSC" 2 320 240))
```

(==> v4ldevice tune `freq`)

tune the tuner to frequency `n` , expressed in 1/16th of a MHz. Tables that map channels to frequencies are available in file `freq-table.lsh` . Tuning to a channel can only be done is the device was opened to listen to the TV tuner (as opposed to the composite video or S-video).

(==> `v4ldevice` grab-into-rgb `img`)

grab an image from the v4l device into a preexisting image (a ubyte-matrix of size heightxwidthxdepth, where depth must be at least 3). Consecutive elements in the last dimension are interpreted as RGB values. Only those elements are modified. If the video image format has not been previously set, this method automatically determines which format is supported by the current device (among RGB24, RGB32 and YUV420P). This allows to support videocards and webcams transparently without having to worry about which formats are supported. Some webcams (like the Logitech Quickcam 3000 pro) only support YUV420 and YUV420P.

(==> `v4ldevice` grab-into-rgbx `img`)

same as `grab-into-rgb` .

(==> `v4ldevice` grab-into-rgba `img alpha`)

grab an image from the v4l device into a preexisting image (a ubyte-matrix of size heightxwidthxdepth, where depth must be at least 3). Consecutive elements in the last dimension are interpreted as RGBA values. Only those 4 elements are modified. The A components are filled with `alpha` . If the video image format has not been previously set, this method automatically determines which format is supported by the current device (among RGB24, RGB32 and YUV420P). This allows to support videocards and webcams transparently without having to worry about which formats are supported. Some webcams (like the Logitech Quickcam 3000 pro) only support YUV420 and YUV420P.

(==> `v4ldevice` grab-rgb24-into-rgbx `img`)

grab an image from the v4l device in RGB24 format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth, where depth must be at least 3). Consecutive elements in the last dimension are interpreted as RGB values. Only those elements are modified.

(==> `v4ldevice` grab-rgb32-into-rgbx `img`)

grab an image from the v4l device in RGB32 format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth, where depth must be at least 3). Consecutive elements in the last dimension are interpreted as RGB values. Only those elements are modified.

(==> `v4ldevice` grab-rgb24-into-rgba `img alpha`)

grab an image from the v4l device in RGB32 format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth with depth equal to at least 4), Consecutive elements in the last dimension are interpreted as RGBA values. The A values are filled with `alpha` .

(==> `v4ldevice` grab-rgb32-into-rgba `img alpha`)

grab an image from the v4l device in RGB32 format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth with depth equal to at least 4), Consecutive elements in the last dimension are interpreted as RGBA values. The A values are filled with `alpha` .

(==> `v4ldevice` grab-yuv422-into-yuv422 `img`)

grab an image from the v4l device in YUV422 format into a preexisting image (a ubyte-matrix of size heightxwidthx2. Consecutive elements are interpreted as YUYVYUYV..... according to the YUV422 definition. so `(img y x 0)` is the luminance component of pixel (x, y).

(==> `v4ldevice` grab-yuv420p-into-rgbx `img`)

grab an image from the v4l device in YUV420P (planar) format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth with depth at least equal to 3). Consecutive elements are interpreted as RGB components. This mode is useful for certain webcam that do not support RGB grabbing (e.g. Logitech Quickam 3000 Pro).

(==> `v4ldevice` grab-yuv420p-into-yuv `img`)

grab an image from the v4l device in YUV420P (planar) format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth with depth at least equal to 3). Consecutive elements are interpreted as YUV components. This mode is useful for certain webcam that support YUV grabbing (e.g. Logitech Quickam 3000 Pro).

(==> `v4ldevice` grab-yuv420p-into-rgba `img alpha`)

grab an image from the v4l device in YUV420P (planar) format into a preexisting image (a ubyte-matrix of size heightxwidthxdepth with depth at least equal to 4). Consecutive elements are interpreted as RGBA components. the A components are filled with `alpha` . This mode is useful for certain webcam that do not support RGB grabbing (e.g. Logitech Quickam 3000 Pro).

(==> `v4ldevice` grab-grey-into-y `img`)

grab an image from the v4l device in GREY (monocrhome) format into a preexisting image (a ubyte-matrix of size heightxwidth.

(==> `v4ldevice` record-into-rgb `movie s`)

record successive frames from the v4l device into a preexisting movie (a 4-dimensional ubyte-matrix of size nframesxheightxwidthx3). Consecutive elements in the last dimension are interpreted as RGBA values. Consecutive slices in the first dimension are individual frames. The `s` parameter controls the temporal subsampling: every `s` video frame is saved in `movie` .

(==> `v4ldevice` monitor `x y z n w h`)

grab video frames and display them in the current Lush window at position `x` , `y` , with zoom factor `z` . The number of frames grabbed is `n` , and their widht/height is `w` / `h` . return the last grabbed frame.

(==> `v4ldevice` grab-rgb24-acc-rgbx `img`)

grab an image from the v4l device in RGB24 format and accumulate the result into a preexisting image (an int-matrix of size `height` x `width` x `depth` where `height` and `width` are the dimensions of the `v4ldevice`) and `depth` must be at least 3. Consecutive elements in the last dimension are interpreted as RGB values.

(==> `v4ldevice` grab-rgb24-acc-rgba `img alpha`)

grab an image from the v4l device and accumulate the result into a preexisting image (an int-matrix of size `height` x `width` x `depth` where `height` and `width` are the dimensions of the `v4ldevice` , and with depth equal to at least

4). Consecutive elements in the last dimension are interpreted as RGBA values. The alpha channel is filled with value `alpha` .

(==> `v4ldevice` grab)

low-level grab of an image from the v4l device. This is a low-level function that simply performs a frame grab in the video format and image size (as set with set-format) with no format conversion. Most users will prefer to use the other grab methods which perform appropriate conversions and write their result into an idx. The present function simply provides a pointer to the frame buffer. The grabbed data can be accessed through the `framebuffer` slot of the `v4ldevice` (a gptr). The number of bytes in the grab frame is given by the product of the `width` , `height` , and `depth` slots. The format depends on what parameter was passed to `set-format` prior to calling `grab` . Please note that the RGB24 and RGB32 formats of video4linux are really BGR formats from Lush's point of view.

