Unicon Programming

Release 0.6.149

Brian Tiffin

Oct 27, 2019

CONTENTS

1 Unicon Programming **3**
- 1.1 The Unicon Programming Language . . . 3
  - 1.1.1 Well met . . . 3
  - 1.1.2 Overview of Unicon . . . 6
  - 1.1.3 Building Unicon from source . . . 12
- 1.2 History . . . 14
  - 1.2.1 The Icon roots . . . 15
  - 1.2.2 And then Unicon . . . 15
  - 1.2.3 People . . . 16

2 Datatypes **19**
- 2.1 Immutable Unicon Datatypes . . . 19
  - 2.1.1 Integer . . . 19
  - 2.1.2 Real numbers . . . 23
  - 2.1.3 Cset . . . 25
  - 2.1.4 String . . . 26
  - 2.1.5 Pattern . . . 27
  - 2.1.6 Regular Expression . . . 27
- 2.2 Non computational types . . . 27
  - 2.2.1 File . . . 27
  - 2.2.2 Window . . . 27
  - 2.2.3 Co-expression . . . 27

3 Data Structures **29**
- 3.1 Unicon mutable data types . . . 29
  - 3.1.1 List (arrays) . . . 29
  - 3.1.2 Set . . . 31
  - 3.1.3 Table . . . 32
  - 3.1.4 Record . . . 34

4 Expressions **37**
- 4.1 Unicon expressions . . . 37
  - 4.1.1 Success and Failure . . . 37
  - 4.1.2 null . . . 38
  - 4.1.3 Precedence . . . 39
  - 4.1.4 Variable scope . . . 41
  - 4.1.5 Semicolon insertion . . . 42
  - 4.1.6 Bound Expressions . . . 43
- 4.2 Unicon Co-Expressions . . . 45
  - 4.2.1 User defined control structures . . . 45

5 **Operators** 47
5.1 Unicon operators . . . 47
5.1.1 Precedence chart . . . 47
5.2 Unary operators . . . 50
5.2.1 ! (generate elements) . . . 50
5.2.2 * (size) . . . 52
5.2.3 + (numeric identity) . . . 53
5.2.4 - (negate) . . . 54
5.2.5 / (null test) . . . 54
5.2.6 \ (not null test) . . . 55
5.2.7 . (dereference) . . . 55
5.2.8 = (anchored or tab match) . . . 55
5.2.9 | (repeated alternation) . . . 56
5.2.10 ? (random element) . . . 56
5.2.11 @ (activation) . . . 56
5.2.12 ~ (cset complement) . . . 56
5.3 Binary Operators . . . 57
5.3.1 & (conjunction) . . . 57
5.3.2 &:= (augmented &) . . . 57
5.3.3 | (alternation) . . . 57
5.3.4 || (concatenation) . . . 58
5.3.5 ||:= (augmented ||) . . . 58
5.3.6 ||| (list concatenation) . . . 58
5.3.7 |||:= (augmented |||) . . . 58
5.3.8 ? (string scan) . . . 58
5.3.9 ?:= (augmented ?) . . . 58
5.3.10 ?? (pattern scan) . . . 58
5.4 Operator idioms . . . 59
5.5 Operator functions . . . 60

6 **Reserved words** 61
6.1 Unicon reserved words . . . 61
6.1.1 Reserved word list . . . 62
6.2 Unicon action words . . . 62
6.2.1 break . . . 62
6.2.2 case . . . 63
6.2.3 create . . . 64
6.2.4 critical . . . 64
6.2.5 every . . . 66
6.2.6 fail . . . 70
6.2.7 if . . . 70
6.2.8 initial . . . 71
6.2.9 initially . . . 71
6.2.10 next . . . 72
6.2.11 not . . . 73
6.2.12 repeat . . . 74
6.2.13 return . . . 74
6.2.14 suspend . . . 75
6.2.15 thread . . . 76
6.2.16 to . . . 78
6.2.17 until . . . 78
6.2.18 while . . . 79
6.3 Declarative expressions . . . 79
6.3.1 abstract . . . 80

6.3.2 class . . . 81
6.3.3 global . . . 82
6.3.4 import . . . 83
6.3.5 invocable . . . 83
6.3.6 link . . . 84
6.3.7 local . . . 85
6.3.8 method . . . 85
6.3.9 package . . . 86
6.3.10 procedure . . . 87
6.3.11 record . . . 88
6.3.12 static . . . 89
6.4 Syntax reserved words . . . 89
6.4.1 all . . . 89
6.4.2 by . . . 90
6.4.3 default . . . 90
6.4.4 do . . . 90
6.4.5 end . . . 90
6.4.6 else . . . 90
6.4.7 of . . . 90
6.4.8 then . . . 91

7 Functions **93**
7.1 Unicon Functions . . . 93
7.1.1 Abort . . . 93
7.1.2 abs . . . 94
7.1.3 acos . . . 94
7.1.4 Active . . . 96
7.1.5 Alert . . . 97
7.1.6 any . . . 98
7.1.7 Any . . . 98
7.1.8 Arb . . . 99
7.1.9 Arbno . . . 99
7.1.10 args . . . 99
7.1.11 array . . . 100
7.1.12 asin . . . 101
7.1.13 atan . . . 103
7.1.14 atanh . . . 104
7.1.15 Attrib . . . 104
7.1.16 Bal . . . 105
7.1.17 bal . . . 106
7.1.18 Bg . . . 107
7.1.19 Break . . . 107
7.1.20 Breakx . . . 108
7.1.21 callout . . . 108
7.1.22 center . . . 109
7.1.23 char . . . 109
7.1.24 chdir . . . 110
7.1.25 chmod . . . 110
7.1.26 chown . . . 111
7.1.27 chroot . . . 111
7.1.28 classname . . . 112
7.1.29 Clip . . . 113
7.1.30 Clone . . . 113
7.1.31 close . . . 114

7.1.32 cofail . . . 115
7.1.33 collect . . . 116
7.1.34 Color . . . 117
7.1.35 ColorValue . . . 117
7.1.36 condvar . . . 118
7.1.37 constructor . . . 119
7.1.38 copy . . . 120
7.1.39 CopyArea . . . 121
7.1.40 cos . . . 121
7.1.41 Couple . . . 123
7.1.42 crypt . . . 124
7.1.43 cset . . . 125
7.1.44 ctime . . . 125
7.1.45 dbcolumns . . . 126
7.1.46 dbdriver . . . 128
7.1.47 dbkeys . . . 129
7.1.48 dblimits . . . 130
7.1.49 dbproduct . . . 131
7.1.50 dbtables . . . 131
7.1.51 delay . . . 132
7.1.52 delete . . . 133
7.1.53 detab . . . 133
7.1.54 display . . . 134
7.1.55 DrawArc . . . 135
7.1.56 DrawCircle . . . 135
7.1.57 DrawCube . . . 136
7.1.58 DrawCurve . . . 137
7.1.59 DrawCylinder . . . 138
7.1.60 DrawDisk . . . 139
7.1.61 DrawImage . . . 140
7.1.62 DrawLine . . . 141
7.1.63 DrawPoint . . . 143
7.1.64 DrawPolygon . . . 144
7.1.65 DrawRectangle . . . 145
7.1.66 DrawSegment . . . 145
7.1.67 DrawSphere . . . 146
7.1.68 DrawString . . . 147
7.1.69 DrawTorus . . . 148
7.1.70 dtor . . . 149
7.1.71 entab . . . 149
7.1.72 EraseArea . . . 150
7.1.73 errorclear . . . 150
7.1.74 Event . . . 151
7.1.75 eventmask . . . 152
7.1.76 EvGet . . . 152
7.1.77 EvSend . . . 153
7.1.78 exec . . . 155
7.1.79 exit . . . 155
7.1.80 exp . . . 156
7.1.81 Eye . . . 156
7.1.82 Fail . . . 157
7.1.83 fcntl . . . 158
7.1.84 fdup . . . 159
7.1.85 Fence . . . 159

7.1.86 fetch . . . 160
7.1.87 Fg . . . 160
7.1.88 fieldnames . . . 161
7.1.89 filepair . . . 162
7.1.90 FillArc . . . 163
7.1.91 FillCircle . . . 164
7.1.92 FillPolygon . . . 164
7.1.93 FillRectangle . . . 165
7.1.94 find . . . 165
7.1.95 flock . . . 166
7.1.96 flush . . . 167
7.1.97 Font . . . 167
7.1.98 fork . . . 168
7.1.99 FreeColor . . . 169
7.1.100 FreeSpace . . . 170
7.1.101 function . . . 170
7.1.102 get . . . 171
7.1.103 getch . . . 172
7.1.104 getche . . . 173
7.1.105 getegid . . . 173
7.1.106 getenv . . . 174
7.1.107 geteuid . . . 174
7.1.108 getgid . . . 174
7.1.109 getgr . . . 175
7.1.110 gethost . . . 175
7.1.111 getpgrp . . . 176
7.1.112 getpid . . . 176
7.1.113 getppid . . . 177
7.1.114 getpw . . . 177
7.1.115 getrusage . . . 178
7.1.116 getserv . . . 179
7.1.117 GetSpace . . . 180
7.1.118 gettimeofday . . . 181
7.1.119 getuid . . . 181
7.1.120 globalnames . . . 182
7.1.121 GotoRC . . . 182
7.1.122 GotoXY . . . 183
7.1.123 gtime . . . 184
7.1.124 hardlink . . . 184
7.1.125 iand . . . 185
7.1.126 icom . . . 185
7.1.127 IdentityMatrix . . . 186
7.1.128 image . . . 187
7.1.129 InPort . . . 187
7.1.130 insert . . . 188
7.1.131 Int86 . . . 188
7.1.132 integer . . . 189
7.1.133 ioctl . . . 189
7.1.134 ior . . . 190
7.1.135 ishift . . . 190
7.1.136 istate . . . 191
7.1.137 ixor . . . 192
7.1.138 kbhit . . . 192
7.1.139 key . . . 193

7.1.140 keyword . . . 194
7.1.141 kill . . . 194
7.1.142 left . . . 195
7.1.143 Len . . . 196
7.1.144 list . . . 196
7.1.145 load . . . 197
7.1.146 loadfunc . . . 198
7.1.147 localnames . . . 200
7.1.148 lock . . . 201
7.1.149 log . . . 201
7.1.150 Lower . . . 202
7.1.151 lstat . . . 202
7.1.152 many . . . 203
7.1.153 map . . . 203
7.1.154 match . . . 204
7.1.155 MatrixMode . . . 205
7.1.156 max . . . 206
7.1.157 member . . . 207
7.1.158 membernames . . . 207
7.1.159 methodnames . . . 208
7.1.160 methods . . . 209
7.1.161 min . . . 209
7.1.162 mkdir . . . 210
7.1.163 move . . . 211
7.1.164 MultMatrix . . . 211
7.1.165 mutex . . . 212
7.1.166 name . . . 213
7.1.167 NewColor . . . 214
7.1.168 Normals . . . 215
7.1.169 NotAny . . . 215
7.1.170 Nspan . . . 215
7.1.171 numeric . . . 216
7.1.172 open . . . 217
7.1.173 opencl . . . 218
7.1.174 oprec . . . 219
7.1.175 ord . . . 220
7.1.176 OutPort . . . 220
7.1.177 PaletteChars . . . 221
7.1.178 PaletteColor . . . 221
7.1.179 PaletteKey . . . 222
7.1.180 paramnames . . . 222
7.1.181 parent . . . 223
7.1.182 Pattern . . . 223
7.1.183 Peek . . . 224
7.1.184 Pending . . . 224
7.1.185 pipe . . . 225
7.1.186 Pixel . . . 226
7.1.187 PlayAudio . . . 227
7.1.188 Poke . . . 227
7.1.189 pop . . . 228
7.1.190 PopMatrix . . . 228
7.1.191 Pos . . . 229
7.1.192 pos . . . 230
7.1.193 proc . . . 230

7.1.194 pull . . . 231
7.1.195 push . . . 231
7.1.196 PushMatrix . . . 232
7.1.197 PushRotate . . . 233
7.1.198 PushScale . . . 233
7.1.199 PushTranslate . . . 234
7.1.200 put . . . 234
7.1.201 QueryPointer . . . 235
7.1.202 Raise . . . 235
7.1.203 read . . . 236
7.1.204 ReadImage . . . 236
7.1.205 readlink . . . 238
7.1.206 reads . . . 238
7.1.207 ready . . . 239
7.1.208 real . . . 239
7.1.209 receive . . . 240
7.1.210 Refresh . . . 240
7.1.211 Rem . . . 241
7.1.212 remove . . . 242
7.1.213 rename . . . 242
7.1.214 repl . . . 243
7.1.215 reverse . . . 243
7.1.216 right . . . 244
7.1.217 rmdir . . . 244
7.1.218 Rotate . . . 245
7.1.219 Rpos . . . 245
7.1.220 Rtab . . . 246
7.1.221 rtod . . . 246
7.1.222 runerr . . . 247
7.1.223 save . . . 248
7.1.224 Scale . . . 248
7.1.225 seek . . . 248
7.1.226 select . . . 249
7.1.227 send . . . 249
7.1.228 seq . . . 250
7.1.229 serial . . . 250
7.1.230 set . . . 251
7.1.231 setenv . . . 251
7.1.232 setgid . . . 252
7.1.233 setgrent . . . 252
7.1.234 sethostent . . . 253
7.1.235 setpgrp . . . 253
7.1.236 setpwent . . . 253
7.1.237 setservent . . . 254
7.1.238 setuid . . . 254
7.1.239 signal . . . 254
7.1.240 sin . . . 255
7.1.241 sort . . . 256
7.1.242 sortf . . . 257
7.1.243 Span . . . 257
7.1.244 spawn . . . 258
7.1.245 sql . . . 259
7.1.246 sqrt . . . 260
7.1.247 stat . . . 261

7.1.248 staticnames . . . 262
7.1.249 stop . . . 262
7.1.250 StopAudio . . . 263
7.1.251 string . . . 263
7.1.252 structure . . . 264
7.1.253 Succeed . . . 265
7.1.254 Swi . . . 265
7.1.255 symlink . . . 265
7.1.256 sys_errstr . . . 266
7.1.257 system . . . 266
7.1.258 syswrite . . . 267
7.1.259 Tab . . . 267
7.1.260 tab . . . 267
7.1.261 table . . . 268
7.1.262 tan . . . 268
7.1.263 Texcoord . . . 270
7.1.264 Texture . . . 270
7.1.265 TextWidth . . . 271
7.1.266 Translate . . . 271
7.1.267 trap . . . 271
7.1.268 trim . . . 272
7.1.269 truncate . . . 272
7.1.270 trylock . . . 273
7.1.271 type . . . 273
7.1.272 umask . . . 274
7.1.273 Uncouple . . . 275
7.1.274 unlock . . . 275
7.1.275 upto . . . 275
7.1.276 utime . . . 276
7.1.277 variable . . . 276
7.1.278 VAttrib . . . 277
7.1.279 wait . . . 277
7.1.280 WAttrib . . . 278
7.1.281 WDefault . . . 281
7.1.282 WFlush . . . 281
7.1.283 where . . . 282
7.1.284 WinAssociate . . . 282
7.1.285 WinButton . . . 283
7.1.286 WinColorDialog . . . 283
7.1.287 WindowContents . . . 284
7.1.288 WinEditRegion . . . 285
7.1.289 WinFontDialog . . . 285
7.1.290 WinMenuBar . . . 286
7.1.291 WinOpenDialog . . . 286
7.1.292 WinPlayMedia . . . 287
7.1.293 WinSaveDialog . . . 287
7.1.294 WinScrollBar . . . 287
7.1.295 WinSelectDialog . . . 288
7.1.296 write . . . 288
7.1.297 WriteImage . . . 289
7.1.298 writes . . . 289
7.1.299 WSection . . . 290
7.1.300 WSync . . . 291

8 Keywords 293
8.1 Unicon Keywords . . . 293
8.1.1 &allocated . . . 293
8.1.2 &ascii . . . 294
8.1.3 &clock . . . 295
8.1.4 &col . . . 295
8.1.5 &collections . . . 296
8.1.6 &column . . . 297
8.1.7 &control . . . 297
8.1.8 &cset . . . 298
8.1.9 ¤t . . . 299
8.1.10 &date . . . 299
8.1.11 &dateline . . . 300
8.1.12 &digits . . . 301
8.1.13 &dump . . . 301
8.1.14 &e . . . 302
8.1.15 &errno . . . 302
8.1.16 &error . . . 303
8.1.17 &errornumber . . . 304
8.1.18 &errortext . . . 304
8.1.19 &errorvalue . . . 305
8.1.20 &errout . . . 306
8.1.21 &eventcode . . . 306
8.1.22 &eventsource . . . 307
8.1.23 &eventvalue . . . 308
8.1.24 &fail . . . 309
8.1.25 &features . . . 310
8.1.26 &file . . . 311
8.1.27 &host . . . 311
8.1.28 &input . . . 312
8.1.29 &interval . . . 312
8.1.30 &lcase . . . 313
8.1.31 &ldrag . . . 313
8.1.32 &letters . . . 314
8.1.33 &level . . . 315
8.1.34 &line . . . 315
8.1.35 &lpress . . . 316
8.1.36 &lrelease . . . 316
8.1.37 &main . . . 317
8.1.38 &mdrag . . . 318
8.1.39 &meta . . . 318
8.1.40 &mpress . . . 319
8.1.41 &mrelease . . . 320
8.1.42 &now . . . 321
8.1.43 &null . . . 321
8.1.44 &output . . . 322
8.1.45 &phi . . . 322
8.1.46 &pi . . . 323
8.1.47 &pick . . . 323
8.1.48 &pos . . . 324
8.1.49 &progname . . . 325
8.1.50 &random . . . 325
8.1.51 &rdrag . . . 326
8.1.52 ®ions . . . 327

8.1.53 &resize . . . 328
8.1.54 &row . . . 328
8.1.55 &rpress . . . 329
8.1.56 &rrelease . . . 330
8.1.57 &shift . . . 331
8.1.58 &source . . . 331
8.1.59 &storage . . . 332
8.1.60 &subject . . . 333
8.1.61 &time . . . 334
8.1.62 &trace . . . 334
8.1.63 &ucase . . . 335
8.1.64 &version . . . 335
8.1.65 &window . . . 336
8.1.66 &x . . . 337
8.1.67 &y . . . 337

9 Preprocessor **339**
9.1 Unicon preprocessor . . . 339
9.1.1 $define . . . 339
9.1.2 $else . . . 340
9.1.3 $endif . . . 340
9.1.4 $error . . . 340
9.1.5 $ifdef . . . 341
9.1.6 $ifndef . . . 342
9.1.7 $include . . . 343
9.1.8 $line . . . 343
9.1.9 $undef . . . 345
9.1.10 #line . . . 346
9.2 Predefined symbols . . . 346
9.2.1 Substitution symbols . . . 347
9.3 EBCDIC transliterations . . . 348

10 Development Tools **349**
10.1 Unicon tools . . . 349
10.1.1 The unicon command . . . 350
10.1.2 The icont command . . . 352
10.1.3 The iconx command . . . 353
10.1.4 Coding conventions and style . . . 356
10.2 Supporting tools . . . 357
10.2.1 make . . . 358
10.2.2 ui . . . 359
10.2.3 UDB . . . 359
10.3 The Icon Virtual Machine . . . 360
10.3.1 ucode . . . 360
10.3.2 icode . . . 364
10.3.3 The Implementation of Icon and Unicon . . . 371
10.4 Editors . . . 371
10.4.1 Vim . . . 371
10.4.2 Emacs . . . 376
10.4.3 Evil . . . 377

11 String Processing **379**
11.1 Unicon String Processing . . . 379
11.1.1 String Scanning . . . 379

12 Patterns **381**
12.1 Unicon Pattern data . . . 381
12.1.1 SNOBOL patterns . . . 381
12.1.2 Regular expressions . . . 383
12.1.3 Pattern operators . . . 383
12.1.4 Regex syntax . . . 383

13 Objects **385**
13.1 Unicon Objects and Classes . . . 385
13.1.1 SOLID . . . 385
13.1.2 IDOL . . . 386

14 Graphics **387**
14.1 Unicon graphics . . . 387
14.2 Colours . . . 387
14.2.1 Unicon colour scheme . . . 389
14.3 Drawing . . . 389
14.4 Events . . . 389
14.5 Attributes . . . 389
14.6 Vidgets . . . 391
14.6.1 On names . . . 392
14.7 Unicon GUI . . . 392
14.8 Plot coordinate pairs . . . 392

15 Database **395**
15.1 Unicon databases . . . 395
15.1.1 Tables . . . 395
15.1.2 DBM . . . 396
15.1.3 ODBC . . . 396

16 Networking **401**
16.1 Unicon Networking . . . 401
16.1.1 Network mode . . . 401
16.1.2 Message mode . . . 401
16.1.3 Verify . . . 406
16.1.4 More HTTPS . . . 406
16.2 CGI . . . 407
16.2.1 CGI 1.1 . . . 410
16.3 AJAX . . . 410
16.4 PHP . . . 411

17 Threading **413**
17.1 Unicon threading . . . 413
17.1.1 Thread creation . . . 413
17.1.2 Hello, threads . . . 413
17.2 Multi-tasking . . . 415

18 Features **417**
18.1 Unicon features . . . 417
18.1.1 Keyboard functions . . . 417
18.1.2 Pseudo terminals . . . 418
18.1.3 libz compression . . . 418

19 Documentation **421**
19.1 Documenting Unicon programs . . . 421

19.1.1 Unicon headers . . . 421
19.1.2 unidoc . . . 423
19.2 Unicon Technical Reports . . . 423

20 Testing **425**
20.1 Testing Unicon . . . 425
20.1.1 Unit testing . . . 425
20.1.2 unitest . . . 426

21 Debugging **435**
21.1 Debugging Unicon . . . 435
21.1.1 Trace . . . 435
21.1.2 UDB . . . 436

22 Execution Monitoring **437**
22.1 Unicon monitoring . . . 437
22.1.1 Visualization . . . 442

23 Performance **443**
23.1 Unicon performance . . . 443
23.2 Summing integers . . . 444
23.2.1 Unicon . . . 444
23.2.2 Python . . . 444
23.2.3 C . . . 445
23.2.4 Ada . . . 445
23.2.5 ALGOL . . . 446
23.2.6 Assembler . . . 446
23.2.7 BASIC . . . 447
23.2.8 C (baseline) . . . 448
23.2.9 COBOL . . . 448
23.2.10 D . . . 448
23.2.11 ECMAScript . . . 449
23.2.12 Elixir . . . 449
23.2.13 Forth . . . 450
23.2.14 Fortran . . . 450
23.2.15 Groovy . . . 451
23.2.16 Java . . . 451
23.2.17 Lua . . . 452
23.2.18 Neko . . . 452
23.2.19 Nickle . . . 453
23.2.20 Nim . . . 453
23.2.21 Perl . . . 454
23.2.22 PHP . . . 454
23.2.23 Python . . . 455
23.2.24 REBOL . . . 455
23.2.25 REXX . . . 455
23.2.26 Ruby . . . 456
23.2.27 Rust . . . 456
23.2.28 Scheme . . . 456
23.2.29 Shell . . . 457
23.2.30 S-Lang . . . 457
23.2.31 Smalltalk . . . 458
23.2.32 SNOBOL . . . 458
23.2.33 Tcl . . . 458
23.2.34 Vala . . . 459

23.2.35 Genie . . . 459
23.2.36 Unicon loadfunc . . . 460
23.2.37 Summary . . . 461
23.3 Development time . . . 464
23.3.1 Downsides . . . 464
23.4 Unicon Benchmark Suite . . . 465
23.4.1 run-benchmark . . . 465

24 Icon Program Library **467**
24.1 IPL . . . 467
24.1.1 Exploring the IPL . . . 467
24.1.2 Useful procedures . . . 468
24.2 Programming Corner . . . 476
24.2.1 Newsletter Catalog . . . 477

25 Unicon Class Library **493**
25.1 UCL . . . 493
25.1.1 JSON . . . 493

26 Use case scenarios **495**
26.1 Scenarios . . . 495
26.1.1 Experience . . . 495
26.1.2 Processing . . . 496
26.1.3 Development . . . 496
26.1.4 Devices . . . 497
26.1.5 Sciences . . . 497
26.1.6 Mathematics . . . 497
26.1.7 Finance . . . 497

27 Programs **499**
27.1 Sample programs and integrations . . . 499
27.1.1 S-Lang . . . 499
27.1.2 COBOL . . . 505
27.1.3 Duktape . . . 511
27.1.4 mruby . . . 517
27.1.5 ficl . . . 519
27.1.6 Lua . . . 526
27.1.7 Fortran . . . 530
27.1.8 Assembler . . . 533
27.1.9 vedis . . . 535
27.1.10 libcox . . . 538
27.1.11 PH7 . . . 541
27.1.12 UnQLite . . . 546
27.1.13 REXX . . . 556
27.1.14 Internationalization and Localization . . . 561
27.1.15 libsoldout markdown . . . 564
27.1.16 ie modified for readline . . . 566
27.1.17 SNOBOL4 . . . 568
27.1.18 fizzbuzz . . . 582
27.1.19 eval . . . 583
27.1.20 unilist . . . 585
27.1.21 tcc . . . 589

28 Multilanguage **593**
28.1 Multilanguage programming . . . 593

28.1.1 loadfunc . . . 594
28.1.2 C Native . . . 594
28.1.3 libffi . . . 608
28.1.4 baconffi . . . 619

29 Theory **623**
29.1 Computer programming theory . . . 623
29.1.1 Unicon and Computer Science . . . 623
29.1.2 Proving Unicon . . . 623
29.1.3 f = ma . . . 624

30 RosettaCode **625**
30.1 Unicon on rosettacode.org . . . 625
30.1.1 Some samples . . . 625

31 Notes **633**
31.1 ABI . . . 633
31.2 API . . . 633
31.3 ASCII . . . 633
31.4 BaCon . . . 634
31.5 BASIC . . . 634
31.6 C . . . 634
31.7 comprehension . . . 634
31.8 Creative Commons . . . 635
31.9 COBOL . . . 635
31.10 DSO . . . 636
31.11 Expect . . . 636
31.12 Farberisms . . . 636
31.13 Forth . . . 636
31.14 GCC . . . 636
31.15 GNU . . . 637
31.16 GnuCOBOL . . . 637
31.17 Graphics Programming in Icon . . . 637
31.18 Help Wanted . . . 637
31.19 Icon . . . 637
31.20 Icon version 9 . . . 638
31.21 JSON . . . 638
31.22 Locale . . . 638
31.23 mutable colour . . . 638
31.24 PHP . . . 639
31.25 POSIX . . . 639
31.26 REXX . . . 639
31.27 Richard Stallman . . . 639
31.28 SEXI . . . 640
31.29 serif . . . 640
31.30 SNOBOL . . . 640
31.31 Tcl/Tk . . . 641
31.32 Tectonics . . . 641
31.33 uniclass . . . 642
31.34 Unix epoch . . . 642
31.35 VAX . . . 642
31.36 VimL . . . 642
31.37 VMS . . . 642
31.38 VOIP . . . 642

31.39 y2k . . . 643
31.40 Year 2038 problem . . . 643
31.40.1 Unicon epoch rollover risk is nearly zero . . . 643

32 ChangeLog **645**

33 License **649**

34 Blog entries **661**
34.1 Execution Monitoring . . . 661
34.2 https with no newline in response . . . 661
34.3 https with redirection . . . 661
34.4 Unicon bug fixing . . . 661
34.5 SNOBOL pattern example . . . 662
34.6 Unicon loadfunc . . . 662
34.6.1 In other news . . . 662
34.7 Impressed . . . 662
34.7.1 The UP docs . . . 663
34.7.2 Unit testing . . . 663
34.8 Invited . . . 665
34.8.1 Recent news . . . 665
34.9 SourceForge . . . 669
34.9.1 Recent news . . . 670
34.10 Unicon FFI . . . 671
34.10.1 libffi . . . 672
34.10.2 C calling Unicon . . . 675
34.11 Unicon Programming docset . . . 676

Index **683**

Author Brian Tiffin

Date Oct 27, 2019. Started 2016-07-29

Status Work in progress, 0.6.149

Unicon The Unified extended dialect of Icon. A very high level programming language.

Abstract Unicon Programming is meant to augment the other documentation available for the Unicon programming language. It will focus on code examples, with downloadable source listings, and try to fill in details that other Unicon project documents have not yet completely covered.

Acknowledgement With many thanks to the Unicon development team, and the Icon project.

Copyright and License Copyright © 2016-2019 Brian Tiffin

This file is part of the Unicon Programming document.

This documentation is free; you can redistribute and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU General Public License along with this document. If not, see <http://www.gnu.org/licenses/>.

While this work is covered under the GNU GPL, the included code listings often come with explicit copyright and licensing notices. When extracted from this documentation, any explicit notices take precedent.

Unicon artwork, Nico the Unicorn, by Serendel Macphereson.

Feedback Correction requests and other feedback can be posted to the Unicon discussion forums at https://sourceforge.net/p/unicon/discussion.

CHAPTER

ONE

UNICON PROGRAMMING

1.1 The Unicon Programming Language

Unicon, Unified extended **Icon**, by *Clinton Jeffery*, Shamim Mohamed, Jafar Al Gharaibeh, Ray Pereda, Robert Parlett, and team.

Icon, an iconoclastic programming language. The design of Icon was led by Dr. Ralph Griswold at the University of Arizona, starting in 1977, as a structured successor to **SNOBOL** and refinement of **SL5**. Icon includes a rich set of datatypes and operators, generators, goal directed evaluation with implicit backtracking, text scanning, integrated graphics, and other complementary very high level features. The design facilitates powerfully concise, yet surprisingly readable program source code. All with an admirable performance profile.

Unicon extends the feature set of Icon by adding classes, a *POSIX* layer, networking, and a host of other modern development productivity enhancements. Development of Unicon is led by *Clinton Jeffery*.

1.1.1 Well met

```
#
# Introductory Unicon
#
procedure main()
    write("Hello, world")
end
```

Example run:

```
prompt$ unicon -s introductory.icn -x
Hello, world
```

And that is the `Hello, world` program for `Unicon`, proof that the basic system is properly installed and functioning.

`Unicon` deserves a little more than that, so this next introduction lets us see what features are included in the installation.

```
#
# Introductory Unicon
#
procedure main()
    write("Hello, world")

    write("\nFeatures in this ", &version)
    every write(&features)
end
```

This time, all the glorious command line details are also included, but don't worry, this will all be second nature before you know it.

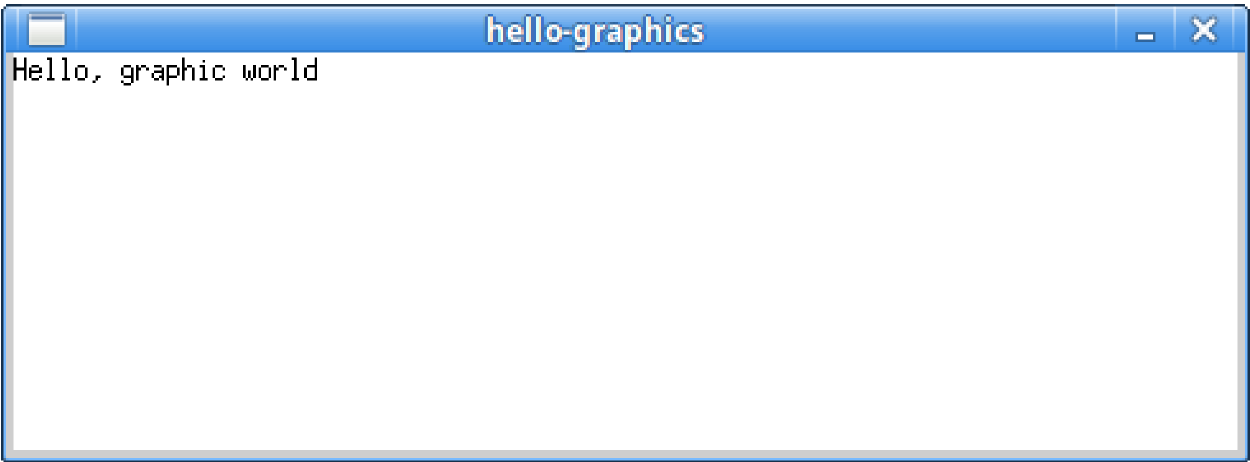
```
prompt$ unicon introductory-features.icn -x
Parsing introductory-features.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O introductory-features.icn /tmp/uni11518157
Translating:
introductory-features.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  introductory-features.u -x
Linking:
Executing:
Hello, world

Features in this Unicon Version 13.1.  August 19, 2019
UNIX
POSIX
DBM
ASCII
co-expressions
native coswitch
concurrent threads
dynamic loading
environment variables
event monitoring
external functions
keyboard functions
large integers
multiple programs
pattern type
pipes
pseudo terminals
system function
messaging
graphics
3D graphics
X Windows
libz file compression
JPEG images
PNG images
SQL via ODBC
Audio
secure sockets layer encryption
CCompiler gcc 5.5.0
```

```
Physical memory: 7808401408 bytes
Revision 6034-743ffd1
Arch x86_64
CPU cores 4
Binaries at /home/btiffin/unicon-git/bin/
```

So, yeah, *wow*, there is a lot of neat stuff in there.

Seeing as there are *graphics*, here is a graphical hello.

```
#
# hello-graphics.icn, graphical hello demonstration
#
procedure main()
    w := open("hello-graphics", "g")
    writes(w, "Hello, graphic world")
    Event(w)
    close(w)
end
```



That example is the basics for graphical output in Unicon.

As simple as console output, to a graphical window.

The next listing is a little more sophisticated, more appropriate for the automatic image captures shown throughout this docset. Note the different font choice.

```
#
# hello-graphics.icn, graphical hello demonstration
#
procedure main()
    if find("graphics", !&features) then stop("no graphics, sorry")

    w := open("hello-graphics", "g", "font=12x24", "rows=1",
              "columns=22", "canvas=hidden") |
         stop("no graphics window")

    writes(w, " Hello, graphic world ")

    WSync(w)
    WriteImage(w, "../images/hello-graphics.png")
    close(w)
end
```

Giving:

```
prompt$ unicon -s hello-graphics.icn -x
```

Hello, graphic world

1.1.2 Overview of Unicon

Note: Unicon encompasses *Icon*. Many things attributed to Unicon here, may have originated in Icon. Attempts will be made to clearly state when Icon is the origin. *Everything* from Icon is in Unicon, and then extended out by the Unicon team. This *Unicon centric view* is not meant as a slight to the Icon project, but merely a convenience when writing about Unicon.

A quick tour of Unicon syntax and semantics.

Unicon is a very high level, network and graphic savvy, multiple paradigm programming language. Elements of imperative, procedural, concurrent, object oriented, reflective, and iterative programming are present and can be freely mixed. Support library sources add more paradigms in a limited fashion, such as interactive, list, logic and XML based programming.

Unicon counts as a curly-bracket language.

The reference implementation is written in C (and a specialized C variant called rtt).

A Java based variant is available, Junicon, http://junicon.sourceforge.net/site/index.html, but this version is not yet complete in terms of runtime function support. Junicon translates core Unicon to Groovy (for an interactive interpreter mode) or Java (for regular compiled mode) on way to the JVM.

The rest of this page describes the reference implementation, although most of the material would be the same for Junicon.

The Icon programming language lies at the heart of Unicon. Icon has been in development in various forms since 1977, with predecessors including SNOBOL and SL5, which date back even further to the 1960s. Unicon builds on the Icon feature set.

Icon is marked as complete, and is in maintenance mode. Unicon continues to evolve, and is in active development.

Unicon includes a vast array of features. One can look at that as tool *bloat*; but a better, more productive, point of view is to look at Unicon as a full tool *box*. A single, unified entity containing a well integrated collection of mechanisms, working together to become greater than the sum of the parts. (Without bulging at the seams).

Virtual Machine

Unicon source can be compiled for a cross platform virtual machine `ucode` readable source and `icode` bytecode.

Or Unicon source can be natively compiled to binary executable form using a `-C` command line option. Native compiles do not yet support all Unicon features.

Expressions

Unicon is an expression language; everything is an expression, including control flow reserved words. Unicon expressions produce a value or fail. Expressions strive to succeed and will attempt alternatives, within the bounds of an expression context, before producing a value or eventually failing. This is termed **goal-directed evaluation**. Goal-directed evaluation works hand in hand with **generators**, *mentioned below* and works best when backtracking is kept in mind.

A nice one word synopsis of Unicon is **determination**.

Values and Types

There are 6 basic immutable types in Unicon:

- null (with a special keyword name, *&null*)
- integer
- real
- string
- cset (short for character set)
- pattern

Unicon is an 8 bit clean *ASCII* system, zero valued bytes are allowed in strings and character sets. Unicon uses one relative indexing in most cases, the first element is 1, not 0.

Values have a type, variables do not. Variables can be freely assigned any value. All values are first class values in Unicon, and can be assigned to variables, passed as arguments, and returned as results.

Strings are immutable, and subscript assignment or concatenate operations will produce a new string, not change an existing string value in place.

Unicon has no pointers, but internally manages references to values.

There are high level, mutable, structured data types:

- list
- set
- table
- record

Unicon includes class and method definitions that internally use record structures for compile time and runtime management.

Unicon also has a set of support data types:

- co-expression (including synchronous tasks and concurrent threads)
- window
- file

Variables not bound to a value default to `null`, represented by the keyword *&null*.

Although Unicon allows variable use without declaration, explicit control over `local`, `global` and `static` scope is always preferred and can avoid unintended name resolution behaviours. This can be a problem when changes to a program introduce a new `global` and previously undeclared, assumed `local`, variables are no longer local to a particular procedure.

Undeclared variable warnings can be produced with the compiler `-u` switch.

Numeric calculations will raise a runtime error when non numeric values (or values that cannot be unambiguously converted to numeric) are included in a computation.

Runtime errors normally cause an abnormal program termination with back trace, but can be transformed and treated as failure conditions through the use of a special keyword, `&error`.

Automatic type coercion is attempted between integer, real, string, and character set values as required by the surrounding expression context.

`null` and the empty string are not promoted to zero in calculations, but cause an aforementioned runtime error.

Division with all integer values produces an integer result. Division by zero causes a runtime error. Zero to the power of zero is deemed an undefined number and causes a runtime error.

Integer literals can include a base radix[1] prefix, and real number literals can include an exponent notation[2] suffix. Base 10 integer literals can also include a scaling suffix. Integer literal scaling includes (case insensitive):

- K scale by 1024, kilo
- M scale by 1024^2, mega
- G scale by 1024^3, giga
- T scale by 1024^4, tera
- P scale by 1024^5, peta

Integer literals can be of arbitrary magnitude.

Internally, large integers are a special type and values exceeding native processor signed value bit widths (commonly 15, 31, or 63 bits) are automatically managed by Unicon with special software routines. (These large integer routines are much slower than native hardware integer computation circuitry and are, by necessity, non-atomic when concurrency is involved). *Although transition from native to large integer handling is automatic, and does not usually require any attention when programming, there may be times when explicit handling is called for.* There is a sample in the *Program Library* for testing when a number has been promoted to large integer form.

String literals use double quotes, i.e. `"a string"`.

Character set literals, called *Cset* in Unicon, use apostrophe (single quotes), i.e. `'aaaxyz'`. Please note, `cset` data is a *set*. That previous example will only contain a single occurrence of the `a` character at runtime.

String literals can be extended across lines by ending the source line with an underscore. Leading spaces on the following line are ignored.

Newlines and other non printable characters can be included in string literals using backslash escape notation.

- \b `backspace`
- \d `delete`
- \e `escape`
- \f `form feed`
- \l `line feed`
- \n `newline`
- \r `carriage return`
- \t `tab`
- \v `vertical tab`
- \' `apostrophe`
- \" `double quote`

[1] Integer literal base radix prefix is specified by a base ten number of a base from 2 through 36 followed by a literal `r` or `R`. For example 8r700 is octal notation, 2r101010 is binary notation.

[2] Real number exponent suffixes are expressed as Ennn, where nnn is a base ten value in the range of -308 to 308 (for 64 bit double precision floating point). The E is case insensitive, `e` or `E`.

- \ backslash
- \ooo octal byte value
- \xhh hexadecimal byte value
- \^x Control-x, x being @, A-Z (or a-z), [, \,], ^, _

Regular expression literals use open and close angle brackets (chevrons), i.e. <[a-c]>, matches a or b or c during a match expression.

List literals are enclosed in square brackets, each element separated by commas. Elements can be any type, including lists or other high level aggregate types. Empty lists are expressed as []. Dangling commas at the end of a list literal create an extra null item in the list, as do empty intermediate entries.

List comprehensions use [: expr :] bracket colon/colon bracket syntax. The resulting list will be all the values generated by expr.

Scope

Unicon has local, static (local), and global scope. There is also support for package variable name spaces.

Undeclared variables inside procedures default to local scope. This can be an issue during long term maintenance. If a global variable is later declared with the same name, an "undeclared" procedure variable is now declared and the procedure may inadvertently change a global value with undesirable outcome.

Long term Unicon development wisdom means fully declaring all local variables even though it is a default semantic.

unicon -u will cause a compile time error when detecting use of an undeclared local variable. You will need to run this pass before adding any new global variables for maximum effect in catching inadvertent name conflicts.

Static variables are local to a procedure but retain values between invocations of the procedure. The *initial* clause is handy for preparing static values when more than simple assignment is required (loading tables from disk, for instance). The initial expression is only evaluated once per procedure during the lifetime of a program.

Procedure names are global. Procedure parameter identifiers are always explicitly local to the procedure, overriding any global identifier lookup. *For those rare times when a procedure needs to access a global variable with the same name as a parameter, the variable built in reflective function can be used.*

Record constructor descriptors are global. Although the inner field names are somewhat local to the record (there can be duplicate member names across different records). All instances of a record type have the same field names.

Class definitions are global. Methods follow the same inner-local sharing as records. All instances of a class have access to all the variable and method names of the class, but different classes may use duplicate names inside.

The *package* declaration specifies that all global symbols within the source unit belong to a named package. All global declarations (variables, procedures records and classes) are invisible outside the package, unless explicitly imported. This feature, along with class definitions, uses an external compile time database to help manage namespace and import linkage of resources within a package.

Package management in Unicon is designed as a "programming in the large" feature. There are some externally influenced management issues to be aware of to ease programming at this scale. See the *package* entry and *Objects* for details.

Functions

Unicon builds with over 300 predefined functions[3].

[3] See *function* for details on listing predefined functions.

User defined functions[4] are called `procedures` in Unicon. These are all `expressions` that produce values or fail.

Unicon does not have a so called `void` type; there is success and a value (possibly `null`) or failure.

Note: Throughout this documentation, the terms *function* and *procedure* are used somewhat interchangeably given the context of Unicon.

In comparison to some other programming languages, such as Pascal where *by definition* procedures do not return values. Or in a mathematical sense where functions do not cause side effects. Neither of those cases hold in current or historic Unicon terminology or jargon. *Nothing will stop you from writing code to meet those stricter definitions if so desired, but they are not commonly used Unicon terms of art*.

Procedure names can be overridden within scoping rules.

As first class values, procedure and function references can be passed as arguments, returned from procedures, and bound to variables.

Procedures are invoked with commonly seen `function(args,...)` parenthesis syntax or `function!list` application.

Programmer defined control structure invocation is also possible with `function{expr1; expr2...}` brace syntax. See *User defined control structures*.

Procedures can also be invoked by string name once enabled (see *invocable*).

More technically, the syntax is actually `expr ()`, where the invocation operator `()` follows an expression, any expression.

When the initial invocation expression in an integer, it becomes a grouped expression and the value determines the parameter to use as the result of the entire comma separated expression group, counted from 1 going left to right. For example: `2(first(), second())` is a grouped expression returning the second result.

Procedures compiled for the Unicon virtual machine can be linked into other Unicon programs with the *link* reserved word. The link phase is fairly smart, and will only include procedures actually used (or set as *invocable*), so there is no worry of overall program bloat when linking to files that contain multiple procedures.

Without an explicit `return` or `suspend`, procedures `fail` by default. That means that procedures that “fall through” to the `end` reserved word, `fail` and produce no value. This has implications that developers should keep in mind. An empty *return* can be used to avoid unexpected nested or chained expression failure.

String scanning

The Unicon language includes a syntax for string scanning, with bookkeeping of subject string and current position managed by the scanning environment.

```
s ? expr
```

`s` is a subject string, and the *expr* scanning expression can be arbitrarily complex. Results can be nested in other scanning expressions.

String scanning can be far more powerful than `regular expressions`. The subject matter does not need to be `regular` or finite or free of context. String scanning expressions can have ever changing contextual state managed within the expression itself, and are free to move the position of interest without constraint. *Theoretically*. You will exceed the limits of RAM and disk space available in the world before you exceed the power inherent in Unicon string

[4] Programmers are sometimes sloppy with the use of the term `function`, *this author is guilty of this technical word crime*. In mathematics a function has no side effects and any given input will always produce the same result. Step wise procedures and algorithms are allowed side effects and state change can influence results. Common vernacular has led to a situation where many (but not all) programmers use the terms `function` and `procedure` interchangeably. “functions” are often not predictably reproducible or free of side effect.

scanning expressions. And if you think you are getting close, Unicon also supports `regular expressions` and SNOBOL style `patterns`, so, the Universe is the limit (and then some).

Generators

Functions and operators can *return* values, or *suspend* values. `return` places a boundary on goal directed evaluation, `suspend` allows the Unicon goal-directed evaluation engine to resume a procedure and attempt alternatives. `return` can be followed by any expression or if omitted, defaults to a `null` value. `suspend` can include an optional *do* phrase that is evaluated when the expression is resumed before following the rest of the normal flow of control. In grand Unicon fashion, the expression that follows `suspend do` can also be a generator.

Co-expressions

Unicon co-expressions are encapsulated expression contexts which allow for parallel evaluation and coroutines, among other power features, like concurrency and multitasking.

Graphics

Unicon builds with 2D, and 3D graphics facilities. Unicon also ships with an object oriented, *class* based, graphical user interface builder. The graphics engine is event driven with features allowing for implicit and explicit event management.

Audio

Unicon supports sound, including a layer for *VOIP* if prerequisites are available during build.

Object oriented

Unicon includes class based object oriented features. This layer is an integration of the earlier IDOL preprocessor into the Unicon core language. Unicon classes are quite simple and yet complex; circular inheritance is supported (*although not overly recommended unless programming routines dealing with things like the wave/particle duality in quantum physics theories*).

Flexibility

Unicon leans to the *more than one way of doing things* school of thinking, but there is also a tendency toward *idiomatic Unicon*, which comes with experience and shared learning by example.

```
# do some thing to each line of an open file
while line := read(f) do {
    thing(line)
}

# same, without bracing
while line := read(f) do thing(line)

# another form with nested expressions
while thing(read(f))

# another form, using the generate elements operator
every line := !f do thing(line)
```

```
# object oriented forms (if f is a class instance encapsulating file)
every f.thing()

# or an alternative
while f.thing()

# others, yes there are other expression forms...

# and perhaps the idiomatic form (opinions may vary)
every thing(!f)
```

You are free to choose. Unicon practitioners will accept all of the above forms as valid source constructs, but that last one sure is concise, sweet looking code.

Unicon expression and operator syntax is so well designed and flexible that even after 40 years of study, advanced programmers are still impressing each other with new techniques. *Ok, maybe that is just this author's impression from initial readings of some of the conversations between the Unicon gurus regarding the secrets yet to discover in Unicon syntax and idioms.*

1.1.3 Building Unicon from source

Now on to getting all this Unicon goodness installed on your computer.

GNU/Linux

In the instructions below, `working-dir`, `install-dir`, and `unicon-projects` are *dealer's choice* directory names (pick a name appropriate for the local system).

Unicon follows the de facto standard `./configure; make; sudo make install` paradigm. There are quite a few options to allow for customized builds. See `./configure --help` for details of all the build options. Enable and disable features and components during the `./configure` phase.

- Get the code

```
prompt$ cd $HOME/working-dir/
prompt$ svn checkout svn://svn.code.sf.net/p/unicon/code/trunk unicon
prompt$ cd unicon
```

- Configure

```
prompt$ ./configure
```

- Make

```
prompt$ make
```

- Prep: (somewhere in `~/.bashrc`[5] add)

[5] There are quite a few options for shell start up scripts. `~/.bashrc` is just one of the options. Your local site setup may use `~/.bash_profile` (which is actually the more current and correct form of interactive shell start up control, but this author has been using `~/.bashrc` for over two decades now, and it's an old, hard to break habit). There are also system wide startup scripts, `/etc/profile` for instance. Shells other than **bash** will have different options, and other operating systems will have completely different details when it comes to setting the **unicon** command path. Consult your local documentation.

```
# add unicon
add-unicon () {
    if [[ ! $PATH =~ unicon ]]; then
        PATH="$HOME/working-dir/unicon/bin:$PATH"
        export PATH
    fi
}
```

You will now have an `add-unicon` shell function available. Set it to load during startup or type the command when you want to have `unicon` tools available.

Or simply use `make install` to install Unicon to the configured locations.

- Invoke

```
prompt$ source ~/.bashrc
prompt$ add-unicon
```

The above step will not be necessary if `make install` was used.

```
prompt$ cd $HOME/unicon-projects/
prompt$ cat hello.icn
```

```
procedure main()
    write("Hello, world")
end
```

```
prompt$ unicon -s hello.icn -x
Hello, world
```

Party, *like it's 2099*.

- Install

The Unicon source build creates a directory structure and command sequence so the only thing you really need to do is set the PATH to the `unicon/bin` directory. This layout knows where the link libraries are, include subdirectories, etcetera. If you like a little more control over the placement, then read through `$HOME/working-dir/unicon/Makefile` for details on setting up an install with:

```
prompt$ sudo make install
```

If using the default prefix (`/usr/local` usually), the above command (usually) requires `sudo` or run the command as the root super user so the installer has permissions for the system directories.

There are older Make rules to control this as shown below, but using the de facto standard DESTDIR and PREFIX settings are now recommended.

```
prompt$ make Install dest=install-dir
```

You can change the `add-unicon` function to set the path to this shiny new `install-dir/bin`, (or, just export that bin path). The `add-unicon` function is handy when initially trying out a Unicon build, but you will probably find that setting the Unicon path becomes part and parcel of your shell login. *I never type add-unicon anymore, the Unicon path setting is built into the shell startup now.*

- Package repository

As of Unicon 13.1, Jafar has created package build rules, which can be used to build your own binary packages. Jafar has also created a PPA (Personal Package Archive), which can be used to automatically stay up to date with binary builds of Unicon on Ubuntu. No sourcing around required.

- sudo add-apt-repository ppa:jafaral/unicon
- sudo apt update
- sudo apt install unicon
- Time to enjoy programming in Unicon

Things should now be ready to go; so on with the show...

But first, a brief word from our sponsors ... the people that created Unicon, not advertising, this is free documentation.

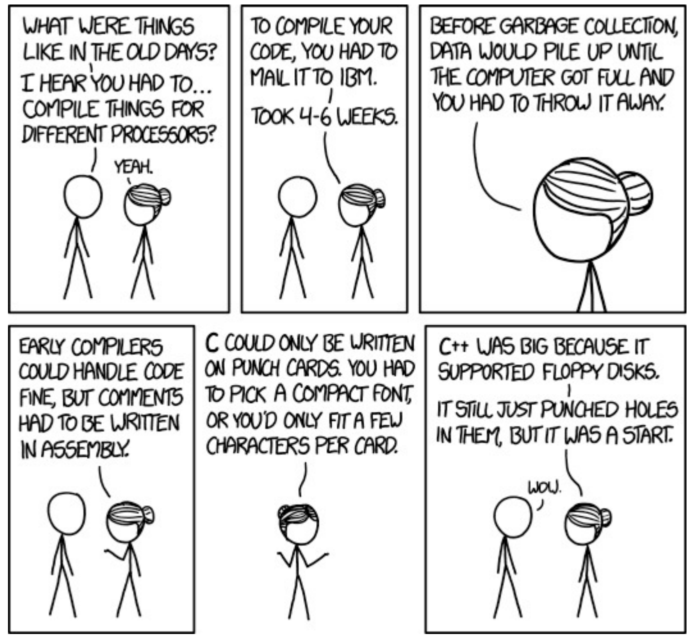
Feel free to skip over some of this *very exciting* background and history information and get right into *Datatypes*, or hit *Development Tools* and leap right into developing, compiling, and running `Unicon` programs.

1.2 History

In the early 1960's, *Ralph Griswold*, *David Farber*, and *Ivan Polonsky*, formed the core team that designed *SNOBOL*.

What follows is part lore, part fact, the facts somewhat lost in the mists of time.

SNOBOL was a renaming of *SEXI*, String EXtraction Interpreter, a name that eventually caused the core team pause, when clerical staff would be smirking while handing over printouts with title pages of SEXI Ralph Griswold, and SEXI Dave Farber. The story goes that SNOBOL, StriNg Oriented symBOlic Language, was a hard found name after hearing "This program doesn't have a snowball's chance in ..." and SNOBOL was so named, and backronymed later.



XKCD http://xkcd.com/1755/ by Randall Munroe *CC BY-NC 2.5*

Less folklore, more factful, now.

Icon was a continuation of the string manipulation and pattern matching strengths of SNOBOL but with a design more in line with ALGOL source code form and other structured programming languages of the mid 1970s.

Icon is just a name, coined before graphical desktops, and before icons as we now know them.

Icon pioneered many key concepts that still influences programming language design and development, and in some ways might be seen as surpassing even the most modern language feature sets. Generators, goal-directed evaluation, chained expressions, string scanning and pattern matching to name a few.

Dr. Griswold had less concern for the commercial success of Icon than the educational potentials of the language. The sources are in the Public Domain, along with the main books about Icon. This may be part of the reason for Icon remaining a relatively obscure programming system; lack of commercial backing meant that there was little vested corporate interest in developments, while at the same time garnering great respect from the professionals and students that got a chance to write code in Icon. Icon was developed, from the ground up, to be very cross-platform, but was not designed to seed any corporate empires.

This author's first exposure was in the mid 1980s. Having convinced management to allow some after hours work on another team's VAX/VMS system. They had a C compiler. We had *VMS* and *Forth*, but the other team had a DEC C compiler. I spent a couple of evenings, off the clock, building an early version of *GCC*. That became our team's C compiler. While we were a Forth shop, having access to C was a boon for some side projects. A few other evenings later, still off the clock, I built a version of Icon v6 for the *VAX*. That soon became an even greater boon for other side and support projects. Icon can easily chew through utterly complex, raw and structured data, outputting formats that suit the need of the moment. Icon was put to great use. As a side benefit, *Farberisms*, *as the application message of the day*, was very well received by the telco engineers we supported at the time.

A few years later, and we had VAX/VMS upgraded with X11 (DECWindows) so the install of Icon was updated with graphics support built in. That is when Clint Jeffery's name started making the rounds with the team.

There were early extensions built for Icon; IDOL an object oriented layer, being one of them. That and other extensions, in particular the POSIX features, were morphed into early versions of Unicon by Clint Jeffery and friends.

1.2.1 The Icon roots

The University of Arizona Computer Science Department is in charge of the Icon project, and it remains the Icon project. Out of respect for the late Dr. Griswold and his desire to freeze the Icon feature set, *Icon* is in maintenance mode, a completed project. Releases are now only produced for minor updates or when surrounding C compiler and operating system changes make it a necessity for clean Icon source code compiles.

1.2.2 And then Unicon

Unicon is not an official updated version or release of Icon, it is a separate project. Unicon builds on the public domain *Icon version 9* sources, and extends those sources out from a very mature, stable code base. Core Unicon is a mature product yet still in development. Clinton Jeffery has been involved with the lead Icon and Unicon development teams since the early days. Now, in 2016, Unicon has established itself well. A feature rich, actively developed programming language, ready and able to take on the internet, and pretty much any problem deserving an automated solution.

Jafar Al-Gharaibeh celebrated his 10th anniversary with the project on 2016-08-06. Nice. *Clinton Jeffery* has got to be approaching a 30th, or more, Icon years included. Active developments abound. SNOBOL and regex patterns added to augment string scanning, all the network messaging protocols, audio and VOIP, and continual polish and new features. *Robert Parlett* has created some terrific code in the `uni` directory that ships with the source code, and highlights the object oriented features of the system. Unicon is a stable, yet still expanding programming environment. Release 13 alpha is looking awesome.

Side blurb: Almost all the samples in this document are captured while the documentation markup is processed into HTML and PDF (and the other output forms). The early pages, August 2016, are all processed with builds of Unicon

Fig. 1.2: *Unicon art, by Serendel Macphereson*

13 alpha, pulled live from the SVN source repository on a regular basis, and rebuilt, before each writing session. Stable. (*Or if not stable, you will be reading a book full of bugs and broken examples. I bet you are not reading a book full of build bugs. There* might *will be some mistakes, but blame the document, not Unicon, for those failures.*)

Note: Please be aware that this document is written by an interested programmer, not by the good folk that work on the internals. Unicon is already a very well documented system, and these pages are meant to augment and by no means replace any previous efforts. If there are discrepancies, treat this document with suspicion, and the original source as likely correct and more accurate.

See

- http://unicon.sourceforge.net/ubooks.html
- http://unicon.sourceforge.net/reports.html

and

- https://www2.cs.arizona.edu/icon/

for access to a vast array of high and low level notes, guides, manuals, references, thesis, and other documentation. Hint hint, TR78-3 is cool. 1978. The Icon Overview.

- https://www2.cs.arizona.edu/icon/ftp/doc/tr78_3.pdf

1.2.3 People

Clinton Jeffery

Team lead for the development of the Unicon programming language.

University of Idaho, Computer Science Department.

Shamim Mohamed

Worked with Clinton on the original creation of Unicon; POSIX layer.

Federico Balbi

Unicon contributing developer; ODBC interface.

Robert Parlett

Unicon contributing developer; Packages, tools, class libraries.

Jafar Al-Gharaibeh

Unicon contributing developer; concurrency, 3D graphics.

Qutaiba Mahmoud

Added support for pseudo terminals in Unicon.

Ralph Griswold

Dr. Ralph Griswold, 1934-2006

Computer scientist, designed and developed SNOBOL, SL5, and Icon

https://en.wikipedia.org/wiki/Ralph_Griswold

David Farber

Professor of computer science.

https://en.wikipedia.org/wiki/David_J._Farber

Famous for many things in the field, including *Farberisms*.

Ivan Polonsky

Computer scientist, designed and developed SNOBOL.

The Unicon Citizen list

This list is taken from the Unicon project page, at

http://unicon.sourceforge.net/citizens.html

It is separately maintained, and these people deserve the pride of place within the Unicon ecosystem. The Citizen list is included here for completeness, but may not always be in synch with the list given above and will include duplicates.

A future release of this docset will merge the two lists and hopefully expand on the entries, as these are the people that we all owe a round of thanks to when programming in Unicon.

Federico Balbi Developer of the ODBC/SQL interface.

Nolan Clayton Contributor to the Unicon IDE.

Sudarshan Gaikaiwari Developer of snobol-style pattern datatype, along with operator overloading.

Jafar al Gharaibeh Dr. al Gharaibeh developed the concurrent programming feature and has been an extensive contributor to the 3D facilities.

Clint Jeffery Father of Unicon, his mission is to make Icon's core expression semantics useful in as many applications and to as many programmers as possible.

Susie Jeffery Contributor to the GUI class library, including (especially) the editabletextlist widget.

Steve Lumos Developer of the messaging facilities. Co-administrator of the Unicon sourceforge site and mailing list.

Kazimir Majorinc Founder of The Generator.

Naomi Martinez Developer of Unicon's 3D graphics facilities.

Shamim Mohamed Developer of the POSIX interface, coauthor of the Unicon book; he coined the name "Unicon".

Shea Newton Author of Unicon 12's benchmarks.

Robert Parlett Author of Unicon's package mechanism. Developer of many class libraries and the IVIB interface tool. Coauthor of the Unicon book chapter on graphic interfaces.

Ray Pereda Ported Berkeley YACC to Icon; his iyacc is used in the Unicon translator. Coauthor of the Unicon book chapters on compilers and genetic algorithms.

Katie Ray Author of the Ulex lexical analyzer generator.

Hani bani Salameh Contributor to the Unicon IDE.

Barry Schwartz Primary contributor to AMD64 port, including its co-expression switch.

Ziad al Sharif The first Unicon Ph.D. and author of the UDB source level debugger.

Phillip Thomas Chief critic, user, tester, and visionary supporter.

Ken Walker Honorary Unicon citizen who developed iconc, the Icon optimizing compiler which forms the basis for "unicon -C" on some Unicon platforms.

Steve Wampler Technical reviewer and major contributor to multiple epochs of Icon and Unicon.

Mike Wilder Wrote uniconc, a port of iconc to support almost all of Unicon on Linux. Implemented vector hashing code which improves type inferencing scalability.

The documentation developed with, and powered by:

GNU/Linux	Vim editor	Sphinx docgen	Unicon

CHAPTER

TWO

DATATYPES

Icon is rich in datatypes. `Unicon` is just that much richer.

2.1 Immutable Unicon Datatypes

`Unicon` starts out with some immutable types:

- Integer (arbitrary size)
- floating point Real numbers
- String
- Cset (sets of characters - ASCII)

Note: *string* is an immutable type. New strings will be formed for operations that look like they are modifying a string in place. *This has consequences*, detailed in the *String* entry.

2.1.1 Integer

Integers in Unicon can be any size, (when the `large integers` feature is compiled in) and are always exact values.

Radix prefix

Integer literals in source code can be of any base from 2 through 36, by using a radix prefix; the default being decimal, base 10. The radix is always a decimal number, followed by the letter `r` or `R` followed by digits of the value.

```
42
2r101010
5r132
16r2A
16R2a
36r16
```

The above, are all valid literals for the value forty-two. Base 0 and base 1 are invalid, and will produce a compile time error

```
#
# Numeric literals, with errors
#
procedure main()
    write("Various forms of the ultimate answer")
    write(42)
    write(2r101010)
    write(5r132)
    write(16r2A)
    write(16R2a)
    write(36r16)

    write(0r42)
    write(1r42)
end
```

Sample run (with errors):

```
prompt$ unicon -s numeric-literals-errors.icn -x
File numeric-literals-errors.icn; Line 20 # invalid radix for integer literal
File numeric-literals-errors.icn; Line 20 # invalid integer literal
numeric-literals-errors.icn:20: # "42": syntax error (104;258)
File numeric-literals-errors.icn; Line 21 # invalid radix for integer literal
File numeric-literals-errors.icn; Line 21 # invalid integer literal
numeric-literals-errors.icn:21: # "42": syntax error (104;258)
```

Getting rid of the troublesome lines (and adding some larger numbers):

```
#
# Numeric literals
#
procedure main()
    write("Various forms of the ultimate answer")
    write(42)
    write(2r101010)
    write(5r132)
    write(16r2A)
    write(16R2a)
    write(36r16)

    ## illegal radix   write(0r42)
    ## illegal radix   write(1r42)

    write("And now some larger values")
    write(36r16K)
    write(36r0to9andAtoZ)
end
```

Sample run:

```
prompt$ unicon -s numeric-literals.icn -x
Various forms of the ultimate answer
42
42
42
42
```

```
42
42
And now some larger values
1532
3013673839525331
```

Base radix also influences the `digits` allowed in the value. For base eight, 8 and 9 are illegal, not being part of the valid set of digit symbols. For base two, only 0 and 1 are allowed. For bases above ten, the alphabet is used. `A (or a)` represents ten, `B/b` is eleven, `Z/z` represents thirty-five with radix 36.

Given

```
write(3r42)
```

`unicon` will report a syntax error

```
File numbers.icn; Line 16 # invalid integer literal
numbers.icn:16: # "42": syntax error (104;258)
```

Case sensitivity

Unlike most of Unicon (being case sensitive), the Radix indicator `R` is case insensitive, `R` and `r` both work. As do upper and lower case letters when used as `digits`. `2A` and `2a` are both valid representations of forty-two in hexadecimal (assuming the `16r` radix is given first). All of `16R2a`, `16R2A`, `16r2a` and `16r2A` represent `42` in base 10.

Octal

Unicon does NOT follow the C convention of `0` prefixed literals being treated as octal (base eight), values. Use `8r0777` if you feel the need to prefix your octal constants with a zero. `8r777` will work just as well for representing five hundred eleven. `042` is forty two, not thirty four as a Unicon numeric literal, unlike C and some other languages.

```
042 ~= 34

042 == 42

042 == 8r52
```

Scaling suffix

And just to add to the flexibility, Unicon supports a trailing suffix that closely resembles the International System of Units `SI` standard, but scaled for binary computers and not the normal decimal base in thousands. Unicon uses `1024` based scaling.

- K (or k) kilo, literal is multiplied by 1024
- M (or m) mega, literal is multiplied by 1024^2
- G (or g) giga, 1024^3
- T (or t) tera, 1024^4
- P (or p) peta, 1024^5

```
write(42)
write(42K)
write(42M)
write(42g)
write(42t)
write(42P)
```

Gives

```
42
43008
44040192
45097156608
46179488366592
47287796087390208
```

To make for some sanity, the suffixes are only supported for decimal (base ten) literals. *Even considering the scaling is actually a binary and not decimal thousands based scaling*. For instance `36r16K` is a base 36 literal of the value `16K` (1532 decimal), not `36r16` modified by a suffix `K`.

The scaling suffixes are also case insensitive.

Positive and negative

The + (plus) and - (minus) signs can be used with any of these literals, and come before any radix specifier.

```
-2r101010 == -42
-16r2A    == -42
-16R2a    == -42
+36r16    == +42
```

There is no such thing as a negative base in `Unicon` so the sign always effects the value, never the radix (or the meaning of the scaling suffix).

Arbitrary magnitude

One of the nicer things with Unicon is the unlimited integer size.

```
n := 1234567890123456789012345678901234567890123456789012345678901234T
write(n)
```

Gives that long literal (scaled by tera, `T`, 1024^4), which displays as

```
1357421750469623887686962388768696238876869623887686956144750755584
```

Unicon Integer data is *always* exact, regardless of magnitude. *Unless you run out of memory, or other error condition has been triggered*.

Play nice

Given all that flexibility, do yourself, and everyone else, a favour and stick with literals that conceptually make sense for the task at hand. Don't use base thirteen literals, just because you can. Stick with the ten fingers for most code, and go to another radix only when it makes sense. Use `8r` octal numbers when dealing with things like Unix permissions, or `2r` for bit patterns. Base twenty-three literals will just cause confusion, for no reason and slow down everyone that wants to read through your program sources. *Using* `unicon` *is proof enough that you are smart cookie.*

2.1.2 Real numbers

Unicon also supports double precision floating point real numbers (in base ten, decimal).

> +/- digits, a decimal point (period), more digits, optional +/- E exponent

```
write(1.23)
write(.23)
write(1.)
write(1.23E42)
write(-1.23E-42)
```

Floating point is an inexact science. The internal representation is an approximation brought on by differences between binary and decimal notation.

`0.5`, is exact, both in decimal and base two arithmetic. But many values, such as `0.3` can cause problems when scaled, multiplied and divided. Be wary of precisions, rounding errors and keep a healthy skepticism when dealing with floating point double precision values.

Don't rely on floating point math for financial calculations. Use fixed point integer math or use something like GnuCOBOL for problems that require bank safe computations.

Having said that, there *is* science behind floating point representation and for most engineering problems, *close enough*, is usually a realistic expectation.

Floating point coercion

Unicon will always attempt to match integer and floating point calculations, promoting values to and from integer and real as hinted at by the code and the datatypes forming the computation.

```
write("Integer division")
every i := 1 to 8 do write(2 / i)

write()

write("Real division")
every i := 1 to 8 do write(2.0 / i)
```

Gives two completely different sets of output. The first `every` loop uses integer division, fractions lost and mostly zeros. The second uses floating point math, the `2.0` literal *forcing* the floating point computation, and a `Real` result:

```
prompt$ unicon numbers.icn -x

Integer division
2
1
0
0
0
0
0
0

Real division
2.0
1.0
0.6666666666666666
0.5
```

```
0.4
0.3333333333333333
0.2857142857142857
0.25
```

Any `Real` value in a computational input will cause a `Real` result. When all the input values are `Integer`, the result is Integer, even if it seems like it should cause a fractional answer.

String to number coercion

Unicon will do similar implicit coercion of data types when given a `String` value as part of a numerical equation. If a `String` can safely be converted to a numeric value, `Integer` or `Real`, it will be. If not, it will raise a run-time error.

```
write("String as Integer division")
every i := 1 to 4 do write("2"/i)
write()

write("String as Real division")
every i := 1 to 4 do write("2.0"/i)
write()

write("String as garbage division")
every i := 1 to 4 do write("2.o"/i)
```

Gives:

```
String as Integer division
2
1
0
0

String as Real division
2.0
1.0
0.6666666666666666
0.5

String as garbage division

Run-time error 102
File numbers.icn; Line 40
numeric expected
offending value: "2.o"
Traceback:
   main()
   {"2.o" / 1} from line 40 in numbers.icn
```

Explicit type conversions

Conversion of data types can also be explicit.

```
write("String as explict Real division")
every i := 1 to 4 do write(real("2")/i)
```

Produces:

```
String as explicit Real division
2.0
1.0
0.6666666666666666
0.5
```

The built-in functions `real(s)` and `integer(s)` can be used to convert `String`, `Real` and `Integer` data to the given numeric form.

`integer(r)` will be a truncation conversion. `integer(2.3)` and `integer(2.9)` both return 2. `real(i)` may lose precision once the integer value exceeds what can be stored in a floating point double precision value.

```
n := 1234567890123456789012345678901234567890123456789012345
write(n)
write(real(n))
write(integer(real(n)))
```

Shows as

```
1234567890123456789012345678901234567890123456789012345
1.234567890123457e+53
1234567890123456779024213753226425954399177206094888896
```

A huge loss of precision occurs after 16 digits of decimal. Don't be firing any NASA spacecraft out toward Jupiter without very careful consideration of floating point Real number precision and accuracy. Unicon is not at fault here, it is in the nature of approximation with floating point representations.

Again, in most day to day real world operations, unless you are trying to fire a rocket with sub-nanometre accuracy across a trillion kilometre distance, Unicon Real values will be close enough. For pure mathematics? *Not even in the same ball park.*

2.1.3 Cset

A character set. A highly efficient datatype for pattern matching.

Csets are limited to single byte values, 0 through 255. There are no duplicates within a Cset. Cset literals use single quotes in Unicon.

```
#
# cset-samples.icn, demonstrate some Csets
#
procedure main()
    noDupes := 'hello'
    write("Given 'hello': ", noDupes, " ", image(noDupes))

    cs := 'abcdef'
    write(cs, " ", image(cs))

    complement := ~cs
    write("Size of complement of ", image(cs), ": ", *complement)

    write("Size of complement of &letters: ", *~&letters)
end
```

Sample run:

```
prompt$ unicon -s cset-samples.icn -x
Given 'hello': ehlo 'ehlo'
abcdef 'abcdef'
Size of complement of 'abcdef': 250
Size of complement of &letters: 204
```

2.1.4 String

This, is where Unicon shines. String manipulation is at the heart and soul of *Icon*, Unicon, and back all the way to *SNOBOL*. Pattern matching, string scanning, slicing, dicing and transformation operations you probably haven't even thought of yet. All nicely packaged in the Unicon executable, very clearly, and very concisely. Ralph Griswold was one of those genius level computer scientists, who led the core Icon developers to exceed expectations and go above and beyond the norm. Clint Jeffery and his team are now pushing those expectations out even further with Unicon. *Release 13 of Unicon has SNOBOL inspired pattern matching operations built in, a huge testament to the legacy and future of Unicon programming. More on that in the SNOBOL patterns chapter.*

In `Unicon`, String data is *immutable*, and is never changed in place. New String data is created, as required.

```
s[2:3] := "D"
```

That expression does not change the existing character of `s` at index 2, but is equivalent to the expression:

```
s := s[1:2] || "D" || s[3:0]
```

Creating a new string by copying existing parts. *Internal memory management means that this is a safe thing to do, as many millions of times as an algorithm may need. The heaps will be efficiently managed by the Unicon runtime.*

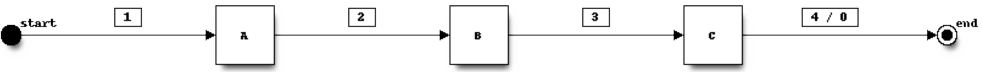
Indexing

String indexing positions (or *subscripting*) is calculated using a cursor that floats "between" characters. Indexing starts at 1, with the virtual cursor positioned before the first character. The end is position "0", and can count backwards, -1 being the position between the last two characters of the string.

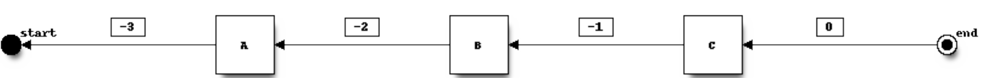
For instance:

```
"ABC"
```

That string has positions: 1, 2, 3, 4 (or 0) counting from start to end. Using negative indexing, the positions are: -3, -2, -1, and 0, counting from the end to the start going backwards.



From the end that is



It is important to get used to the idea of position values being *before*, *between* and *after* characters, and that zero is the position past the end of a string.

2.1.5 Pattern

Unicon now supports SNOBOL based pattern data.

2.1.6 Regular Expression

Along with patterns, Unicon has also added regex literals for string matching. Basic regex at this point.

2.2 Non computational types

Unicon also supports a variety of *non computational* types. These vary from File, to Window to other internally managed (usually) *non-mutable* types.

Note: Unicon does not have pointers, but does manage internal references.

2.2.1 File

A value returned from *open* when using file modes of `r`, `w`, `a`.

2.2.2 Window

A graphics context.

2.2.3 Co-expression

A very handy code datatype.

Next up, structures

`Icon` and `Unicon` then add a very nice set of aggregate *structures* and other high level datatypes. Most of these types are *mutable*. Slices will be changed in place and not copied as frequently as *string* data.

From this point on, Icon won't be mentioned as often, this is a Unicon book.

CHAPTER

THREE

DATA STRUCTURES

One of the features that elevates Unicon from a high level language to a *very* high level language is the range of structured data types.

3.1 Unicon mutable data types

3.1.1 List (arrays)

Unicon lists are *mutable*, ordered collections of data. Each element can be any datatype. Lists are created with square bracket syntax, or the *list* function. `list()` accepts an optional size parameter, followed by an optional default initial value.

&null is the default initial value if not specified.

Appending to a list is supported with the `|||` *(list concatenation)* operator, which can be augmented with assignment (as is the case for almost all Unicon operators).

List functions include stack and queue like operations, along with arbitrary positioning.

- *push*, *pop* (front of the list)
- *put*, *pull* (end of the list)
- *insert*(L, i, x) will insert *x* at position *i*
- *delete* (L, i,...) will delete the elements at position *i* and other given indexes.

Subscript indexing of a list is supported with square bracket syntax. `List[index]`. This is where Unicon lists, and arrays look very similar. Lists are one-dimensional vectors. Subscripting will fail if the requested element(s) are out of current list size range.

Create new list sections with square bracket and colon syntax, `L[start:end]`. *This syntax creates new lists.*

Subscript range access creates new lists. List sections are not variable; the original list is not modifiable using sections. Where as `Groceries[1] := "Milk"` modifies the `Groceries` list, `Groceries[1:3] := "Milk"` is invalid; the `Groceries[1:3]` section expression is not a variable and assignment does not apply.

The *get* function will remove and return multiple elements from the head of a list.

list comprehension

While the normal list creation syntax is `[v1, v2, ...]`, Unicon also features list *comprehension*. The syntax is `[: expr :]`, all values from the expression become values in the returned list.

```
#
# list-operations.icn, some simple List examples
#
link fullimag

procedure main()
    local L1, L2
    # L1 is 5 elements of 0
    L1 := list(5, 0)
    L2 := ["a", "b", "c", "c", 1, 1.0, ["sublist"]]

    # lists are ordered, mutable structures
    write(fullimage(L1), " size of L1: ", *L1)
    write(fullimage(L2), " size of L2: ", *L2)

    # Indexing
    write("\nL2[3]: ", L2[3])
    write("L2[1:3]: ", fullimage(L2[1:3]))

    # Concatenation
    write("\nL1 ||| L2")
    write(fullimage(L1 ||| L2))

    # Membership
    if member(L1 ||| L2, "d") then write("\nd is in the list")

    # Insertion
    write("\nInsertion")
    write(fullimage(insert(L2, 1, "first")))

    # List comprehensions
    write("\nComprehension with [: 1 to 10 :]")
    write(fullimage([: 1 to 10 :]))
    write("Comprehension with [: 1 to (1 to 5) :]")
    write(fullimage([: 1 to (1 to 5) :]))
end
```

```
prompt$ unicon -s list-operations.icn -x
[0,0,0,0,0] size of L1: 5
["a","b","c","c",1,1.0,<2>["sublist"]] size of L2: 7

L2[3]: c
L2[1:3]: ["a","b"]

L1 ||| L2
[0,0,0,0,0,"a","b","c","c",1,1.0,<2>["sublist"]]

Insertion
["first","a","b","c","c",1,1.0,<2>["sublist"]]

Comprehension with [: 1 to 10 :]
[1,2,3,4,5,6,7,8,9,10]
Comprehension with [: 1 to (1 to 5) :]
```

```
[1,1,2,1,2,3,1,2,3,4,1,2,3,4,5]
```

3.1.2 Set

Unicon Set data is an unordered, *mutable* data structure that follows set theory. Sets are initialized with the *Set* function.

Union, intersection, difference, insertion and membership testing are all supported. Set elements are unique within a set.

Cset is an internally efficient form of character sets.

```
#
# set-operations.icn, some simple Set data examples
#
link fullimag

procedure main()
    S1 := set("a", "b", "c", "c", "d")
    S2 := set("d", "e", "f", "g", "h")

    # sets have a uniqueness propery, "c" is only held once
    write("Uniqueness")
    write(fullimage(S1))
    write(fullimage(S2))

    # Union
    write("\nUnion S1 ++ S2")
    write(fullimage(S1 ++ S2))

    # Intersection
    write("\nIntersection S1 ** S2")
    write(fullimage(S1 ** S2))

    # Difference
    write("\nDifference S1 -- S2")
    write(fullimage(S1 -- S2))

    write("Difference S2 -- S1")
    write(fullimage(S2 -- S1))

    # Membership
    if member(S1 ** S2, "d") then write("\nd is in both sets")

    # Insertion
    write("\nInsertion")
    write(fullimage(insert(S1, "i")))
end
```

```
prompt$ unicon -s set-operations.icn -x
Uniqueness
set("a","b","c","d")
set("d","e","f","g","h")

Union S1 ++ S2
```

```
set("a","b","c","d","e","f","g","h")

Intersection S1 ** S2
set("d")

Difference S1 -- S2
set("a","b","c")
Difference S2 -- S1
set("e","f","g","h")

d is in both sets

Insertion
set("a","b","c","d","i")
```

3.1.3 Table

Unicon tables are associative arrays, key-value pairs. Keys can be any type, values can be any type. Created with the *Table* function, access is by square bracket subscripting (where the subscript is a key specifier).

insert, *delete* are supported.

Structure operators such as * size and ? random element operators (returning a random table value, not a key) are also built to work with tables.

The generate elements operator, generates values not keys.[1] The function *key*(T) generates the keys of a table.

Subscripted access will return a default value if lookup does not find the key. The default value is the optional argument of the *Table* declaration function, or *&null*.

The *member* function succeeds if the given key exists in the table, or fails otherwise.

```
#
# table-operations.icn, some simple table() examples
#
link ximage

procedure main()
    T1 := table()               # the default default is &null
    T2 := table(0)              # default element is 0

    # tables are mutable, key value pair structures
    # accessing an unknown key returns the table default value
    # setting an unknown key inserts the key-value pair

    write("T1[\"unknown\"]: ", image(T1["unknown"]))
    write("T2[\"unknown\"]: ", image(T2["unknown"]))

    T1["unknown"] := "now known"
    write("T1[\"unknown\"]: ", image(T1["unknown"]))

    # T2 elements default to 0, math is ok for unknown elements
    write("T2[\"unknown\"] + 42: ", image(T2["unknown"] + 42))
```

[1] The fact that the *!* operator generates values and not keys for table data was deemed an early Icon language design mistake. Before Icon v7, you needed to convert a table to a paired list to get at keys. The *key* function was added to get around this design flaw. Key value pair associative data tables were originally in *SNOBOL*. This extremely useful Unicon data structure has a long history,

```
    write("\nximages:")
    write(ximage(T1))
    write(ximage(T2))

    # Access is by square bracket operator
    # keys can be any type, values can be any type
    T2[3] := "42"
    T2["3"] := 42
    write("\nElements:\nT2[3]: ", image(T2[3]))
    write("T2[\"3\"]: ", image(T2["3"]))

    # Size, note: T2["unknown"] was not assigned during the computation
    write("size of T1: ", *T1)
    write("size of T2: ", *T2)

    # element generator gives values
    write("\nT2 values")
    every write(image(!T2))

    # key() function, generates the keys
    write("\nT2 keys")
    every write(image(key(T2)))
    write(ximage(T2))

    # Membership
    if member(T2, 3) then write("\ninteger 3 is a key in T2")

    # removal
    v := delete(T2, 3)
    write("\ndelete T2[3], return value is: ", type(v))
    # membership test will now fail, no output written
    if member(T2, 3) then write("\ninteger 3 is a key in T2")
    write("T2 keys after delete")
    every write(image(key(T2)))

    # table values can also be functions
    fT := table()
    fT["add"] := adder
    operation := "add"
    # add two values, invoked from table key
    write("\noperation ", operation, " returns ", fT[operation](1,2))
    write(ximage(fT))
end

#
# adder function
#
procedure adder(a,b)
    return a + b
end
```

```
prompt$ unicon -s table-operations.icn -x
T1["unknown"]: &null
T2["unknown"]: 0
T1["unknown"]: "now known"
T2["unknown"] + 42: 42
```

```
ximages:
T1 := table(&null)
   T1["unknown"] := "now known"
T2 := table(0)

Elements:
T2[3]: "42"
T2["3"]: 42
size of T1: 1
size of T2: 2

T2 values
"42"
42

T2 keys
3
"3"
T2 := table(0)
   T2[3] := "42"
   T2["3"] := 42

integer 3 is a key in T2

delete T2[3], return value is: table
T2 keys after delete
"3"

operation add returns 3
T6 := table(&null)
   T6["add"] := procedure adder
```

Todo

more on tables

3.1.4 Record

Records are fixed length[2], named field structures, created with a globally named initializer defined with the *record* reserved word at compile time. Unicon record structures can also be defined at runtime with the *constructor* function.

Compile time `record` definitions occur outside of procedures in source code. The initializer is global to all procedures within a compile unit. *constructor* definitions are contained within the same scope as the variable used to store the return value from the function.