```
(libload "video4linux/v4l")
(libload "libc/libc")
(new-window)
;; these parameters are for Quickcam 3000 pro webcam
(setq v (new v4ldevice "/dev/video" "AUTO" 0 320 240))
(setq m (ubyte-matrix 240 320))
;; set format to YUV420P (planar).
(==> v set-format VIDEO_PALETTE_YUV420P 320 240)
(==> v grab)
;; copy the Y component into the matrix m
(memcpy (idx-ptr m) :v:framebuffer (* 240 320))
(rgb-draw-matrix 0 0 m)
```

TV Frequency tables

these tables contain the frequencies (in 1/16th of a MHz) that can be passed to the "tune" method of a v4l-device to set the channel of the tuner.

freqtable-us-bcast

frequency table vector for US broadcast TV. The i-th element is the frequency of the channel i. When the frequency is 0, the channel does not exist.

freqtable-us-cable

frequency table vector for US cable TV. The i-th element is the frequency of the channel i. When the frequency is 0, the channel does not exist.

10.4.4 Demos

Simple Video Capture Demos

(v4l-demo-rgb `in width height`**)**

monitors NTSC video captured from /dev/video on on a Lush graphic screen. `in` is the card's input from which the video will be captured (=1 for composite input, =2 for S-video input on some bttv cards).

```
(v4l-demo-rgb 1 320 240)
```

bug: the function must be interrupted with CTRL-C.

feature: the function is 6 lines of lush.

(v4l-demo2 `in1 in2 width height`**)**

play captured videos on a Lush graphic screen from two video source simultaneously. This assumes that two video capture devices are present in the computer. `in1` and `in2` are the card's inputs from which the videos will be captured (=2 for S-video input on some bttv cards).

```
(v4l-demo2 2 2 320 240)
```

bug: the function must be interrupted with CTRL-C.

(effect-sdl-demo `eff`**)**

This starts a real-time video effect demo that uses video4linux for grabbing video and SDL for displaying. The `eff` parameter must be an instance of a subclass of `effect` . Several subclasses are provided including edge detection, laplacian, image difference, etc...

This function uses three global variables to configure the video grabbing. Their names are: `*video-device*` `*video-standard*` and `*video-input*` , and their default values are respectively "/dev/video", "NTSC", 1 (1 is the composite input on most TV cards). The default values can be changed by `defvar` ing those variables to the desired value before loading the present file. Example:

```
(effect-sdl-demo (new effect-trail 320 240 20))
(effect-sdl-demo (new effect-edge 320 240))
```

Video Effects

Video effect classes are subclasses of the `effect` class. Video effects are used by calling the `fprop` method which takes one input (RGBA image) and one output (RGBA image). The effect is applied to the input image and written into the output image. The input and output must be `height` x `width` x4 idx3 of ubytes. If used with effect-sdl-demo The alpha channel of the output image will determine the blending of the output image with the previously displayed image. Set all the alphas to 255 for full opacity. Effect subclasses must have a constructor that takes at least a width and height parameter (and possibly other initialization parameters). They must have width and height methods that return the width and height.

(new effect `width height`**)**

noop effect: simply copies the input to the output unaffected

(new effect-edge `width height`**)**

performs a simple edge detection by applying an on-center/off-surround convolution filter and rectifying, amplifying and saturating the result.

(new effect-trail `width height hl`**)**

simply change the transparency to create a "trail" effect. `hl` is the "half life" of the trail: 0 leaves no trail, 1 leaves a trail that decays by half at each frame, and 20 looks funny. Example:

```
(new effect-diff 320 240 20)
```

(new effect-laplace `width height gain`**)**

this applies a sort-of laplacian filter (really a set-of-your-pant on-center/off-surround filter) to each RGB component of the image. The `gain` paramter determines the gain of the output. Values around 1 are reasonable.

(new effect-diff `width height alpha`**)**

computes a simple difference between consecutive images. `alpha` is the transparency with which the difference image is painted. Setting it to 255 will make it opaque, setting it to 128 will leave a short trail.

(new effect-bgsubst `width height imagefile erosion dilation thres`**)**

performs background substitution. This compares the current grabbed image with an image computed as the average of the first 10 frames. If the Euclidean distance in RGB space of a pixel to the average is larger than `thres` , the corresponding pixel of the currently grabbed image is drawn, otherwise the corresponding pixel in the image `imagefile` is drawn. `thres` should be between 500 and 5000 for normal lighting conditions. `erosion` and `dilation` are used to erode and dilate the mask so as to eliminate rogue foreground pixels. Suggested values are 3 and 2 respectively.

realtime video effects

a simple GUI application that shows several real-time video effects, including: color manipulation, image difference, background substitution, edge detection, and image warping.

10.5 BLAS: Basic Linear Algebra Subroutine

Lush has an interface to the Basic Linear Algebra Subroutines library (BLAS). Please refer to the Lush on-line documentation tool `(helptool)` for more details.

10.6 LAPACK

Lush has an interface to part of the Linear Algebra Package library (LAPACK).Please refer to the Lush on-line documentation tool `(helptool)` for more details.

10.7 ALSA: Advanced Linux Sound Architecture

Lush has a full interface to the Advanced Linux Sound Architecture (ALSA). Please refer to the Lush on-line documentation tool tool `(helptool)` for more details.

Chapter 11

Applications

Applications are Lush programs (generally with a graphical user interface) that are useful by themselves without without requiring any programming.

11.1 Graphic Tools

Chapter 12

Tutorials

12.1 Tutorial: Writing Games with Lush and SDL.

Author(s): Yann LeCun

Lush provides a simple way to write 2D real-time games using the SDL library (Simple Directmedia Layer). What follows is a gentle tutorial on how to do that.

This tutorial was written to be understandable by motivated high school students with some basic familiarity with Lush but with little programing experience. Seasoned programmers will probably want to skip certain paragraphs.

The code for this tutorial is available at .

12.1.1 A Quick Reminder on Basic Lush Programming

In this tutorial, we make use of some simple object oriented concepts. Creating a new instance of a class is performed with a call to `new` :

```
(setq an-instance (new a-class arg1 arg2 ...))
```

For example, creating a new sprite (a movable object on a graphic screen) is done with:

```
(setq a-new-sprite (new sdl-sprite scr 0))
```

Objects have slots and methods. Slots are variables that are attached to the object, while methods are functions that are attached to the class of the object. Calling a method on an object is like telling it to carry out a particular action. This is done with the function ==>. Here is an example:

```
(==> a-new-sprite move 300 400)
(==> a-new-sprite draw)
```

This can be interpreted as "tell `a-new-sprite` to `move` to position 300, 400", then "tell `a-new-sprite` to `draw` itself on the screen".

12.1.2 A Simple Lunar Lander

As a case study, we will describe how to write the core of a simple Lunar Lander game. The amazing thing is that the whole game will fit in less than a page of Lush code.

The code described in this section is available in `/home/gemi/Projects/fedora/packages/lush/f14/`

First, make a working directory, say `game` and cd to it.

Creating the Ship and Background Images

Before we start writing code, we need to create images for the ship and the background. This can be done with The Gimp. Start The Gimp, and open a new drawing with a transparent background. Paint your object (or modify an existing digital picture), and save the image as a PNG file. The advantage of using the PNG format is that it is relatively compact, uses lossless compression, and supports an alpha channel for transparency.

For the time being, instead of creating your own art, you can simply copy the files lem.png and moon.png from the directory `/home/gemi/Projects/fedora/packages/lush/f14/lush` into your working directory.

First Implementation of Lander

Getting started

Start your favorite text editor (say emacs), and open a new program file, say lander.lsh.

At the top of the file, we put the following line:

```
(libload "sdl/libsdl")
```

which will load the SDL library.

Next we need to define a function that will run our game:

```
(de lander1 ()
```

initializing SDL

The first line of our function MUST be this:

```
  (sdl-initialize)
```

This will initialize the SDL subsystem. It's OK to call this function multiple times (it only initialize SDL once).

opening the SDL screen

Opening an SDL screen is done as follows:

```
(setq scr (new sdl-screen 640 480 "Lander"))
```

now the variable `scr` contains the screen object.

creating the background and ship sprites

First, we create a new sprite on the screen `scr` and give it the ID 0. Sprites are composed of one of several frames that can be loaded with images. We can load the image file `moon.png` into frame 0, and set the hot-point at coordinates 0 0, (which is the upper-left corner). The hot point is the "handle" by which the sprite is "held". When we move the sprite to coordinate (x, y), we move its handle to (x, y). Here, we will move this sprite to (0, 0).

```
;; create background sprite
(setq bgd (new sdl-sprite scr 0))
;; load moon image into frame 0 of bgd sprite
(==> bgd load-frame "moon.png" 0 0 0)
;; move sprite to 0 0
(==> bgd move 0 0)
```

The move method does not actually draw anything on the screen, it merely sets the internal coordinate variables of the sprite to (0, 0).

Now let's create the ship sprite and create two variables to hold its position:

```
;; create ship sprite
(setq ship (new sdl-sprite scr 1))
;; load lem image into frame 0 of ship sprite
(==> ship load-frame "lem.png" 0 40 35)
;; set position of ship
(setq x 10) (setq y 20)
```

The handle of the ship sprite is a coordinate 40, 35 (i.e. 40 pixel to the right, and 35 pxiels down from the upper left corner of the ship image), which is roughly at the center of the ship.

creating the event handler

We are going to need to grab inputs from the keyboard. To do so, we create an `sdl-event` object, and create an integer vector with 3 elements to hold the result of grabbing the events:

```
(setq event (new sdl-event))
(setq xyk (int-array 3))
```

the main loop
The main loop of our game will look something like this:

```
- clear the screen
- draw the background sprite
- read the keyboard
- move the ship sprite according to the keys pressed
- draw the ship sprite at its new location
- flip the screens (see below)
- repeat the above step until the user quits
```

The SDL screen opened through the `sdl-screen` object is *double buffered* . What that means is that all the drawing commands do not directly happen on the visible screen, but happen in a "hidden" screen (often called a back buffer).When all the objects have been drawn, we swap the visible screen and the invisible screen: the screen we just drew into is now visible, and the previously visible screen is now available for drawing into without affecting what's shown on the screen. This technique allows us to take our time drawing all the object without the user seeing a mess of partially drawn things. In other words, the "double buffer" technique avoids the "flickering" that happens on the screen when objects are drawn one by one in sequence. Flipping the screens is performed by calling the `flip` method of the `sdl-screen` . Here the code of our main loop:

```
(while (not stop)
  (==> scr clear)                 ; fill image with black
  (==> bgd draw)                  ; draw moon ground
  (==> event get-arrows xyk)      ; read keyboard
  (when (= (xyk 2) @@SDLK_q) (setq stop t)) ; stop when q is pressed
  (setq x (+ x (* 10 (xyk 0))))   ; compute new X coordinate
  (setq y (+ y (* 10 (xyk 1))))   ; compute new Y coordinate
  (==> ship move x y)             ; move ship sprite to new position
  (==> ship draw)                 ; draw ship in back buffer
  (==> scr flip)                  ; flip screen buffers
  )                               ; loop
```

The "get-arrow" method fills the three elements of the "xyk" vector as follows:

- (xyk 0): -1 if left arrow pressed, +1 if right arrow pressed, 0 otherwise
- (xyk 1): -1 if up arrow pressed, +1 if down arrow pressed, 0 otherwise
- (xyk 2): key symbol of any other key that is pressed simultaneously.

So, the expression `(setq x (+ x (* 10 (xyk 0))))` will move the ship by 10 pixels to the left or right when the left or right arrow keys are pressed.

The expression `(when (= (xyk 2) @@SDLK_q) (setq stop t))` tests if the "q" key was pressed, and sets the "stop" variable to true if it was pressed. `SDLK_q` is a constant. The value of a constant is accessed by prepending one or two "@" characters to the name. The `while` loop tests the `stop` variable and exits if it is true (i.e. if the "q" key has been pressed).

Putting it all together

Here is the complete code:

```
(de lander01 ()
  ;; initialize the SDL subsystem. DONT FORGET THIS!!!
  (sdl-initialize)
  (setq scr (new sdl-screen 640 480 "Lander")) ; open screen
  ;; create background sprite
  (setq bgd (new sdl-sprite scr 0))
  (==> bgd load-frame "moon.png" 0 0 0)
  (==> bgd move 0 0)
  ;; create lem sprite
  (setq ship (new sdl-sprite scr 1))
  (==> ship load-frame "lem.png" 0 40 35)
  ;; set position of ship
  (setq x 200) (setq y 100)
  ;; create event object
  (setq event (new sdl-event))
  (setq xyk (int-array 3))

  (while (not stop)
    (==> scr clear)                    ; fill image with black
    (==> bgd draw)                    ; draw moon ground
    (==> event get-arrows xyk)             ; read keyboard
    (when (= (xyk 2) @@SDLK_q) (setq stop t)) ; stop when q is pressed
    (setq x (+ x (* 10 (xyk 0))))
    (setq y (+ y (* 10 (xyk 1))))
    (==> ship move x y)                    ; move ship sprite to position
    (==> ship draw)                    ; draw ship
    (==> scr flip)                    ; flip screens
    ))
```

This code has three major problems:

- It's not a real Lunar Lander game, since the ship does not obey Newtonian mechanics.
- It's written in a terribly un-clean style: the code uses lots of global variables, which is universally recognized as bad practice.
- nothing prevents the ship from getting off the screen.

Second Lander: Gravity and Newtonian mechanics

The new implementation replaces all the global variables by local variables created using the `let*` construct. Local variables created with `let*` (or `let`) disappear after the `let*` evaluation completes.

The new implementation also obeys Newtonian mechanics with (gravity, inertia an such). This is done very simply with the following sequence of operations inside the main loop:

```
1 - read the keyboard arrows and determine the engines thrusts
2 - compute accelerations from thrust and gravity (apply accel=force/mass):
    - set X-acceleration = X-thrust / ship's mass
    - set Y-acceleration = Y-thrust / ship's mass + gravity
3 - compute new velocity from acceleration (time integration):
    - set new X-velocity = old X-velocity + X-acceleration * deltat
    - set new Y-velocity = old Y-velocity + Y-acceleration * deltat
4 - compute new position from velocity (time integration):
    - set X-position = X-position + X-velocity * deltat
    - set Y-position = Y-position + Y-velocity * deltat
```

The `deltat` variable is the expected time it takes to go around the main loop of the game (more on this below). Here is how the above equations work. **Step 2** computes the accelerations using Newton's law *acceleration equals force divided by mass* . We must add the acceleration of gravity (which is independent from the mass, as Galileo showed by dropping things from the tower of Pisa) to the Y component of the acceleration. Our objects will be positioned on the screen, where the unit of distance is the pixel. Therefore, our unit of speed will be the pixel per second, and our unit of acceleration the pixel per second per second (i.e. by how many *pixels per second* does our speed change every second). Rather than divide the forces by the mass, we precompute the inverse of the mass `mass-inv` , and multiply the force by this value. We do this because mutliplication is a lot faster than division. The Lush code segment is:

```
(setq ax (* mass-inv side-thrust (xyk 0)))
(setq ay (+ grav (* mass-inv main-thrust (xyk 1))))
```

Step 3 computes the velocities from the accelerations. The idea is the following, if horizontal acceleration is `X-acceleration` , and we maintain that acceleration for `deltat` seconds, our velocity will have increased by `X-acceleration` times `deltat` . If we assume that the acceleration was constant while our program went around the loop, and that it took `deltat` seconds to go around that loop, then we must increase the velocity by `(* ax deltat)` . Here is the appropriate Lush code:

```
(setq vx (+ vx (* ax deltat)))  ; update X-velocity
(setq vy (+ vy (* ay deltat)))  ; update Y-velocity
```

Step 4 computes the position from the velocities. The idea is similar: During the time `deltat` that it takes our program to go around its main loop, we assume that the velocity is constant. The position of the ship during that time has changed by the velocity times `deltat` . This is true for the horizontal velocity and position as well as the vertical velocity and position:

```
(setq x  (+ x (* vx deltat)))   ; update X-position
(setq y  (+ y (* vy deltat)))   ; update Y-position
```

So, if going around the loop takes 0.05 seconds (20 frames per second), `deltat` should be 0.05. For example, if the X-velocity is 40 pixels per second and going around the loop takes 0.05 seconds, then the X-position should be incremented by 40*0.05 = 2 pixels each time we go around the loop, hence the formula above. How do we know how long it takes to go around the loop? Fortunately, the `flip` method of `sdl-screen` object sets the the `deltat` slot of the `sdl-screen` object to the number of seconds since the last call to flip (most likely, and hopefully a number much smaller than 1). So as long as we do one screen flip per cycle around the loop, we can simply set our `deltat` to the screen's `deltat` which we can access with `:scr:deltat` . The complete update code for our main loop is now:

```
[...get keyboard input into xyk here...]
(setq ax (* mass-inv side-thrust (xyk 0))) ; update X-acceleration
(setq ay (+ grav (* mass-inv main-thrust (xyk 1)))) ; update Y-accel
(setq vx (+ vx (* ax deltat)))  ; update X-velocity
(setq vy (+ vy (* ay deltat)))  ; update Y-velocity
(setq x  (+ x (* vx deltat)))   ; update X-position
(setq y  (+ y (* vy deltat)))   ; update Y-position
[...draw all the objects here...]
(==> scr flip)                  ; flip screens
(setq deltat :scr:deltat)       ; get time between screen flips
```

Next, we need some code to bounce the ship around or have it wrap around the screen when it goes off the boundaries. Here is the code below. We are assuming that the variable "ground" contains the value 360 or so (near the bottom of the screen):

```
(when (< x -40) (setq x (+ 640 (- x -40)))) ; wrap around left side
(when (> x 680) (setq x (+ -40 (- x 640)))) ; wrap around right side
(when (> y ground)                  ; bounce on ground
  (setq vy (* -0.5 vy))           ; divide vertical speed by 2
  (setq vx (* 0.25 vx))           ; divide horiz speed by 4
  (setq y ground))                ; set vert position to ground altitude
```

Here is the new complete code:

```
(de lander02 ()
  ;; initialize the SDL subsystem. DONT FORGET THIS!!!
  (sdl-initialize)
  (let* ((scr (new sdl-screen 640 480 "Lander")) ; open screen
        (bgd (new sdl-sprite scr 0))       ; create background sprite
        (ship (new sdl-sprite scr 1))       ; create lem sprite
        ;; set position, velocity, acceleration of ship
        (x 200) (y 100) (vx 4) (vy 0) (ax 0) (ay 0)
        ;; set mass, inverse mass, and deltat of ship
        (mass 1) (mass-inv (/ 1 mass)) (deltat 0.01)
        (side-thrust 200)              ; set side engine thrust
        (main-thrust 400)              ; set main engine thrust
        (grav 200)              ; set gravity coefficient in pixels/s/s
        (stop ())
        (event (new sdl-event))
        (xyk (int-array 3))
        (ground 360))
    (==> bgd load-frame "moon.png" 0 0 0)
    (==> bgd move 0 0)
    (==> ship load-frame "lem.png" 0 40 35)
    (while (not stop)
      (==> scr clear)                     ; fill image with black
      (==> bgd draw)                     ; draw moon ground
      (==> event get-arrows xyk)       ; read keyboard
      (when (= (xyk 2) @@SDLK_q) (setq stop t))       ; stop when q is pressed
      (setq ax (* mass-inv side-thrust (xyk 0))) ; update acceleration
      (setq ay (+ grav (* mass-inv main-thrust (xyk 1)))) ; update acceleration
      (setq vx (+ vx (* ax deltat)))       ; update velocity
      (setq vy (+ vy (* ay deltat)))       ; update velocity
      (setq x  (+ x (* vx deltat)))       ; update position
      (setq y  (+ y (* vy deltat)))       ; update position
      (when (< x -40) (setq x (+ 640 (- x -40)))) ; wrap around left side
      (when (> x 680) (setq x (+ -40 (- x 640)))) ; wrap around right side
      (when (> y ground)              ; bounce on ground
       (setq vy (* -0.5 vy))           ; divide vertical speed by 2
       (setq vx (* 0.25 vx))           ; divide horiz speed by 4
       (setq y ground))                ; set vert position to ground altitude
      (==> ship move x y)              ; move ship sprite to position
      (==> ship draw)                     ; draw ship
      (==> scr flip)                     ; flip screens
      (setq deltat :scr:deltat)       ; update deltat to time between screen flips
      )))
```

Third Lander: a flame and a shadow

Let's add a little coolness and realism. We will add a flame when the engine is on, and shadow of the ship projected on the ground. We will create two additional sprites, one for the flame, one for the shadow. The flame will be drawn at the same location as the ship, whenever the motor is on. The shadow will drawn at the same horizontal position as the ship, but at the same vertical position as the ground. Here are the relevant lines:

```
(let* ([... allocate screen and bg sprite ...]
       (ship (new sdl-sprite scr 1))       ; create lem sprite
       (flame (new sdl-sprite scr 1))       ; create flame sprite
       (shadow (new sdl-sprite scr 3)) ; create shadow sprite
       [... more initializations...]
      )
  [... load background image ...]
  ;; load image and put the handle at the center of the sprite (40,35)
  (==> ship load-frame "lem.png" 0 40 35)
  ;; the flame is designed to have the handle at the same place
  (==> flame load-frame "lem-flame.png" 0 40 35)
  ;; here is the shadow
  (==> shadow load-frame "lem-shadow.png" 0 40 -6)
  (while (not stop)                   ; main loop
    [... compute all the coordinates...]
    (==> shadow move x 360)              ; move shadow sprite to position
    (==> flame move x y)              ; move flame sprite to position
    (==> ship move x y)              ; move ship sprite to position
    (==> shadow draw)                  ; draw shadow
    (when (<> 0 (xyk 1)) (==> flame draw)) ; draw flame if engine is on
    (==> ship draw)                    ; draw ship
    (==> scr flip)                      ; flip screens
    (setq deltat :scr:deltat)       ; update deltat to time between screen flips
    ))
```

Here is the complete code (which can be found in /home/gemi/Projects/fedora/packages/lush/f14/lush-2.0/pac
)

```
(de lander03 ()
  ;; initialize the SDL subsystem. DONT FORGET THIS!!!
  (sdl-initialize)
  (let* ((scr (new sdl-screen 640 480 "Lander")) ; open screen
       (bgd (new sdl-sprite scr 0))       ; create background sprite
       (ship (new sdl-sprite scr 1))       ; create lem sprite
       (flame (new sdl-sprite scr 1))       ; create flame sprite
       (shadow (new sdl-sprite scr 3)) ; create shadow sprite
```

```
    ;; set position, velocity, acceleration of ship
    (x 200) (y 100) (vx 4) (vy 0) (ax 0) (ay 0)
    ;; set mass, inverse mass, and deltat of ship
    (mass 1) (mass-inv (/ 1 mass)) (deltat 0.01)
    (side-thrust 200)              ; set side engine thrust
    (main-thrust 400)              ; set main engine thrust
    (grav 200)              ; set gravity coefficient in pixels/s/s
    (stop ())
    (event (new sdl-event))
    (xyk (int-array 3))
    (ground 360))
(==> bgd load-frame "moon.png" 0 0 0)
(==> bgd move 0 0)
;; put the handle at the center of the sprite (40,35)
(==> ship load-frame "lem.png" 0 40 35)
(==> flame load-frame "lem-flame.png" 0 40 35)
(==> shadow load-frame "lem-shadow.png" 0 40 -6)
(while (not stop)
  (==> scr clear)                    ; fill image with black
  (==> bgd draw)                    ; draw moon ground
  (==> event get-arrows xyk)       ; read keyboard
  (when (= (xyk 2) @@SDLK_q) (setq stop t))       ; stop when q is pressed
  (setq ax (* mass-inv side-thrust (xyk 0))) ; update acceleration
  (setq ay (+ grav (* mass-inv main-thrust (xyk 1)))) ; update acceleration
  (setq vx (+ vx (* ax deltat)))       ; update velocity
  (setq vy (+ vy (* ay deltat)))       ; update velocity
  (setq x  (+ x (* vx deltat)))       ; update position
  (setq y  (+ y (* vy deltat)))       ; update position
  (when (< x -40) (setq x (+ 640 (- x -40)))) ; wrap around left side
  (when (> x 680) (setq x (+ -40 (- x 640)))) ; wrap around right side
  (when (> y ground)              ; bounce on ground
   (setq vy (* -0.5 vy))
   (setq vx (* 0.25 vx))
   (setq y ground))
  (==> shadow move x 360)              ; move ship sprite to position
  (==> flame move x y)              ; move ship sprite to position
  (==> ship move x y)              ; move ship sprite to position
  (==> shadow draw)
  (when (<> 0 (xyk 1)) (==> flame draw)) ; draw flame if engine is on
  (==> ship draw)
  (==> scr flip)                    ; flip screens
  (setq deltat :scr:deltat)       ; update deltat to time between screen flips
  )))
```

Fourth Lander: the ship rotates

More realistic lunar landers have no side thruster, but rotate around to direct their main thruster. We will now explain how to create rotated images of the ship. To do so, the `sdl-sprite` has a method called `rotscale-frame` . This method is called as follows:

```
(==> ship rotscale-frame src-frame dst-frame angle scale)
```

This takes the image of a source frame, identified by its number `src-frame` , rotates it by `angle` degrees (clockwise), scale it by `scale` , and write the resulting image in the frame with index `dst-frame` . The handle (or hotpoint) is left unchanged in the process, i.e., the handle in the transformed frame is at the same location within the object as in the source frame.

In the code segment below, we take the 0-th frame and create rotated version of it every 10 degrees:

```
(==> ship load-frame "lem.png" 0 40 35)
(let ((i 1))
  ;; make an image every 10 degrees from 10 to 350
  (for (angle 10 350 10)
    ;; take frame 0, rotate by angle, scale by 1,
    ;; and copy into frame i
    (==> ship rotscale-frame 0 i angle 1)
    (incr i)))
```

Within the main loop, we must read the keyboard, and use the result from the left and right arrow keys to rotate the ship. We will use a variable `theta` to store the current angle of the ship. This variable is an integer between 0 and 35 which, when multiplied by 10 gives the ship's angle with the vertical. `theta` is used as the index of the frame for the ship sprite:

```
(while (not stop)
  [... stuff deleted...]
  (==> event get-arrows xyk)       ; read keyboard
  ;; update the angle from the left-right arrow keys
  (setq theta (+ theta (xyk 0)))
  ;; bring the angle back to the 0-360 interval
  (while (>= theta 36) (setq theta (- theta 36)))
  (while (< theta 0) (setq theta (+ theta 36)))
  ;; set frame of ship and flame to one matching the angle
  (==> ship set-frame (int theta))
  [... stuff deleted...]
  )
```

Next, we must modify the computation of the X and Y accelerations to take into account the fact that the thrust of the main engine is at an angle. When the angle of the ship with the vertical is `theta` times 10 degrees, the horizontal component of the thrust is `main-thrust * sin( -pi/180 * 10 * theta)` , which in lush is written `(* main-thrust (sin (* pi/180 -10 theta)))` . We must multiply the angle in degree by `pi/180` because the `sin` functions takes angles in radians. Rather than computing `pi/180` every time, we precompute the value and put the result in a global variable called `pi/180` :

```
(defvar pi/180 (/ 3.1415927 180))
```

The horizontal acceleration is the horizontal thrust divided by the ship's mass (or multiplied by the inverse of the mass)

```
(setq ax (* mass-inv main-thrust (xyk 1) (sin (* pi/180 -10 theta))))
```

The vertical component of the acceleration is derived similarly, except the `sin` is now a `cos` , and the gravity component is added:

```
;; compute x acceleration
(setq ax (* mass-inv main-thrust (xyk 1) (sin (* pi/180 -10 theta))))
;; compute y acceleration
(setq ay (+ grav (* mass-inv main-thrust
                    (xyk 1) (cos (* pi/180 10 theta)))))
```

The rest of the code is identical to the previous version.

Here is the complete code, which can be found in `/home/gemi/Projects/fedora/packages/lush/f14`):

```
(setq pi/180 (/ 3.1415927 180))
(de lander04 ()
  ;; initialize the SDL subsystem. DONT FORGET THIS!!!
  (sdl-initialize)
  (let* ((scr (new sdl-screen 640 480 "Lander")) ; open screen
         (bgd (new sdl-sprite scr 0))       ; create background sprite
         (ship (new sdl-sprite scr 1))       ; create lem sprite
         (flame (new sdl-sprite scr 1))       ; create flame sprite
         (shadow (new sdl-sprite scr 3)) ; create shadow sprite
         ;; set position, velocity, acceleration of ship
         (x 200) (y 100) (vx 4) (vy 0) (ax 0) (ay 0)
         ;; angle of ship
         (theta 0)
         ;; set mass, inverse mass, and deltat of ship
         (mass 1) (mass-inv (/ 1 mass)) (deltat 0.01)
```