`record` data plays a background role in the object oriented features of Unicon, but programmers don't need to worry about that, details managed by the compiler.

```
#
# record-operations.icn, demonstrate records
```

[2] Records are fixed length in terms of always having the same number of named fields. The field elements themselves can be any data type, of variant sizes. This is not the same as a COBOL fixed length record, which always have the same number of bytes in each field and in each record hierarchy.

```
#
link ximage

# record initializers are in the global name space
record sample(field1, field2, field3)

procedure main()
    # records are created by the initializer
    R1 := sample("value1", "value2", 3)

    # record field access uses the dot operator
    write("R1.field1: ", R1.field1)
    write("R1.field2: ", R1.field2)
    write("R1.field3: ", R1.field3)

    # Unicon also allows runtime creation of record constructors
    Rcon := constructor("runtime", "f1", "f2", "f3")

    # The new "runtime" initializer is now available as Rcon
    R2 := Rcon("v1", "v2", R1)

    write("\nR2.f1: ", R2.f1)
    write("R2.f2: ", R2.f2)
    # like all Unicon structures, fields can be arbitrarily nested
    write("R2.f3.field1: ", R2.f3.field1)
    write("type of R2: ", type(R2))
    write("type of R2.f3: ", type(R2.f3))
end
```

```
prompt$ unicon -s record-operations.icn -x
R1.field1: value1
R1.field2: value2
R1.field3: 3

R2.f1: v1
R2.f2: v2
R2.f3.field1: value1
type of R2: runtime
type of R2.f3: sample
```

CHAPTER

FOUR

EXPRESSIONS

4.1 Unicon expressions

Everything in Unicon is an expression. Well just about everything, *the source files are actually just text until processed*, for instance.

Expressions can be chained in Unicon, and the overall *multiple expression* expression still counts as an expression in terms of the lexical syntax.

For example: The *every* reserved word is syntactically defined as

```
every expr1 [do expr2]
```

That can be

```
every 1 to 5 do writes(".")
```

Or it can be

```
every i := 1 to 5 do write(i)
```

The assignment expression, inserted in the `every` still counts as $expr_1$ for `every`, from a compiler syntax point of view. This is a powerfully concise feature of Unicon and almost all programs take advantage of this grouped expressions within an expression paradigm.

Expression blocks, contained within braces, also count as a single expression, in terms of the Unicon parser. Some operators, such as alternation and conjunction, and other syntax elements provide even more flexibility when forming expressions.

4.1.1 Success and Failure

All Unicon expressions produce a value, or fail. This is a very powerful computational paradigm. Procedures can return a full range of values, and need not worry about reserving sentinel values for error codes or out-of-band results.

For errors, or simple end-of-file status, the procedure can simply fail. *When error status codes are required, variables can be set.*

Procedures always produce values, *unless they fail*. The default value for *return* when no expression is given is *&null*. Falling off the end of a procedure without a `return` or `suspend` signals failure.

```
#
# default-return-value.icn, default expression for return, is &null
#
procedure main()
    write(type(subproc()))
end

# this actually returns &null, not void/nothing
procedure subproc()
    return
end
```

```
prompt$ unicon -s default-return-value.icn -x
null
```

There is no practical use for the concept of *void* in Unicon. *But fear not*, the foreign function interface layer *loadfunc*, can include wrappers that manage the C concept of void returns and transform things to usable Unicon values, when necessary. *&null* is not void. The closest thing to `void` in Unicon is `failure`.

Failure propagation

With goal-directed evaluation, Unicon expressions always strive to produce a result for surrounding expressions. Failure will propagate outwards when no result can be produced. When an inner expression fails, a surrounding expression will attempt any possible alternates, until it either succeeds or must also fail. This propagation will continue until a *bound expression* marker terminates any goal-directed backtracking.

4.1.2 null

The null value, represented by *&null* is the default value for unset variables or omitted arguments.

The null value is treated specially inside most expressions. Many expressions will cause a runtime error when a null value is dereferenced as part of a computation, for instance.

```
#
# null value runtime error
#
procedure main()
    a := b + c
end
```

The sample above abends with a runtime error when the null values of `b` and `c` are used in a calculation. See *&error* for ways of controlling how Unicon handles runtime errors. Runtime errors can be converted to expression failure, allowing programmer control of abend states.

```
prompt$ unicon -s runtime-null.icn -x

Run-time error 102
File runtime-null.icn; Line 12
```

```
numeric expected
offending value: &null
Traceback:
   main()
   {&null + &null} from line 12 in runtime-null.icn
```

Through this document, `null`, and `&null` are used interchangeably, null is the value, represented by the keyword *&null*.

4.1.3 Precedence

It is very much worthwhile getting used to the precedence rules with Unicon, It is not as tricky as it may first seem, but the level of complexity of some Unicon expressions will be easier to read with a sound understanding of the order of precedence rules. When in doubt, use parentheses to control the evaluation order of complex expressions.

Updated for Unicon and reformatted slightly from

https://www2.cs.arizona.edu/icon/refernce/exprlist.htm#expressions[1]

Shown in order of decreasing precedence. Items in groups (as separated by empty lines) have equal precedence (left to right).[2]

For instance: exponentiation ^ is higher than addition +, so

```
a + b ^ c
```

parses as

```
a + (b ^ c)
```

Left to right associativity means

```
a ++ b -- c
```

parses as

```
(a ++ b) -- c
```

```
High Precedence Expressions

    (expr)                          # grouping
    {expr1;expr2;...}               # compound
    x(expr1,expr2,...)              # process argument list
    x{expr1,expr2,...}              # process co-expression list
    [expr1,expr2,...]               # list
    expr.F                          # field reference
    expr1[expr2]                    # subscript
    expr1[expr2,expr3,...]          # multiple subscript
    expr1[expr2:expr3]              # section
    expr1[expr2+:expr3]             # section
    expr1[expr2-:expr3]             # section
```

[1] ProIcon was a commercial Macintosh Icon extension venture by Bright Forest and Catspaw. When ProIcon was taken off the market, materials were placed in the public domain for redistribution by the Icon project. *Unicon has deep roots in computing for the public good.*

[2] The generator limitation expression is a rare Unicon infix operator that is evaluated right to left. The limit needs to be known before the first result is suspended by the generator. *Like all nifty Unicon expressions, the limit can actually be a generator as well.*

```
Prefix Expressions

    not expr                          # success/failure reversal
    | expr                            # repeated alternation
    ! expr                            # element generation
    * expr                            # size
    + expr                            # numeric value
    - expr                            # negative
    . expr                            # value (dereference)
    / expr                            # null
    \ expr                            # non-null
    = expr                            # match and tab
    ? expr                            # random value
    ~ expr                            # cset complement
    @ expr                            # activation (&null transmitted)
    ^ expr                            # refresh
    .> expr                           # pattern cursor position assign

Infix (some postfix Unicon 13) Expressions

    expr2 \ expr1                    # limitation 2
    expr1 @ expr2                    # transmission
    expr1 ! expr2                    # invocation
    expr1 >@ expr2                   # send to other out-box
    expr1 >>@ expr2                  # blocking send to other out-box
    expr1 <@ expr2                   # receive from other out-box
    expr1 <<@ expr2                  # blocking receive from other out-box

    expr1 ^ expr2                    # exponentiation
    expr >@                           # send to default out-box
    expr >>@                          # blocking send
    expr <@                           # receive
    expr <<@                          # blocking receive

    expr1 * expr2                    # multiplication
    expr1 / expr2                    # division
    expr1 % expr2                    # remainder
    expr1 ** expr2                   # cset or set intersection

    expr1 + expr2                    # addition
    expr1 - expr2                    # subtraction
    expr1 ++ expr2                   # cset or set union
    expr1 -- expr2                   # cset or set difference
    expr1 -> expr2                   # conditional pattern assignment
    expr1 => expr2                   # immediate pattern assignment

    expr1 || expr2                   # string concatenation
    expr1 ||| expr2                  # list concatenation

    expr1 < expr2                    # numeric comparison
    expr1 <= expr2                   # numeric comparison
    expr1 = expr2                    # numeric comparison
    expr1 >= expr2                   # numeric comparison
```

```
expr1 > expr2                   # numeric comparison
expr1 ~= expr2                  # numeric comparison
expr1 << expr2                  # string comparison
expr1 <<= expr2                 # string comparison
expr1 == expr2                  # string comparison
expr1 >>= expr2                 # string comparison
expr1 >> expr2                  # string comparison
expr1 ~== expr2                 # string comparison
expr1 === expr2                 # value comparison
expr1 ~=== expr2                # value comparison

expr1 | expr2                   # alternation
expr1 || expr2                  # pattern concatenation

expr1 to expr2 by expr3        # step wise number generation
expr1 .| expr2                  # pattern alternate

expr1 := expr2                  # assignment
expr1 <- expr2                  # reversible assignment
expr1 :=: expr2                 # exchange
expr1 <-> expr2                 # reversible exchange
expr1 op:= expr2                # (augmented assignments)

expr1 ?? expr2                  # pattern match

expr1 ? expr2                   # string scanning

expr1 & expr2                   # conjunction

Low Precedence Expressions

    break [expr]                     # break from loop
    case expr of {                   # case selection
       expr1 : expr2
       ...
       [default: expr3]
       }
    create expr                      # co-expression creation
    every expr1 [do expr2]          # iterate over generated values
    fail                              # failure of procedure
    if expr1 then expr2 [else expr3]   # if-then-else
    next                              # go to top of loop
    repeat expr                      # loop
    return expr                      # return from procedure
    suspend expr1 [do expr2]        # suspension of procedure
    until expr1 [do expr2]          # until-loop
    while expr1 [do expr2]          # while-loop
```

4.1.4 Variable scope

Unicon supports variables with the following scope:

- local
- global
- static
- package

All procedures are global in scope. Methods are local to the enclosing class. Parameters to a procedure are local to the body of the procedure. `local` can be used to create localized references within a procedure hiding any global variables. Static local variables retain their values between invocations of the enclosing procedure.

Packages add another layer to the scoping rules of Unicon, and act as namespace containers for all procedures belonging to a package.

Todo

add more details to package scoping

4.1.5 Semicolon insertion

Unicon expressions are separated by semicolons.

```
i := 3 + 3; write(i);
```

That code is the same as

```
i := 3 + 3;
write(i);
```

Which, during `unicon` translation, is the same as

```
i := 3 + 3
write(i)
```

The lexer makes life easier, by automatically inserting semicolons at the end of a line if the last token could legally end an expression, and the first token of the next line is legal at the beginning of an expression. *This has implications*. Although automatic semicolon insertion is usually an invisible assistive feature, you need to be careful with expressions that span lines. They need to break at a point that avoids the automatic semicolon.

```
#
# semicolon.icn example of semicolon insertion issues
#
procedure main()
    i := 3 +
         3
    write(i)
    i := 3
         + 3
    write(i)
end
```

```
prompt$ unicon -s semicolon.icn -x
6
3
```

The second result is equivalent to

```
i := 3; +3; write(i);
```

Unicon will gladly accept +3 as an expression, then discard the result, unused.

In the sample, the first evaluation, ending with +, which is not a legal expression ending token, avoided the automatic semicolon and set i to 6.

```
i := 3+3; write(i);
```

Discarding results is a feature of the Unicon implementation and happens all the time. All parameters to a function or procedure are evaluated, but only the required ones are dereferenced and passed. Conjugation evaluates $expr_1$, and produces the value from $expr_2$ if $expr_1$ succeeds, but the value of $expr_1$ itself is discarded. *Side effects may happen, but the result is discarded in terms of the surrounding context*.

Just a thing to be wary of when splitting expressions across lines. The Unicon lexical analyzer is very flexible and does the right thing in the vast majority of cases, but white space is not the only concern when splitting expressions across line breaks. The computer will always do what it is told, which may not always match what a human intended.

Visible semicolon insertion

To verify how the compiler treats multi-line expressions, the preprocessor output can always be used to see where semicolons are inserted:

```
prompt$ unicon -E -s semicolon.icn
#line 0 "/tmp/uni16417806"
#line 0 "semicolon.icn"












procedure main();
    i := 3 +
         3;
    write(i);
    i := 3;
         + 3;
    write(i);
end
```

4.1.6 Bound Expressions

Along with the order of precedence, and goal directed evaluation, bounded expressions can be used to control the level of backtracking. A lot of bounded expressions are implicit. Intuitive, but nearly invisible with the automatic semicolon insertion that occurs at the end of each line (or multi line expression as explained in *automatic semicolon insertion*).

Bound expressions block backtracking, a complex feature *(for the Unicon language designers and compiler authors)*. This features allows goal directed evaluation to work, without rewinding all the way back to the main entry point every time an alternative is required.

```
#
# backtrack-bound.icn
#
procedure main()
    # not found; a keeps the value 1, alternate not used
    a := (1 | 2)
    if find("h", "this") == a then
        write("found at ", a)
    else
        write("not found: a is ", a)

    # found; alternative is allowed when striving for the goal
    if find("h", "this") == (a := (1 | 2)) then
        write("found at ", a)
    else
        write("not found: a is ", a)
end
```

The first expressions is implicitly bound at the newline between the `a` assignment and the `if find`, in the first code fragment. Unicon will not backtrack to reassign `a` to `2` when attempting to satisfy the `find` goal. *This is a good thing*. In the second fragment, backtracking is not bound as part of the `a` assignment, and `find` is allowed to try `2` when striving for the goal.

```
not found: a is 1
found at 2
```

Curiousities

The available Unicon documentation, which includes the Icon document collection, is a vast treasure hoard of gems and jewels. The Unicon mailing list is also a source of learning and shared knowledge. Bruce Rennie asked about this one:

```
#
# suspended conjugation
#
procedure main()
    every write(subproc())
end

procedure subproc()
    every (suspend 1 to 3) & 4
end
```

That code produces:

```
1
2
3
```

Normally conjugation (the `&` operator), returns $expr_2$ when $expr_1$ and $expr_2$ succeed.

In the above instance, the parenthesized *suspend* actually fails when resumed after the 3 is delivered. The conjugation then fails and the 4 is never used. The *write* eventually fails, and the *every* in main fails after the *to* generator *write*

side effects.

```
#
# suspended conjugation
#
procedure main()
    every write(subproc())
end

procedure subproc()
    every suspend 1 to 3 & 4
end
```

That code will display the 4 three times, and only the 4s, the $expr_2$ result of the conjugation which succeeds three times given then the *to*.

```
4
4
4
```

The key expression here is grouped as

```
every suspend ((1 to 3) & 4)
```

`1 to 3` succeeds, and conjugation produces $expr_2$ (the 4), when $expr_1$ succeeds.

4.2 Unicon Co-Expressions

Co-expressions are another *ahead of its time* feature of Icon/Unicon. They are usually used in the context of generators, allowing sequence results to be generated when needed.

Co-expressions are also a key part of the multi-tasking features built into the Unicon virtual machine. See *Multi-tasking* for more information on this aspect of Unicon programming potentials. Multiple programs can be loaded into a single instance of a running Unicon machine and task co-expression activation can make for some very interesting possibilities.

Todo

co-expression entries

4.2.1 User defined control structures

Todo

control structure examples

CHAPTER

FIVE

OPERATORS

5.1 Unicon operators

Unicon is an operator rich programming language, very rich. As Unicon is also an *everything is an expression language*, operators can be chained to provide very concise computational phrases. With the small price of requiring developers to understand the rules of *Precedence* and order of operations.

5.1.1 Precedence chart

Updated for Unicon and reformatted slightly from

https://www2.cs.arizona.edu/icon/refernce/exprlist.htm#expressions[1]

Shown in order of decreasing precedence. Items in groups (as separated by empty lines) have equal precedence (left to right).[2]

For instance: exponentiation ^ is higher than addition +, so

```
a + b ^ c
```

parses as

```
a + (b ^ c)
```

Left to right associativity means

```
a ++ b -- c
```

parses as

[1] ProIcon was a commercial Macintosh Icon extension venture by Bright Forest and Catspaw. When ProIcon was taken off the market, materials were placed in the public domain for redistribution by the Icon project. *Unicon has deep roots in computing for the public good.*

[2] The generator limitation expression is a rare Unicon infix operator that is evaluated right to left. The limit needs to be known before the first result is suspended by the generator. *Like all nifty Unicon expressions, the limit can actually be a generator as well.*

```
(a ++ b) -- c
```

```
High Precedence Expressions

    (expr)                          # grouping
    {expr1;expr2;...}               # compound
    x(expr1,expr2,...)              # process argument list
    x{expr1,expr2,...}              # process co-expression list
    [expr1,expr2,...]               # list
    expr.F                          # field reference
    expr1[expr2]                    # subscript
    expr1[expr2,expr3,...]          # multiple subscript
    expr1[expr2:expr3]              # section
    expr1[expr2+:expr3]             # section
    expr1[expr2-:expr3]             # section

Prefix Expressions

    not expr                        # success/failure reversal
    | expr                          # repeated alternation
    ! expr                          # element generation
    * expr                          # size
    + expr                          # numeric value
    - expr                          # negative
    . expr                          # value (dereference)
    / expr                          # null
    \ expr                          # non-null
    = expr                          # match and tab
    ? expr                          # random value
    ~ expr                          # cset complement
    @ expr                          # activation (&null transmitted)
    ^ expr                          # refresh
    .> expr                         # pattern cursor position assign

Infix (some postfix Unicon 13) Expressions

    expr2 \ expr1                   # limitation [2]
    expr1 @ expr2                   # transmission
    expr1 ! expr2                   # invocation
    expr1 >@ expr2                  # send to other out-box
    expr1 >>@ expr2                 # blocking send to other out-box
    expr1 <@ expr2                  # receive from other out-box
    expr1 <<@ expr2                 # blocking receive from other out-box

    expr1 ^ expr2                   # exponentiation
    expr >@                         # send to default out-box
    expr >>@                        # blocking send
    expr <@                         # receive
    expr <<@                        # blocking receive

    expr1 * expr2                   # multiplication
    expr1 / expr2                   # division
    expr1 % expr2                   # remainder
```

```
    expr1 ** expr2                    # cset or set intersection

    expr1 + expr2                     # addition
    expr1 - expr2                     # subtraction
    expr1 ++ expr2                    # cset or set union
    expr1 -- expr2                    # cset or set difference
    expr1 -> expr2                    # conditional pattern assignment
    expr1 => expr2                    # immediate pattern assignment

    expr1 || expr2                    # string concatenation
    expr1 ||| expr2                   # list concatenation

    expr1 < expr2                     # numeric comparison
    expr1 <= expr2                    # numeric comparison
    expr1 = expr2                     # numeric comparison
    expr1 >= expr2                    # numeric comparison
    expr1 > expr2                     # numeric comparison
    expr1 ~= expr2                    # numeric comparison
    expr1 << expr2                    # string comparison
    expr1 <<= expr2                   # string comparison
    expr1 == expr2                    # string comparison
    expr1 >>= expr2                   # string comparison
    expr1 >> expr2                    # string comparison
    expr1 ~== expr2                   # string comparison
    expr1 === expr2                   # value comparison
    expr1 ~=== expr2                  # value comparison

    expr1 | expr2                     # alternation
    expr1 || expr2                    # pattern concatenation

    expr1 to expr2 by expr3          # step wise number generation
    expr1 .| expr2                    # pattern alternate

    expr1 := expr2                    # assignment
    expr1 <- expr2                    # reversible assignment
    expr1 :=: expr2                   # exchange
    expr1 <-> expr2                   # reversible exchange
    expr1 op:= expr2                  # (augmented assignments)

    expr1 ?? expr2                    # pattern match

    expr1 ? expr2                     # string scanning

    expr1 & expr2                     # conjunction

Low Precedence Expressions

    break [expr]                       # break from loop
    case expr of {                     # case selection
       expr1 : expr2
       ...
       [default: expr3]
       }
    create expr                        # co-expression creation
```

```
every expr1 [do expr2]              # iterate over generated values
fail                                 # failure of procedure
if expr1 then expr2 [else expr3]   # if-then-else
next                                 # go to top of loop
repeat expr                         # loop
return expr                         # return from procedure
suspend expr1 [do expr2]           # suspension of procedure
until expr1 [do expr2]             # until-loop
while expr1 [do expr2]             # while-loop
```

5.2 Unary operators

5.2.1 ! (generate elements)

Generate elements.

! *x* : any*

Generate elements of *x*. Values produced depends on *type* of *x*.

For *x* of

Type	! produces
integer	equivalent to `(1 to x)` for integer x
real	digits from the real number (including .)
string	one character substrings of string in order
file	lines/records of resource
list	elements of list
record	field values of record
set	elements of set, in undefined order
table	values of table, in undefined order
variable	fields that can be used for assignment

For *record* and *Table*, use *key* to retrieve the key fields, *which can be used directly to get at the values*.

```
#
# generate-elements, unary !
#
link ximage

procedure main()
    write("!3")
    every write(!3)

    write("\n!\"abc\"")
    every write(!"abc")

    write("\n![1,2,3]")
    every write(![1,2,3])

    write("\n!&ucase\\3")
    every write(!&ucase\3)

    write("\n!this file\\3")
```

```
    f := open(&file, "r")
    every write(!f\3)
    close(f)

    s := "abc"
    write("\nevery !s := 3  # with s starting as ", s)
    every !s := 3
    write("s now ", s)

    L := ["abc", "def", "ghi"]
    write("\nevery !L := \"xyz\"  # with L starting as ", ximage(L))
    every !L := "xyz"
    write(ximage(L))
end
```

```
!3
1
2
3

!"abc"
a
b
c

![1,2,3]
1
2
3

!&ucase\3
A
B
C

!this file\3
##-
# Author: Brian Tiffin
# Dedicated to the public domain

every !s := 3  # with s starting as abc
s now 333

every !L := "xyz"  # with L starting as L2 := list(3)
   L2[1] := "abc"
   L2[2] := "def"
   L2[3] := "ghi"
L2 := list(3,"xyz")
```

Note that `!"abc"` produced `"a", "b", "c"`, while `!s` produced `s[1], s[2], s[3]`, which can be assigned to, as can the element references of a *List (arrays)* variable (or other structured) type. The generate elements operator doesn't just generate values, it generates the references to the values if applicable to the surrounding expression and the source, *x*.

Note: There may be times when multi-threading changes the state of *x* during generation, and it will be up the

programmer to account for these times.

5.2.2 * (size)

Size of operator. **expr* Returns the size of the *expr*.

**x* : *integer*

- integer, size of the value coerced to string
- real, size of the value coerces to string
- string, number of characters in string
- cset, number of characters in cset
- structure, number of elements in list, set, table, fields of record
- co-expression, number of activations since last reset
- thread, messages in in-box

```
#
# size-operator.icn, demonstrate size operator for various expressions
#
# tectonics: unicon -s size-operator.icn -x
#
link printf

record R(f1, f2, f3)
procedure main()

format := "%-12s %-20s is %2d %s\n"
printf(format, "integer:", "*12", *12, "as string")

printf(format, "real:", "*12.12", *12.12, "as string")

printf(format, "string:", "*\"abc\"", *"abc", "")

printf(format, "cset:", "*'aabbcd'", *'aabbcd', "")

printf(format, "list:", "*[1,2,3]", *[1,2,3], "")
printf(format, "list:", "*[: 1 to 5 :]", *[: 1 to 5 :], "")

r1 := R(1,2,3)
printf(format, "record:", "*record(f1,f2,f3)", *r1, "")

printf(format, "keyword:", "*&file", *&file, "as name: " || &file)

db := open("unicon", "o", "", "")
printf(format, "odbc:", "*db of ODBC file", *db, "")
sql(db, "select * from contacts")
printf(format, "odbc:", "*fetch() of contacts", *fetch(db), "")
close(db)

coexp := create seq()\3
printf(format, "activations:", "*coexp", *coexp, "none from seq()\\3")
```

```
@coexp
printf(format, "activations:", "*coexp", *coexp, "after one")
every 1 to 6 do @coexp
printf(format, "activations:", "*coexp", *coexp, "after seven")
end
```

```
integer:      *12                  is  2 as string
real:         *12.12               is  5 as string
string:       *"abc"               is  3
cset:         *'aabbcd'            is  4
list:         *[1,2,3]             is  3
list:         *[: 1 to 5 :]        is  5
record:       *record(f1,f2,f3)    is  3
keyword:      *&file               is 17 as name: size-operator.icn
odbc:         *db of ODBC file     is -1
odbc:         *fetch() of contacts is  3
activations: *coexp                is  0 none from seq()\3
activations: *coexp                is  1 after one
activations: *coexp                is  3 after seven
```

5.2.3 + (numeric identity)

Numeric identity, does not change the value other than to convert to a required numeric type.

+*x* is *x* as a number if possible.

+"123" returns integer 123, +"abc" raises a run-time conversion error.

```
#
# numeric-identity.icn, demonstrate the prefix plus operator
#
procedure main()
write(+-1)
write(+--1)
write(+++---1)

# the size of -1 is 2 as in "-1", identity does not change that
write(*+-1)

# the size of -0 is 1 as in "0", identity does not change that
write(*+-0)

write("type(\"123\"): " || type("123"))
write("type(+\"123\"): " || type(+"123"))

# raises a run-time error
write(+"abc")
end
```

```
-1
1
-1
2
1
type("123"): string
```

```
type(+"123"): integer

Run-time error 102
File numeric-identity.icn; Line 29
numeric expected
offending value: "abc"
Traceback:
   main()
   {+"abc"} from line 29 in numeric-identity.icn
```

5.2.4 - (negate)

Negate. Reverses the sign of the given operand.

-*x* is the signed reverse of *x* as a number if possible.

-"-123" returns integer 123. —1 is -1. -"abc" raises a run-time conversion error.

```
#
# negate-operator.icn, demonstrate the prefix minus operator
#
procedure main()
write(-"-123")
write(-+1)
write(--1)
write(---1)
end
```

```
123
-1
1
-1
```

5.2.5 / (null test)

Null test.

/x : *x* or &fail

Test *expr* for null. Succeeds and returns *x* if *x* is null.

This doesn't just return the value *&null*, but a reference to the expression if appropriate, which can be set to a value. This is very useful for optional arguments and optional default values.

```
/var := "abc"
```

The null test operator will succeed and produce a reference to `var` if it is *&null*, and the surrounding assignment can then set a new value. If `var` is already set, the null test will fail and the assignment operation is not attempted. That is not the only use for the null test operator, but it the most common.

5.2.6 \ (not null test)

Not null test.

\ *x* : *x* or &fail

Test *expr* for nonnull. Succeeds and returns *x* if *x* is not null.

A handy operator when dealing with possibly unset values.

```
L |||:= [\var]
```

The nonnull test will return `var`, unless it is *&null* in which case the operator fails. Only values that have been set will be appended to the *List (arrays)* `L` in the expression shown above.

5.2.7 . (dereference)

Dereference.

. *x* : *x*

The dereference operator. It returns the value *x*.

Unless required by context, Unicon will pass references to variables (so they can be set to new values). The dereference operator returns the actual value of *x* even if the surrounding context may prefer a reference.

5.2.8 = (anchored or tab match)

Tab match or anchored pattern match. Historically this unary operator has been called tab-match. *Even though it is extended for use with Unicon pattern types with different semantics, and not just string scanning, it will likely always be spoken as tab-match.*

=*x* : *string* | *pattern*

Equivalent to `tab(match(s))` for string *s*, or anchors a pattern match when *x* is a pattern, `=p`.

```
#
# tabmatch-operator.icn, tab match and anchored pattern match
#
procedure main()
subject := "The quick brown fox jumped over the lazy dog."
space := ' '

subject ? {
    ="The"
    =space
    ="quick"
    =space
    ="brown"
    =space
    animal := tab(upto(space))
}
write(animal)
end
```

```
fox
```

5.2.9 | (repeated alternation)

Repeated generation.

| *x* : *x**

The repeated alternation operator. `|x` generates results from evaluating the expression *x* over and over again, in an infinite loop.

5.2.10 ? (random element)

Random element.

Returns a random element from *x*.

- Number, returns a number from 0 to n - 1.
- String, returns a random single character subscript of *s*.
- List, returns a random element from the list.
- Table, returns a random value from the table - not the key.

Todo

fill out all the random element types.

5.2.11 @ (activation)

Activate co-expression. Shown here as a unary operator, but it is actually a binary operator with an optional initial transmitted value.

When used in a unary prefix mode the default value *&null* is supplied to the co-expression.

5.2.12 ~ (cset complement)

cset complement.

~*x* returns the cset consisting of all the characters not found in *x*.

Unicon csets include byte values from 0 to 255, and the complement will be limited to that range.

Other *Set* data cannot really use a complement operator, as that would result in an unbounded infinite set.

```
#
# complement-operator.icn, demonstrate the cset complement operator
#
procedure main()
cs := 'ABCDEVWXYZabcdevwxyz'
write(*~cs)
write((~cs)[48+:60])
end
```

```
236
/0123456789:;<=>?@FGHIJKLMNOPQRSTU[\]^_`fghijklmnopqrstu{|}~
```

5.3 Binary Operators

Unicon includes a rich set of binary operators, and many can be augmented with assignment, making for a lengthy list of operator symbols.

Note: In terms of precedence rules, augmented assignment operators are equal in precedence to assignment, not the level of precedence for the operator being augmented.

For instance: `a *:= b + c` parses as `a := a * (b + c)`

The source expression: `a := a * b + c` parses as `a := (a * b) + c`

Some few binary operators have optional left hand side parameters, and those are also included in the Unary Operator list above for completeness (co-expression activation for instance).

5.3.1 & (conjunction)

Conjunction. *x1* & *x2* produces *x2* if *x1* succeeds.

5.3.2 &:= (augmented &)

Augmented conjunction. *x1* &:= *x2* produces *x2* if *x1* succeeds and assigns the result to *x1*. If *x1* fails, no reassignment occurs.

5.3.3 | (alternation)

Alternation. *x1* | *x2* yields the result of *x1* followed by *x2*, in that order. The alternation operator is a generator.

5.3.4 || (concatenation)

Concatenation. *x* || *y* produces a new value that is the concatenation of the strings (or patterns) *x* and *y*.

5.3.5 ||:= (augmented ||)

Augmented concatenation. *x* || *y* produces a new value that is the concatenation of the strings (or patterns) *x* and *y*, and assigns the result to *x*.

5.3.6 ||| (list concatenation)

List concatenation operator.

5.3.7 |||:= (augmented |||)

Augmented list concatenation operator.

5.3.8 ? (string scan)

One of the key features of Unicon. *x*1 ? *x*2 invokes the string scanner. The expression *x*1 is the *&subject* and *x*2 can be arbitrarily complex. See *String Scanning*.

5.3.9 ?:= (augmented ?)

Augmented string scanning. The result of the scan is assigned to *x*1.

5.3.10 ?? (pattern scan)

Pattern scan operator. Similar to string scanning, but *expr* is a *Pattern* not a general Unicon expression.

5.4 Operator idioms

The chaining capabilities inherent in the nature of Unicon expressions can lead to some very powerful programming techniques. Some of these techniques are common practise and become idioms, some are yet to be discovered and some few are just curiosities best left alone as they may be more confusing than separating the expression for the sake of clarity.

The order of operations *Precedence* rules can play a large role when building up expression chains.

For example, string scanning has a lower precedence than assignment:

```
result := subject ? {
    step1()
    step2()
}
```

In the code above, `result` is set to the original contents of `subject`, not the actual result produced by the scanning environment.

The above is equivalent to:

```
(result := subject) ? {
    step1()
    step2()
}
```

To get the scanning result, a rather odd looking order of operations control ends up as:

```
result := (subject ? {
    step1()
    step2()
})
```

The idiomatic Unicon expression uses augmented assignment:

```
subject ?:= {
    step1()
    step2()
}
```

With the downside that `subject` (the data referenced by the variable name used) is changed after scanning. Another option is to place the scan inside a procedure and use

```
suspend subject ? {
    step1()
    step2()
}
```

Or, get used to seeing parenthesis control around these often times rather complex operations. You will see the `r := (s ? {code})` form relatively frequently in *IPL* listings.

Todo

more examples and clarification required.

5.5 Operator functions

Almost all Unicon operators can be used as function call expressions.

A few examples, summing a list of arguments and demonstrating the right hand side value given a condition test, along with some straight up arithmetic.

```
#
# opfuncs.icn, Calling operators as functions
#
# tectonics: unicon -s opfuncs.icn -x
#
link lists
procedure main()
    write("Multiply: ", "*"(3,5))
    write("Greater than (RHS): ", ">"(5,3))
    write("Less than (RHS): ", "<"(5,10))

    write("cset intersection: ", "**"('abcdefg', 'efghijk'))
    L := ['abcdefg', 'efghijk']
    write("cset union from list: ", "++"!L)


    result := 0
    L := [1,2,3,4,5,6,7,8,9]
    every result := "+"(result, !L)
    write("Sum of list: ", result)

    write("Section: ", limage("[:]"(L,3,6)))

    writes("ToBy: ")
    every writes("..."(1,9,2) || " ")
    write()
end
```

```
prompt$ unicon -s opfuncs.icn -x
Multiply: 15
Greater than (RHS): 3
Less than (RHS): 10
cset intersection: efg
cset union from list: abcdefghijk
Sum of list: 45
Section: [3,4,5]
ToBy: 1 3 5 7 9
```

CHAPTER

SIX

RESERVED WORDS

6.1 Unicon reserved words

Unicon uses a fair number of reserved words. There are 38 reserved words in Unicon version 13. That might seem like a lot, and it is, until you compare Unicon with the likes of *COBOL*[1].

Emphatics[2] is used here to denote action causing reserved words. In many other languages, they would be called *statements*, but Unicon has no *statements*. These reserved words all count as *expressions* and return results along with triggering control flow actions to occur. *Almost* everything[3] in Unicon is an *expression*. Listed here as emphatic, these action causing reserved words act, and react, as expressions.

While most Unicon reserved words are actionable, flow control expressions, some others take a supporting role, as either *declarative* expressions, or as syntax markers for optional phrasing. For example, the *by* reserved word is only usable when paired with *to*, and will not work alone as a valid expression. *else* is only applicable within an *if* expression. *then* is a marker word that separates *expr*1 from *expr*2 in an *if* expression. Other words inform the compiler about data management, with variables being declared *global*, *local* or *static*.

Reserved words cannot be used as identifier names.[4]

All reserved words count as expressions (or assist in defining an overall expression).

[1] COBOL has over 500 reserved words, some implementations listing over 1,000.

[2] This chapter was originally called "Statements", but that is a technically incorrect and harmfully misleading naming convention. Unicon has no statements. All expressions return results, emphatic reserved words lead flow control and count as expressions.

[3] If you try hard, you may find a Unicon syntax element that does not count as an expression; *Unicon preprocessor* directives come to mind, as would the *all* reserved word that is part of *invocable* in `invocable all`.

[4] Unicon can be *tricky* sometimes. Everything is an expression, and variables do not need to be declared before use. From the Programming Corner, Icon Newsletter 35, March 1st, 1991:

while next := read() do write(next)

That code fragment causes an endless loop. The `next` is not parsed as a new variable identifier, but is valid as the *expr*1 part of the assignment operator *expr*1 `:=` *expr*2. Unicon accepts that code, which at runtime evaluates the `next`, as part of the assignment expression, immediately transferring control back to the top of the while loop; an infinite loop.

6.1.1 Reserved word list

abstract	*else*	*link*	*return*
all	*end*	*local*	*static*
break	*every*	*method*	*suspend*
by	*fail*	*next*	*then*
case	*global*	*not*	*thread*
class	*if*	*of*	*to*
create	*import*	*package*	*until*
critical	*initial*	*procedure*	*while*
default	*initially*	*record*	
do	*invocable*	*repeat*	

6.2 Unicon action words

Once again, all these reserved words are *expressions* and need to be thought of that way for effective Unicon programming. *Verb* might be an appropriate name for these action words, but the term used in this document is *emphatic*.

6.2.1 break

Exit the nearest enclosing loop and evaluate optional expression.

break [*expr*]

If the *expr* includes `break`, multiple levels of nesting can be exited.

For instance; `break break next` will exit two inner loop levels and restart at the top of an outer loop. The example below breaks out of one level, jumping to the top of the outer.

```
#
# break.icn, loop exit sample
#
# this sample is utterly contrived
#
procedure main()
    every i := 1 to 3 do {
        every j := 1 to 4 do {
            write("i is ", i, ", j is ", j)
            # when i is 2, restart the outer loop
            if i == 2 then break next

            # when j is 2, just break out of the inner loop
            if j == 2 then break write("poor j, never gets to 3")
        }
        write("after the inner loop, i is ", i)
    }
end
```

Sample run:

```
prompt$ unicon -s break.icn -x
i is 1, j is 1
```

```
i is 1, j is 2
poor j, never gets to 3
after the inner loop, i is 1
i is 2, j is 1
i is 3, j is 1
i is 3, j is 2
poor j, never gets to 3
after the inner loop, i is 3
```

6.2.2 case

case *expr* **of** {

*expr*1: *expr*2

...

*expr*3: *expr*4

[**default**: *expr*n]

}

A selection expression where *expr* is compared to the compound expressions that follow. First matching *expr*1 (or *expr*3, etc) will evaluate and produce the resulting value of the paired *expr*2. If no matches are found, the optional `default` *expr*n case is evaluated. With no `default` and no comparison matches, the `case` expression will fail.

```
#
# case.icn, demonstrate case selection
#
procedure main()
    c := "initial value"
    t := "e"
    # no match, no default, case will fail, c not re-assigned
    c := case t of {
            "a": "b"
            "c": "d"
         }
    write(image(c))

    # no match, with default, c will be assigned to "f"
    c := case t of {
            "a": "b"
            "c": "d"
            default: "f"
         }
    write(image(c))

    # first match satisfies case, c is assigned to "b"
    c := case t of {
            map("E"): "b"
            "e"     : "d"
            default : "f"
         }
    write(image(c))

end
```

Sample run:

```
prompt$ unicon -s case.icn -x
"initial value"
"f"
"b"
```

See *default*, *of*.

6.2.3 create

create *expr* : *co-expression*

Create a co-expression for *expr*. Not evaluated until explicitly requested by the invocation operation or *spawn* function.

Co-expressions are a very powerful feature of Unicon. A feature that is still missing from many other programming languages.

See @ for information on co-expression activation.

```
#
# create.icn, Demonstrate co-expression create
#
procedure main()
    coex := create(1 to 3)

    write("co-expression now ready to be invoked, at will")
    write(@coex)

    write("Do some other computing")
    write(@coex)

    write("get last expected result")
    write(@coex)

    if v := @coex then
        write("co-expression has delivered more results than expected")
end
```

Giving:

```
co-expression now ready to be invoked, at will
1
Do some other computing
2
get last expected result
3
```

6.2.4 critical

critical `mtx : expr` is the equivalent of *lock*(mtx); *expr* ; *unlock*(mtx), where `mtx` is a mutual exclusion facility control for structure `x` defined with `mtx :=` *mutex* `(x)`.

This allows for more concise control of critical memory sections when multi-threading.

For instance:

```
x := x + 1
```

That code is actually at risk in threaded applications. It is a three step process. Fetch x, increment x, assign to x. Multiple threads, without synchronization might all evaluate the *fetch x* part of those steps at the same time. The final result would be incorrect.

```
critical mtx : x := x + 1
```

That code will ensure that each thread is allowed to complete all steps of the expression before allowing another thread access to the critical section.

```
#
# critical.icn, Demonstrate mutually exclusive expression control
#
global x

procedure main()
    x := 0

    mtx := mutex()
    T1 := thread report(mtx)
    T2 := thread other(mtx)

    # due to the nature of threading
    # the values displayed in report
    # might be 1,2,3,4 or 1,10,11,12 or 1,2,11,12 etc
    wait(T1 | T2)

    # final result will always be 12
    write("x is ", x)
end

# increment and report
procedure report(mtx)
    every 1 to 4 do {
        critical mtx : x := x + 1
        write(x)
    }
end

# just increment
procedure other(mtx)
    every 1 to 8 do
        critical mtx : x := x + 1
end
```

Giving:

```
1
10
11
12
x is 12
```

6.2.5 every

every *expr*1 [**do** *expr*2]

`every` will produce all values from a generator. Internally, the `every` statement always fails, causing *expr*1 to be resumed for more results. An optional *do* *expr*2 will be evaluated before each resumption of *expr*1.

`every` is a one of the work horse words in Unicon.

Of note: *while* and *every* play different roles in Unicon, and the distinction can be confusing at first, especially when dealing with *read*.

```
every read() do stuff()

while read() do stuff()
```

That first expression is wrong, or at least it may not do what a new Unicon programmer expects. *No Unicon expression that compiles is ever wrong, it just does what it is told.* The example below reads its own source, to highlight the effect:

```
# ## top-line read by every ##
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: September, 2016
# Modified: 2016-10-03/22:51-0400
##+
#
# every.icn, Demonstrate the every emphatic
#
procedure main()
    # so many things to show for every
    # semantics wise, every continuously fails, causing the Unicon goal
    # directed evaluation engine to seek alternatives for a successful
    # expression outcome

    # write some to-by numbers
    every writes(0 to 8 by 2) do writes(" ")

    # write some alternatives
    write()
    every write(1 | 2 | "abc")

    # every out a string, one character at a time
    every write(!"xyzzy")

    # write out this file; not always a job for every
    write()
    f := open(&file, "r") | stop("Cannot open ", &file, " for read")

    write("read in an every loop may not do what you expect")
    # don't do this
    every now := read(f) do stuff(now)
    # read returns a result, not suspend
    # every succeeds, job complete, after the first record

    # do this, with while
    write()
    write("### read (the rest) in a while loop ###")
```

```
    write()
    while now := read(f) do stuff(now)
    close(f)
end

procedure stuff(now)
    write(now)
end
```

Sample run:

```
prompt$ unicon -s every.icn -x
0 2 4 6 8
1
2
abc
x
y
z
z
y

read in an every loop may not do what you expect
# ## top-line read by every ##

### read (the rest) in a while loop ###

##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: September, 2016
# Modified: 2016-10-03/22:51-0400
##+
#
# every.icn, Demonstrate the every emphatic
#
procedure main()
    # so many things to show for every
    # semantics wise, every continuously fails, causing the Unicon goal
    # directed evaluation engine to seek alternatives for a successful
    # expression outcome

    # write some to-by numbers
    every writes(0 to 8 by 2) do writes(" ")

    # write some alternatives
    write()
    every write(1 | 2 | "abc")

    # every out a string, one character at a time
    every write(!"xyzzy")

    # write out this file; not always a job for every
    write()
    f := open(&file, "r") | stop("Cannot open ", &file, " for read")

    write("read in an every loop may not do what you expect")
```

```
    # don't do this
    every now := read(f) do stuff(now)
    # read returns a result, not suspend
    # every succeeds, job complete, after the first record

    # do this, with while
    write()
    write("### read (the rest) in a while loop ###")
    write()
    while now := read(f) do stuff(now)
    close(f)
end

procedure stuff(now)
    write(now)
end
```

There is an alternative to `while` and `read`. The *generate elements* operator `!`, can be used with the file type, so it becomes `every` friendly. No `read` is called (which returns a result, that return satisfies the every). With the `!` operators, Unicon files generate the elements within the resource.

```
# ## top-line read by every ##
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: September, 2016
# Modified: 2016-10-03/22:51-0400
##+
#
# every-gen.icn, Demonstrate the every emphatic with generate elements
#
procedure main()
    # write out this file; not always a job for every
    write()
    f := open(&file, "r") | stop("Cannot open ", &file, " for read")

    write("read in an every loop does not do what you may expect")
    # don't do this
    every now := read(f) do stuff(now)
    # read returns a result, not suspend
    # every succeeds, job complete, after the first record

    # now using the file itself with the ! operator
    write()
    write("### generate (the rest) ###")
    write()
    every now := !f do stuff(now)
    close(f)
end

procedure stuff(now)
    write(now)
end
```

Every with a file content generator:

```
prompt$ unicon -s every-filegen.icn -x

read in an every loop does not do what you may expect
# ## top-line read by every ##

### generate (the rest) ###

##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: September, 2016
# Modified: 2016-10-03/22:51-0400
##+
#
# every-gen.icn, Demonstrate the every emphatic with generate elements
#
procedure main()
    # write out this file; not always a job for every
    write()
    f := open(&file, "r") | stop("Cannot open ", &file, " for read")

    write("read in an every loop does not do what you may expect")
    # don't do this
    every now := read(f) do stuff(now)
    # read returns a result, not suspend
    # every succeeds, job complete, after the first record

    # now using the file itself with the ! operator
    write()
    write("### generate (the rest) ###")
    write()
    every now := !f do stuff(now)
    close(f)
end

procedure stuff(now)
    write(now)
end
```

The conceptual difference between *return* and *suspend* is an important one for all Unicon programmers to understand. It can take a few practice runs before it sinks in. *Traditional* programming experiences can make this even harder, and you may find yourself slipping back without realizing it from time to time; before an initial program run reminds you of the difference. No harm done, *as long as alpha testing is always part and parcel of your Unicon development cycle*. Unicon has features that set it apart from most other programming languages. It is important to *think Unicon* to take maximum advantage of these features.

`every` feeds on generators by forcing the goal-directed evaluation engine to keep trying alternatives. If the expression completes (either by returning a result (with *return*) or failing, `every` terminates. `while` on the other hand, is a procedural expression with a more traditional flow control paradigm. Unicon allows for both methods of loop management.

See also:

repeat, *suspend*, *until*, *while*, *break*, *next*, *return*, *!*

6.2.6 fail

fail causes the enclosing procedure or method to terminate immediately and produce no result, signalling failure. The invocation may not be resumed. `fail` is equivalent to `return &fail`.

```
#
# fail.icn, demonstrate the fail emphatic expression
#
procedure main()
    # resumption will terminate after the hard fail expression
    every i := subproc() do write(i)
end

procedure subproc()
    # will only deliver 1 and 2
    suspend j := 1 to 4 do if j = 2 then fail
end
```

Sample run:

```
1
2
```

6.2.7 if

if *expr*1 **then** *expr*2 [**else** *expr*3]

The Unicon `if` emphatic expression evaluates *expr*1 and if it succeeds, proceeds to evaluate *expr*2, given after the `then` reserved word. If the optional `else` clause is present, then a failed *expr*1 will proceed to evaluate *expr*3 instead. The overall result is either failure, or the result of *expr*2 (or *expr*3) depending on the outcome of *expr*1.

```
#
# if.icn, Demonstrate the if emphatic reserved word expression
#
procedure main()
   if 1 = 1 then write("yes, one equals one")
   if 1 = 2 then write("not displayed")
   else write("one is not equal to two")

   a := 1
   a := if 1 = 2 then write("if fails, a not set, remains 1")
   write(a)

   # a will get the then expression result
   a := if 1 = 1 then 2
   write(a)

   # a will get the else expression result
   a := if 1 = 2 then 2 else 3
   write(a)
end
```

Giving:

```
prompt$ unicon -s if.icn -x
yes, one equals one
one is not equal to two
1
2
3
```

6.2.8 initial

An **initial** *expr* is only evaluated the first time a procedure is entered. Mainly used for initializing *static* variables, when an initializing expression is not sufficient. Initializing a *local* variable in an initial expression is usually not desirable, `local` variables will lose the value before second entry.

```
#
# initial.icn, demonstrate an initial expression
#
procedure main()
    c := subproc()
    write(image(c))

    c := subproc()
    write(image(c))
end

procedure subproc()
    local a
    static b
    # a will only be set on the first call
    initial {
        a := "abc"
        b := "bcd"
    }
    write(b)
    return a
end
```

Giving:

```
bcd
"abc"
bcd
&null
```

See *procedure*, *static*.

6.2.9 initially

initially `[(parameters)]` creates a special method that is invoked when an object is instantiated. If the `initially` section has declared parameters, they are used as the parameters of the constructor for the objects of that class.

```
#
# initially.icn, demonstrate class initialization control
#
$define space ' '
class person(first, last, middle)
    method show()
        writes(first)
        if \middle then writes(space, middle)
        write(space, last)
    end

    initially
        /first := "Jane"
        /last := "Doe"
end

procedure main()
    c := person("Clinton", "Jeffery", "L.")
    c.show()

    c := person()
    c.show()

    c := person("John")
    c.show()
end
```

Giving:

```
Clinton L. Jeffery
Jane Doe
John Doe
```

6.2.10 next

The **next** expression forces a loop to immediately skip to its next iteration, bypassing any subsequent code that makes up the remainder of the loop body. Control flow returns to the beginning of the loop expression.

```
#
# next.icn, Demonstrate the next emphatic reserved word expression
#
procedure main()
    # displays 1, 3, 4
    every i := 1 to 4 do {
        if i = 2 then next
        write(i)
    }

    # displays i 1,3,4 paired with inner j 1,3
    every i := 1 to 4 do {
        every j := 1 to 3 do {
            # break out of the j loop and reiterate the i loop
            if i = 2 then break next
            # reiterate the j loop
```

```
            if j = 2 then next
            write("i: ", i, " j: ", j)
        }
    }
end
```

Giving:

```
prompt$ unicon -s next.icn -x
1
3
4
i: 1 j: 1
i: 1 j: 3
i: 3 j: 1
i: 3 j: 3
i: 4 j: 1
i: 4 j: 3
```

See also:

break, *every*, *repeat*, *suspend*, *while*, *until*

6.2.11 not

not *expr* reverses the success or failure status of *expr*.

Some care must be shown when using `not` in Unicon. It is reversing the sense of success and failure, not acting as a boolean operator as used with most other programming languages. For instance:

```
not 1
```

Does not produce a zero, as might be assumed by a C programmer, it signals failure, reversing the successful `1` result to fail.

See the bitwise functions *iand*, *icom*, *ior*, *ishift*, and *ixor* when bit twiddling is called for.

icom(i) would be the closest Unicon equivalent to **not** in most boolean value control flow languages.

```
not &fail
```

The code above will produce a *successful* *&null* result.

```
#
# not.icn, Demonstrate the success/failure reversal reserved word
#
procedure main()
    # no write occurs, failure propogates
    write(not 1)

    a := not &fail
    write("not &fail is ", image(a))

    if not every 1 to 2 then write("turn every failure into success")
end
```

Giving:

```
prompt$ unicon -s not.icn -x
not &fail is &null
turn every failure into success
```

6.2.12 repeat

repeat `expr` introduces and infinite loop of *expr*. It is up to an element inside *expr* to exit the loop or terminate the program.

```
#
# repeat.icn, demonstrate the infinite repeat loop
#
procedure main()
    i := 0
    repeat {
        i +:= 1
        case i of {
            2: next
            4: break
        }
        write(i)
    }
    write("loop exited at ", i)
end
```

Sample run:

```
1
3
loop exited at 4
```

See also:

every, *return*, *suspend*, *until*, *while*, *break*, *next*, *exit*, *stop*

6.2.13 return

return `[expr]` exits a procedure or method, producing *expr* as its result. The invocation may not be resumed. Default value for the optional *expr* is *&null*.

```
#
# return.icn, demonstrate how return terminates a generator
#
procedure main()
    every write(r := subproc())

    # set an exit code from last given result; 42
    exit(r)
end
```

```
# generate sequence 1 to n, until the result of that sequence hits 3
procedure subproc()
    suspend i := !seq() do {
        if i = 3 then return 42
    }
end
```

Sample run:

```
prompt$ unicon -s return.icn ; ./return ; echo "shell status: $?"
1
1
2
1
2
3
42
shell status: 42
```

See also:

procedure, *method*, *suspend*

6.2.14 suspend

suspend *expr* [**do** *expr*1]

Suspend a value from a procedure. When resumed, the optional do *expr*1 will be evaluated before control passes to resumption point. If *expr* is a generator, resumption of the suspended procedure will resume the generator and suspend again with the result produced.

```
#
# suspend-sample
#
procedure main()
    if susproc() == 5 then write("got 5") else write("no 5")
end

procedure susproc()
    local val
    suspend val := seq() do write("val was: ", val)
end
```

Sample run

```
prompt$ unicon -s -t suspend-sample.icn -x
             :       main()
nd-sample.icn:   12  | susproc()
nd-sample.icn:   17  | susproc suspended 1
nd-sample.icn:   12  | susproc resumed
val was: 1
nd-sample.icn:   17  | susproc suspended 2
nd-sample.icn:   12  | susproc resumed
val was: 2
nd-sample.icn:   17  | susproc suspended 3
```

```
nd-sample.icn:   12  | susproc resumed
val was: 3
nd-sample.icn:   17  | susproc suspended 4
nd-sample.icn:   12  | susproc resumed
val was: 4
nd-sample.icn:   17  | susproc suspended 5
got 5
nd-sample.icn:   13  main failed
```

See also:

every, *repeat*, *until*, *while*, *break*, *next*

6.2.15 thread

thread `expr` is equivalent to `spawn(create expr)`. Create a thread and then start it running. The thread handle is produced, of type *Unicon Co-Expressions*.

Threading in Unicon is a wide ranging topic, best described by the feature author, *Jafar Al-Gharaibeh*, in Unicon technical report UTR14, http://unicon.org/utr/utr14.pdf. This document is also shipped with the Unicon sources as `doc/utr/utr14.docx`

A small producer/consumer example:

```
#
# thread.icn, Demonstrate thread messaging
#
# requires Concurrency build of Unicon
#
procedure main()
    pTr := thread producerRace()
    cTr := thread consumerRace(pTr)
    every wait(pTr | cTr)
    write("racing producer/consumer complete")
    write()

    pT := thread producer()
    cT := thread consumer(pT)
    every wait(pT | cT)
    write("main complete")
end

#
# This code can easily trigger incorrect results due to a race condition
#

# send messages to the thread out-box
procedure producerRace()
    write("producer entry")
    every !6@>
end

# receive messages from the out-box of the producer thread
procedure consumerRace(T)
    write("consumer entry")
    while write(<@T)
```

```
end

# What follows is the suggested update from Jafar
# It alleviates the race condition where consumerRace
# can complete before the producerRace even starts
#
# an original capture:
#
# JMBPro:proj jafar$ ./thrd
# producer entry
# consumer entry
# racing producer/consumer complete

#
# This is the better code...
#

# send messages to the thread out-box
procedure producer()
    write("synch producer entry")
    every !6@>
    # produce &null (&null@>)  to signal the end
    @>
end

# receive messages from the out-box of the producer thread
procedure consumer(T)
    write("blocking consumer entry")
    # blocking receive.
    while write(\<<@T)
end
```

Sample run:

```
prompt$ unicon -s thread.icn -x
producer entry
consumer entry
1
2
3
4
5
6
racing producer/consumer complete

synch producer entry
blocking consumer entry
1
2
3
4
5
6
main complete
```

See also:

create, *spawn*, *wait*

6.2.16 to

*expr*1 **to** *expr*2 [**by** *expr*3] produces the integer sequence from *expr*1 to *expr*2 with an optional step value given by *expr*3. The default step is `1`.

The initial value of *expr*1 is always included in the sequence. The last result will be less than or equal to *expr*2, depending on the step size of *expr*3.

```
#
# to.icn, demonstrate to-by sequence generator
#
procedure main()
    writes("1 to 3:        ")
    every writes((1 to 3) || " ")

    writes("\n1 to 5 by 2:  ")
    every writes((1 to 5 by 2) || " ")

    writes("\n1 to 6 by 2:  ")
    every writes((1 to 6 by 2) || " ")

    writes("\n1 to 7 by 2:  ")
    every writes((1 to 7 by 2) || " ")

    writes("\n3 to 1:")
    every writes((3 to 1) || " ")

    writes("\n3 to 1 by -1: ")
    every writes((3 to 1 by -1) || " ")

    write()
end
```

Giving:

```
prompt$ unicon -s to.icn -x
1 to 3:       1 2 3
1 to 5 by 2:  1 3 5
1 to 6 by 2:  1 3 5
1 to 7 by 2:  1 3 5 7
3 to 1:
3 to 1 by -1: 3 2 1
```

See also:

by

6.2.17 until

until *expr*1 [**do** *expr*2] loops until *expr*1 succeeds evaluating the optional *expr*2 after each test.

```
#
# until.icn, Demonstrate loop until success
#
procedure main()
    i := 1
    until i = 4 do {
        write(i)
        i +:= 1
    }
end
```

Giving:

```
1
2
3
```

See also:

every, *repeat*, *while*, *break*, *next*

6.2.18 while

while *expr*1 [**do** *expr*2] loops until *expr*1 fails, evaluating the optional *expr*2 after each test.

```
#
# while.icn, Demonstrate loop until fail
#
procedure main()
    i := 1
    while i < 4 do {
        write(i)
        i +:= 1
    }
end
```

Sample run:

```
1
2
3
```

See also:

every, *repeat*, *until*, *break*, *next*

6.3 Declarative expressions

A fair number of the Unicon reserved words are of a declarative nature:

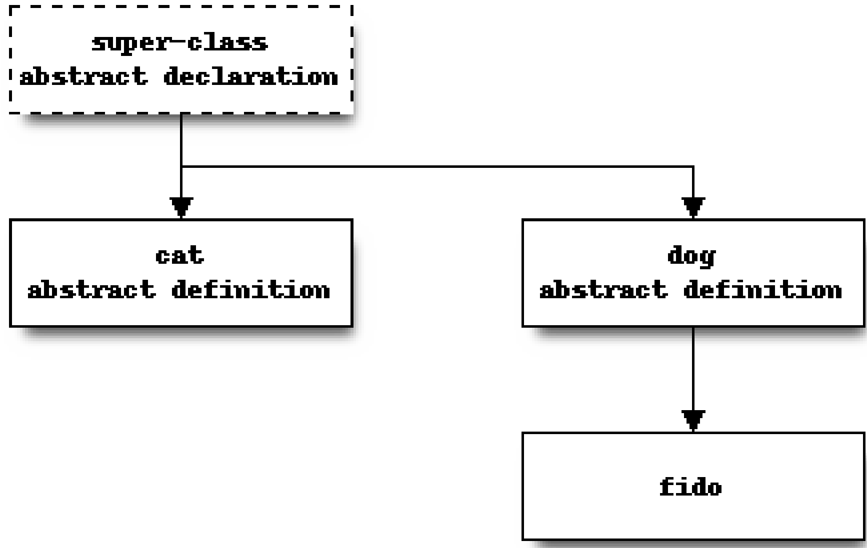
- informing the compiler how defined items are to be treated

- for the creation of program elements at compile time
- and other construction details.

6.3.1 abstract

A *method* attribute. Allows a *class* hierarchy to include methods that must be defined lower in the inheritance chain.

abstract methods have no implementation at the point of declaration. A sub-class is responsible for defining implementation code.




```
#
# abstract.icn, demonstrate an abstract method declaration
#
class animal(tag, position)
    # all animals conceptually move by changing position
    method moving(d)
        position +:= d
        write(tag, " moved to ", position)
    end

    # a generic animal needs a specific way of speaking
    abstract method speak(words)

    initially
        position := 0
end

# the dog class
class dog : animal(tag)
    method speak(words)
        write("bark? woof? grr? eloquent soliloquy?")
```

```
    end
end

# create a dog, move it and ask it to speak
procedure main()
   fido := dog("fido")
   fido.moving(4)
   fido.speak()
end
```

Sample run:

```
fido moved to 4
bark? woof? grr? eloquent soliloquy?
```

See *Unicon GUI* by *Robert Parlett* for more practical examples of `abstract` methods when designing graphical user interface hierarchies.

6.3.2 class

class *name* [*: superclass :...*] `(attributes)` is a Unicon object programming declarative, defining *name*. *name* can then be used to create instances of the class.

Class hierarchies are defined with an (optional) colon separated list of parent classes.

`class` definitions include

- *abstract* method declarations
- *method* definitions
- *initially* attribute initialization blocks

`class` definitions end with the *end* reserved word.

```
#
# class.icn, demonstrate a class declaration
#
# THIS CODE IS UNFINISHED BUSINESS

$define space ' '
class creator(field)
    method show(f)
        write(image(r := (\f | field)))
        return r
    end
end

procedure p()
    r := "abc"
    return r
end

procedure main()
    c := creator("name")
    write(type(c), " ", image(c))
    a := c.show()
```

```
    b := c.show('a')
    pr := p()
    write(image(a), space, image(b), space, image(pr))
end
#$include "abstract.icn"
```

Sample run:

```
creator__state object creator_1(1)
"name"
'a'
"name" 'a' "abc"
```

See *abstract* for more details on the code sample.

In Unicon, `class` can be compared to *record*; declaring a name that is used later to define storage. Object oriented programming is orders of magnitude more powerful, encapsulating code along with the data values, all within an inheritance hierarchy.

6.3.3 global

global var [, var...] declares one or more global variables. Placed outside of procedures, the named variables will be accessible inside all procedures and methods of the program.

```
#
# global.icn, demonstrate global variables
#
global var, other
procedure main()
    var := other := 42
    subproc()
    # var will be changed, other will not
    write(var)
    write(other)
end

# access the global "var", but create a local "other"
procedure subproc()
    local other
    write(var)
    var := 84
    other := 21
end
```

Giving:

```
42
84
42
```

See also:

Variable scope, *local*, *static*

6.3.4 import

import `package [package...]` imports one or more packages. This is a programming in the large Unicon feature. Using packages relies on an external databases, `uniclass` to help manage the namespace resolution.

```
#
# import.icn, demonstrate a package import
#
import example

procedure main()
    example()
end
```

Sample run:

```
prompt$ unicon import.icn -x
Parsing import.icn: ..
/home/btiffin/unicon-git/bin/icont -c   -O import.icn /tmp/uni51075390
Translating:
import.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  import.u -x
Linking:
Executing:
in example procedure of example package
```

See also:

package

6.3.5 invocable

invocable `[all | name,...]` allow string invocation of the given name(s), or any procedure with the *all* reserved word.

```
#
# invocable.icn, Demonstrate string invocation
#

# an alternative is invocable all to allow any string invocations
invocable "subproc"

procedure main()
    # allowed
    f := "subproc"
    f(42)

    # will cause a runtime error, not listed as invocable
    f := "other"
    f(42)
end

procedure subproc(in)
```

```
    write("in subproc with: ", in)
end

procedure other(in)
    write(in)
end
```

Sample run will end in a purposeful error, “other” is not set invocable.

```
prompt$ unicon -s invocable.icn -x
in subproc with: 42

Run-time error 106
File invocable.icn; Line 22
procedure or integer expected
offending value: "other"
Traceback:
   main()
   "other"(42) from line 22 in invocable.icn
```

6.3.6 link

link `library [, library...]` links other ucode into this program.

Unicon ucode files are created with `unicon -c source.icn`. This option creates a library that can be linked into other programs.

The *IPL* provides hundreds of ready to link library procedures.

```
#
# link.icn, demonstrate Unicon link
#

# link in the reflective extended image function
link ximage

procedure main()
    L := [1,2,3]
    write("image of L:")
    write(image(L))

    write("\nximage of L:")
    write(ximage(L))
end
```

Sample run:

```
prompt$ unicon -s link.icn -x
image of L:
list_1(3)

ximage of L:
L1 := list(3)
   L1[1] := 1
   L1[2] := 2
   L1[3] := 3
```

```
prompt$ unicon -s -t link.icn -x
              :       main()
image of L:
list_1(3)

ximage of L:
link.icn      :   21  | ximage(list_1 = [1,2,3],&null,&null)
ximage.icn    :  201  | ximage returned "L1 := list(3)\n  ..."
L1 := list(3)
   L1[1] := 1
   L1[2] := 2
   L1[3] := 3
link.icn      :   22  main failed
```

See also:

Icon Program Library

6.3.7 local

local `var [, var...]` declares one or more variable as being within local scope of a procedure or method body.

```
#
# local.icn, demonstrate local override of global variables
#
global var
procedure main()
    var := 42
    subproc()
    # var will not be changed
    write("var in main: ", var)
end

# create a local "var", global var not normally accessible inside subproc
procedure subproc()
    local var
    var := 84
    write("var in subproc: ", var)
end
```

Sample run:

```
var in subproc: 84
var in main: 42
```

6.3.8 method

method `name` declares a method within a *class*. An object oriented feature of Unicon.

```
#
# method.icn, demonstrate a class method definition
#
class sample()
    method sample()
        write("in sample of sample")
    end
end

procedure main()
    instance := sample()
    instance.sample()
end
```

Sample run:

```
prompt$ unicon -s method.icn -x
in sample of sample
```

6.3.9 package

package `name` declares the current source unit to be part of a package, `name`. This defines a namespace for all procedures within the package.

Package access uses the *import* reserved word.

Packages are a programming in the large feature of Unicon. Using packages relies on an external database, `uniclass`, that manages the namespace resolution.

```
#
# package.icn, Demonstrate Unicon packages (poorly)
#
package example

procedure main()
    write("part of package sample")
end

procedure example()
    write("in example procedure of example package")
end
```

Giving:

```
prompt$ unicon package.icn
Parsing package.icn: package.icn is already in Package example
...main is already in Package example
example is already in Package example

/home/btiffin/unicon-git/bin/icont -c   -O package.icn /tmp/uni12420122
Translating:
package.icn:
  example__main
  example__example
No errors
```

```
/home/btiffin/unicon-git/bin/icont  package.u
Linking:
```

```
prompt$ unicon -c package.icn
Parsing package.icn: package.icn is already in Package example
...main is already in Package example
example is already in Package example

/home/btiffin/unicon-git/bin/icont -c   -O package.icn /tmp/uni51075390
Translating:
package.icn:
  example__main
  example__example
No errors
```

See also:

import

6.3.10 procedure

procedure `name([parameter,...])` defines a procedure. The body of the procedure continues until the *end* reserved word is encountered.

User defined procedures are semantically equivalent to built in functions; they always produce a value or fail. Results can be *suspend*ed, or *return*ed.

```
#
# procedure.icn, demonstrate a Unicon procedure definition
#

# procedure bodies continue up until the mandatory end reserved word
procedure main(arglist)
    write("All Unicon programs require a main procedure")
    write()
    write("This program was invoked by the operating system with:")
    every write(!arglist)

    # if the arglist is empty, subproc will not be called
    subproc(\arglist[1])
end

procedure subproc(arg)
    write("This procedure \"subproc\" was invoked with: ", arg)
end
```

Sample run:

```
prompt$ unicon -s procedure.icn -x Determination
All Unicon programs require a main procedure

This program was invoked by the operating system with:
Determination
This procedure "subproc" was invoked with: Determination
```

main

Unless compiled with `-c` or similar, *all* Unicon programs require a `procedure main()`. The program will be given a list of command line arguments, but that usage is optional.

```
procedure main() ... end

procedure main(arglist) ... end
```

Are both valid Unicon main procedures.

For any given program there can only be one `main`. The operating system is given a zero result, even if *return* is used at the end of main. Other platform dependent status codes can be set with the *exit* function. Error status will be set when the *stop* function is used or when an unconverted runtime error abnormal end occurs.

6.3.11 record

record `name(field,...)` defines a record constructor for `name` with one or more fields.

```
#
# record.icn, demonstrate record data
#

record sample(a, b, c)

procedure main()
    R := sample(1, 2, 3)
    write(R.a)
    write(R.b)
    write(R.c)
end
```

Giving:

```
1
2
3
```

Trivia: `record` can be used to form the shortest known complete Unicon program.

```
procedure main()
end
```

That is not the shortest valid program. This is:

```
record main()
```

The shortest known complete program.[5] It creates a `record` constructor that just happens to provide the `main` procedure, a required element of all valid Unicon programs. When executed the program evaluates the constructor expression, creating a record of type `main` and terminates normally.

---

[5] From the Icon Newsletter, Programming Corner, Issue 32 (question), and 33 (answer).

6.3.12 static

static `var [, var...]` declares local procedure or method variables to be persistent while still local. Values will be remembered between invocations during the entire program execution.

```
#
# static.icn, demonstrate static variable declarations
#
procedure main()
    localproc()
    staticproc()
    write()
    localproc()
    staticproc()
end

# a is reset on each invocation
procedure localproc()
    /a := "initial value"
    writes("enter localproc  with a as ")
    write(image(a))
    a := "new value"
end

# a is remembered across each invocation
procedure staticproc()
    static a
    /a := "initial value"
    writes("enter staticproc with a as ")
    write(image(a))
    a := "new value"
end
```

Giving:

```
prompt$ unicon -s static.icn -x
enter localproc  with a as "initial value"
enter staticproc with a as "initial value"

enter localproc  with a as "initial value"
enter staticproc with a as "new value"
```

6.4 Syntax reserved words

Unicon also includes a few reserved words that work together with some of the other *Unicon action words* and *Declarative expressions* words.

6.4.1 all

all is an optional clause for the *invocable* declarative. Used to allow string invocation and any user defined procedure or built-in function.

See *invocable* for more details.

6.4.2 by

by is an optional step value clause for the *to* expression.

See *to* for more details. The default `by` step is `1`.

6.4.3 default

default specifies a default case for the *case* `of` selection expression.

See *case* for more details.

6.4.4 do

do is an optional clause to specify the expression to be executed for each iteration of a loop construct.

See *every*, *suspend while* for more details. *And, yes,* `suspend` *counts as a looping construct.*

6.4.5 end

end is the reserved word to signify the end of a *class*, *method*, or *procedure* body.

6.4.6 else

else evaluates an alternative *expr*3 when an *if* *expr*1 fails to produce a result. An `else` clause is optional with an `if` statement.

`if` *expr*1 `then` *expr*2 `else` *expr*3

See: *if*, *then* for more details.

6.4.7 of

of is the reserved word that precedes a special compound expression consisting of a sequence of *case* branches.

`case` *expr* `of` ...

See: *case*, *default* for more details.

6.4.8 then

then expressions are executed when (and only when) an *if* *expr* produces a result. The `then` reserved word is not optional when constructing a valid `if` expression.

See: *if* and *else* for more details.

CHAPTER

SEVEN

FUNCTIONS

Unicon has a large repertoire of built-in functions.

A function is equivalent to a user defined *procedure*, in terms of syntax, argument management, and semantics.

Functions cover a wide range of usage; from initializing data structures (`list` and `table`, for instance) to adding utility to Unicon (`left` and `right` formatting), to supporting Execution Monitoring, and graphics, along with many other operations and system services that make Unicon what it is.

7.1 Unicon Functions

There are 310 functions built into Unicon (as of 2016-08-20), but no builds will include all of them, as some few are platform dependent. This list includes all core and all optional functions that can be part of Unicon.

Note: A lot of the reference material here is courtesy of Programming with Unicon, Second edition; by Clinton Jeffery, Shamim Mohamed, Jafar Al Gharaibeh, Ray Pereda, Robert Parlett.

That book ships with the Unicon sources in the `doc/book/` subdirectory.

7.1.1 Abort

Abort()

Type *pattern*

Versionadded Unicon 13.

A *SNOBOL* inspired pattern matching operation. Causes immediate match failure, no further alternatives are attempted.

```
#
# Abort.icn, demonstrate SNOBOL style pattern match abort
#
procedure main()
```

```
    write("match a or b, but abort (fail) if 1 is found first")
    pat := Any("ab") .| "1" || Abort()
    tests := ["ab1", "b1a", "1ab", "xzyab1", "xzy1ba"]

    every s := !tests do
        if s ?? pat then
            write(s || " matched") else write(s || " aborted")
end
```

Sample run:

```
match a or b, but abort (fail) if 1 is found first
ab1 matched
b1a matched
1ab aborted
xzyab1 matched
xzy1ba aborted
```

7.1.2 abs

abs (*n : number*) → number

Type number, *integer* or *real*

Parameters **n** – numeric value

Returns absolute value

The absolute value function. Produces the maximum of n and -n, $|n|$.

```
#
# abs.icn, demonstrate the absolute value function
#
procedure main()
    p := 1
    n := -1
    r := -1.23
    write(p, ", abs(", p, ") = ", abs(p))
    write(n, ", abs(", n, ") = ", abs(n))
    write(r, ", abs(", r, ") = ", abs(r))
end
```

Sample run:

```
1, abs(1) = 1
-1, abs(-1) = 1
-1.23, abs(-1.23) = 1.23
```

7.1.3 acos

acos (*r : real*) → real

Type real

Parameters **r** – real, $-1 \leq r \leq 1$

Returns arc cosine of r, $arccos(r)$

Produces the inverse cosine of r, result in radians.

"arc" is an abbreviation of "arcus", inverse circular.

```
#
# acos.icn, demonstrate the the arc cosine function
#
procedure main()
    write("Arc Cosine: Domain -1 <= x <= 1; result in radians")
    every r := -1.0 to 1.0 by 0.25 do {
        write(left("acos(" || trim(left(r, 6)) || ")", 12),
              " = ", acos(r))
    }
end
```

Sample run:

```
Arc Cosine: Domain -1 <= x <= 1; result in radians
acos(-1.0)   = 3.141592653589793
acos(-0.75)  = 2.418858405776378
acos(-0.5)   = 2.094395102393196
acos(-0.25)  = 1.823476581936975
acos(0.0)    = 1.570796326794897
acos(0.25)   = 1.318116071652818
acos(0.5)    = 1.047197551196598
acos(0.75)   = 0.7227342478134157
acos(1.0)    = 0.0
```

Todo

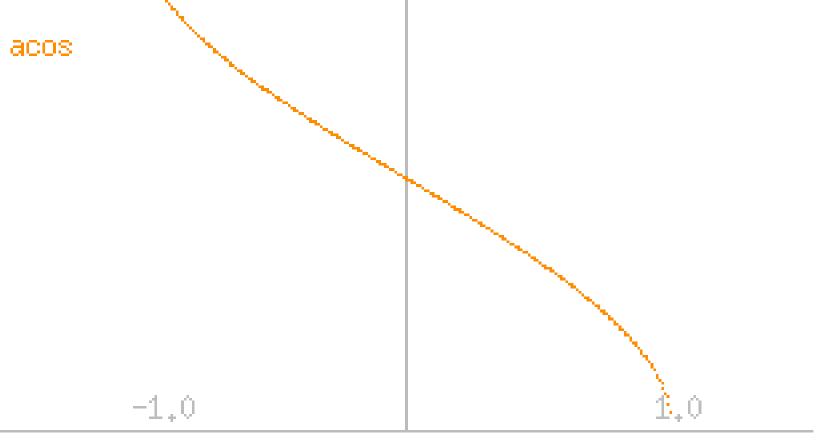
fix plot range and domain handling

Graphical plot:

```
#
# plot-function, trigonometric plotting, function from command line
#
$define points 300
$define xoff   154
$define base   164
$define yscale 60
$define xscale 100

invocable "asin", "acos", "atan"

# plot the given function, default to sine
procedure main(args)
    func := map(args[1]) | "acos"
    if not func == ("asin" | "acos" | "atan") then func := "acos"

    # range of pixels for 300 points, y scaled at +/- 60, 4 pixel margins
    &window := open("Plotting", "g", "size=308,168", "canvas=hidden")

    # tan may cause runtime errors
```

```
    if func == "atan" then &error := 6

    color := "vivid orange"
    Fg(color)
    write(&window, "\n " || func)

    Fg("gray")
    DrawLine(2, base, 306, base)
    DrawLine(xoff, 2, xoff, 164)
    #DrawString(8, 10, "1.5+", 8, 69, "0", 2, 126, "-1.5-")
    DrawString(xscale / 2 + 2, 160, "-1.0")
    DrawString(points - (xscale / 2) - 2, 160, "1.0")

    Fg(color)
    step := 2.0 / points
    every x := -1.0 to 1.0 by step do {
        DrawPoint(xoff + (x * xscale), base - (yscale * func(x)))
    }
    WSync()
    WriteImage("../images/plot-" || func || ".png")
    close(&window)
end
```

```
prompt$ unicon -s plot-arccos.icn -x acos
```



See also:

asin, *atan*, *sin*, *cos*, *tan*

7.1.4 Active

`Active()` : *window*

`Active()` produces a Window value that has one or more events pending. If there are no events pending, `Active()` will block and wait. `Active()` produces different windows on each call to help ensure windows are serviced and don't starve for attention.

```
#
# Active.icn, demonstrate active event testing
```

```
#
link enqueue, evmux
procedure main()
    window := open("Active sample", "g", "size=90,60", "canvas=hidden")
    Enqueue(window, &lpress, 11, 14, "", 2)
    Enqueue(window, &lrelease, 12, 15, "", 3)
    w := Active()
    write(image(w))
    e := Event(w, 1)
    write("event ", e, " at ", &x, " ", &y)
    e := Event(w, 1)
    write("event ", e, " at ", &x, " ", &y)
    close(window)
end
```

Sample run:

```
prompt$ unicon -s Active.icn -x
window_1:1(Active sample)
event -1 at 11 14
event -4 at 12 15
```

7.1.5 Alert

`Alert()` → window

Type *window*

Produces a visual flash or audible sound to signal notable events.

`Alert()` : *window*

`Alert()` produces a visual flash or audible sound to signal notable events.

```
#
# Alert.icn, visual/audible alert
#
link ximage
procedure main()

    # Alert needs to have a default window, &window needs to be non null
    write("Expect an error 140")
    &error := 1
    w := Alert()
    write("Error ", &errornumber, ": ", &errortext)

    # now Alert will have a resource to use
    &window := open("title", "g", "canvas=hidden")
    w := Alert()
    write(ximage(w))
    close(&window)
end
```

Sample run:

```
Expect an error 140
Error 140: window expected
window_1:1(title)
```

7.1.6 any

`any(c, s, i, i)` : *integer* or *fail*

`any()` is a string scanning function that produces the first index, $i1 + 1$, if $s[i1:i2][1]$ is in the *cset* c. It fails otherwise. *Match this character to any of the characters in the cset.*

s defaults to *&subject* and the indexes default to the character at *&pos*.

```
#
# any.icn, demonstrate string scanning any() function
#
procedure main()
    str := "ABCdef"
    str ? any(&lcase) & write("Match lower case true")
    str ? any(&ucase) & write("Match upper case true")
end
```

Sample run:

```
Match upper case true
```

7.1.7 Any

`Any(c)` : *pattern*

`Any(c)` is the SNOBOL pattern that *any* is based on. Produces a *Pattern* that matches the next subject character if it appears in the *cset*.

```
#
# Any.icn, demonstrate the SNOBOL based pattern Any() function
#
procedure main()
    str := "ABCdef"
    ups := Any(&ucase)
    write("Type of ups: ", type(ups))
    str ?? ups -> intermediate
    write(intermediate)
end
```

Sample run:

```
Type of ups: pattern
A
```

7.1.8 Arb

`Arb()` : *success*

`Arb()` is part of the SNOBOL inspired pattern feature set. Matches an arbitrary number of characters from the subject string. `Arb()` matches the shortest possible substring, which includes the empty string. Patterns on either side of `Arb()` determine what is actually matched.

```
#
# Arb.icn, demonstrate Arb() SNOBOL, always succeeds matching
#
procedure main()
    sub := "This is a test"
    pat := "This" || Arb() -> inter || "a test"
    write(sub ?? pat)
    write((sub[1:-2] ?? pat) | "no match")
    sub ?? pat; write(inter)
end
```

Sample run:

```
This is a test
no match
 is
```

7.1.9 Arbno

`Arbno(pat)` : *string*

`Arbno()` matches an arbitrary number of the pattern `pat`, including zero matches.

```
#
# Arbno.icn, demonstrate Arbno() SNOBOL pattern
#
procedure main()
    sub := "This is is is a test"
    pat := "This" || Arbno(" is") -> inter || " a test"
    sub ?? pat
    write(inter)
end
```

Sample run:

```
 is is is
```

7.1.10 args

`args(x, i)` : *any*

`args(p)` produces the number of arguments expected by procedure *p*. *This is sometimes referred to as the* `arity` *of a function*. For variable argument prototypes, `args(p)` will return a negative number, representing the final list argument position. E.g.

```
procedure sample(x, y, z[])
    write(args(sample))
end
```

In that case, `args(sample)` returns `-3`.