```
    (side-thrust 200)              ; set side engine thrust
    (main-thrust 400)              ; set main engine thrust
    (grav 200)              ; set gravity coefficient in pixels/s/s
    (stop ())
    (event (new sdl-event))
    (xyk (int-array 3))
    (ground 360))
(==> bgd load-frame "moon.png" 0 0 0)
(==> bgd move 0 0)
;; put the handle at the center of the sprite (40,35)
(==> ship load-frame "lem.png" 0 40 35)
(==> flame load-frame "lem-flame.png" 0 40 35)
(==> shadow load-frame "lem-shadow.png" 0 40 -6)
;; fill up frames with rotated lems
(let ((i 1))
  ;; make an image every 10 degrees from 10 to 350
  (for (angle 10 350 10)
      ;; take frame 0, rotate by angle, scale by 1,
      ;; and copy into frame i
      (==> ship rotscale-frame 0 i angle 1)
      ;; same for flame
      (==> flame rotscale-frame 0 i angle 1)
      (incr i)))
(while (not stop)
  (==> scr clear)                    ; fill image with black
  (==> bgd draw)                    ; draw moon ground
  (==> event get-arrows xyk)       ; read keyboard
  (when (= (xyk 2) @@SDLK_q) (setq stop t))       ; stop when q is pressed
  ;; update the angle from the left-right arrow keys
  (setq theta (+ theta (xyk 0)))
  ;; bring the angle back to the 0-360 interval
  (while (>= theta 36) (setq theta (- theta 36)))
  (while (< theta 0) (setq theta (+ theta 36)))
  ;; set frame of ship and flame to one matching the angle
  (==> ship set-frame (int theta))
  (==> flame set-frame (int theta))
  ;; compute x acceleration
  (setq ax (* mass-inv main-thrust (xyk 1) (sin (* pi/180 -10 theta))))
  ;; compute y acceleration
  (setq ay (+ grav (* mass-inv main-thrust
                  (xyk 1) (cos (* pi/180 10 theta)))))
  (setq vx (+ vx (* ax deltat)))       ; update velocity
  (setq vy (+ vy (* ay deltat)))       ; update velocity
  (setq x  (+ x (* vx deltat)))       ; update position
  (setq y  (+ y (* vy deltat)))       ; update position
  (when (< x -40) (setq x (+ 640 (- x -40)))) ; wrap around left side
```

```
(when (> x 680) (setq x (+ -40 (- x 640)))) ; wrap around right side
(when (> y ground)              ; bounce on ground
 (setq vy (* -0.5 vy))
 (setq vx (* 0.25 vx))
 (setq theta 0)
 (setq y ground))
(==> shadow move x 360)              ; move shadow sprite to position
(==> flame move x y)              ; move flame sprite to position
(==> ship move x y)              ; move ship sprite to position
;; now draw sprites in the right order, bottom sprite first
(==> shadow draw)                    ; draw shadow
(when (<> 0 (xyk 1)) (==> flame draw)) ; draw flame if engine is on
(==> ship draw)                     ; draw ship
(==> scr flip)                     ; flip screens
(setq deltat :scr:deltat)       ; update deltat to time between screen flips
)))
```

12.1.3 SpaceWar: missiles and collision detection

Two ships

[under construction]

Shooting missiles

[under construction]

Collision Detection

[under construction]

Chapter 13

FAQ (Frequently Asked Questions)

13.1 What to do when I get this error?

13.1.1 Module has undefined references

when you load a partially linked object file, or when a function you compiled calls a C function that is not defined in any previously loaded library or module you get the above error. To find out which functions are not defined, type `(mod-undefined)` .

13.1.2 compiler : Unknown Type in: ()

This message may occur for a variety of reasons, but the most common one is that the compiler can't figure out the type of the return value of an expression. A frequent cause is an hash-brace construct (inline C code segment) that is the last expression of a function (and therefore its return value). Lush cannot know the type of the value returned by a hash-brace unless you cast it.

Example 1:

```
;; this causes an error
(de foo1-broken (x) ((-double-) x) #{ $x*$x #})
;; this is correct
(de foo1-correk (x) ((-double-) x) (to-double #{ $x*$x #}))
```

Example 2:

```
;; this causes an error (dunno the return type).
(de foo2-broken (x) ((-idx1- (-int-)) x) #{ *(IDX_PTR($x,int)) = 34 #})
;; this is correct (return value is nil).
```

```
(de foo2-correk (x) ((-idx1- (-int-)) x) #{ *(IDX_PTR($x,int)) = 34 #} ())
```

13.2 How Do I ... ?

13.2.1 Read lines from a file into a list

```
(de read-lines(f)
 (reading f
   (let ((ans ()))
     (while (<> (skip-char "\\n\\r\\f") "\\e")
       (setq ans (cons (read-string) ans)))
     (reverse ans))))
```

13.2.2 Apply a function to all elements of a vector

interpreted lisp way:

```
(idx-bloop ((x v)) (func x))
```

efficient compiled C code way:

```
(cidx-bloop ("i" ("v" v)) #{ *v = my_c_fun(*v); #}) ())
```

13.2.3 Get a pointer to the raw data of an idx

just use the `idx-ptr` function.

```
(idx-ptr m}
```

13.2.4 Get a pointer to a function written in Lisp

Some functions in popular libraries take function pointers as argument (a typical example is the GSL function minimization routines). To obtain a pointer to the compiled version is a function written in Lisp, simply use the function `to-gptr` :

```
(de myfunc (x) ((-double-) x) (- (* x x) 2))
(dhc-make () myfunc)
(some-root-finding-function-in-C (to-gptr myfunc))
```

13.2.5 Know if a function can be used in compiled code

Just use `compilablep` :

```
? (compilablep +)
= t
```