`args(C)` produces the number of arguments in the current instance of *co-expression* *C*. `args(C, i)` produces the *i*th argument within co-expression *C*.

```
#
# args.icn, Demonstrate the args arity function
#
procedure main()
    write("args for main: ", args(main))
    write("args for subproc: ", args(subproc))
    subproc()
    coex := create subproc()
    write("args for co-expression (subproc): ", args(coex))
    @coex
end

procedure subproc(a:string:"Hello", b:integer:5)
    write(a, ", ", b)
end
```

Sample run:

```
args for main: 0
args for subproc: 2
Hello, 5
args for co-expression (subproc): 2
Hello, 5
```

args as common main argument

Of special note with `args`. Historically, many Unicon programs have defined main as

```
procedure main(args)
```

That definition will hide the `args` built-in function. When maintaining such a program, a developer can either rename the list variable passed into `main` (and all references), or fallback to using *proc* to retrieve the built-in function.

```
procedure main(args)
    write("arity of main: ", proc("args", 0)(main))
end
```

7.1.11 array

> **Attention:** multi-dimensional array referencing is a pending feature

`array()` : *array*

`array()` produces an efficient *list* of homogeneous numeric data (*integer* or *real*).

```
#
# array.icn, Demonstrate a single dimensional numeric array allocation
#
link ximage
procedure main()
    # allocate an array of integers, default 42
    a := array(6, 42)
    write(type(a), " is ", ximage(a))
    write("a[5] = ", a[5])
    every i := 1 to 10 do a[i] := i
    write("a[5] = ", a[5])

    r := array(6, 42.0)
    write(type(r), " is ", ximage(r))
    write("r[5] = ", r[5])
    every i := 1 to 10 do r[i] := real(i)
    write("r[5] = ", r[5])
end
```

Sample run:

```
list is L1 := list(6,42)
a[5] = 42
a[5] = 5
list is L5 := list(6,42.0)
r[5] = 42.0
r[5] = 5.0
```

7.1.12 asin

`asin(r)` : *real*

`asin(r)` produces the arc sine of the angle *r*, given in radians as a *real*.

```
#
# asin.icn, demonstrate the asine function
#
procedure main()
    write("asin(r): -pi/2 to pi/2, Range: -1.0 <= x <= 1.0")
    every r := -1.0 to 1.0 by 0.25 do {
        write(left("asin(" || r || ")", 24), " = ", asin(r), " radians")
    }
end
```

Sample run:

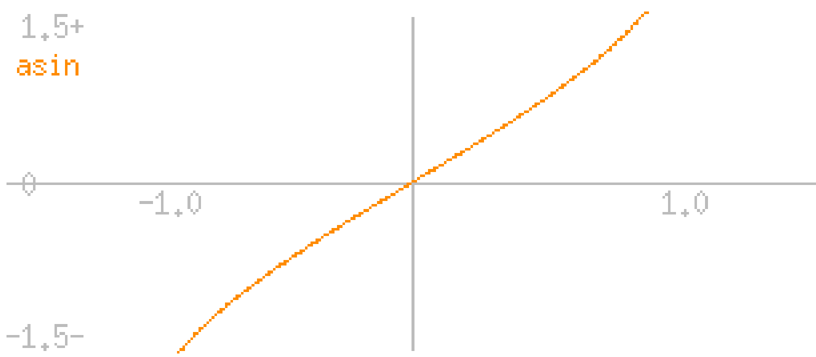
```
asin(r): -pi/2 to pi/2, Range: -1.0 <= x <= 1.0
asin(-1.0)              = -1.570796326794897 radians
asin(-0.75)             = -0.848062078981481 radians
asin(-0.5)              = -0.5235987755982989 radians
asin(-0.25)             = -0.2526802551420786 radians
asin(0.0)               = 0.0 radians
```

```
asin(0.25)              = 0.2526802551420786 radians
asin(0.5)               = 0.5235987755982989 radians
asin(0.75)              = 0.848062078981481 radians
asin(1.0)               = 1.570796326794897 radians
```

Graphical plot:

```
#
# plot-function, trigonometric plotting, function from command line
#
$define points 300
$define xoff   154
$define base   64
$define yscale 60
$define xscale 100

invocable "asin", "acos", "atan"

# plot the given function, default to sine
procedure main(args)
    func := map(args[1]) | "asin"
    if not func == ("asin" | "acos" | "atan") then func := "asin"

    # range of pixels for 300 points, y scaled at +/- 60, 4 pixel margins
    &window := open("Plotting", "g", "size=308,128", "canvas=hidden")

    # tan may cause runtime errors
    if func == "atan" then &error := 6

    color := "vivid orange"
    Fg(color)
    write(&window, "\n " || func)

    Fg("gray")
    DrawLine(2, base, 306, base)
    DrawLine(xoff, 2, xoff, 126)
    DrawString(8, 10, "1.5+", 8, 69, "0", 2, 126, "-1.5-")
    DrawString(xscale / 2 + 2, 76, "-1.0")
    DrawString(points - (xscale / 2) - 2, 76, "1.0")

    Fg(color)
    step := 2.0 / points
    every x := -1.0 to 1.0 by step do {
        DrawPoint(xoff + (x * xscale), base - (yscale * func(x)))
    }
    WSync()
    WriteImage("../images/plot-" || func || ".png")
    close(&window)
end
```

```
prompt$ unicon -s plot-arcfunction.icn -x asin
```



See also:

acos, *atan*, *sin*, *cos*, *tan*

7.1.13 atan

`atan(r, r:1.0)` : *real*

`atan(r)` produces the arc tangent of *r*. `atan(r1, r2)` produces the arc tangent of *r1* and *r2*. Arguments and given in radians.

```
#
# atan.icn, demonstrate the atan function
#
procedure main()
    write("atan(r): -pi/2 to pi/2, Range: -1.0 <= x <= 1.0")
    every r := -1.0 to 1.0 by 0.25 do {
        write(left("atan(" || r || ")", 24), " = ", atan(r), " radians")
    }
end
```

Sample run:

```
prompt$ unicon -s atan.icn -x
atan(r): -pi/2 to pi/2, Range: -1.0 <= x <= 1.0
atan(-1.0)              = -0.7853981633974483 radians
atan(-0.75)             = -0.6435011087932844 radians
atan(-0.5)              = -0.4636476090008061 radians
atan(-0.25)             = -0.2449786631268641 radians
atan(0.0)               = 0.0 radians
atan(0.25)              = 0.2449786631268641 radians
atan(0.5)               = 0.4636476090008061 radians
atan(0.75)              = 0.6435011087932844 radians
atan(1.0)               = 0.7853981633974483 radians
```

No image available

7.1.14 atanh

`atanh(r)` : *real*

`atanh(r)` produces the inverse hyperbolic tangent of *r*. Argument given in radians.

```
#
# atanh.icn, demonstrate the atan function
#
procedure main()
    write("atanh(r): -pi/2 to pi/2, Range: -1.0 <= x <= 1.0")
    every r := -1.0 to 1.0 by 0.25 do {
        write(left("atanh(" || r || ")", 24), " = ", atanh(r), " radians")
    }
end
```

Sample run:

```
prompt$ unicon -s atanh.icn -x
atanh(r): -pi/2 to pi/2, Range: -1.0 <= x <= 1.0
atanh(-1.0)              = -inf.0 radians
atanh(-0.75)             = -0.9729550745276566 radians
atanh(-0.5)              = -0.5493061443340548 radians
atanh(-0.25)             = -0.2554128118829954 radians
atanh(0.0)               = 0.0 radians
atanh(0.25)              = 0.2554128118829954 radians
atanh(0.5)               = 0.5493061443340548 radians
atanh(0.75)              = 0.9729550745276566 radians
atanh(1.0)               = inf.0 radians
```

Todo

plot image of atanh

7.1.15 Attrib

`Attrib(T, i, x,...)` : *any* [*Concurrency*]

`Attrib()` read/write thread attributes for thread handle `T`.

```
#
# Attrib.icn, Demonstrate thread attributes
#
# requires Concurrency build of Unicon
#
$include "threadh.icn"
import threads

global pT, cT

procedure main()
    # spin up the producer, then await completion
    pT := thread producer()
    wait(pT)
```

```
    # spin up the consumer, then await completion
    cT := thread consumer(pT)
    wait(cT)

    write("main complete")
end

procedure producer()
    write("pT OUTBOX_SIZE: ", Attrib(pT, OUTBOX_SIZE))
    Attrib(pT, OUTBOX_LIMIT, 3)

    # send 10 numbers, but limit is 3, non blocking
    write("producer sending")
    every !10@>
    write("pT OUTBOX_SIZE: ", Attrib(pT, OUTBOX_SIZE))
end

procedure consumer(T)
    write("consumer receiving from producer public outbox")
    while write(<@T)
end
```

Sample run:

```
pT OUTBOX_SIZE: 0
producer sending
pT OUTBOX_SIZE: 3
consumer receiving from producer public outbox
1
2
3
main complete
```

7.1.16 Bal

`Bal()` : *type* [*Patterns*]

`Bal()` SNOBOL style balanced parentheses pattern match

```
#
# Bal.icn, demonstrate the SNOBOL style balanced parentheses pattern
#
procedure main()
    pat := Len(2) || Bal() -> b || Rem() -> r
    write(image(pat))

    tests := ["a=(b*c)+d", "a=b*c+d", "a=()+d"]
    every subject := !tests do {
        write("Matching     : ", subject)
        subject ?? pat
        write("balanced part: ", b)
        write("remainder    : ", r)
    }
end
```

Sample run:

```
pattern_6(7) = Len(2) || (Bal()) -> b || (Rem()) -> r
Matching     : a=(b*c)+d
balanced part: (b*c)
remainder    : +d
Matching     : a=b*c+d
balanced part: b
remainder    : *c+d
Matching     : a=()+d
balanced part: ()
remainder    : +d
```

7.1.17 bal

`bal(c1:&cset, c2:')', c3:')', s, i1, i2)` : *integer**

`bal()` is a string scanning function that generates the integer positions in `s` where a member of `c1` in `s[i1:i2]` is balanced with respect to `c2` and `c3`.

```
#
# bal.icn, demonstrate the string scanning bal() function
#
procedure main()
    L := list()
    s := "(1 + 2) * ((3 + 4) * 5) - 6"
    carets := repl(" ", *s)

    # simple
    write("scanning: ", image(s), " with bal()")
    s ? every write(bal())

    # nicely formatted for visual effect
    s ? every carets[bal()] := "^"
    write()
    write(s)
    write(carets)
end
```

Sample run:

```
scanning: "(1 + 2) * ((3 + 4) * 5) - 6" with bal()
1
8
9
10
11
24
25
26
27

(1 + 2) * ((3 + 4) * 5) - 6
^      ^^^^             ^^^^
```

7.1.18 Bg

`Bg(w, s)` : *string*

`Bg(w)` retrieves the background colour of the window, `w`. `Bg(w, s)` attempts to set the background colour by name, rgb or *mutable colour* value. Fails if the background cannot be set to the specified colour.

```
#
# Bg.icn, demonstrate background colour settings
#
procedure main()
    window := open("Bg example", "g", "size=110,75", "canvas=hidden")
    write(window, Bg(window))
    Bg(window, "vivid orange")
    write(window, Bg(window))
    Bg(window, "10000,20000,30000")
    write(window, Bg(window))
    Bg(window, "transparent green")
    write(window, Bg(window))
    if c := NewColor(window, "40000,50000,60000") then {
        Bg(window, c)
        write(window, Bg(window))
    }
    else {
        Bg(window, "white")
        write(window, "no mutable colours")
    }
    WSync(window)
    WriteImage(window, "../images/Bg.png")
    FreeColor(\c)
    close(window)
end
```

Sample run:

```
prompt$ unicon -s Bg.icn -x
```

white
vivid orange
10000,20000,30000
transparent green
no mutable colours

7.1.19 Break

`Break(c)` : *string*? [*Patterns*]

`Break(c)` matches any characters in the subject string up to but not including any of the characters in the *cset* `c`.

```
#
# Break.icn, demonstrate the Break pattern match
#
procedure main()
    s := "integers"
```

```
    s ?? Break('e') -> found
    write("Matched: ", found)
end
```

Sample run:

```
Matched: int
```

See *Breakx*.

7.1.20 Breakx

`Breakx(c)` : *string*?

`Breakx(c)` is an extended *Break*. It will match any characters in the subject string up to any of the subject characters in the *Cset*, `c`. `Breakx` will search beyond the break position for any possible longer match if resumed due to subsequent pattern failure. This means that `Breakx` might return characters in `c`, unlike the *Break* that will not.

```
#
# Breakx.icn, demonstrate the extended Break pattern match
#
procedure main()
    # this Break match stops at the first "e"
    s := "integers"
    s ?? Break("e") -> found || "er"
    write("Matched: ", found)

    # this Breakx match tries twice, a second "e" is followed by "er"
    s ?? Breakx("e") -> found || "er"
    write("Matched: ", found)
end
```

Sample run:

```
Matched: g
Matched: integ
```

See *Break*.

7.1.21 callout

`callout()` : *type*

Todo

entry for function callout

`callout()`

Sample run:

7.1.22 center

`center(s, i:1, s2:" ")` : *string*

`center()` produces a centred string, `s` within width `i`, padded on either side by `s2`. If `i` is less than the size of `s`, the centre `i` characters of `s` are produced.

```
#
# center.icn, Demonstrate string centring.
#
procedure main()
    s := " this is a test "
    write(center(s, 64, "="))
    write(center("center(s,8) of " || image(s) || " is " ||
          image(center(s, 8)), 64))
end
```

Sample run:

```
======================== this is a test ========================
        center(s,8) of " this is a test " is "s is a t"
```

7.1.23 char

`char(i)` : *string*

`char()` produces a string consisting of the character encoded by the integer `i`. Unicon is *ASCII* based. `char(0)` is the null byte, the first ASCII character, and anything outside of the range 0-255 will cause a runtime error.

```
#
# char.icn, demonstrate the char() function.
#
procedure main()
    write(image(char(0)))
    write(image(char(1)))
    write(char(65))
    # run-time error, out of range
    write(image(char(257)))
end
```

This sample run fails with a runtime error when asking for `char(257)`, an out of range integer.

```
"\x00"
"\x01"
A

Run-time error 205
File char.icn; Line 16
invalid value
offending value: 257
Traceback:
   main()
   char(257) from line 16 in char.icn
```

7.1.24 chdir

`chdir(s)` : *string*

`chdir(s)` changes the current working directory to `s`. `s` will be operating system dependent. `chdir()` produces the current working directory as a *string*

```
#
# chdir.icn, change and return current working directory
#
procedure main()
    write(chdir())
    chdir("..")
    write(chdir())
end
```

Sample run:

```
/home/btiffin/wip/writing/unicon/examples
/home/btiffin/wip/writing/unicon
```

7.1.25 chmod

`chmod(f, m)` : *?*

`chmod()` changes a file access mode. `f` can be a string name or open file handle. Mode `m` is encoded with +/- (permit/deny)

- r, read
- w, write
- x, execute

Who can access is also encoded by

- u, user, (owner) of the file
- g, group membership of the file
- o, other
- a, all

Mode bits also include

- s, setuid or setgid for allowing the process to assume a uid/gid when executing the resource
- t, sticky bit. Asks the kernel to retain the process image in memory when executing the resource terminates. Sticky bits on the directory prevents renaming, moving or deleting the files in the directory if not the owning user.

POSIX also allows a numeric, octal value for mode settings. rwx as bit 2, 1, 0 of an octal value. 8r7 being 2r111, rwx, 8r5 being 2r101, rx, no w, and 8r0 is no permission in that grouping. Values are by owner, group and other, 3 octal digits. 8r707 is rwx for user, no group access, rwx for other. *A poor example; not sharing with friends, but allowing outsiders.*

```
#
# chmod.icn, Demonstrate chmod, the file access permission function
#
procedure main()
    # set "other" read permissions
    # others outside owner (u) and group (g)
    file := "chmod.icn"
    write("chmod call: ", image(chmod(file, "o+r")))

    write("mode of ", file, " ", stat(file).mode)
end
```

Sample run:

```
chmod call: &null
mode of chmod.icn -rw-rw-r--
```

7.1.26 chown

`chown(f, u, g)` : *null*

`chown()` change or retrieve (if permitted) the owner/group of the filename `f`. `u` and `g` can be strings or numeric identifiers as known by the operating system.

```
#
# chown.icn, Demonstrate chown, the file ownership function
#
procedure main()
    file := "chown.icn"
    # Change owner/group of file, usually only superusers can do this
    # so expect failure of this sample, file will remain owned by btiffin
    write(image(chown(file, "nobody", "nobody")))
    write("Owner of ", file, " ", stat(file).uid)
end
```

Sample run, very likely to fail without permissions.

```
Owner of chown.icn btiffin
```

7.1.27 chroot

`chroot(s)` : *null*

`chroot(f)` changes the root directory, `/`, of the current filesystem to `f`.

`chroot` was an early Unix attempt at protecting filesystems. Still in use by some applications, web servers for instance, to ensure that files outside of a permissible tree are inaccessible by normal users. Modern equivalents BSD jails and new LCX/LXD Linux container technology have superseded use of `chroot`. This is normally a privileged operation.

```
#
# chroot.icn, Demonstrate chroot, the filesystem root directory function
#
# Seeing as this program is evaluated during document generation it is
# very likely to fail.  The book is never built using root permissions.
#
procedure main()
    # Change filesystem root to protect outside directories
    #  from unprivileged access.  Requires CAP_SYS_CHROOT capabilities.
    if newroot := chroot("/home/btiffin") then {
        write(image(newroot))
        chdir("/")
        write(chdir())
        d := open(".", "r")
        while write(read(d))
        close(d)
    } else write("no permission to set new root directory")
end
```

Sample run, very likely to fail without permissions.

```
no permission to set new root directory
```

7.1.28 classname

`classname(r)` : *string*

`classname(r)` will produce the name of the *class* of `r`.

```
#
# classname.icn, demonstrate the classname function
#
class sample()
    method inner()
        write("in method inner of class ", classname(self))
    end
end

procedure main()
    r := sample()
    r.inner()
    write("r is of class ", classname(r))
end
```

This sample run includes the linkage details to give an example of how Unicon classes are managed.

```
prompt$ unicon classname.icn -x
Parsing classname.icn: ..
/home/btiffin/unicon-git/bin/icont -c   -O classname.icn /tmp/uni12296156
Translating:
classname.icn:
  sample_inner
  sample
  sampleinitialize
  main
```

```
No errors
/home/btiffin/unicon-git/bin/icont  classname.u -x
Linking:
Executing:
in method inner of class sample
r is of class sample
```

7.1.29 Clip

`Clip(w:&window, x:0, y:0, wid:0, h:0)` : *window* [*graphics*]

`Clip()` sets a clipping rectangle for a window. Top left corner starts at `x`, `y`. If width `wid` is 0, the clipping region extends from `x` to the right side of the window. When height `h` is 0, the clip region extends from `y` to the bottom of the window. Graphic writes outside of the clipping region will not occur.

```
#
# Clip.icn, demonstrate clipping region
#
procedure main()
    window := open("Clip example", "g", "size=110,75", "canvas=hidden")
    write(window, "Clipping")
    Clip(window, 0, 0, 50, 50)
    Fg(window, "vivid orange")
    FillCircle(window, 50, 50, 30)
    WSync(window)
    WriteImage(window, "../images/Clip.png")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s Clip.icn -x
```

Clipping

7.1.30 Clone

`Clone(w1, attributes...)` : *window* [*graphics*]

`Clone()` creates a window that couples the drawing canvas of `w1` with a new graphics context. Attributes from `w1` are inherited, overridden by any attributes specified in the `Clone()` function.

```
#
# Clone.icn, demonstrate window cloning, canvas coupling
#
procedure main()
```

```
    window := open("Clone one", "g", "size=110,60", "canvas=hidden")
    write(window, "Cloning")
    write(window)

    other := Clone(window, "fg=vivid orange", "font=sans")
    write(other, "From the clone")

    WSync(window)
    WriteImage(window, "../images/Clone.png")
    close(other)
    close(window)
end
```

Sample run:

```
prompt$ unicon -s Clone.icn -x
```

Cloning

From the clone

7.1.31 close

`close(x)` : *file | integer*

`close()` a file, pipe, window, network, message or database connection. Resources are freed. Closing a window will cause it to disappear, but will remain active until all open bindings are closed. If a pipe or network connection is closed, an integer exit status is returned, otherwise the value produced is the closed file handle.

```
#
# close.icn, demonstrate the close function
#
procedure main()
    # open the source file resource
    f := open(&file, "r")
    # show the first meaningful comment line
    read(f)
    write(read(f))

    # close the resource, display the expression value
    write(image(close(f)))

    &error := 1
    # next line will cause a runtime error, converted to failure
    write(read(f))
    write("Error: ", &errornumber, " - ", &errortext)

    # open a pipe and check exit status
    write()
    write("open an 'ls' pipe, show first few entries, skip the rest")
    p := open("ls", "p")

    # show first few entries
```

```
    every 1 to 6 do write(read(p))
    # spin past the rest
    while read(p)

    # display the close expression value
    write("Exit status from ls pipe: ", image(close(p)))
end
```

Sample run:

```
# Author: Brian Tiffin
file(close.icn)
Error: 212 - attempt to read file not open for reading

open an 'ls' pipe, show first few entries, skip the rest
1to4
1to4.icn
Abort
Abort.icn
abs
abs.icn
Exit status from ls pipe: 0
```

7.1.32 cofail

`cofail(CE)` : *any*

`cofail(ce)` activates *co-expression* *ce*, transmitting failure instead of a result.

```
#
# cofail.icn, demonstrate cofail, transmit failure to a co-expression
#
procedure main()
    ce := create 1 to 4

    write(@ce)
    write(@ce)

    cofail(ce)
    # this will cause a runtime error now, the co-expression empty
    write(@ce)
end
```

Sample run (ends with an error):

```
1
2
System error at line 12 in cofail.icn
empty activator stack
```

7.1.33 collect

`collect(i1:0, i2:0)` : *null*

`collect()` runs the garbage collector to ensure that `i2` bytes are available in region `i1`, where `i1` can be:

- 0, no region in particular
- 1, static region
- 2, string region
- 3, block region

```
#
# show collections, create and remove a string, reshow collections
#
procedure main()
    collections()
    s := repl(&letters, 1000)
    s := &null
    collect()
    write("\nAfter string create/remove")
    collections()
end

# Display current memory region allocations
procedure collections()
    local collects

    collects := [] ; every put(collects, &collections)

    write("Collections, ", *collects, " values generated")
    write(repl("-", 30 + **collects))
    write("Heap   : ", collects[1])
    write("Static : ", collects[2])
    write("String : ", collects[3])
    write("Block  : ", collects[4])
end
```

Sample run:

```
Collections, 4 values generated
-------------------------------
Heap   : 0
Static : 0
String : 0
Block  : 0

After string create/remove
Collections, 4 values generated
-------------------------------
Heap   : 1
Static : 0
String : 0
Block  : 0
```

7.1.34 Color

`Color(w, i, s,...)` : *window* [*graphics*]

`Color(w, i)` produces the current setting of mutable colour `i`. `Color(w, i, s,...)` set the colour map entries referenced by `i[j]` to the colours specified by `s[j]`..

```
#
# Color.icn, demonstrate colour references and settings
#
procedure main()
    window := open("Color example", "g", "size=200,110", "canvas=hidden")
    write(window, Color(window, 1)) | write(window, "no mutable colours")
    Color(window, 1, "vivid orange")
    write(window, Color(window, 1)) | {
        write(window, "still no colour map")
        write(window)
        write(window, "but don't despair")
        write(window, "colour maps have been superseded")
        write(window, "by display hardware that has")
        write(window, "all the colours, all the time")
    }
    WSync(window)
    WriteImage(window, "../images/Color.png")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s Color.icn -x
```

```
no mutable colours
still no colour map

but don't despair
colour maps have been superseded
by display hardware that has
all the colours, all the time
```

7.1.35 ColorValue

`ColorValue(w, s)` : *string* [*graphics*]

`ColorValue(w, s)` converts the string colour name `s` into a string with three 16-bit integer values representing the RGB components. `ColorValue()` fails if the string `s` is not a recognized name, or valid RGB decimal or hex encoded colour.

```
#
# ColorValue.icn, demonstrate colour name to value
#
procedure main()
    window := open("ColorValue example", "g", "size=240,80", "canvas=hidden")
    write(window, right("vivid orange is ", 20),
```

```
        ColorValue(window, "vivid orange")) |
        write(window, "invalid colour name")
    write(window, right("1,2,3 is ", 20),
        ColorValue(window, "1,2,3")) |
        write(window, "invalid colour encoding")
    write(window, right("#aaaabbbbcccc is ", 20),
        ColorValue(window, "#aaaabbbbcccc")) |
        write(window, "invalid colour hex encoding")
    writes(window, right("made up colour is ", 20))
    write(window, ColorValue(window, "made up colour")) |
        write(window, "invalid colour name")
    WSync(window)
    WriteImage(window, "../images/ColorValue.png")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s ColorValue.icn -x
```

```
    vivid orange is 65535,16383,0
           1,2,3 is 1,2,3
   #aaaabbbbcccc is 43690,48059,52428
  made up colour is invalid colour name
```

7.1.36 condvar

`condvar(mutex)` : *condition variable*

`condvar()` creates a new condition variable for use with *wait* or *signal*. The optional `mutex` argument associated the condition variable with the mutually exclusive lock.

The returned variable can be used with `wait(cv)` to block the current thread until a `signal(cv)` is invoked (from another thread).

```
#
# condvar.icn, demonstrate a threading condition variable
#

# taken from the Programming with Unicon book, page 146
procedure main()
    mtx := mutex()
    L := mutex([], mtx)
    cv := condvar(mtx)
    p := thread produce(L, cv)
    c := thread consume(L, cv)
    every wait(p | c)
end

procedure produce(L, cv)
    every put(L, !10) & *L=1 & signal(cv)
end
```

```
procedure consume(L, cv)
    i := 0
    while i < 10 do
        if x := get(L) then
             i +:= 1 & write(x)
        else
            critical cv: until *L>0 do wait(cv)
end
```

Sample run:

```
1
2
3
4
5
6
7
8
9
10
```

7.1.37 constructor

`constructor(s, ...)` : *procedure*

`constructor(label, field, field, ...)` creates a new *record* type, named `label` with fields named from the subsequent arguments (as strings). A constructor procedure is returned for creating records of this type.

```
#
# constructor.icn, Create a constructor procedure for a new record type
#
link ximage

record one(a,b,c)

procedure main()
    # start with a pre-compiled record
    r1 := one(1,2,3)
    write("r1.a ", r1.a)

    # add a new record type at runtime
    rc := constructor("newrec", "d", "e", "f")
    r2 := rc(4,5,6)
    write("r2.d ", r2.d)
    write(ximage(r2))
end
```

Sample run:

```
r1.a 1
r2.d 4
R_newrec_1 := newrec()
   R_newrec_1.d := 4
```

```
   R_newrec_1.e := 5
   R_newrec_1.f := 6
```

7.1.38 copy

`copy(any)` : *any*

`copy(x)` produces a copy of `x`. For immutable types, this is a no-op. For structures, a one-level deep copy of the object is made.

The *IPL* contains a `deepcopy` procedure when a nested structure needs to be copied.

```
#
# copy.icn, Demonstrate the one-level copy function
#
link ximage

record newrec(a,b,c)
procedure main()
    i := copy(5)
    write("copy(5) is ", i)
    L := [[1,2], 3, 4]
    L2 := copy(L)
    write(ximage(L2))

    write()
    write("Only one level copied:")
    R := newrec(1, 2, L)
    R2 := copy(R)
    write(ximage(R))
    write()
    write("The inner list is a reference")
    write("Changing the original, changes the copy")
    L[1] := 5
    write(ximage(R))
end
```

Sample run:

```
copy(5) is 5
L3 := list(3)
   L3[1] := L1 := list(2)
      L1[1] := 1
      L1[2] := 2
   L3[2] := 3
   L3[3] := 4

Only one level copied:
R_newrec_1 := newrec()
   R_newrec_1.a := 1
   R_newrec_1.b := 2
   R_newrec_1.c := L2 := list(3)
      L2[1] := L1 := list(2)
         L1[1] := 1
         L1[2] := 2
```

```
      L2[2] := 3
      L2[3] := 4

The inner list is a reference
Changing the original, changes the copy
R_newrec_1 := newrec()
   R_newrec_1.a := 1
   R_newrec_1.b := 2
   R_newrec_1.c := L2 := list(3)
      L2[1] := 5
      L2[2] := 3
      L2[3] := 4
```

7.1.39 CopyArea

`CopyArea(w1, w2, x:0, y:0, wid:0, h:0, x2:0, y2:0)` : *window* [*graphics*]

`CopyArea()` copies the rectangle `x,y,wid,h` from `w1` to `x2,y2` of `w2`.

```
#
# CopyArea.icn, demonstrate graphic area copy
#
procedure main()
    window := open("CopyArea example", "g", "size=150,125", "canvas=hidden")
    write(window, "CopyArea")
    Fg(window, "vivid orange")
    FillCircle(window, 50, 50, 30)
    CopyArea(window, window, 20,20,50,50, 60,60)
    WSync(window)
    WriteImage(window, "../images/CopyArea.png")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s CopyArea.icn -x
```

7.1.40 cos

`cos(r)` : *real*

`cos(r)` returns the Cosine of the given angle, *r* (in radians)

```
#
# cos.icn, demonstrate the the Cosine function
#
procedure main()
    write("cos(r): Domain all real repeating within 0 <= r <= 2pi (in radians),␣
↪Range: -1 <= x <= 1")
    every r := 0.0 to &pi * 2 by &pi/4 do {
        write(left("cos(" || r || ")", 24), " radians = ", cos(r))
    }
end
```

Sample run:

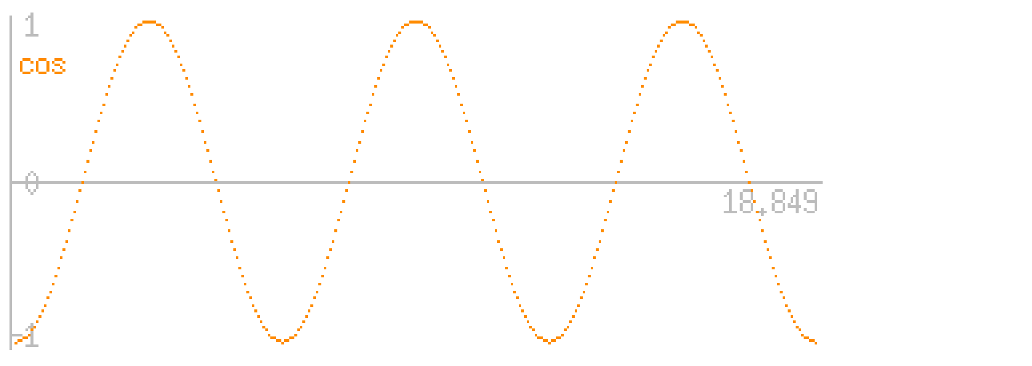
```
cos(r): Domain all real repeating within 0 <= r <= 2pi (in radians), Range: -1 <= x
↪<= 1
cos(0.0)                 radians = 1.0
cos(0.7853981633974483)  radians = 0.7071067811865476
cos(1.570796326794897)   radians = 6.123233995736766e-17
cos(2.356194490192345)   radians = -0.7071067811865475
cos(3.141592653589793)   radians = -1.0
cos(3.926990816987241)   radians = -0.7071067811865477
cos(4.71238898038469)    radians = -1.83697019872103e-16
cos(5.497787143782138)   radians = 0.7071067811865474
cos(6.283185307179586)   radians = 1.0
```

Graphical plot:

```
#
# plot-function, trigonometric plotting, function from command line
#
$define points 300
$define xoff   4
$define base   64
$define yscale 60
$define xscale 100

invocable "sin", "cos", "tan"

# plot the given function, default to sine
procedure main(args)
    func := map(args[1]) | "sin"
    if not func == ("sin" | "cos" | "tan") then func := "sin"

    # range of pixels for 300 points, y scaled at +/- 60, 4 pixel margins
    &window := open("Plotting", "g", "size=308,128", "canvas=hidden")

    # tan may cause runtime errors
    if func == "tan" then &error := 6

    color := "vivid orange"
    Fg(color)
    write(&window, "\n " || func)

    Fg("gray")
    DrawLine(2, 64, 306, 64)
    DrawLine(2, 2, 2, 126)
    DrawString(8, 10, "1", 8, 69, "0", 2, 126, "-1")
```

```
    DrawString(270, 76, left(points * 2 * &pi / 100, 6))

    Fg(color)
    every x := 0 to points do
        DrawPoint(xoff + x, base + yscale * func((2 * &pi * x) / xscale))

    WSync()
    WriteImage("../images/plot-" || func || ".png")
    close(&window)
end
```

```
prompt$ unicon -s plot-function.icn -x cos
```



7.1.41 Couple

`Couple(w1, w2)` : *window* [*graphics*]

`Couple(w1, w2)` produces a new value that binds the window associated with *w1* to the graphics context of *w2*.

```
#
# Couple.icn, demonstrate w1 coupling
#
procedure main()
    w1 := open("window 1", "g", "size=110,60", "canvas=hidden")
    write(w1, "Couple 1")
    write(w1)

    w2 := open("window 2", "g", "size=110,60", "canvas=hidden")
    Fg(w2, "vivid orange")
    write(w2, "Couple 2")

    other := Couple(w1, w2)
    write(other, "From the Couple")

    # save image of w1
    WSync(other)
    WriteImage(other, "../images/Couple.png")
    close(other, w2, w1)
end
```

Sample run:

```
Couple 1

From the Couple
```

7.1.42 crypt

`crypt(s1, s2)` : *string* [*POSIX*]

`crypt(s1, s2)` encrypts the password `s1` using the salt `s2`. The first two characters of the result string will be the salt.

```
#
# crypt.icn, Demonstrate the POSIX crypt function
#
procedure main()
    password := "#secretpassword!"
    salt := "SV"
    write(image(p := crypt(password, salt)))

    # input routine would get user password
    # compare both encrypted values
    attempt := crypt(password, salt)
    if attempt == p then
        write("User verified by password")
end
```

Sample run:

```
"SVI6LCAF1jj.6"
User verified by password
```

A word of warning. Modern computers are fast. Short passwords are very susceptible to brute force trials, especially when a bad actor gets hold of a copy of the encrypted form. It makes it easy to run through millions of attempts, comparing guesses that are `crypt`ed to the encrypted form.

Do yourself a favour and use strong passwords. Every letter and symbol makes it 96 times harder[1] to guess (assuming printable ASCII character codes are used). One character, 96 tries maximum, two characters, 96 * 96 tries. When you get to eight, the numbers start to be reasonable for thwarting casual bad actors. 7,213,895,789,838,336 potential combinations.

Modern machines can attempt millions of brute force guesses per second.

---

[1] That's over simplifying the issue. There are highly sophisticated algorithms in play now, with many common human behavioural aspects programmed in. And machines are fast.

7.1.43 cset

`cset(any)` : *cset*?

`cset(x)` attempts to convert `x` to a *cset*. Fails if the conversion can not be performed.

```
#
# cset-function.icn, Demonstrate the convert to cset function
#
record makefail(a,b,c)
procedure main()

   # convert a string
   write("convert \"xyzzy\" to cset")
   write(cset("xyzzy"))

   # convert a number
   write("convert 3.141 to cset")
   write(cset(3.141))

   # attempt to convert a record
   R := makefail(1,2,3.0)
   if not write(cset(R)) then write("record cannot be converted")
end
```

Sample run:

```
convert "xyzzy" to cset
xyz
convert 3.141 to cset
.134
record cannot be converted
```

7.1.44 ctime

`ctime(i)` : *string*

`ctime(i)` converts the integer time `i`, given in seconds since the epoch of Jan 1st, 1970 00:00:00 Greenwich Mean Time, into a string using the local timezone.

```
#
# ctime.icn, Demonstrate the ctime function
#
procedure main()
    # convert epoch start to a formatted time
    write(ctime(0))

    # convert time of run to formatted time
    write(ctime(&now))

    # two days in the future (relative to time of run)
    write(ctime(&now + 48 * 60 * 60))
end
```

Sample run:

```
Wed Dec 31 19:00:00 1969
Sun Oct 27 04:52:44 2019
Tue Oct 29 04:52:44 2019
```

See also:

&clock, *&dateline*, *gtime*, *&now*

7.1.45 dbcolumns

`dbcolumns(D, s)` : *list*

`dbcolumns(db, tablename)` returns a *List (arrays)* of *record* data.

```
#
# dbcolumns.icn, ODBC table column information
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # display ODBC view of table schema
    write("\ndbtables:")
    every write(ximage(dbtables(db)))

    # show dbcolumns information from first table
    tables := dbtables(db)
    write("\ndbcolumns.", tables[1].name, ":")
    every write(ximage(dbcolumns(db, tables[1].name)))

    # access some of the reflective record data
    write()
    write("First column")
    dbc := dbcolumns(db, tables[1].name)
    write("tablename: ", dbc[1].tablename,
          ", colname: ", dbc[1].colname,
          ", type: ", dbc[1].typename)

    write("Second column")
    write("tablename: ", dbc[2].tablename,
          ", colname: ", dbc[2].colname)

    # generate a query
    write()
    write("Query for ", dbc[2].colname, " (using dbc[2].colname)")
    sql(db, "select " || dbc[2].colname || " from " || dbc[2].tablename)

    # I know my name is first, but Jafar deserves the press
```

```
    rec := fetch(db)
    rec := fetch(db)

    # skipping generic info now, the column is known and it is 'name'
    write("Access name field from fetched record")
    write("Name: ", rec.name)

    close(db)
end
```

Sample run:

```
dbtables:
L1 := list(1)
   L1[1] := R__1 := ()
      R__1.qualifier := ""
      R__1.owner := ""
      R__1.name := "contacts"
      R__1.type := ""
      R__1.remarks := ""

dbcolumns.contacts:
L6 := list(3)
   L6[1] := R__1 := ()
      R__1.catalog := ""
      R__1.schema := ""
      R__1.tablename := "contacts"
      R__1.colname := "id"
      R__1.datatype := 4
      R__1.typename := "integer"
      R__1.colsize := 9
      R__1.buflen := 10
      R__1.decdigits := 10
      R__1.numprecradix := 0
      R__1.nullable := 1
      R__1.remarks := ""
   L6[2] := R__2 := ()
      R__2.catalog := ""
      R__2.schema := ""
      R__2.tablename := "contacts"
      R__2.colname := "name"
      R__2.datatype := 12
      R__2.typename := ""
      R__2.colsize := 0
      R__2.buflen := 255
      R__2.decdigits := 10
      R__2.numprecradix := 0
      R__2.nullable := 1
      R__2.remarks := ""
   L6[3] := R__3 := ()
      R__3.catalog := ""
      R__3.schema := ""
      R__3.tablename := "contacts"
      R__3.colname := "phone"
      R__3.datatype := 12
      R__3.typename := ""
      R__3.colsize := 0
      R__3.buflen := 255
```

```
      R__3.decdigits := 10
      R__3.numprecradix := 0
      R__3.nullable := 1
      R__3.remarks := ""

First column
tablename: contacts, colname: id, type: integer
Second column
tablename: contacts, colname: name

Query for name (using dbc[2].colname)
Access name field from fetched record
Name: jafar
```

`dbcolumns` is one of the reflective ODBC functions available to a Unicon programmer. As can be seen in the example above, it can be used for self-documenting purposes (make a run to get all the record fields, then code to suit), or for writing utility applications with user defined naming.

See *ODBC* for details on the example setup.

7.1.46 dbdriver

`dbdriver(D)` : *record*

`dbdriver(db)` produces a record of current ODBC driver information:

```
record driver(name, ver, odbcver, connections, statements, dsn)
```

Or, write a program and look at the data fields using the reflective properties inherent in Unicon.

```
#
# dbdriver.icn, ODBC table column information
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # display ODBC driver info
    write("\nODBC driver information:")
    write(ximage(dbdriver(db)))

    # and a direct field access, (hint: "unicon")
    write("Data Source Name: ", dbdriver(db).dsn)

    close(db)
end
```

Sample run:

```
prompt$ unicon -s dbdriver.icn -x

ODBC driver information:
R__1 := ()
   R__1.name := "sqlite3odbc.so"
   R__1.ver := "0.9992"
   R__1.odbcver := "03.00"
   R__1.connections := 0
   R__1.statements := ""
   R__1.dsn := "unicon"
Data Source Name: unicon
```

See *ODBC* for details on the example ODBC setup.

7.1.47 dbkeys

`dbkeys(D, string)` : *list* [*ODBC*]

`dbkeys(db, tablename)` produces a list of record pairs (col, seq) containing information about the primary keys in the given table.

```
#
# dbkeys.icn, ODBC table column information
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # display table key information
    write("\nODBC table contacts key information:")
    write(ximage(dbkeys(db, "contacts")))

    write("contacts first key: ", dbkeys(db, "contacts")[1].col)

    close(db)
end
```

Sample run:

```
prompt$ unicon -s dbkeys.icn -x

ODBC table contacts key information:
L1 := list(1)
   L1[1] := R__1 := ()
      R__1.col := "id"
```

```
      R__1.seq := 1
contacts first key: id
```

7.1.48 dblimits

`dblimits(D)` : *record* [*ODBC*]

`dblimits(db)` produces a record that contains the upper bounds of many parameters associated with the given database.

```
#
# dblimits.icn, ODBC internal limit information
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # display ODBC limit info
    write("\nODBC limit information:")
    write(ximage(dblimits(db)))

    # and a direct field access
    write("Max SQL statement length: ", dblimits(db).maxstmtlen)

    close(db)
end
```

Sample run:

```
prompt$ unicon -s dblimits.icn -x

ODBC limit information:
R__1 := ()
   R__1.maxbinlitlen := 0
   R__1.maxcharlitlen := 0
   R__1.maxcolnamelen := 255
   R__1.maxgroupbycols := 0
   R__1.maxorderbycols := 0
   R__1.maxindexcols := 0
   R__1.maxselectcols := 0
   R__1.maxtblcols := 0
   R__1.maxcursnamelen := 255
   R__1.maxindexsize := 0
   R__1.maxownnamelen := 255
   R__1.maxprocnamelen := 0
   R__1.maxqualnamelen := 255
   R__1.maxrowsize := 0
```

```
   R__1.maxrowsizelong := "N"
   R__1.maxstmtlen := 16384
   R__1.maxtblnamelen := 255
   R__1.maxselecttbls := 0
   R__1.maxusernamelen := 16
Max SQL statement length: 16384
```

7.1.49 dbproduct

`dbproduct(D)` : *record* [*ODBC*]

`dbproduct(db)` produces a record the gives the name and version of the DBMS product containing the given database.

```
#
# dbproduct.icn, ODBC product and version information
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    write("\nODBC product information:")
    write(ximage(dbproduct(db)))

    # and a direct field access
    write("ODBC version: ", dbproduct(db).ver)

    close(db)
end
```

Sample run:

```
prompt$ unicon -s dbproduct.icn -x

ODBC product information:
R__1 := ()
   R__1.name := "SQLite"
   R__1.ver := "3.9.2"
ODBC version: 3.9.2
```

7.1.50 dbtables

`dbtables(D)` : *list* [*ODBC*]

`dbtables(db)` returns a list of records that describe all the tables in the given database.

```
#
# dbtables.icn, ODBC table list
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # display ODBC table list
    write("\nODBC table list:")
    write(ximage(dbtables(db)))

    # and a direct field access, (hint: "contacts")
    write("First table name: ", dbtables(db)[1].name)

    close(db)
end
```

Sample run:

```
prompt$ unicon -s dbtables.icn -x

ODBC table list:
L1 := list(1)
   L1[1] := R__1 := ()
      R__1.qualifier := ""
      R__1.owner := ""
      R__1.name := "contacts"
      R__1.type := ""
      R__1.remarks := ""
First table name: contacts
```

7.1.51 delay

`delay(i)` : *null*

`delay(i)` pauses a program for at least *i* milliseconds.

```
#
# delay.icn, demontrate the millisecond delay function
#
procedure main()
    # retrieve a microsecond clock value (millionths)
    write(gettimeofday().usec)
    # pause for 10 (or more) milliseconds (thousandths)
    delay(10)
    write(gettimeofday().usec)
end
```

A sample run:

```
prompt$ unicon -s delay.icn -x
517257
527389
```

7.1.52 delete

`delete(x1, x2,...)` : *x1*

`delete(x1, x2,...)` removes one or more elements *x2, [x3....]* from structure *x1*. *x1* can be list, set, table, DBM database, or POP connection.

```
#
# delete.icn, demonstrate delete from structure
#
link fullimag, ximage

procedure main()
    L := [1,2,"abc"]
    write("delete element 3 from list")
    write("Before: ", fullimage(L))
    write("After : ", fullimage(delete(L, 3)))

    T := table()
    T["abc"] := "xyz"
    T["xyz"] := "abc"
    write("delete key abc from table")
    write("Before: ", ximage(T))
    write("After : ", ximage(delete(T, "abc")))
end
```

Sample run:

```
prompt$ unicon -s delete.icn -x
delete element 3 from list
Before: [1,2,"abc"]
After : [1,2]
delete key abc from table
Before: T5 := table(&null)
   T5["abc"] := "xyz"
   T5["xyz"] := "abc"
After : T5 := table(&null)
   T5["xyz"] := "abc"
```

7.1.53 detab

`detab(string, i:9,...)` : *string*

`detab(s, i,...)` replaces tabs with spaces with stops are columns indicated by the second and following arguments, which must be integers. Tabs stops are extended using the interval between the last two specified stops.

```
#
# detab.icn, demonstrate tabs to spaces
#
procedure main()
    write("detab with stops at 4, 8, 12")
    s := "\t\t\tThree tabs in\n\t\tTwo tabs in\n\tOne tab in\nNo tabs"
    write(detab(s, 4,8,12))
end
```

Sample run:

```
prompt$ unicon -s detab.icn -x
detab with stops at 4, 8, 12
           Three tabs in
       Two tabs in
   One tab in
No tabs
```

7.1.54 display

`display(i:&level, f:&errout, CE:¤t)` : *null*

`display()` writes the local variables of the *i* most recent procedure activations from co-expression *CE*, plus all global variables, to the file *f*

```
#
# display.icn, demonstrate the display debugging aid
#
global g,h
procedure main()
    local i,j,k
    g := "a global"
    i := 0
    j := 2
    k := 4
    subproc(i,j,k)
end

procedure subproc(l, m, n)
    display()
end
```

Sample run:

```
prompt$ unicon -s display.icn -x
co-expression_1(1)

subproc local identifiers:
   l = 0
   m = 2
   n = 4
main local identifiers:
   i = 0
   j = 2
   k = 4
```

```
global identifiers:
   display = function display
   g = "a global"
   main = procedure main
   subproc = procedure subproc
```

7.1.55 DrawArc

`DrawArc(w, x, y, wid, h, a1:0.0, a2:&pi*2, ...)` : *window* [*graphics*]

`DrawArc(w, x, y, width, height, a1, a2, ...)` draws arcs or ellipses. Each arc is defined by 4 given and 2 derived coordinates. *x*, *y*, *width*, **height* define a bounding rectangle around the arc; the centre of the arc is the point (*x*+(*width*)/2, *y*+(*height*)/2. Angle *a1* is the starting position of the arc. The angle *a2* is not an end position but specifies the direction and extent of the arc. Angles are given in radians. Multiple arcs can be drawn with one call to the function.

```
#
# DrawArc.icn, demonstrate drawing an Arc
#
procedure main()
    &window := open("DrawArc", "g",
                    "size=65,40", "canvas=hidden")
    # An arc
    Fg(&window, "vivid orange")
    DrawArc(&window, 10, 10, 40, 20)

    Fg(&window, "blue")
    DrawArc(&window, 26, 15, 10, 10, 0.0, &pi)

    # save image for the document
    WSync()
    WriteImage("../images/DrawArc.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawArc.icn -x
```

7.1.56 DrawCircle

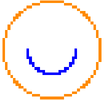
`DrawCircle(w, x, y, radius, a1:0.0, a2:&pi*2, ...)` : *window* [*graphics*]

`DrawCircle(w, x, y, r, a1, a2)` draws a circle centred at *x*,*y*. Otherwise similar to *DrawArc* with width equal to height.

```
#
# DrawCircle.icn, demonstrate drawing a circle
#
procedure main()
    w := open("DrawCircle", "g", "size=40,40", "canvas=hidden")

    # A full circle
    Fg(w, "vivid orange")
    DrawCircle(w, 20, 20, 18)

    # A partial circle
    Fg(w, "blue")
    DrawCircle(w, 20, 20, 9, 0.0, &pi)

    # save image for the document
    WSync(w)
    WriteImage(w, "../images/DrawCircle.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s DrawCircle.icn -x
```



7.1.57 DrawCube

`DrawCube(w, x, y, z, len, ...)` : *record* [*3D graphics*]

`DrawCube(w, x, y, z, len)` draws a cube with sides of length *len* at the position *x*, *y*, *z* on the 3D window *w*. The display list element is returned. This procedure fails if the graphic context attribute *dim* is set to 2.

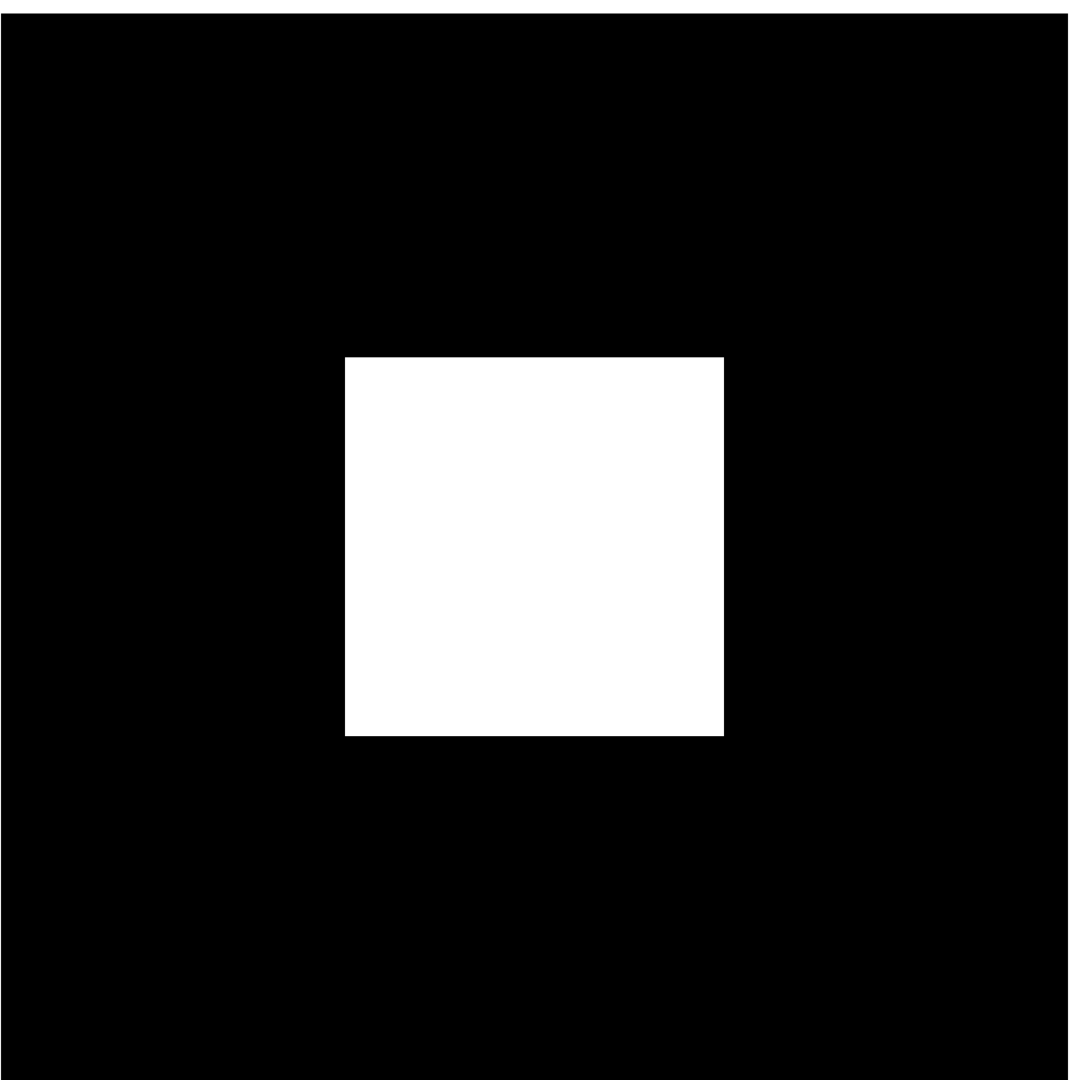
Todo

Not working

```
#
# DrawCube.icn, demonstrate drawing a cube
#
procedure main()
    window := open("DrawCube", "gl", "bg=black", "buffer=on",
                    "size=400,400", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A cube
    DrawCube(window, 0.0, 0.0, -2.0, 0.6)

    # save image for the document
    Refresh(window)
    WSync(window)
```

```
    WriteImage(window, "../images/DrawCube.png")
    close(window)
end
```

Sample run:



7.1.58 DrawCurve

`DrawCurve(w, x1, y1, ...)` : *window*

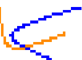
`DrawCurve(w, x1,y1, x2,y2, x3,y3, ...)` draws a smooth curve between each *x,y* pair in the argument list. If the first and last point are the same, the curve is smoothed and closed at that point.

```
#
# DrawCurve.icn, demonstrate drawing an Arc
#
procedure main()
    w := open("DrawCurve", "g", "size=65,40", "canvas=hidden")
```

```
    # Some curves
    Fg(w, "vivid orange")
    DrawCurve(w, 10,10, 15,25, 35,20)

    Fg(w, "blue")
    DrawCurve(w, 30,30, 15,20, 40,10)

    # save image for the document
    WSync(w)
    WriteImage(w, "../images/DrawCurve.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s DrawCurve.icn -x
```



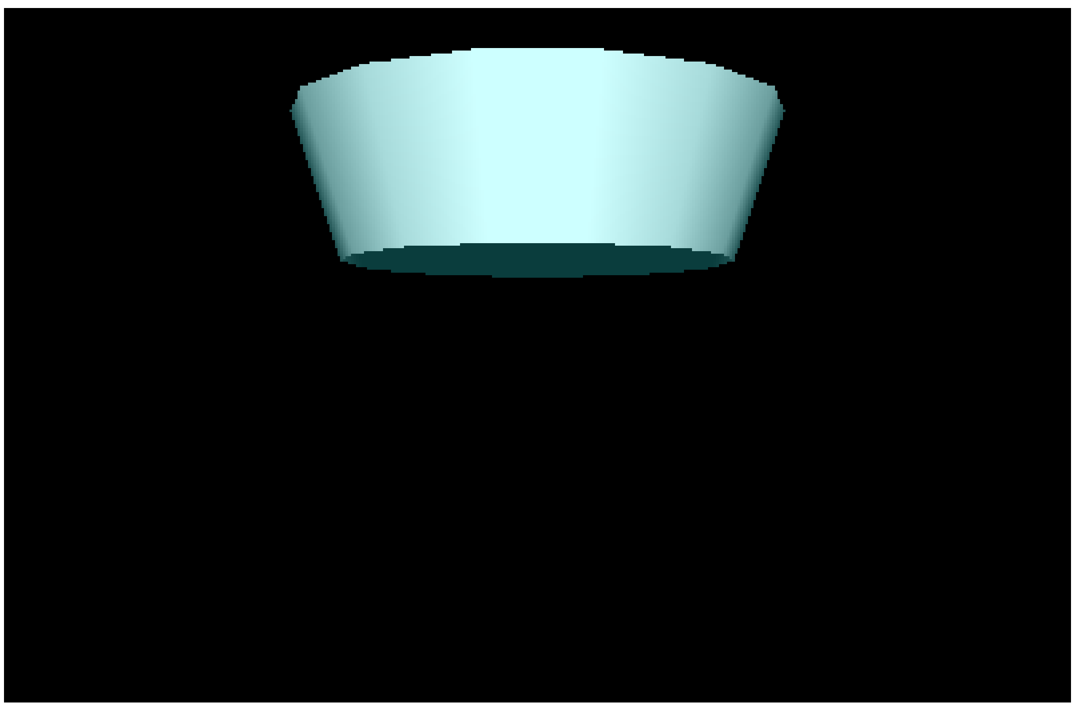
7.1.59 DrawCylinder

`DrawCylinder(w, x,y,z, h, r1,r2,...)` : *record* [*3D graphics*]

`DrawCylinder(w, x,y,z, h, rt,rb)` draws a cylinder with a top radius *rt*, a bottom with radius *rb*, a height *h* centred at *x,y,z* on window *w*. A display list element is returned. `DrawCylinder` fails if window attribute `dim` is set to 2.

```
#
# DrawCylinder.icn, demonstrate drawing a cylinder
#
procedure main()
    window := open("DrawCylinder", "gl", "bg=black", "buffer=on",
                    "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")
    # A cube
    DrawCylinder(window, 0.0, 0.19, -2.2, 0.3, 0.4, 0.5)

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/DrawCylinder.png")
    close(window)
end
```

Sample output:



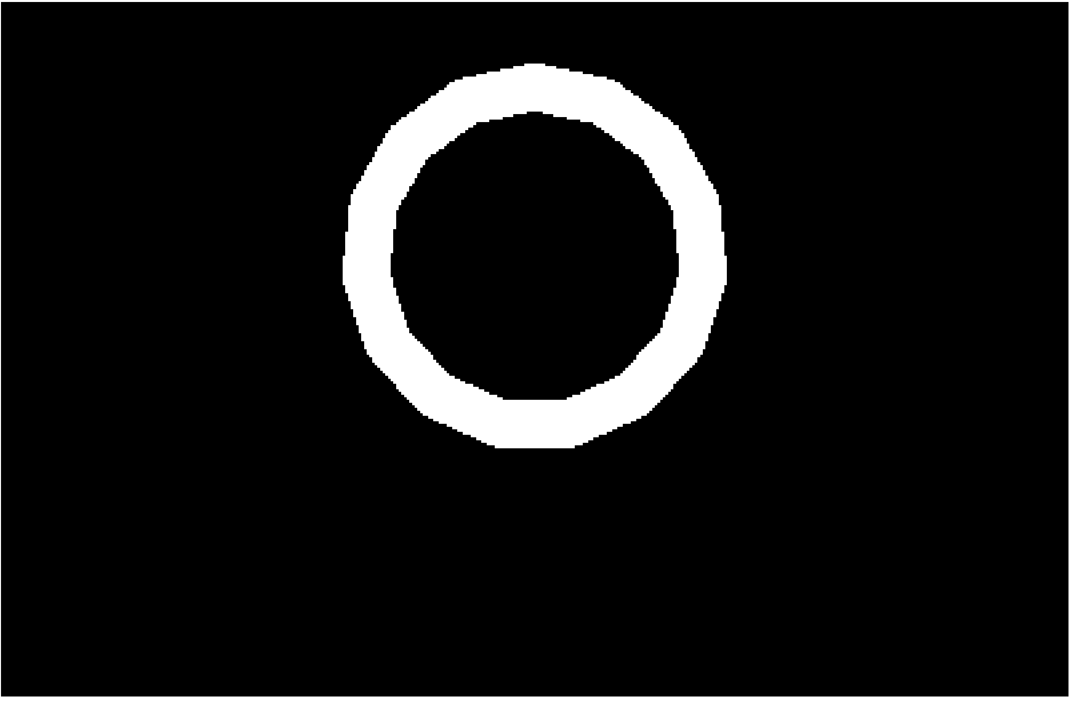
7.1.60 DrawDisk

`DrawDisk(w, x,y,z, r1,r2, a1,a2,...)` : *record*

`DrawDisk(w, x,y,z, ri, ro, astart, asweep)` draws a (partial) disk centred at *x,y,z*, with an inner circle of radius *ri*, an outer circle of radius *ro*, a starting angle of *astart*, and a sweeping angle of *asweep*, on window *w*. The parameters *a1* and *a2* are optional, and a full disk is rendered if they are not provided. The display list element is returned.

```
#
# DrawDisk.icn, demonstrate drawing a Disk
#
procedure main()
    window := open("DrawDisk", "gl", "bg=black", "buffer=on",
                        "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A disk
    DrawDisk(window, 0.0,0.19,-2.2, 0.3,0.4)

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/DrawDisk.png")
    close(window)
end
```

Sample output:



7.1.61 DrawImage

`DrawImage(w, x, y, s)` : *window*

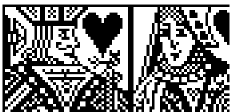
`DrawImage(w, x, y, s)` draws image string *s* at *x,y* in window *w*.

Unicon image strings are of the form "*width*, *palette*, *pixels*".

```
#
# DrawImage.icn, demonstrate drawing image strings
#
link cardbits
procedure main()
    &window := open("DrawImage", "g", "size=85,40", "canvas=hidden")

    # image data is "width, palette, pixels"
    img := cardbits()
    DrawImage(0, 0, img)

    # save image for the document
    WSync()
    WriteImage("../images/DrawImage.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawImage.icn -x
```



7.1.62 DrawLine

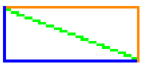
`DrawLine(w, x1, y1, z1, ...)` : *window* | *list* [*graphics/3D graphics*]

`DrawLine(w, x1, y1,...,xn,yn)` draws lines between each adjacent *x*, *y* pair of arguments. In 3D, `DrawLine` takes from 2-4 coordinate per vertex and returns a list the represents the lines on the display list *for refresh purposes*.

```
#
# DrawLine.icn, demonstrate drawing a line
#
procedure main()
    &window := open("DrawLine", "g",
                    "size=70,40", "canvas=hidden")
    # An Line
    Fg("vivid orange")
    DrawLine(11,10, 60,10, 60,30)

    Fg("blue")
    DrawLine(10,10, 10,30, 59,30)

    Fg("green")
    DrawLine(11,11, 59,29)

    # save image for the document
    WSync()
    WriteImage("../images/DrawLine.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawLine.icn -x
```



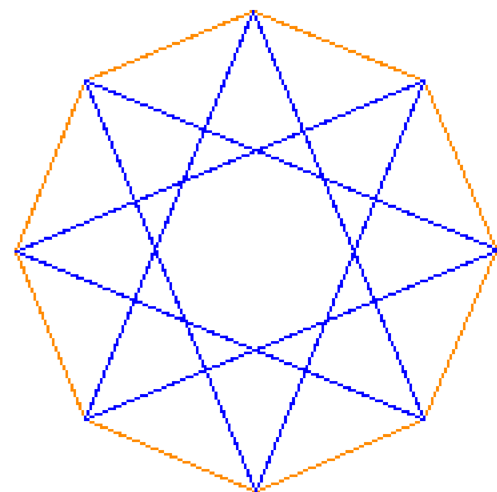
A more sophisticated line drawing example is in the *Graphics Programming in Icon* book, pages 73-75. Using `DrawLine` to produce polygons (see also *DrawPolygon*) and star shapes; with a single function that depends on a `skip` value to draw the stars.

```
#
# linedrawing.icn, from the Graphics Programming in Icon book
#
# Draw a regular polygon with the specified number of vertices and
# radius, centered at (cx,cy).
#
procedure main()
    &window := open("linedrawing", "g", "size=200,200", "canvas=hidden") |
               stop("Cannot open graphics window")

    Fg("vivid orange")
    rpolystars(100, 100, 90, 8)
```

```
    Fg("blue")
    rpolystars(100, 100, 90, 8, 3)

    WSync()
    WriteImage("../images/linedrawing.png")
    close(&window)
end

procedure rpolystars(cx, cy, radius, vertices, skips)
    local theta, incr, xprev, yprev, x, y

    theta := 0                              # initial angle
    /skips := 1
    incr := skips * 2 * &pi / vertices
    xprev := cx + radius * cos(theta)       # initial position
    yprev := cy + radius * sin(theta)

    every 1 to vertices do {
        theta +:= incr
        x := cx + radius * cos(theta)       # new position
        y := cy + radius * sin(theta)
        DrawLine(xprev, yprev, x, y)
        xprev:= x                           # update old position
        yprev:= y
    }

    return
end
```

With a run sample of:

```
prompt$ unicon -s linedrawing.icn -x
```



7.1.63 DrawPoint

`DrawPoint(w, x1,y1, ...)` : *window* [list]

`DrawPoint(w, x1,y1,...,xn,yn)` draws points given by *x,y* pairs on window *w*. With 3D graphics, `DrawPoint()` takes from 2 to 4 coordinates per vertex and returns the list that represents the points on the display list for refresh purposes.

```
#
# DrawPoint.icn, demonstrate drawing points in 2D
#
procedure main()
    &window := open("DrawPoint", "g", "size=45,30", "canvas=hidden")

    # Some points
    Fg("vivid orange")
    every x := 10 to 22 by 4 do
        every y := 10 to 16 do
            DrawPoint(x,y)
    # and a crossing line
    DrawLine(8,12, 24,14)

    # save image for the document
    WSync()
    WriteImage("../images/DrawPoint.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawPoint.icn -x
```

Todo

3D points

With 3D graphics, points are x,y,z

```
#
# DrawPoint-3D.icn, demonstrate drawing points in 3D
#
link ximage
procedure main()
    &window := open("DrawPoint-3D","gl", "bg=white", "fg=orange",
                    "buffer=on", "size=400,200") #, "canvas=hidden")

    # Some 3D points
    every x := 0 to 0.3 by 0.05 do
        every y := 0.1 to 0.3 by 0.1 do
            every z := -2.0 to -1.8 by 0.005 do
                DrawPoint(x,y,z)

    # save image for the document
    Refresh()
```

```
    WSync()
    WriteImage("../images/DrawPoint-3D.png")
    close(&window)
end
```

Sample output:

7.1.64 DrawPolygon

`DrawPolygon(w, x1,y1[,z1], ..., xn,yn[,zn])` : *window* | *list*

`DrawPolygon(w, x1,y1, x2,y2, xn,yn)` draws a polygon connecting each *x,y* pair (in 2D). In 3D, `DrawPolygon()` takes from 2 to 4 coordinates per vertex and returns the list that represents the polygon on the display list.

```
#
# DrawPolygon.icn, demonstrate drawing polygons
#
procedure main()
    &window := open("DrawPolygon", "g", "size=45,30", "canvas=hidden")

    # a polygon using the procedural apply operator
    Fg("vivid orange")
    points := [5,5, 17,17, 5,17, 5,5]
    DrawPolygon!points

    # save image for the document
    WSync()
    WriteImage("../images/DrawPolygon.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawPolygon.icn -x
```

Todo

3D polygons

7.1.65 DrawRectangle

`DrawRectangle(w, x1,y1, wid1, h1, ...)` : *window*

`DrawRectangle(w, x, y, width, hieght)` draws a rectangle with a top right corner of *x,y* and a perceived width and height. Actual rectangle is *width+1* pixels wide, and *height+1* pixels high.

```
#
# DrawRectangle.icn, demonstrate drawing polygons
#
procedure main()
    &window := open("DrawRectangle", "g", "size=45,40", "canvas=hidden")

    Fg("vivid orange")
    DrawRectangle(10,10, 20, 10)

    # save image for the document
    WSync()
    WriteImage("../images/DrawRectangle.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawRectangle.icn -x
```

7.1.66 DrawSegment

`DrawSegment(w, x1,y1[,z1], x2,y2[,z2],...)` : *window|list*

`DrawSegment(w, x1,y1, x2,y2,...)` draws lines between alternating *x,y* pairs in the argument list. In 3D, `DrawSegment` takes from 2 to 4 coordinates per vertex and returns the list that represents the segments.

```
#
# DrawSegment.icn, demonstrate drawing points in 2D
#
procedure main()
    &window := open("DrawSegment", "g", "size=45,30", "canvas=hidden")

```

```
    Fg("vivid orange")
    every x := 10 to 22 by 4 do
        every y := 10 to 16 do
            DrawSegment(x,y, x+5,y-5)

    # save image for the document
    WSync()
    WriteImage("../images/DrawSegment.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s DrawSegment.icn -x
```

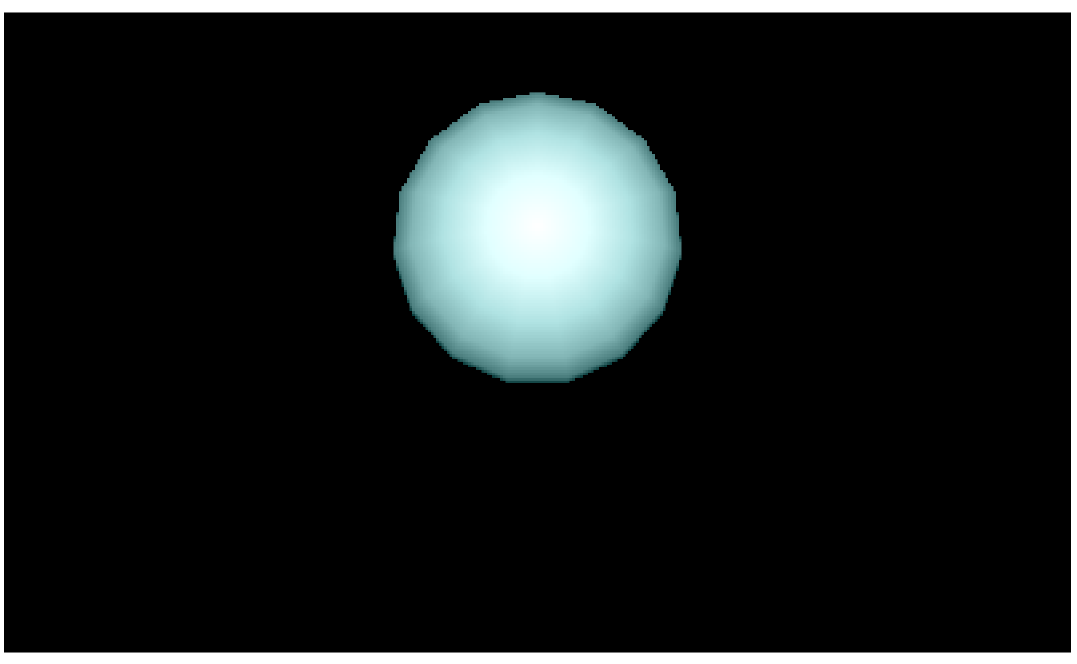
7.1.67 DrawSphere

`DrawSphere(w, x,y,z, r,...)` : *record* [*Graphics, 3D*]

`DrawSphere(w, x,y,z, r,...)` draws a sphere with radius *r* centred at *(x,y,z)* on 3D window *w*. The display list is returned. Fails when used on windows with *WAttrib* "dim=2".

```
#
# DrawSphere.icn, demonstrate drawing a sphere
#
procedure main()
    window := open("DrawSphere", "gl", "bg=black", "buffer=on",
                   "size=400,240", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A sphere
    DrawSphere(window, 0.0, 0.19, -2.2, 0.3)

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/DrawSphere.png")
    close(window)
end
```

Sample output:



7.1.68 DrawString

`DrawString(w, x1, y1, s1,...)` : *window*

`DrawString(w, x, y, s)` draws string *s* on window *w* at *x*, *y*, without effecting the current text cursor position. When used with `drawop=reverse` it is possible to draw erasable test. No background is drawn, only the actual pixels of the characters.

```
#
# DrawString.icn, demonstrate a boxed string
#
procedure main()
    w := open("DrawString", "g", "size=162,30",
              "linestyle=solid", "canvas=hidden")
    Font(w, "Liberation Mono")

    text := "Important message"
    fh := WAttrib(w, "fheight")
    tw := TextWidth(w, text)

    Fg(w, "purple")
    DrawString(w, 10, 20, text)
    Fg(w, "black")
    DrawRectangle(w, 5, 5, tw + 8, fh + 6)

    WSync(w)
    WriteImage(w, "../images/DrawString.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s DrawString.icn -x
```

Important message

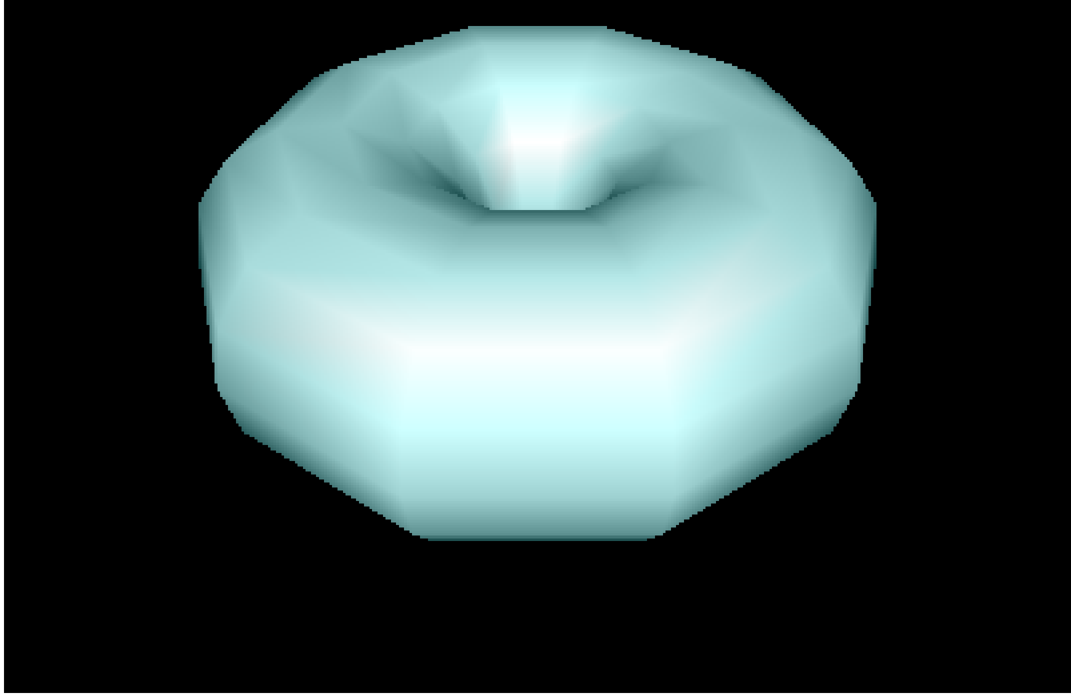
7.1.69 DrawTorus

`DrawTorus(w, x,y,z, r1,r2,...)` : *record*

`DrawTorus(w, x,y,z, ri, ro)` draws a torus with inner radius *ri*, outside radius *r2*, centred at *x,y,z* on 3D window *w*. The display list element is returned. `DrawTorus` fails if the window attribute `dim` is set to 2.

```
#
# DrawTorus.icn, demonstrate drawing a Torus
#
procedure main()
    window := open("DrawTorus", "gl", "bg=black", "buffer=on",
                    "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/DrawTorus.png")
    close(window)
end
```

Sample output:



7.1.70 dtor

`dtor(r)` : *real*

`dtor(r)` produces the equivalent of *r* degrees, in radians.

```
#
# dtor.icn, demonstrate degrees to radians
#
link numbers

# uses decipos from numbers, align decimal within field
procedure main()
    write("Degrees Radians")
    every r := 0.0 to 360.0 by 45.0 do
        write(decipos(r, 4, 8), decipos(dtor(r), 2, 20))
end
```

Sample run:

```
prompt$ unicon -s dtor.icn -x
Degrees Radians
  0.0   0.0
 45.0   0.7853981633974483
 90.0   1.570796326794897
135.0   2.356194490192345
180.0   3.141592653589793
225.0   3.926990816987241
270.0   4.71238898038469
315.0   5.497787143782138
360.0   6.283185307179586
```

7.1.71 entab

`entab(s, i:9,...)` : *string*

`entab(s, i,...)` replaces spaces with tabs, with stops at the columns indicated. Tab stops are extended using the interval between the last two specified stops. Defaults give 8 space tabs, with stops at 1, 9, 17, etcetera.

```
#
# entab.icn, demonstrate spaces to tabs
#
procedure main()
    write("entab with stops at 4, 8, 12")
    s := repl(" ", 12) || "Three tabs in\n" ||
         repl(" ", 8) || "Two tabs in\n" ||
         "    One tab in\nNo tabs"
    write(entab(s, 4,8,12))
end
```

Sample run:

```
prompt$ unicon -s entab.icn -x | cat -T
entab with stops at 4, 8, 12
^I^I^I Three tabs in
^I^I Two tabs in
^I One tab in
No tabs
```

7.1.72 EraseArea

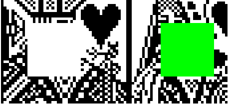
`EraseArea(w, x:0,y:0, wid:0, h:0, ...)` : *window*

`EraseArea(w, x,y, width,height)` erases a rectangular area to the background colour. If *width* is 0, the region extends from *x* to the right. If *height* is 0, the region extends from *y* to the bottom. In 3D, `EraseArea(w)` clears the contents of the entire window.

```
#
# EraseArea.icn, demonstrate erasing part of a window
#
link cardbits
procedure main()
    &window := open("EraseArea", "g", "size=85,40", "canvas=hidden")

    # image data is "width, palette, pixels"
    img := cardbits()
    DrawImage(&window, 0, 0, img)
    EraseArea(&window, 10,10, 20,20)
    Bg("green")
    EraseArea(&window, 60,10, 20,20)

    # save image for the document
    WSync()
    WriteImage("../images/EraseArea.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s EraseArea.icn -x
```



7.1.73 errorclear

`errorclear()` : *null*

`errorclear()` resets the keywords *&errornumber*, *&errortext*, *&errorvalue* to indicate that no error is present.

```
#
# errorclear.icn, Demonstrate error keyword condition reset
#
link printf
procedure main()
    &error := 1
    nonexistent()

    # source line tracking not optimal here
    if &errornumber ~= 0 then
        write(printf("%s:%d Runtime error %d:%s", &file, &line-5,
                                    &errornumber, &errortext))

    errorclear()
    if &errornumber ~= 0 then
        write(printf("%s:%d Runtime error %d:%s", &file, &line-9,
                                    &errornumber, &errortext))
end
```

A contrived example:

```
prompt$ unicon -s errorclear.icn -x
errorclear.icn:13 Runtime error 106:procedure or integer expected
```

7.1.74 Event

`Event(w, i:infinity)` : *string|integer*

`Event(w, i)` produces the next event available for window *w*. If no events are available, `Event()` waits *i* milliseconds. Keyboard events are returned as *String* while mouse events are returned as *Integer*. When an event is retrieved the keywords *&x*, *&y*, *&row*, *&col*, *&interval*, *&control*, *&shift*, and *&meta* are also set. If the 3D attribute "pick=on" is active, *&pick* is also set. `Event()` fails is there is a timeout before an event is available.

```
#
# Event.icn, demonstrate event returns
#
link enqueue, evmux
procedure main()
    window := open("Event", "g", "size=20,20", "canvas=hidden")

    # insert an event into the queue, left press, control and shift
    Enqueue(window, &lpress, 11, 14, "cs", 2)
    e := Event(window)
    write(image(e))

    # a side effect of the Event function is keywords settings
    write("&x:", &x)
    write("&y:", &y)
    write("&row:", &row)
    write("&col:", &col)
    write("&interval:", &interval)
    write("&control:", &control)
    write("&shift:", &shift)
    write("&meta:", &meta)
```

```
    close(window)
end
```

Sample run, right-click while holding down shift and control keys:

```
prompt$ unicon -s Event.icn -x
-1
&x:11
&y:14
&row:2
&col:2
&interval:2
&control:
&shift:
```

7.1.75 eventmask

`eventmask(CE, cset, T)` : *cset* | *null*

`eventmask(ce)` returns the event mask associated with the program that created *ce*, or *&null* if there is no event mask. `eventmask(ce, cs)` sets that program's event mask to *cs*. These settings control the *Unicon monitoring* triggers of a running program. A third *Table* argument, *T*, specifies a different value mask for each event code in the table; handy for filtering virtual machine instructions (which is event code `E_Opcode` as defined in `evdefs.icn`). Individual instruction codes are defined in `opdefs.icn` a file designed to be used with *$include*.

```
#
# eventmask.icn, display default execution monitor eventmask
#
procedure main()
    write("in main with eventmask ", image(eventmask(&current)))
    coex := create(write(image(eventmask(&current))))
    @coex
end
```

Sample run:

```
prompt$ unicon -s eventmask.icn -x
in main with eventmask &null
&null
```

See also:

Unicon monitoring

7.1.76 EvGet

`EvGet(c, flag)` : *string* [*Execution Monitoring*]

`EvGet(c, flag)` activates a program being monitored until an event in *cset* mask *c* occurs. Normally `EvGet()` returns a one character event code. *c* default is all events encoded in *eventmask*. *flag* controls the handling of out-of-band data, a null flag rejects out-of-band event data.

```
#
# EvGet.icn, demonstrate Execution Monitoring event get
# Needs 1to4.icn as the monitoring target
#

$include "evdefs.icn"
link evinit, printf

procedure main()
    # initialize the Target Process (a loaded task co-expression)
    EvInit("1to4")

    # loop across events until the task completes
    perline := 0
    while ec := EvGet() do {
        # keep the event code display all lined up
        if (perline +:= 1) > 16 then write() & perline := 1
        if ec == E_Fret then perline := 1
        # event code display in octal, to match evdefs.icn defines
        printf("%03o ", ord(ec))
    }
    write()
end
```

Sample run:

```
prompt$ unicon -s EvGet.icn -x
275 271 240 240 272 103 274 273 240 240 242 241 274 273 240 241
240 240 240 277 240 240 277 240 240 277 240 273 240 260 275 271
263 273 240 260 111 113 116 111 113 163 163 261 273 240 272 363
143 252 111 113 116
1
255 240 240 251 264 276 271 275 271 263 273 240 260 111 113 116
111 113 163 163 261 273 240 272 363 143 252 111 113 116
2
255 240 240 251 264 276 271 275 271 263 273 240 260 111 113 116
111 113 163 163 261 273 240 272 363 143 252 111 113 116
3
255 240 240 251 264 276 271 275 271 263 273 240 260 111 113 116
111 113 163 163 261 273 240 272 363 143 252 111 113 116
4
255 240 240 251 264 276 271 262 251 273 240 274 273 240 246 240
130
```

7.1.77 EvSend

`EvSend(i, x, CE)` : *any*

`EvSend(c, v, ce)` transmits event code *c* with value *v* to a monitored co-expression *ce*.

```
#
# EvSend.icn, demonstrate Execution Monitoring event send
# Needs 1to4.icn as the monitoring target
#
# WARNING: This is not a safe example, still learning
```

```
#

$include "evdefs.icn"
link evinit, printf

procedure main()
    # initialize the Target Process
    EvInit("1to4")

    # loop across the execution events
    perline := 0
    while ec := EvGet() do {
        # keep a neat event code display
        if (perline +:= 1) > 16 then write() & perline := 1
        if ec == E_Fret then perline := 1
        # event code display in octal to match evdefs defines
        printf("%03o ", ord(ec))

        #
        # DON'T DO THIS AT HOME
        #
        # if the current event is a conversion attempt for the 3
        # being working on in 1to4, tell the co-expression the
        # the conversion failed, (this skips over a conversion target
        # event) probably bugging out the VM
        #
        if ec == E_Aconv & &eventvalue === 3 then {
            writes("sending a conversion fail event, returned: ")
            write(image(EvSend(E_Fconv, &eventvalue, &eventsource)))
        }
    }
    write()
end
```

Sample run (DANGEROUS, as this author doesn't get the whole picture yet):

```
prompt$ unicon -s EvSend.icn -x
275 271 240 240 272 103 274 273 240 240 242 241 274 273 240 241
240 240 240 277 240 240 277 240 240 277 240 273 240 260 275 271
263 273 240 260 111 113 116 111 113 163 163 261 273 240 272 363
143 252 111 113 116
1
255 240 240 251 264 276 271 275 271 263 273 240 260 111 113 116
111 113 163 163 261 273 240 272 363 143 252 111 113 116
2
255 240 240 251 264 276 271 275 271 263 273 240 260 111 113 116
111 sending a conversion fail event, returned: "K"
163 163 261 273 240 272 363 143 252 111 113 116
3
255 240 240 251 264 276 271 275 271 263 273 240 260 111 113 116
111 113 163 163 261 273 240 272 363 143 252 111 113 116
4
255 240 240 251 264 276 271 262 251 273 240 274 273 240 246 240
130
```

7.1.78 exec

`exec(string, string,...)` : *null* [*POSIX*]

`exec(s, arg0, arg1,...)` replaces the currently executing Unicon program with a new program, named *s*. Other arguments are passed to the program as the argument list. *s* must be a path to a binary executable program. To evaluate scripts, the *s* should be a shell such as `/bin/sh`.

```
#
# exec.icn, demonstrate the POSIX exec function
#
procedure main()
    exec("./exec-replacement", "argv0", "arglist[1]", "arglist[2]")
end
```

```
#
# exec-replacement.icn, the program replaced by the exec function demo
#
procedure main(arglist)
    write(&progname)
    every write(!arglist)
end
```

Sample run:

```
prompt$ unicon -s -C exec-replacement.icn
```

```
prompt$ unicon -s exec.icn -x
argv0
arglist[1]
arglist[2]
```

7.1.79 exit

`exit(i)`

`exit()` terminates the current program execution, returning a normal termination status code to the operating system (which will be system dependent, normally zero). `exit(i)` returns status code *i*.

```
#
# exit.icn, demonstrate shell exit status
#
procedure main()
    exit(42)
end
```

Sample run:

```
prompt$ unicon -s exit.icn ; ./exit ; echo $?
42
```

7.1.80 exp

`exp(r)` : *real*

`exp(r)` returns *&e* raised to the power *r*.

```
#
# exp.icn, demonstrate the natural exponential function
#
procedure main()
    every r := 0 | 1 | 2 | &e do write("exp(", r, ") ", exp(r))
end
```

Sample run:

```
prompt$ unicon -s exp.icn -x
exp(0) 1.0
exp(1) 2.718281828459045
exp(2) 7.38905609893065
exp(2.718281828459045) 15.15426224147926
```

7.1.81 Eye

`Eye(w, s)` : *window*

`Eye(w)` retrieves the current 3D eye parameters. `Eye(w, s)` sets the `eyedir` (direction), `eyepos` (position), `eyeup` (up vector) graphic attributes. Each of these is an *x,y,z* coordinate; defaulting to "0,0,0,0,0,-1,0,1,0". For a 3D scene, changing the `Eye()` values will cause the entire window to be rendered from the new point of view.

- eyepos, where the eye (camera) is located in *x,y,z*. Default is 0,0,0.
- eyedir, where the eye is looking, defaults to looking into the *negative* *z* axis. 0,0,-1.
- eyeup, what point in the 3D space is the up direction, defaults to the positive *y* axis. 0,1,0.

```
#
# Eye.icn, display the viewport reference
#
procedure main()
    w := open("Eye", "gl", "bg=red",
                "size=400,300")#, "canvas=hidden")
    WAttrib(w,"light0=on, ambient blue-green","bg=white", "fg=ambient yellow")
    # A disk
    DrawDisk(w, 0.4,-0.5,-4.0, 0.3,1.0)
    write(Eye(w))
    # save image for the document
    WSync(w)
    Refresh(w)
    WAttrib(w, "eyepos=4,-4,8.0")
    WSync(w)
    Refresh(w)
    write(Eye(w))
    WAttrib(w, "eyedir=1,0,0.4")
    Refresh(w)
    write(Eye(w))
    WriteImage(w, "../images/Eye.png")
```

```
    close(w)
end
```

Sample run (not auto captured):

```
prompt$ unicon -s Eye.icn -x
0.00,0.00,0.00,0.00,0.00,-100.00,0.00,1.00,0.00
4.00,-4.00,8.00,0.00,0.00,-100.00,0.00,1.00,0.00
4.00,-4.00,8.00,1.00,0.00,0.40,0.00,1.00,0.00
```

See also:

WAttrib

7.1.82 Fail

`Fail()` : *fail* [*Patterns*]

The `Fail()` pattern signals a failure in a local portion of a pattern match. It causes the goal-directed evaluation engine to backtrack and seek alternatives. This is different from *Abort* which stops pattern matching, `Fail()` tells the system to back and try alternatives.

```
#
# Fail.icn, demonstrate Fail() SNOBOL pattern, forces backtracking
#
# Display one character per line by forcing alternatives for Len()
procedure main()
    sub := "Force backtrack"
    out := &output
    sub ?? Len(1) => out || Fail()
end
```

Sample run (taken from Unicon Technical Report 18, page 15):

```
prompt$ unicon -s Fail.icn -x
F
o
r
c
e

b
a
c
k
t
r
a
c
k
```

7.1.83 fcntl

`fcntl(f, s, s)` : *integer|string|record* [*POSIX*]

`fcntl(file, cmd, arg)` performs miscellaneous operations on the open file *file*. Directories and DBM files cannot be arguments to `fcntl()`. See *fcntl(2)*.

The following characters are possible values for *cmd*:

- f, Get flags (F_GETFL)
- F, Set flags (F_SETFL)
- x, Get close on exec flags (F_GETFD)
- X, Set close on exec flags (F_SETFD)
- l, Get file lock (F_GETLK)
- L, Set file lock (F_SETLK)
- W, Set file lock and wait (F_SETLKW)
- o, Get file owner or process group (F_GETOWN)
- O, Set file owner or process group (F_SETOWN)

In the case of `L`, the *arg* value should be a string that describes the lock, otherwise *arg* value is an *Integer*.

The lock string consists of three parts separated by commas:

- the type of lock (r, w, or u)
- the starting position
- the length

The starting position can be an offset from the beginning of the file, *n*, the end of the file *-n*, or a relative position *+n*. A length of 0 means lock till EOF. These characters represent the file flags set by F_SETFL and retrieved with F_SETFL:

- d, FNDELAY
- s, FASYNC
- a, FAPPEND

```
#
# fcntl.icn, demonstrate the POSIX fcntl function
#
link ximage
procedure main()
    f := open(&file, "r") | stop("Cannot open " || &file)
    every cmd := "f" | "x" | "o" do
        write(image(fcntl(f, cmd, 0)))
    write(ximage(fcntl(f, "l", "r,0,0")))
    close(f)
end
```

Sample run:

```
prompt$ unicon -s fcntl.icn -x
""
0
0
```

```
R_posix_lock_1 := posix_lock()
   R_posix_lock_1.value := "u,0,0"
   R_posix_lock_1.pid := ""
```

7.1.84 fdup

`fdup(f, f)` : *?* [*POSIX*]

`fdup(src, dst)` duplicates a file descriptor, by closing *dst* and setting *src* to that file descriptor; the *dst* fd is replaced by the *src* fd. See *dup2(2)*. Commonly used with *exec* to manage standard in and standard out streams.

```
#
# fdup.icn, duplicate a POSIX file descriptor
#
procedure main()
    src := open("oldfile.txt", "r") #| stop("cannot open oldfile.txt")
    dst := open("newfile.txt", "r") #| stop("cannot open newfile.txt")
    write("org newfile: ", read(dst))
    if fdup(src, dst) then {
        write("read from newfile.txt will now look like oldfile.txt")
        write("dup newfile: ", read(dst))
        write("dup newfile: ", read(dst))
    }
    else
        write("fdup(src, dst) failed")
    close(src)
    close(dst)
end
```

Sample run:

```
prompt$ unicon -s fdup.icn -x
org newfile: newfile line 1
read from newfile.txt will now look like oldfile.txt
dup newfile: oldfile line 1
dup newfile: oldfile line 2
```

7.1.85 Fence

`Fence()` : *type*

Todo

entry for function Fence

`Fence()`

Sample run:

7.1.86 fetch

`fetch(D, s)` : *string* | *row?*

`fetch(db, k)` fetches the value corresponding to key *k* from a DBM or SQL database *db*. The result is a *String* for DBM or a *row* for SQL. If the key *k* is omitted for a current SQL query, `fetch(db)` produces the next row of the selection and advances the cursor to the next row. A *row* is a *record* where the field names and types are determined by the current query. `fetch` will fail if there are no more rows. Typically a call to *sql* will be followed by a while loop that calls `fetch(db)` until it fails.

```
#
# fetch.icn, demonstrate retrieving rows from ODBC
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # make a query
    sql(db, "SELECT name, phone FROM contacts")

    while row := fetch(db) do {
        write("Contact: ", row.name, ", ", row.phone)
    }

    close(db)
end
```

Sample run:

```
prompt$ unicon -s fetch.icn -x
Contact: brian, 613-555-1212
Contact: jafar, 615-555-1213
Contact: brian, 615-555-1214
Contact: clint, 615-555-1215
Contact: nico, 615-555-1216
```

7.1.87 Fg

`Fg(w, s)` : *string* [*graphics*]

`Fg(w)` retrieves the current foreground colour. `Fg(w, s)` sets the foreground colour by name or value. `Fg()` fails if the foreground cannot be set to the given colour.

In 3D graphics, `Fg(w, s)` changes the material properties of subsequently drawn objects to those requested by *s*. The string *s* must be a semi-colon separated list of material properties. A material property is of the form:

```
diffuse | ambient | specular | emission | colour name | shininess n
```

Where shininess values range from 0 thru 128. `Fg(w)` retrieves the current values of the material properties.

```
#
# Fg.icn, demonstrate foreground colour settings
#
procedure main()
    window := open("Fg example", "g", "size=110,75", "canvas=hidden")
    write(window, Fg(window))

    Fg(window, "vivid orange")
    write(window, Fg(window))

    Fg(window, "10000,20000,30000")
    write(window, Fg(window))

    Bg(window, "black")
    Fg(window, "transparent green")
    write(window, Fg(window))
    Bg(window, "white")

    if c := NewColor(window, "40000,50000,60000") then {
        Fg(window, c)
        write(window, Fg(window))
    }
    else {
        Fg(window, "black")
        write(window, "no mutable colours")
    }

    WSync(window)
    WriteImage(window, "../images/Fg.png")
    FreeColor(\c)
    close(window)
end
```

Sample run:

```
prompt$ unicon -s Fg.icn -x
```

7.1.88 fieldnames

`fieldnames(R)` : *strings**

`fieldnames(r)` generates the names of the fields in the *record* *r*.

```
#
# fieldnames.icn, demonstrate reflective fieldnames function
#

record sample(a,b,c,thing, other)

procedure main()
    R := sample(1,2,3,"xyz", [1,2,3])
    every write(fieldnames(R))
end
```

Sample run:

```
prompt$ unicon -s fieldnames.icn -x
a
b
c
thing
other
```

7.1.89 filepair

`filepair()` : *list* [*POSIX*]

`filepair()` creates a bi-directional pair of file, analogous to the POSIX `socketpair(2)` function. It returns a list of two *indistinguishable* files. Writes on one will be avaialable on the other. The connection is bi-directional, unlike that of the *pipe* function.

File pairs are typically created just before a *fork* operation. After forking, one process should close *L[1]* and the other should close *L[2]* for end of file notifications to work properly.

```
#
# filepair.icn, connected file streams
#
link ximage

procedure main()
    pair := filepair() | stop("filepair failed: ", sys_errstr(&errno))

    pid := fork() | stop("fork failed")
    if pid < 0 then stop("fork failed: ", pid)
    if pid = 0 then {
        close(pair[2])
        child(pair[1])
    } else {
        close(pair[1])
        parent(pair[2])
    }
    write("both: ", pid)
end

#
# child, writer half
#
procedure child(out)
    write("Child ", image(out))
    every i := 1 to 10 do {
        write(out, "this is filepair write test ", i) |
            stop("failed write")
        flush(out)
    }
    close(out)
end

#
# parent, reader half
#
procedure parent(in)
```

```
    write("Parent ", image(in))
    # needs to be a non-blocking read
    fcntl(in, "F", "d")

    limiter := 0
    until (limiter +:=1) > 200 do {
        if result := ready(in) then {
            write("parent received: ", image(result))
            break
        } else {
            delay(10)
        }
    }
    close(in)
end
```

Sample run (skipped, bug report pending):

7.1.90 FillArc

`FillArc(w, x, y, wid, h, a1:0.0, a2:&pi*2, ...)` : *window* [*graphics*]

`FillArc(w, x, y, width, height, a1, a2, ...)` draws filled arcs or ellipses. Each arc is defined by 4 given and 2 derived coordinates. *x*, *y*, *width, *height* define a bounding rectangle around the arc; the centre of the arc is the point (*x*+(*width*)/2, *y*+(*height*)/2. Angle *a1* is the starting position of the arc. The angle *a2* is not an end position but specifies the direction and extent of the arc. Angles are given in radians. Multiple arcs can be drawn with one call to the function.

```
#
# FillArc.icn, demonstrate drawing a filled Arc
#
procedure main()
    &window := open("FillArc", "g",
                    "size=65,40", "canvas=hidden")
    # A filled arc, as ellipse
    Fg(&window, "vivid orange")
    FillArc(&window, 10, 10, 40, 20)

    # A partial arc, half ellipse
    Fg(&window, "blue")
    FillArc(&window, 26, 15, 10, 10, 0.0, &pi)

    # save image for the document
    WSync()
    WriteImage("../images/FillArc.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s FillArc.icn -x
```

7.1.91 FillCircle

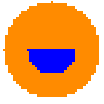
`FillCircle(w, x, y, radius, a1:0.0, a2:&pi*2, ...)` : *window* [*graphics*]

`FillCircle(w, x, y, r, a1, a2)` fills a circle centred at *x,y*. Otherwise similar to *FillArc* with width equal to height.

```
#
# FillCircle.icn, demonstrate drawing a filled circle
#
procedure main()
    w := open("FillCircle", "g", "size=40,40", "canvas=hidden")

    # A full circle, filled
    Fg(w, "vivid orange")
    FillCircle(w, 20, 20, 18)

    # A partial circle, filled
    Fg(w, "blue")
    FillCircle(w, 20, 20, 9, 0.0, &pi)

    # save image for the document
    WSync(w)
    WriteImage(w, "../images/FillCircle.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s FillCircle.icn -x
```



7.1.92 FillPolygon

`FillPolygon(w, x1,y1[,z1], ..., xn,yn[,zn])` : *window* | *list*

`FillPolygon(w, x1,y1, x2,y2, xn,yn)` draws a filled polygon connecting each *x,y* pair (in 2D). In 3D, `FillPolygon()` takes from 2 to 4 coordinates per vertex and returns the list that represents the polygon on the display list.

```
#
# FillPolygon.icn, demonstrate drawing filled polygons
#
procedure main()
    &window := open("FillPolygon", "g", "size=45,30", "canvas=hidden")

    # a polygon using the procedural apply operator
    Fg("vivid orange")
    points := [5,5, 17,17, 5,17, 5,5]
    FillPolygon!points

    # save image for the document
```

```
    WSync()
    WriteImage("../images/FillPolygon.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s FillPolygon.icn -x
```

7.1.93 FillRectangle

`FillRectangle(w, x1,y1, wid1, h1, ...)` : *window*

`FillRectangle(w, x, y, width, hieght)` fills a rectangle with a top right corner of *x,y* and a perceived width and height. Actual rectangle is *width+1* pixels wide, and *height+1* pixels high.

```
#
# FillRectangle.icn, demonstrate drawing filled rectangles
#
procedure main()
    &window := open("FillRectangle", "g", "size=45,40", "canvas=hidden")

    Fg("vivid orange")
    FillRectangle(10,10, 20, 10)

    # save image for the document
    WSync()
    WriteImage("../images/FillRectangle.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s FillRectangle.icn -x
```

7.1.94 find

`find(s, s:&subject, i:&pos, i:0)` : *integer**

The string scanning function `find(s1, s2, i1, i2)` generates the positions where *s1* occurs within *s2[i1:i2]*

```
#
# find.icn, demonstrate the find string scanning position generator
#
procedure main()
    if (p := find("xyz", "abc xyz def ghi xyz", 1, 0)) > 5 then write(p)

    write()
    s := "The rain in Spain stays mainly in the plain"
    write("Subject: ", s)
    write("substring: in")
    s ? every write(find("in"))
end
```

Sample run:

```
prompt$ unicon -s find.icn -x
17

Subject: The rain in Spain stays mainly in the plain
substring: in
7
10
16
27
32
42
```

7.1.95 flock

`flock(f, s)` : *?*

`flock(f, s)` applies an advisory lock to file *f*, given options *s*. Advisory locks can be shared or exclusive, but do not enforce exclusive access. The options can be

- “s”, shared lock
- “x”, exclusive lock
- “b”, don’t block when locking
- “u”, unlock

`flock()` cannot be used with window, directory or database files. See *flock(2)* for more information.

```
#
# flock.icn, demonstrate the POSIX flock advisory file locking function
#
procedure main()
    fn := "newfile.txt"
    f := open(fn, "r")
    write("requesting lock for ", fn)
    if flock(f, "xb") then
        write("got exclusive lock on ", fn)
    else
        write("lock attempt on ", fn, " failed")
    flock(f, "u")
```

```
    close(f)
end
```

Sample run:

```
prompt$ unicon -s flock.icn -x
requesting lock for newfile.txt
got exclusive lock on newfile.txt
```

7.1.96 flush

`flush(f)` : *file*

`flush(f)` flushes all pending or buffered output to file *f*.

This will be hard to demonstrate in a document, but flushing output buffers can help with keeping full lines of output in synch on screen when threading, mixing I/O models or other times when buffering would get in the way of application goals.

```
#
# flush.icn, demonstrate flush of buffer output streams
#
procedure main()
    writes("flushing")
    flush(&output)
end
```

Sample run:

```
prompt$ unicon -s flush.icn -x
flushing
```

7.1.97 Font

`Font(w, s)` : *string*

`Font(w)` produces the name of the current font for window *w*.

`Font(w, s)` sets the font to *s* for window *w* and produces the name, or fails if the font name is invalid.

Unicon ships with four portable fonts:

- sans, proportional font without *serif*s
- serif, proportional font with serifs
- mono (or fixed), mono spaced font without serifs
- typewriter, mono spaced font with serifs

Most other font names are system-dependent, and follow the format `family[,styles],size`. Styles can optionally add `bold` and/or `italic`. `Font()` fails if the requested font name does not exist.

```
#
# Font.icn, demonstrate Font setting and retrieval
#
procedure main()
    w := open("Font", "g", "size=666,160",
              "linestyle=solid", "canvas=hidden")
    write(w, "Default font: " || Font(w))

    # the Unicon portable fonts
    every font := "sans" | "serif" | "mono" | "typewriter" do {
        Font(w, font)
        write(w, font || ": " || &letters || &digits)
    }
    write(w)
    # Some specific GNU/Linux X11 fonts
    every font := "courier" | "r14" | "rk16" do {
        Font(w, font)
        write(w, font || ": " || &letters || &digits)
    }

    WSync(w)
    WriteImage(w, "../images/Font.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s Font.icn -x
```

Default font: fixed
sans: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
serif: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
mono: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
typewriter: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789

courier: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
r14: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
rk16: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789

7.1.98 fork

`fork()` : *integer*

`fork()` creates a new process that is effectively identical to the current process except in the return value from `fork` (and some small amount of operating system “paperwork”). The current process will receive the PID (see *getpid*) of the child process. The forked process will receive a `0`. Negative return values denote an error, a failure when the operating system attempted to create a new process.

```
#
# fork.icn, demonstrate process forking
#
procedure main()
    local sharedvar := 42
```

```
    # parent process
    i := fork() | stop("fork() fail")
    if i < 0 then stop("fork fail with ", i)

    # at this point, two nearly identical processes are running
    # each will have a unique pid and "i" result
    # the variable "sharedvar" will be a copy. Both processes will
    # see it as 42 at first.  Child divides to 21, parent doubles to 84
    if i = 0 then {
        write("child process: ", right(i, 6), ", ", sharedvar)
        sharedvar /:= 2
    } else {
        write("parent process: ", right(i, 5), ", ", sharedvar)
        sharedvar *:= 2
    }
    # both processes will display this line and then end
    write("sharedvar for ", if i = 0 then "child " else "parent",
          " process is now: ", sharedvar)
end
```

Sample run:

```
prompt$ unicon -s fork.icn -x
parent process: 15262, 42
sharedvar for parent process is now: 84
child process:      0, 42
sharedvar for child  process is now: 21
```

7.1.99 FreeColor

`FreeColor(w, s,...)` : *window*

`FreeColor(w, s)` release colour *s* from the window system color map entries. If a colour is still in use when it is freed, unpredictable results will occur.

This is not deprecated, but is unnecessary with modern display technology. *NewColor* and `FreeColor` were initially developed at a time when display screens and graphics cards could only handle a limited palette of colours at any one time. A feature not required with monitors and cards that can handle millions of simultaneous colours.

```
#
# FreeColor.icn, demonstrate freeing created colours
#
procedure main()
    window := open("FreeColor", "g", "size=110,24", "canvas=hidden")
    if c := NewColor(window, "40000,50000,60000") then {
        Fg(window, c)
        write(window, Fg(window))
    }
    else {
        Fg(window, "black")
        write(window, "no mutable colours")
    }

    WSync(window)
    WriteImage(window, "../images/FreeColor.png")
```

```
    FreeColor(\c)
    close(window)
end
```

Sample run:

```
prompt$ unicon -s FreeColor.icn -x
```

```
no mutable colours
```

7.1.100 FreeSpace

`FreeSpace(A)` : *null* [*MS-DOS*]

This is an outdated MS-DOS specific feature of Unicon.

`FreeSpace(A)` frees an allocated memory block returned from *GetSpace*.

```
#
# FreeSpace.icn, an MS-DOS specific free a memory region
#
# This sample is UNTESTED, which means it counts as broken.
#
procedure main()
    s := "Hello, distant past"
    mem := GetSpace(*s)
    Poke(mem, s)
    from := Peek(mem, *s)
    write(image(from))
    # free the memory region
    FreeSpace(mem)
end
```

Sample run (skipped on this GNU/Linux build machine):

See also:

GetSpace, *Peek*, *Poke*, *Int86*

7.1.101 function

`function()` : *string**

`function()` generates the names of the built-in functions.

```
#
# function.icn, demontrate the function() list of built-ins generator
#
link wrap
procedure main()
    write("Functions built into ", &version)
    write()
```

```
    wrap()
    every write(wrap(function() || ", ", 72))
    write(wrap()[1:-2])
end
```

Sample run:

```
prompt$ unicon -s function.icn -x
Functions built into Unicon Version 13.1.  August 19, 2019

Abort, Active, Alert, Any, Arb, Arbno, Attrib, Bal, Bg, Break, Breakx,
Clip, Clone, Color, ColorValue, CopyArea, Couple, DrawArc, DrawCircle,
DrawCube, DrawCurve, DrawCylinder, DrawDisk, DrawImage, DrawLine,
DrawPoint, DrawPolygon, DrawRectangle, DrawSegment, DrawSphere,
DrawString, DrawTorus, EraseArea, EvGet, EvSend, Event, Eye, Fail,
Fence, Fg, FillArc, FillCircle, FillPolygon, FillRectangle, Font,
FreeColor, GotoRC, GotoXY, IdentityMatrix, Len, Lower, MatrixMode,
MultMatrix, NewColor, Normals, NotAny, Nspan, PaletteChars,
PaletteColor, PaletteKey, Pattern, Pending, Pixel, PlayAudio,
PopMatrix, Pos, PushMatrix, PushRotate, PushScale, PushTranslate,
QueryPointer, Raise, ReadImage, Refresh, Rem, Rotate, Rpos, Rtab,
Scale, Span, StopAudio, Succeed, Tab, Texcoord, TextWidth, Texture,
Translate, Uncouple, VAttrib, WAttrib, WDefault, WFlush, WSection,
WSync, WinAssociate, WinButton, WinColorDialog, WinEditRegion,
WinFontDialog, WinMenuBar, WinOpenDialog, WinPlayMedia, WinSaveDialog,
WinScrollBar, WinSelectDialog, WindowContents, WriteImage, abs, acos,
any, args, array, asin, atan, atanh, bal, callout, center, char, chdir,
chmod, chown, chroot, classname, close, cofail, collect, condvar,
constructor, copy, cos, crypt, cset, ctime, dbcolumns, dbdriver,
dbkeys, dblimits, dbproduct, dbtables, delay, delete, detab, display,
dtor, entab, errorclear, eventmask, exec, exit, exp, fcntl, fdup,
fetch, fieldnames, filepair, find, flock, flush, fork, function, get,
getch, getche, getegid, getenv, geteuid, getgid, getgr, gethost,
getpgrp, getpid, getppid, getpw, getrusage, getserv, gettimeofday,
getuid, globalnames, gtime, hardlink, iand, icom, image, insert,
integer, ioctl, ior, ishift, istate, ixor, kbhit, key, keyword, kill,
left, list, load, loadfunc, localnames, lock, log, lstat, many, map,
match, max, member, membernames, methodnames, methods, min, mkdir,
move, mutex, name, numeric, open, oprec, ord, paramnames, parent,
pattern_alternate, pattern_assign_immediate, pattern_assign_onmatch,
pattern_boolfunccall, pattern_boolmethodcall, pattern_concat,
pattern_match, pattern_setcur, pattern_stringfunccall,
pattern_stringmethodcall, pattern_unevalvar, pindex_image, pipe, pop,
pos, proc, pull, push, put, read, readlink, reads, ready, real,
receive, remove, rename, repl, reverse, right, rmdir, rtod, runerr,
seek, select, send, seq, serial, set, setenv, setgid, setgrent,
sethostent, setpgrp, setpwent, setservent, setuid, signal, sin, sort,
sortf, spawn, sql, sqrt, stat, staticnames, stop, string, structure,
symlink, sys_errstr, system, syswrite, tab, table, tan, trap, trim,
truncate, trylock, type, umask, unlock, upto, utime, variable, wait,
where, write, writes
```

7.1.102 get

`get(L, i:1)` : *any?*

`get(L)` returns an element, which is removed from the head of the queue *L*. `get(L, i)` removes the first *i* elements, returning the last one removed.

```
#
# get.icn, demonstrate the queue get from head function
#
link lists
procedure main()
    L := []
    put(L, 1, 2, 3)
    item := get(L)
    write(item)
    write(limage(L))
end
```

Sample run:

```
prompt$ unicon -s get.icn -x
1
[2,3]
```

7.1.103 getch

`getch()` : *string?*

`getch()` waits (if necessary) for a character types at the console keyboard, *even if standard input is redirected*. The character is not echoed.

Note: On GNU/Linux getch doesn't wait when stdin redirected

```
#
# getch.icn, wait for a keystroke
#
procedure main()
    writes("read: ")
    line := read()
    write(line)
    writes("getch: ")
    ch := getch()
    write()
    write("line from &input: ", line)
    write("character from getch(): ", image(ch))
end
```

A sample run (captured outside document generation in this case)

```
prompt$ unicon -s getch.icn -x
read: stdin data
stdin data
getch:
line from &input: stdin data
character from getch(): "w"
```

7.1.104 getche

`getche()` : *type*

`getche()` waits (if necessary) for a character types at the console keyboard, *even if standard input is redirected*. The character is echoed.

Note: On GNU/Linux getche doesn't wait when redirected, fails if no keypresses are pending

```
#
# getche.icn, wait for a keystroke, with echo
#
procedure main()
    writes("read: ")
    line := read()
    write(line)
    writes("getche: ")
    ch := getche()
    write()
    write("line from &input: ", line)
    write("character from getche(): ", image(ch))
end
```

A sample run (captured outside document generation in this case)

```
read: typed from stdin
typed from stdin
getche: g
line from &input: typed from stdin
character from getche(): "g"
```

7.1.105 getegid

`getegid()` : *string*

`getegid()` produces the effective group identity of the current process. A name is returned if available, otherwise the numeric code is returned.

```
#
# getegid.icn, display the effective group identity
#
procedure main()
    write(getegid())
end
```

Sample run:

```
prompt$ unicon -s getegid.icn -x
btiffin
```

7.1.106 getenv

`getenv(s)` : *string*

`getenv(s)` retrieves the value of the environment variable *s* from the current process space.

```
#
# getenv.icn, demonstrate retrieving an environment variable
#
procedure main()
    write(getenv("SHELL"))
end
```

Sample run:

```
prompt$ unicon -s getenv.icn -x
/bin/bash
```

See also:

setenv

7.1.107 geteuid

`geteuid()` : *string*

`geteuid()` returns the effective user identity for the current process. A name is returned if available, otherwise the numeric code is returned.

```
#
# geteuid.icn, display the effective user identity
#
procedure main()
    write(geteuid())
end
```

Sample run:

```
prompt$ unicon -s geteuid.icn -x
btiffin
```

7.1.108 getgid

`getgid()` : *string*

`getgid()` produces the current group identity for the current process. A name is returned if available, otherwise a numeric code is produced.

```
#
# getgid.icn, display the current group identity
#
procedure main()
    write(getgid())
end
```

Sample run:

```
prompt$ unicon -s getgid.icn -x
btiffin
```

7.1.109 getgr

`getgr(g)` : *record*

`getgr(g)` produces a record the contains group file information for group *g*, a string group name of the integer group code. If *g* is null, each successive call to `getgr()` returns the next entry. As in *read* this value is given with *return* and is not a generator *suspend*.

Record is `record posix_group(name, passwd, gid, members)`

```
#
# getgr.icn, display the group file information
#
link fullimag, ximage

procedure main()
    limiter := 0
    #write(ximage(getgr("nobody")))
    while (limiter +:= 1) <= 10 & write(fullimage(getgr()))
end
```

Sample run:

```
prompt$ unicon -s getgr.icn -x
posix_group("root","x",0,"")
posix_group("daemon","x",1,"")
posix_group("bin","x",2,"")
posix_group("sys","x",3,"")
posix_group("adm","x",4,"syslog,btiffin")
posix_group("tty","x",5,"")
posix_group("disk","x",6,"")
posix_group("lp","x",7,"")
posix_group("mail","x",8,"")
posix_group("news","x",9,"")
```

7.1.110 gethost

`gethost(x)` : *record|string*

`gethost(n)` for network connection *n* produces a string containing the IP address and port number this machine is using for the connection. `gethost(s)` returns a record that contains the host information for the name *s*. If *s* is null, each successive call to `gethost()` returns the next entry. *sethostent* resets the sequence to the beginning. Aliases and addressed are comma separated lists (in `a.b.c.d` format).

The record type returned is `record posix_hostent(name, aliases, addresses)`.

As in *read* this value is given with *return* and is not a generator *suspend*.

```
#
# gethost.icn, display host entities.
#
link fullimag, ximage

procedure main()
    limiter := 0
    write(ximage(gethost("localhost")))
    while (limiter +:= 1) <= 10 & write(fullimage(gethost()))
end
```

Sample run:

```
prompt$ unicon -s gethost.icn -x
R_posix_hostent_1 := posix_hostent()
   R_posix_hostent_1.name := "localhost"
   R_posix_hostent_1.aliases := "localhost"
   R_posix_hostent_1.addresses := "127.0.0.1"
posix_hostent("localhost","","127.0.0.1")
posix_hostent("btiffin-CM1745","","127.0.1.1")
posix_hostent("ip6-localhost","ip6-loopback","127.0.0.1")
```

7.1.111 getpgrp

`getpgrp()` : *integer*

`getpgrp()` produces the process group of the current process.

```
#
# getpgrp.icn, display the process group
#
procedure main()
    write(getpgrp())
end
```

Sample run:

```
prompt$ unicon -s getpgrp.icn -x
13976
```

7.1.112 getpid

`getpid()` : *integer*

`getpid()` produces the process identification (pid) of the current process.

```
#
# getpid.icn, display the current process identifier
#
procedure main()
    write(getpid())
end
```

Sample run:

```
prompt$ unicon -s getpid.icn -x
15380
```

7.1.113 getppid

`getppid()` : *integer?*

`getppid()` produces the process id of the parent of the current process.

```
#
# getppid.icn, display the parent process identification
#
procedure main()
    write(getppid())
end
```

Sample run:

```
prompt$ unicon -s getppid.icn -x
15390
```

7.1.114 getpw

`getpw(u)` : *record*

`getpw(u)` produces a record that contains account password file information. *u* can be a numeric uid or a user name. If *u* is null, each successive call to `getpw()` returns the next entry. As in *read* this value is given with *return* and is not a generator *suspend*. *setpwent* resets the sequence to the beginning.

Record type is:

```
record posix_password(name, passwd, uid, gid, age,
                      comment, gecos, dir, shell)
```

Most systems now contain a special marker `x` for the passwd field and the actual data is saved in a shadow file, safe from prying eyes. Even with access to the shadow data, the password is stored using *crypt*. The encrypted form makes it just that little bit easier to brute force guess the original password so the shadow data file was developed, with more restrictive permissions than the global passwd file that `getpw` accesses.

```
#
# getpw.icn, display passwd records.
#
link fullimag, ximage

procedure main()
    limiter := 0
    write(ximage(getpw("nobody")))
    # not showing this data
    #while write(fullimage(getpw()))
end
```

Sample run:

```
prompt$ unicon -s getpw.icn -x
R_posix_passwd_1 := posix_passwd()
   R_posix_passwd_1.name := "nobody"
   R_posix_passwd_1.passwd := "x"
   R_posix_passwd_1.uid := 65534
   R_posix_passwd_1.gid := 65534
   R_posix_passwd_1.gecos := "nobody"
   R_posix_passwd_1.dir := "/nonexistent"
   R_posix_passwd_1.shell := "/usr/sbin/nologin"
```

7.1.115 getrusage

`getrusage(s)` : *record*

`getrusage(s)` produces resource usage for *s*, where *s* can be "self", "thread" or "children". `getrusage` fails if resource usage cannot be retrieved for *s*.

Record type is:

```
record posix_rusage(utime, stime, maxrss, minflt, majflt,
                    inblock, oublock, nvcsw, nivcsw)
```

The `utime` and `stime` fields are `record posix_timeval(sec, usec).`

From *getruage(2)*:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};                           /**/
```

```
#
# getrusage.icn, retrieve resource usage.
#
link fullimag, ximage

procedure main()
    every who := "self" | "thread" | "children" | "random" do {
        write(who)
        write(ximage(getrusage(who)))
```

```
        write(fullimage(getrusage(who)))
        write()
    }
end
```

Sample run:

```
prompt$ unicon -s getrusage.icn -x
self
R_posix_rusage_1 := posix_rusage()
   R_posix_rusage_1.utime := R_posix_timeval_1 := posix_timeval()
      R_posix_timeval_1.sec := 0
      R_posix_timeval_1.usec := 4000
   R_posix_rusage_1.stime := R_posix_timeval_2 := posix_timeval()
      R_posix_timeval_2.sec := 0
      R_posix_timeval_2.usec := 4000
   R_posix_rusage_1.maxrss := 6452
   R_posix_rusage_1.minflt := 571
   R_posix_rusage_1.majflt := 0
   R_posix_rusage_1.inblock := 0
   R_posix_rusage_1.oublock := 56
   R_posix_rusage_1.nvcsw := 2
   R_posix_rusage_1.nivcsw := 4
posix_rusage(posix_timeval(0,4000),posix_timeval(0,4000),6452,576,0,0,56,2,4)

thread

children
R_posix_rusage_3 := posix_rusage()
   R_posix_rusage_3.utime := R_posix_timeval_5 := posix_timeval()
      R_posix_timeval_5.sec := 0
      R_posix_timeval_5.usec := 0
   R_posix_rusage_3.stime := R_posix_timeval_6 := posix_timeval()
      R_posix_timeval_6.sec := 0
      R_posix_timeval_6.usec := 0
   R_posix_rusage_3.maxrss := 2484
   R_posix_rusage_3.minflt := 181
   R_posix_rusage_3.majflt := 0
   R_posix_rusage_3.inblock := 0
   R_posix_rusage_3.oublock := 0
   R_posix_rusage_3.nvcsw := 5
   R_posix_rusage_3.nivcsw := 1
posix_rusage(posix_timeval(0,0),posix_timeval(0,0),2484,181,0,0,0,5,1)

random
```

7.1.116 getserv

`getserv(s, s)` : *record*

`getserv(service, proto)` retrieves the service database entry named *s* using protocol *proto*.

If *s* is null, each successive call to `getserv()` returns the next entry. *setservent* resets the sequence to the beginning.

As in *read* these values are given with *return* and not a generator *suspend*.

The record type returned is `record posix_servent(name, aliases, port, proto)`.

```
#
# getserv.icn, display service information.
#
link fullimag, ximage

procedure main()
    limiter := 0
    write(ximage(getserv("echo", "udp")))
    write(ximage(getserv("echo", "tcp")))
    while sr := getserv() do if (limiter +:= 1) < 10 then
        write(fullimage(sr))
    write(limiter, " services in the database")
end
```

Sample run:

```
prompt$ unicon -s getserv.icn -x
R_posix_servent_1 := posix_servent()
   R_posix_servent_1.name := "echo"
   R_posix_servent_1.aliases := ""
   R_posix_servent_1.port := 7
   R_posix_servent_1.proto := "udp"
R_posix_servent_2 := posix_servent()
   R_posix_servent_2.name := "echo"
   R_posix_servent_2.aliases := ""
   R_posix_servent_2.port := 7
   R_posix_servent_2.proto := "tcp"
posix_servent("tcpmux","",1,"tcp")
posix_servent("echo","",7,"tcp")
posix_servent("echo","",7,"udp")
posix_servent("discard","sink,null",9,"tcp")
posix_servent("discard","sink,null",9,"udp")
posix_servent("systat","users",11,"tcp")
posix_servent("daytime","",13,"tcp")
posix_servent("daytime","",13,"udp")
posix_servent("netstat","",15,"tcp")
557 services in the database
```

For GNU/Linux the services data set is usually a plain text file, `/etc/services`.

7.1.117 GetSpace

`GetSpace(i)` : *A* [*MSDOS*]

This is an outdated MS-DOS specific feature of Unicon.

`GetSpace(i)` retrieves *i* bytes of memory outside normal Unicon control and garbage collection. The return value is an "address", effectively a long integer.

```
#
# GetSpace.icn, an MS-DOS specific memory region allocator
#
# This sample is UNTESTED, which means it counts as broken.
#
```

```
procedure main()
    s := "Hello, distant past"
    # get some memory, the size of s
    mem := GetSpace(*s)
    Poke(mem, s)
    from := Peek(mem, *s)
    write(image(from))
    FreeSpace(mem)
end
```

Sample run (skipped on this GNU/Linux build machine):

See also:

FreeSpace, *Peek*, *Poke*, *Int86*

7.1.118 gettimeofday

`gettimeofday()` : *record*

`gettimeofday()` returns the current time in seconds and microseconds since the epoch, January 1st, 1970 at 00:00:00, Greenwich Mean Time. The `sec` field may be converted to a date string with the *ctime* or *gtime* functions.

Returns `record posix_timeval(sec, usec).`

```
#
# gettimeofday.icn, retrieve the second and microsecond clock values
#
procedure main()
    R := gettimeofday()
    write(R.sec," seconds since the Unix epoch")
    write(R.usec, " microsecond counter")
    write(ctime(R.sec))
end
```

Sample run:

```
prompt$ unicon -s gettimeofday.icn -x
1572166384 seconds since the Unix epoch
413976 microsecond counter
Sun Oct 27 04:53:04 2019
```

See also:

&clock, *&dateline*, *&now*, *gtime*, *ctime*

7.1.119 getuid

`getuid()` : *string* [*POSIX*]

`getuid()` produces the real user identity of the current process.

```
#
# getuid.icn, display the current user identity
#
procedure main()
    write(getuid())
end
```

Sample run:

```
prompt$ unicon -s getuid.icn -x
btiffin
```

7.1.120 globalnames

`globalnames(CE)` : *string**

`globalnames(ce)` generates the names of the global variable in the program that create the co-expression *ce*.

```
#
# globalnames.icn, generate globalnames
#
global var, other
procedure main()
    local lv
    static sv
    lv := 1
    sv := 1
    var := 1
    every write(globalnames(&main))
end
```

Sample run:

```
prompt$ unicon -s globalnames.icn -x
main
var
write
globalnames
```

Note how `other` isn't listed. It would need to be set to show up.

See also:

localnames, *staticnames*

7.1.121 GotoRC

`GotoRC(w, row:1, col:1)` : *window* [*Graphics*]

`GotoRC(w, r, c)` moves the text cursor to row *r*, column *c*, on window *w*. Row and column are given in character positions. The upper left is `(1,1)`. The column is calculated using the pixel width of the widest character in the current font. Works best with fixed width fonts. Row is determined by the height of the current font.

```
#
# GotoRC.icn, set graphic text cursor to row and column
#
procedure main()
    w := open("GotoRC", "g", "size=130,60", "canvas=hidden")
    text := "Initial message"
    write(w, text)

    text := "Placed message"
    GotoRC(w, 3,5)
    Fg(w, "purple")
    write(w, text)

    WSync(w)
    WriteImage(w, "../images/GotoRC.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s GotoRC.icn -x
```

Initial message

Placed message

7.1.122 GotoXY

`GotoXY(w:&window, x:0, y:0)` : *window* [*Graphics*]

`GotoXY(w, x, y)` moves the text cursor to a specific cursor location, *x, y* (given in pixels) on window *w*.

```
#
# GotoXY.icn, set graphic text cursor to x,y
#
procedure main()
    w := open("GotoXY", "g", "size=130,60", "canvas=hidden")
    text := "Initial message"
    write(w, text)

    text := "Placed message"
    GotoXY(w, 10,40)
    Fg(w, "purple")
    write(w, text)

    WSync(w)
    WriteImage(w, "../images/GotoXY.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s GotoXY.icn -x
```

Initial message

Placed message

7.1.123 gtime

`gtime(i)` : *string*

`gtime(i)` converts the integer time `i`, given in seconds since the epoch of Jan 1st, 1970 00:00:00 Greenwich Mean Time, into a string, based on GMT.

```
#
# gtime.icn, Demonstrate the gtime function
#
procedure main()
    # convert epoch start to a formatted time
    write(gtime(0))

    # convert time of run to formatted time
    write(gtime(&now), " Greenwich Mean Time")

    # two days in the future (relative to time of run)
    write(gtime(&now + 48 * 60 * 60))
end
```

Sample run:

```
Thu Jan  1 00:00:00 1970
Sun Oct 27 08:53:05 2019 Greenwich Mean Time
Tue Oct 29 08:53:05 2019
```

See also:

&clock, *ctime*, *&dateline*, *&now*

7.1.124 hardlink

`hardlink(s, s)` : *?* [*POSIX*]

`hardlink(src, dst)` creates a hard link on the file system, *dst* becomes a directory entry referencing the same file system i-node as *src*. This creates a new name for *src*, and changes to *dst* will effect *src*.

```
#
# hardlink.icn, demonstrate the POSIX hard link() function
#
procedure main()
    hardlink("tt.src", "tt.link")
end
```

Sample run:

```
prompt$ unicon -s hardlink.icn -x
```

With that link, if `tt.src` started out containing:

```
tt.src file data
```

A change to `tt.dst`, say with:

```
prompt$ echo 'tt.dst file data'
```

The disk data blocks in `tt.src` have been changed:

```
prompt$ cat tt.src
tt.dst file data
```

See *symlink* for the more frequently used soft link.

7.1.125 iand

`iand(i, i)` : *integer*

`iand(i1, i2)` produces the bitwise AND of *i1* and *i2*.

```
#
# iand.icn, bitwise AND of two integers
#
link printf
procedure main()
    i1 := 7
    i2 := 3
    write(printf("iand(%d, %d) = %d", i1, i2, iand(i1, i2)))

    # create some large integers
    i1 := 2^66 + 7
    i2 := 2^67 + 2^66 + 3
    write(printf("iand(%d, %d) = %d", i1, i2, iand(i1, i2)))
end
```

Sample run:

```
prompt$ unicon -s iand.icn -x
iand(7, 3) = 3
iand(73786976294838206471, 221360928884514619395) = 73786976294838206467
```

7.1.126 icom

`icom(i)` : *integer*

`icom(i)` produces the bitwise one's complement of *i*.

```
#
# icom.icn, bitwise one's complements of integer
#
link printf
procedure main()
    i := 7
    write(printf("icom(%d) = %d", i, icom(i)))

    # create a large integers
    i := 2^66 + 7
    write(printf("icom(%d) = %d", i, icom(i)))
end
```

Sample run:

```
prompt$ unicon -s icom.icn -x
icom(7) = -8
icom(73786976294838206471) = -73786976294838206472
```

7.1.127 IdentityMatrix

IdentityMatrix(*window*) → record

Type *pattern*

Requires *3D graphics*

Replaces the current 3D graphic matrix to the identity matrix. Returns the display list element.

```
#
# IdentityMatrix.icn, demonstrate matrix transforms
# Curerntly a work in progress
#
link fullimag
procedure main()
    window := open("IdentityMatrix", "gl", "bg=black", "buffer=on",
                    "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)
    Refresh(window)
    MatrixMode(window, "projection")
    PushMatrix(window)

    write(image(IdentityMatrix(window)))
    write(fullimage(PopMatrix(window)))

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/IdentityMatrix.png")
    close(window)
end
```

Sample run (not auto captured):

```
prompt$ unicon -s IdentityMatrix.icn -x
gl_pushmatrix("PopMatrix",224)
```

See also:

PushMatrix, *PopMatrix*, *Eye*

7.1.128 image

`image(x)` : *string*

`image(x)` produces a string image of *x*.

```
#
# image.icn, demonstrate string image function
#
link ximage, fullimag
procedure main()
    s := "abc"
    cs := 'cba'
    L := [1,2,3]

    write("s is ", s, " image is ", image(s))
    write("cs is ", cs, " image is ", image(cs))
    write("list L image is ", image(L))
    write("fullimage(L) is ", fullimage(L))
    write("ximage(L) is ", ximage(L))
end
```

Sample run:

```
prompt$ unicon -s image.icn -x
s is abc image is "abc"
cs is abc image is 'abc'
list L image is list_1(3)
fullimage(L) is [1,2,3]
ximage(L) is L1 := list(3)
   L1[1] := 1
   L1[2] := 2
   L1[3] := 3
```

7.1.129 InPort

`InPort(i)` : *integer* [*MS-DOS*]

`InPort(i)` will read a byte value from port *i*. This is an MS-DOS specific feature of Unicon.

```
#
# InPort.icn, read a value from an MS-DOS port
#
procedure main()
    write(InPort(1))
end
```

Sample run (skipped on this GNU/Linux build machine):

See also:

OutPort

7.1.130 insert

`insert(x1, x2, x3:&null, ...)` : *x1*

`insert(x1, x2, x3)` inserts element *x2* into *List (arrays)*, *Set*, *Table* or DBM database *x1*, if it is not already a *member*.

For lists, tables and databases the assigned value for *x2* is *x3*. For lists, *x2* is an integer index, and for other types *x2* is a key. For sets, *x3* is taken as another element to insert. `insert()` always succeeds and returns *x1*. In Unicon, multiple elements can be inserted in one call.

```
#
# insert.icn, demonstrate element insert function
#
link fullimag
procedure main()
    L := list()
    S := set()
    T := table()
    k := 1
    v := 2

    insert(L, k, v, 3, 4)
    insert(S, k, v, 3, 4)
    insert(T, k, v, 3, 4)

    write(fullimage(L))
    write(fullimage(S))
    write(fullimage(T))
end
```

Sample run:

```
prompt$ unicon -s insert.icn -x
[2]
set(1,2,3,4)
table(1->2,3->4)
```

7.1.131 Int86

`Int86(list)` : *list* [*MS-DOS*]

This is an MS-DOS specific Unicon feature.

`Int86(L)` performs an MS-DOS interrupt routine. The list *L* must contain *nine* integer values. The first value is a *flag* value, the rest will be stored in *ax,bx,cx,dx,si,di,es,ds* CPU registers, in that order. After the dispatch to the interrupt vector in *flag*, via the `int386x` C library function, the returned list will be set with the *cflag* and register values as set by DOS.

Attention: This feature is specific to MS-DOS on Intel chips and is relatively dangerous. A programmer must know what are safe values for the registers before using this, now mostly outdated, feature.

```
#
# Int86.icn, perform an MS-DOS interrupt 21h support routine.
#
# This sample is UNTESTED, and the feature itself is outdated.
#
procedure main()
    # Get disk transfer address
    flag := 16r21
    ax := 16r2f00
    result := Int86([flag, ax] ||| list(7, 0))
    # filename location is 16 * es + cx (result[8], [3]) + 30
    # further access will require use of GetSpace, Peek and Poke
end
```

Sample run (skipped on this GNU/Linux build machine):

See `ipl/procs/io.icn` for an old working example of `Int86()`.

See also:

GetSpace, *Peek*, *Poke*

7.1.132 integer

`integer(any)` : *integer?*

`integer(x)` converts the value *x* to an integer, or fails if the conversion cannot be performed.

```
#
# integer.icn, convert to integer
#
procedure main()
    every x := 123 | "123" | "abc" | 123.123 do
        write("integer(", image(x), ") is ", integer(x) | "not converted")
end
```

Sample run:

```
prompt$ unicon -s integer.icn -x
integer(123) is 123
integer("123") is 123
integer("abc") is not converted
integer(123.123) is 123
```

7.1.133 ioctl

`ioctl(f, i, s)` : *integer*

`ioctl(f, i, s)` passes the options in *s* to the open special device file channel *f*, given an integer action specified in *i*.

```
#
# ioctl.icn, special device, driver contol function
#
# ** work in progress **
procedure main()
    action := 12345
    options := "option block"
    f := open("/dev/null", "r")
    ioctl(f, action, options)
    close(f)
end
```

Todo

ioctl demo

Sample run (pending):

7.1.134 ior

`ior(i, i)` : *integer*

`ior(i1, i2)` produces the bitwise OR of *i1* and *i2*.

```
#
# ior.icn, bitwise OR of two integers
#
link printf
procedure main()
    i1 := 7
    i2 := 3
    write(printf("ior(%d, %d) = %d", i1, i2, ior(i1, i2)))

    # create some large integers
    i1 := 2^66 + 7
    i2 := 2^67 + 2^66 + 3
    write(printf("ior(%d, %d) = %d", i1, i2, ior(i1, i2)))
end
```

Sample run:

```
prompt$ unicon -s ior.icn -x
ior(7, 3) = 7
ior(73786976294838206471, 221360928884514619395) = 221360928884514619399
```

7.1.135 ishift

`ishift(i, i)` : *integer*

`ishift(i, j)` produces the value from shifting *i* by *j* bit positions. Shift left for positive *j* and shift right for negative *j*. Zero bits are shifted in.

```
#
# ishift.icn, bitwise shift
#
link printf
procedure main()
    shift := 3
    every i := 7 | -7 do {
        # shift right, low bit zero fill
        write(printf("ishift(%d, %d) = %d", i, shift, ishift(i, shift)))
    }

    # shift left, high bit zero fill
    shift := -3
    every i := 7 | -7 do {
        # shift right, low bit zero fill
        write(printf("ishift(%d, %d) = %d", i, shift, ishift(i, shift)))
    }

    # using large integer
    i := 2^66 + 7
    write(printf("ishift(%d, %d) = %d", i, shift, ishift(i, shift)))
end
```

Sample run:

```
prompt$ unicon -s ishift.icn -x
ishift(7, 3) = 56
ishift(-7, 3) = -56
ishift(7, -3) = 0
ishift(-7, -3) = -1
ishift(73786976294838206471, -3) = 9223372036854775808
```

7.1.136 istate

`istate(CE, s)` : *integer*

`istate(ce, attrib)` reports selected virtual machine interpreter state information for *ce*. Used by monitors. *attrib* must be one of:

- "count"
- "ilevel"
- "ipc"
- "ipc_offset"
- "sp"
- "efp"
- "gfp"

Todo

what do the attribute fields actually mean

```
#
# istate.icn, report selected vm interpreter state information
#
procedure main()
    every attr := "count" | "ilevel" | "ipc" | "ipc_offset" |
                  "sp" | "efp" | "gfp" do {
        write("attribute ", attr, " = ", istate(&main, attr))
    }
end
```

Sample run:

```
prompt$ unicon -s istate.icn -x
attribute count = 0
attribute ilevel = 1
attribute ipc = 29794496
attribute ipc_offset = 640
attribute sp = 140501955478448
attribute efp = 140501955478312
attribute gfp = 0
```

7.1.137 ixor

`ixor(i, i)` : *integer*

`ixor()` produces the bitwise exclusive OR (XOR) of *i1* and *i2*.

```
#
# ixor.icn, bitwise XOR of two integers
#
link printf
procedure main()
    i1 := 7
    i2 := 3
    write(printf("ixor(%d, %d) = %d", i1, i2, ixor(i1, i2)))

    # create some large integers
    i1 := 2^66 + 7
    i2 := 2^67 + 2^66 + 3
    write(printf("ixor(%d, %d) = %d", i1, i2, ixor(i1, i2)))
end
```

Sample run:

```
prompt$ unicon -s ixor.icn -x
ixor(7, 3) = 4
ixor(73786976294838206471, 221360928884514619395) = 147573952589676412932
```

7.1.138 kbhit

`kbhit()` : *?*

`kbhit()` checks to see if there is a keyboard character waiting to be read. Returns null or fails when no events are pending.

```
#
# kbhit.icn, see if there is a keyboard event pending
#
procedure main()
    # eat any stray characters
    while kbhit() do getch()

    delay(1000)
    write()

    if kbhit() then write("key press pending:", image(getch()))
    else write("no keys pressed during delay")
end
```

Sample run (captured outside documentation generation)

```
prompt$ unicon -s kbhit.icn -x
a
key press pending:"a"
```

The echo of `a` (tapped during the delay) was from the GNU/Linux console, `stty` set to echo.

7.1.139 key

`key(x)` : *any**

`key(T)` generates the key values from *Table* *T*. `key(L)` generates the indices from `1` to `*L` in *List (arrays)* *L*. `key(R)` generates the string field names of *record* *R*.

```
#
# key.icn, generate the keys of a structure
#
link wrap
record sample(x,y,z)
procedure main()
    L := [1,2,3]
    R := sample(1,2,3)
    T := table()
    insert(T, "a", 1, "b", 2, "c", 3)

    write("keys of list L, record R, table T")
    wrap()
    every f := L | R | T do {
        every writes(wrap(key(f) || ", "))
        write(wrap()[1:-2])
    }
end
```

Sample run:

```
prompt$ unicon -s key.icn -x
keys of list L, record R, table T
```

```
1, 2, 3
x, y, z
c, b, a
```

7.1.140 keyword

`keyword(s, CE:¤t,i:0)` : *any**

`keyword(s, ce, i)` produces the value(s) of keyword *s* in the context of *ce* execution, *i* levels up the stack from the current point of execution. Used in execution monitors

```
#
# keyword.icn, retrieve value(s) of keywords from co-expressions
#
procedure main()
    ce1 := create |write(keyword("&line", &current, 0))
    @ce1
    write(&line)
    @ce1
end
```

Sample run:

```
prompt$ unicon -s keyword.icn -x
12
14
12
```

7.1.141 kill

`kill(i, x)` : *?* [*POSIX*]

`kill(pid, signal)` sends a signal to the process specified by *pid*. The *signal* parameter can be a string name or the integer code for the signal to be sent.

From `src/runtime/rposix.r`

```
stringint signalnames[] = {
   { 0,          40 },
   { "SIGABRT",     SIGABRT },
   { "SIGALRM",         SIGALRM },
   { "SIGBREAK",        SIGBREAK },
   { "SIGBUS",          SIGBUS },
   { "SIGCHLD",         SIGCHLD },
   { "SIGCLD",      SIGCLD },
   { "SIGCONT",         SIGCONT },
   { "SIGEMT",          SIGEMT },
   { "SIGFPE",          SIGFPE },
   { "SIGFREEZE",   SIGFREEZE },
   { "SIGHUP",          SIGHUP },
   { "SIGILL",          SIGILL },
   { "SIGINT",          SIGINT },
```

```
    { "SIGIO",          SIGIO },
    { "SIGIOT",         SIGIOT },
    { "SIGKILL",        SIGKILL },
    { "SIGLOST",        SIGLOST },
    { "SIGLWP",     SIGLWP },
    { "SIGPIPE",        SIGPIPE },
    { "SIGPOLL",    SIGPOLL },
    { "SIGPROF",        SIGPROF },
    { "SIGPWR",         SIGPWR },
    { "SIGQUIT",        SIGQUIT },
    { "SIGSEGV",        SIGSEGV },
    { "SIGSTOP",        SIGSTOP },
    { "SIGSYS",         SIGSYS },
    { "SIGTERM",        SIGTERM },
    { "SIGTHAW",    SIGTHAW },
    { "SIGTRAP",        SIGTRAP },
    { "SIGTSTP",        SIGTSTP },
    { "SIGTTIN",        SIGTTIN },
    { "SIGTTOU",        SIGTTOU },
    { "SIGURG",         SIGURG },
    { "SIGUSR1",        SIGUSR1 },
    { "SIGUSR2",        SIGUSR2 },
    { "SIGVTALRM",      SIGVTALRM },
    { "SIGWAITING", SIGWAITING },
    { "SIGWINCH",       SIGWINCH },
    { "SIGXCPU",        SIGXCPU },
    { "SIGXFSZ",        SIGXFSZ },
};
```

```
#
# kill.icn, send signal to PID
#
procedure main()
    write("send signal to kill current process")
    kill(getpid(), "SIGKILL")
    write("won't get here")
end
```

Sample run (with error termination code, due to kill):

```
prompt$ unicon -s kill.icn -x
send signal to kill current process
Killed
```

7.1.142 left

`left(s, i:1, s:" ")` : *string*

`left(s1, i, s2)` formats *s1* to be a string of length *i*. If *s1* has more than *i* characters, it is truncated. If *s1* has less than *i* characters then it is padded to the right with as many copies of *s2* needed to increase the length to *i*. Last copy of *s2* is left side truncated if necessary ("filler" will pad as "ller" for instance).

```
#
# left.icn, fill out a string to length, truncate or pad fill on right
```

```
#
procedure main()
    s := "abcdefghij"
    write(":", left(s, 5), ":")
    write(":", left(s, 10), ":")
    write(":", left(s, 20), ":")
    write(":", left(s, 20, "filler"), ":")
end
```

Sample run:

```
prompt$ unicon -s left.icn -x
:abcde:
:abcdefghij:
:abcdefghij          :
:abcdefghijllerfiller:
```

7.1.143 Len

`Len(i)` : *string* [*Patterns*]

`Len(n)` is a SNOBOL pattern that matches the next *n* characters, or fails if not enough characters remain in the subject. `Len(0)` matches the empty string.

```
#
# Len.icn, demonstrate Len() SNOBOL pattern matching n characters
#
# Display the first 12 characters of subject
procedure main()
    sub := "IMPORTANT   Message for tag"
    sub ?? Len(12) => tag
    write(":", tag, ": ", *tag)
end
```

Sample run:

```
prompt$ unicon -s Len.icn -x
:IMPORTANT   : 12
```

7.1.144 list

`list(i:0, any:&null)` : *list*

`list(i, x)` creates a list of size *i*, with all initial values set to *x*. If *x* is a mutable value, such as a list, all elements refer to the same value, not a separate copy.

```
#
# list.icn, demonstrate the list creation function
#
link lists
procedure main()
```

```
    L1 := [1,2,3]
    L2 := list(5, L1)
    every Lt := !L2 do write(limage(Lt))
    delete(L1, 2)
    every Lt := !L2 do write(limage(Lt))
end
```

Sample run:

```
prompt$ unicon -s list.icn -x
[1,2,3]
[1,2,3]
[1,2,3]
[1,2,3]
[1,2,3]
[1,3]
[1,3]
[1,3]
[1,3]
[1,3]
```

7.1.145 load

`load(s, L, f:&input, f:&output, f:&errout,i,i,i)` : *co-expression*

`load(s, arglist, input, output, error, block, string, stack)` loads the *icode* file named *s* and returns the program as a co-expression, ready to start in the loaded `main()` procedure with *arglist* as the command line arguments. The three file parameters are used as the *&input*, *&output* and *&errout* for the program co-expression. The three integers are used to set initial memory region sizes for `block`, `string`, and `stack`.

```
#
# load.icn, demonstrate loading icode files into the multitasker
#
procedure main()
    write("Loading ./load-module")
    lp := load("load-module", ["a", "b", "c"])
    write("Evaluating main of load-module")
    @lp
    write("Back in initial program")
end
```

```
#
# load-module.icn, a demonstration module for use with load sample.
#
procedure main(arglist)
    every write(!arglist)
end
```

Sample run:

```
prompt$ unicon -s load-module.icn
```

```
prompt$ unicon -s load.icn -x
Loading ./load-module
Evaluating main of load-module
a
b
c
Back in initial program
```

7.1.146 loadfunc

loadfunc(*libname:string*, *funcname:string*) → procedure

`loadfunc()` reads `libname` to load a C foreign function `funcname` returning a *procedure* value. Uses the *IPL* support file, `icall.h`.

```
#
# loadfunc.icn, demonstrate loadfunc
#
procedure main(arglist)
    ffi := loadfunc("./loaded.so", "loaded")
    write("Type of ffi: ", type(ffi))

    # default arg of 21, half of the ultimate answer
    param := arglist[1] | 21
    write("loaded function mutiply by two, ffi(", param, ") = ",
          ffi(param))
end
```

All loadable functions require a Unicon compatible prototype:

int **cfunction**(int *argc*, *descriptor *argv*)

or, *equivalent*

int **cfunction**(int *argc*, struct *descriptor argv[]*)

That is, a function returning an integer that accepts a count of arguments and a pointer to a special array of structures that hold encoded argument values.

The C function is expected to return an integer, 0 for success, -1 for failure with positive values being error codes.

Arguments are marshalled to and from Icon with a special bit encoded structure with slots for data or pointer to data. Unicon Integer, Real, String and List data can be encoded for passing between C and Unicon.

descriptor

A `descriptor` is an opaque internal data structure.

```
typedef long word;
typedef struct descrip {
   word dword;
   union {
      word integr;              /*   integer value */
#if defined(DescriptorDouble)
      double realval;
#endif
      char *sptr;               /*   pointer to character string */
```

```
        union block *bptr;          /*   pointer to a block */
        struct descrip *descptr;    /*   pointer to a descriptor */
        } vword;
} descriptor, *dptr;                /**/
```

argv[0] is reserved for the actual value delivered to Unicon on return from the external C function.

```
/*
 loaded.c, a Unicon loadable external function
 tectonics: gcc -shared -fpic -o loaded.so loaded.c
 icall.h copied from unicon tree ipl/cfuncs
*/

#include <stdio.h>
#include "icall.h"

int
loaded(int argc, descriptor *argv)
{
#ifdef DEBUG
    /* look at count and the pointer */
    printf("argc: %d, argv: %p\n", argc, argv);
    fflush(stdout);
#endif

    /* Ensure an integer arg 1 */
    if (argc < 1) {
        ArgError(argc, 101);
    }

    ArgInteger(1);

#ifdef DEBUG
    /* show the operational step */
    printf("argv[argc]: %ld\n", IntegerVal(argv[argc]));
#endif

    RetInteger(IntegerVal(argv[argc]) * 2);
}
```

That code exercises some of the helper macros defined in `icall.h`.

void **ArgInteger** (int *index*)

`ArgInteger` ensures the argument at `argv[index]` is an native integer, or fails and returns an errorcode. Marked `void` as `ArgInteger` is actually a code fragment macro, not an actual function.

void **ArgError** (int *index*, int *errorcode*)

`ArgError` returns argv[index] as an offending value, with errorcode. Marked `void` as `ArgError` is actually a code fragment macro, not an actual function.

int **RetInteger** (int *result*)

`RetInteger` returns an integer for use as a Unicon function result.

Sample run:

```
prompt$ gcc -shared -fpic -o loaded.so loaded.c
```

```
prompt$ unicon -s loadfunc.icn -x
Type of ffi: procedure
loaded function mutiply by two, ffi(21) = 42
```

Sample with an erroneous value passed:

```
prompt$ unicon -s loadfunc.icn -x "abc"
Type of ffi: procedure

Run-time error 101
File loadfunc.icn; Line 17
integer expected or out of range
offending value: "abc"
Traceback:
   main(list_1 = ["abc"])
   loaded("abc") from line 17 in loadfunc.icn
```

See *Programs* for some examples of loading scripting engines, like Ruby, S-Lang and Javascript, into Unicon and integration with other languages, like *GnuCOBOL*.

7.1.147 localnames

`localnames(C:co-expression, i:integer:0)` : *string**

`localnames(C, i)` generates the names of local variables in co-expression *C*, *i* levels up from the current procedure invocation. The default, level 0, generates names in the currently active procedure inside *C*.

```
#
# localnames.icn, generate local names
#
global var, other
procedure main(arglist)
    local lv
    static sv
    lv := 1
    sv := 1
    var := 1
    every write(localnames(&main))
end
```

Sample run:

```
prompt$ unicon -s localnames.icn -x
lv
```

See also:

globalnames, *staticnames*

7.1.148 lock

`lock(x)` : *x*

`lock(x)` locks the mutex *x*, or the mutex associated with the thread-safe object *x*.

```
#
# lock.icn, demonstrate mutual exclusive region locking
#
global x

procedure main()
    mx := mutex()
    x := 0
    t1 := thread incrementor(mx)
    t2 := thread incrementor(mx)
    every wait(t1 | t2)
    write("x = ", x)
end

procedure incrementor(m)
    lock(m)
    # a non atomic increment expression; fetch x,  add to x, store x
    x := x + 1
    unlock(m)
end
```

Sample run:

```
prompt$ unicon -s lock.icn -x
x = 2
```

7.1.149 log

`log(r, r:&e)` : *real*

`log(r, b)` generates the logarithm of *r* in base *b*.

```
#
# log.icn, return a logarithm in a given base, default natural logarithm
#
procedure main()
    write("log(10) base &e default: ", log(10))
    write("log(10, 10) ", log(10, 10))
    write("log(100, 10) ", log(100, 10))
end
```

Sample run:

```
prompt$ unicon -s log.icn -x
log(10) base &e default: 2.302585092994046
log(10, 10) 1.0
log(100, 10) 2.0
```

7.1.150 Lower

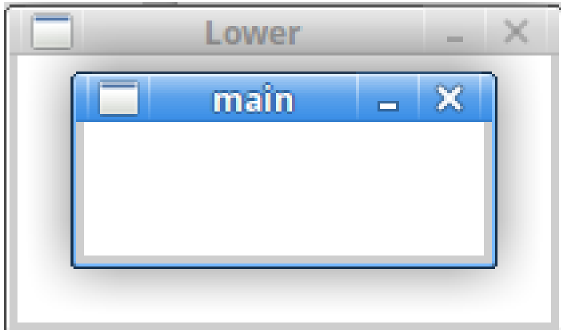
`Lower(w)` : *window*

`Lower(w)` moves window *w* to the bottom of the window stack. The window may end up obscured, hidden behind other windows.

```
#
# Lower.icn, lower a window to the bottom of the window stack
#
procedure main()
    w := open("main", "g", "size=150,50", "canvas=hidden")
    lower := open("Lower", "g", "size=200,100", "canvas=hidden")
    Lower(lower)
    WSync(w, lower)
    close(lower, w)
end
```

Sample run:

```
prompt$ unicon -s Lower.icn -x
```



See also:

Raise

7.1.151 lstat

`lstat(f)` : *record?* [*POSIX*]

`lstat(f)` returns a record of filesystem information for file (or path) *f*. Does not follow symbolic links, if *f* is a symlink, then information about the symlink is returned. See *stat*.

Return record is:

```
record posix_stat(dev, ino, mode, nlink, gid, rdev, size,
                  atime, mtime, ctime, blksize, blocks, symlink)
```

The `atime`, `ctime`, and `mtime` fields may be formatted with the *ctime*() and *gtime*() functions. `mode` is a string form similar to the output of `ls -l`. `lstat()` will fail if the file or path *f* does not exist.

```
#
# lstat.icn, File status information, does not follow symbolic links.
#
```

```
link ximage
procedure main()
    fn := "/usr/bin/cc"
    write("lstat(", fn, "): ", ximage(lstat(fn)))
end
```

Sample run:

```
prompt$ unicon -s lstat.icn -x
lstat(/usr/bin/cc): R_posix_stat_1 := posix_stat()
   R_posix_stat_1.dev := 2049
   R_posix_stat_1.ino := 4325501
   R_posix_stat_1.mode := "lrwxrwxrwx"
   R_posix_stat_1.nlink := 1
   R_posix_stat_1.uid := "root"
   R_posix_stat_1.gid := "root"
   R_posix_stat_1.rdev := 0
   R_posix_stat_1.size := 20
   R_posix_stat_1.atime := 1572162119
   R_posix_stat_1.mtime := 1452760995
   R_posix_stat_1.ctime := 1452760995
   R_posix_stat_1.blksize := 4096
   R_posix_stat_1.blocks := 0
```

7.1.152 many

`many(c, s, i, i)` : *integer?*

`many(c, s, i1, i2)` produces the position in *s* after the longest initial sequence of members of *c* within *s[i1:i2]*. A goal directed generator, but returns after the first match.

```
#
# many.icn, string scanning function scans for matches of many characters
#
procedure main()
    cs := &lcase
    s := "this is abcde of fghijklmnop"
    s ? write(many(cs))
end
```

Sample run:

```
prompt$ unicon -s many.icn -x
5
```

7.1.153 map

`map(s, s:&ucase, s:&lcase)` : *string*

`map(s1, s2, s3)` maps *s1* using *s2* and *s3*. The resulting string will be a copy of *s1* with any characters that appear in *s2* replaced by characters in the position from *s3*. The defaults allow for upper case to lower case conversions of *s1*.

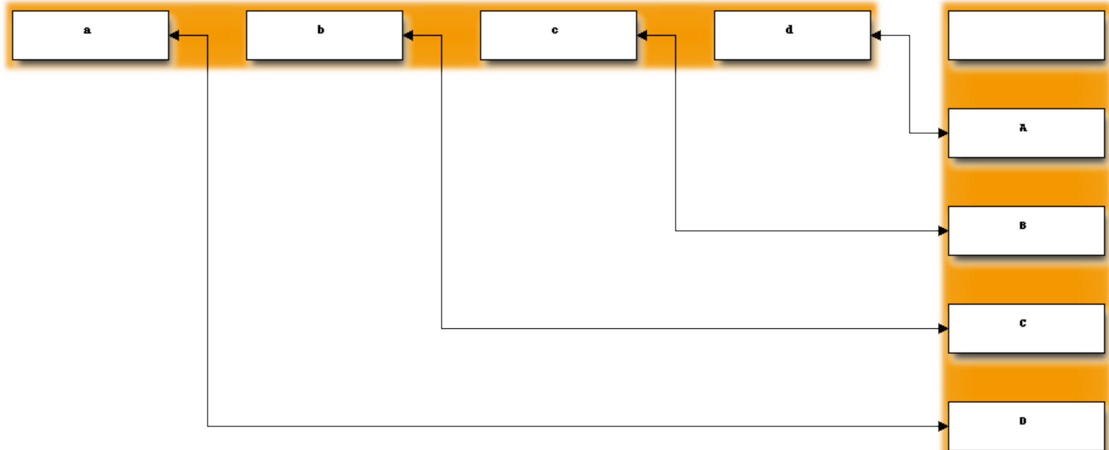
Map transforms are a powerful feature of Unicon. Permutations are possible when `map()` is called with *s1* and *s2* as constants and *s3* being a transform target.

```
#
# map.icn, demonstrate map and tranform
#
procedure main()
    # upper to lower case mapping
    up := "This is a TEST"
    write(map(up))

    # permutation mapping, allowing s3 to be the variable
    # s2 and s3 must be equal length
    s3 := "abcde"
    # '0' in s1 (index 1) maps to '0' in s2, (index 5)
    write("01234 43210: ", s3, " to ", map("01234", "43210", s3))
    write("Multiple s1: ", s3, " to ", map("012343210", "43210", s3))
end
```

Sample run:

```
prompt$ unicon -s map.icn -x
this is a test
01234 43210: abcde to edcba
Multiple s1: abcde to edcbabcde
```

`map()` is usually thought of as source, what-to-change, to-what. When the `to-what` is changed to be a variable, Unicon returns a string where character markers in *s1* are replaced by positionally matched characters from *s2*. Characters of the variable *s3* are sourced to feed an *s1 s2* positional transform.



Characters of s3 mapped by positional matches between s1 and s2.

7.1.154 match

`match(s, s:&subject, i:&pos, i:0 :` *integer*

`match(s1, s2, i1, i2)` produces *i1+*s1* if *s1 == s2[i1 +: *s1]* but fails otherwise. A string scanning function that returns the index at the end of a match starting at current position (by default).

```
#
# match.icn, demonstrate string scanning match
#
procedure main()
    s := "this is a test"
    # match will return the index after the matched string
    s ? this := match("this")
    write(this)

    # no match, variable that remains null, no write
    s ? that := match("that")
    write(\that)

    # match is built into the unary equal operator as tab(match(s))
    s ? data := ="this"
    write(":", data, ":")
    # equivalent to
    s ? data := tab(match("this"))
    write(":", data, ":")
end
```

Sample run:

```
prompt$ unicon -s match.icn -x
5
:this:
:this:
```

See also:

tab, *= (anchored or tab match)*

7.1.155 MatrixMode

MatrixMode (*w*, *s*) → record

Argument window

Argument string, "projection" or "modelview"

Returns display list record

Sets the current matrix stack mode. The "projection" stack can hold 2 matrices, the "modelview" stack can hold 32 matrices.

```
#
# MatrixMode.icn, demonstrate the two matrix modes
# Curerntly a work in progress
#
link fullimag
procedure main()
    window := open("MatrixMode", "gl", "bg=black", "buffer=on",
                   "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")
```

```
    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)
    Refresh(window)
    MatrixMode(window, "projection")
    PushMatrix(window)
    write(image(IdentityMatrix(window)))
    write(fullimage(PopMatrix(window)))

    MatrixMode(window, "modelview")
    PushMatrix(window)
    write(image(IdentityMatrix(window)))
    write(fullimage(PopMatrix(window)))

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/MatrixMode.png")
    close(window)
end
```

Sample run (not auto captured):

7.1.156 max

`max(n, ...)` : *number*

`max()` returns the largest value from the list of arguments. The arguments do not need to be numeric, but reasoning about comparison rules between different structures of possibly different types can be quite tricky.

```
#
# max.icn, demonstrate the maximum value function
#
link fullimag
procedure main()
    write(max(1,2,3))
    write()

    # using the apply operator comes in handy for max
    L := [1,2,3,4,5]
    write(max!L)
    # equivalent to
    write(max(1,2,3,4,5))
    write()

    # max also accepts structures directly
    write(max(L))

    # although max accepts structures, results are somewhat "fuzzy"
    L1 := ["abc", "def"]
    L2 := ["uvw", "xyz", "123"]
    L3 := ["UVW", "XYZ"]
    L4 := ["ZZZ", "457", "002"]
    L5 := ["YYY", "456", "001"]
```

```
    write(fullimage(max(L1, L2, L3, L4, L5)))
end
```

Sample run:

```
prompt$ unicon -s max.icn -x
3

5
5

5
["YYY","456","001"]
```

7.1.157 member

`member(x, ...)` : *x?*

`member(x, ...)` returns *x* if all other arguments are members of the *Set*, *cset*, *list*, or *table* *x*, but fails otherwise. If *x* is a `cset` all of the characters in subsequent string arguments must be present in *x* in order to succeed.

```
#
# member.icn, demonstrate structure membership test
#
procedure main()
    S := [1,2,3,4]
    if member(S, 1,2,3) then write("all members of set")

    T := table()
    insert(T, "key1", "abc", "key2", "def")
    if member(T, "key1") then write("have key key1")

    # this test will fail, table membership is by key
    if member(T, "abc") then write("have value abc")

    cs := 'abcde'
    if member(cs, "a","b","c") then write("all members of cset")
end
```

Sample run:

```
prompt$ unicon -s member.icn -x
all members of set
have key key1
all members of cset
```

7.1.158 membernames

`membernames(x)` : *list*

`membernames(x)` produces a list containing the string names of the fields of *x*, where *x* is either an object or a string name of a *class*.

```
#
# membernames.icn, demonstrate class/object membernames
#
link fullimag

class sample(a,b,c,d)
end

procedure main()
    # this initial write fails if sample is never instantiated
    write(fullimage(membernames("sample")))

    c := sample(1,2,3,4)
    write(fullimage(membernames(c)))
end
```

Sample run:

```
prompt$ unicon -s membernames.icn -x
["a","b","c","d"]
["__s","__m","a","b","c","d"]
```

7.1.159 methodnames

`methodnames()` : *type*

`methodnames(x)` produces a list containing the string names of the fields of *x*, where *x* is either an object or a string name of a *class*.

```
#
# methodnames.icn, demonstrate class/object methodnames
#
link fullimag

class sample(a,b,c,d)
    method one()
    end
    method two()
    end
end

procedure main()
    # this initial write is an empty list if sample is never instantiated
    write(fullimage(methodnames("sample")))

    c := sample(1,2,3,4)
    write(fullimage(methodnames(c)))
end
```

Sample run:

```
prompt$ unicon -s methodnames.icn -x
["sample_one","sample_two"]
["sample_one","sample_two"]
```

7.1.160 methods

`methods(x)` : *list*

`methods(x)` produces a *list* containing the procedure values of the methods of *x*, where *x* is either an object or the string name of a class.

```
#
# methods.icn, demonstrate class/object procedure values of methods list
#
link fullimag

class sample(a,b,c,d)
    method one()
    end
    method two()
    end
end

procedure main()
    # this initial write is an empty list if sample is never instantiated
    write(fullimage(methods("sample")))

    c := sample(1,2,3,4)
    write(fullimage(methods(c)))
end
```

Sample run:

```
prompt$ unicon -s methods.icn -x
[procedure sample_one,procedure sample_two]
[procedure sample_one,procedure sample_two]
```

7.1.161 min

`min(n, ...)` : *number*

`min()` returns the smallest value from the list of arguments, which must be numeric.

```
#
# min.icn, demonstrate the minimum value function
#
link fullimag
procedure main()
    write(min(1,2,3))
    write()

    # using the apply operator comes in handy for min
    L := [1,2,3,4,5]
    write(min!L)
    # equivalent to
    write(min(1,2,3,4,5))
    write()

    # min also accepts structures directly
    write(min(L))
```

```
    # although min accepts structures, results are somewhat "fuzzy"
    L1 := ["abc", "def"]
    L2 := ["uvw", "xyz", "123"]
    L3 := ["UVW", "XYZ"]
    L4 := ["ZZZ", "457", "002"]
    L5 := ["YYY", "456", "001"]
    write(fullimage(min(L1, L2, L3, L4, L5)))
end
```

Sample run:

```
prompt$ unicon -s min.icn -x
1

1
1

1
["abc","def"]
```

7.1.162 mkdir

`mkdir(s, x)` : *?*

`mkdir(path, mode)` attempts to create directory *path* with permissions *mode*. *mode* can be an integer or a string of the form:

```
[ugoa]*[+-=][rwxRWXstugo]*
```

See *chmod* for more details on mode values.

Getting the normal 8r755 value used for most directories is difficult with the mode string, as you need to set `u=rwx` but `go=rx`. It is easiest to leave *mode* as a default, or use the octal notation.

```
#
# mkdir.icn, demonstrate the create directory function
#
link convert
procedure main()
    path := "mkdir-sample"
    mode := 8r755 # rwxr-xr-x
    if mkdir(path, mode) then {
        write("created directory ", path,
              " with 8r", exbase10(mode, 8))
        system("ls -ld " || path)
        rmdir(path)
    }
    else
        write("directory create ", path,
              " with 8r", exbase10(mode, 8), " failed")
end
```

Sample run:

```
prompt$ unicon -s mkdir.icn -x
created directory mkdir-sample with 8r755
drwxr-xr-x 2 btiffin btiffin 4096 Oct 27 04:53 mkdir-sample
```

7.1.163 move

`move(i:1)` : *string*

`move(i)` moves the string scanning position *&pos* *i* characters from the current position and returns the substring of *&subject* between the old and new positions. This function will reset an old value if it is resumed during goal-directed evaluation.

The `move()` function makes little sense outside a string scanning environment but will effect `&pos` and read any explicitly set `&subject`.

```
#
# move.icn, demonstrate the string scanning move function
#
procedure main()
    # display first letter of each word, using counted moves
    s := "this is a test"
    s ? {
        write(move(1), " ", &pos)
        move(4)
        write(move(1), " ", &pos)
        move(2)
        write(move(1), " ", &pos)
        move(1)
        write(move(1), " ", &pos)
    }

    # outside of string scanning, move makes little sense
    # but can be used by explicitly setting &pos and &subject
    &pos := 1
    &subject := "not scanning"
    write(move(3))
end
```

Sample run:

```
prompt$ unicon -s move.icn -x
t 2
i 7
a 10
t 12
not
```

7.1.164 MultMatrix

MultMatrix(*w*, *L*) → record

Returns Transformation matrix record.

Multiplies the current transformation matrix used in 3D window *w* by the 4x4 matrix represented as a list of 16 values in *L*.

```
#
# MultMatrix.icn, demonstrate matrix transforms
# Curerntly a work in progress
#
link fullimag
procedure main()
    window := open("MultMatrix", "gl", "bg=black", "buffer=on",
                    "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)
    Refresh(window)
    MatrixMode(window, "projection")
    PushMatrix(window)

    MultMatrix(window, [[0,0,0,1], [0,0,1,0], [0,1,0,0], [1,0,0,0]])
    write(fullimage(PopMatrix(window)))

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/MultMatrix.png")
    close(window)
end
```

Sample run:

See also:

MatrixMode, *PopMatrix*, *PushMatrix*, *IdentityMatrix*

7.1.165 mutex

`mutex(x, y)` : *x*

`mutex(x)` creates a new mutual exclusion control variable for structure *x*, which can be a *list*, *set* or *table*. `mutex(x, y)` associates an existing mutex *y* (or mutex associated with protecting resoure *y*) with structure *x*. `mutex()` returns a new non-associated control variable, which can be used with *critical*.

```
#
# mutex.icn, Demonstrate mutually exclusive gating variables.
#
global x

procedure main()
    x := 0

    mtx := mutex()
    T1 := thread report(mtx)
    T2 := thread other(mtx)

    # due to the nature of threading
```

```
    # the values displayed in report
    # might be 1,2,3,4 or 1,10,11,12 or 1,2,11,12 etc
    wait(T1 | T2)

    # final result will always be 12
    write("x is ", x)
end

# increment and report
procedure report(mtx)
    every 1 to 4 do {
        critical mtx : x := x + 1
        write(x)
    }
end

# just increment
procedure other(mtx)
    every 1 to 8 do
        critical mtx : x := x + 1
end
```

Sample run:

```
prompt$ unicon -s mutex.icn -x
1
10
11
12
x is 12
```

See also:

critical

7.1.166 name

`name(v, CE:¤t)` : *type*

`name(v, ce)` returns the name of variable *v* within the program that created co-expression *ce* . Keyword variables are recognized. `name()` returns the base type and subscript or field information for variables that are elements within other values, but the returned string will unlikely match the source code for such variables.

```
#
# filename.icn, purpose
#
procedure main()
    a := 1
    write(name(a), " is ", a)

    L := [[1], ["abc"]]
    write(name(L[2]))

    # a dereferenced value will cause a runtime error
    write(name(.a))
end
```

Sample run (ends with error):

```
prompt$ unicon -s name.icn -x
a is 1
list_3[2]

Run-time error 111
File name.icn; Line 19
variable expected
offending value: 1
Traceback:
   main()
   name(1,&null) from line 19 in name.icn
```

7.1.167 NewColor

`NewColor(w, s)` : *integer* [*Graphics*]

`NewColor(w, s)` allocates a mutable colour entry in the palette map for the current windowing system, initializing the colour to *s*, and returns a small negative integer for this entry. The colour integer can be used as a colour specification. `NewColor()` fails if the entry cannot be allocated/

This is not deprecated, but mostly unnecessary with modern display hardware. Most current graphic displays can handle millions of simultaneous colours, something not possible when older graphics programs had a limited palette of colours to use at any given time, limited by the electronics of the day.

```
#
# NewColor.icn, demonstrate creating colour map entries.
#
procedure main()
    window := open("NewColor", "g", "size=110,24", "canvas=hidden")
    # RGB values can be from 0 to 65535
    if c := NewColor(window, "40000,50000,60000") then {
        Fg(window, c)
        write(window, Fg(window))
    }
    else {
        Fg(window, "black")
        write(window, "no mutable colours")
    }

    WSync(window)
    WriteImage(window, "../images/NewColor.png")
    FreeColor(\c)
    close(window)
end
```

Sample run:

```
prompt$ unicon -s NewColor.icn -x
```

no mutable colours

7.1.168 Normals

Normals (*w, s|l*) → *List*

Returns Element display list

Requires 3D graphics

Sets texture coordinates to those defined in the argument list or string.

```
#
# Normals.icn, demonstrate texture mapping coordinate setting
#
procedure main()
    &window := open("Normals", "gl", "size=200,200")


    close(&window)
end
```

Sample run (pending):

7.1.169 NotAny

NotAny(c) : *[Pattern]*

NotAny(c) is a SNOBOL based pattern that matches any single character *not* contained in the character set *c*, appearing in the subject string.

```
#
# NotAny.icn, demonstrate the SNOBOL based pattern NotAny() function
#
procedure main()
    str := "ABCdef"
    ups := NotAny(&lcase)
    write("Type of ups: ", type(ups))
    str ?? ups -> intermediate
    write(intermediate)
end
```

Sample run:

```
prompt$ unicon -s NotAny.icn -x
Type of ups: pattern
A
```

7.1.170 Nspan

Nspan(cset) : *string*

A SNOBOL inspired pattern matching function.

Nspan(cs) will match zero or more subject characters from the *cset* *cs*. It is equivalent to the pattern Span(cs) .| "" (matching the empty string after failing to span across any of the available characters).

```
#
# Nspan.icn, pattern matching function, zero or more characters
#
procedure main()
    "string" ?? Nspan('pqrstuv') => result
    write(image(result))

    # Nspan requires no anchor, it will match to zero occurrences
    "no character match at start of string" ?? Nspan('pqrstuv') => second
    write(image(second))
end
```

Sample run:

```
prompt$ unicon -s Nspan.icn -x
"str"
""
```

See also:

Span, *break*.

7.1.171 numeric

`numeric(any)` : *number*

`numeric(x)` produces an integer or real number resulting from type conversion of *x*, or fails if the conversion is not possible.

```
#
# numeric.icn, produce an integer or real from type conversion
#
procedure main()
    i := 123
    r := 123.123
    s := 123.0
    b := "36rXYZ"
    e := "abc"

    write(numeric(i) * 2)
    write(numeric(r) * 2)
    write(numeric(s) * 2)
    write(numeric(b) * 2)

    # no write occurs as e is not numeric, conversion fails
    write(numeric(e) * 2)

end
```

Sample run:

```
prompt$ unicon -s numeric.icn -x
246
246.246
246.0
88054
```

7.1.172 open

`open(s, s:"rt", ...)` : *file?*

`open(s1, s2, ...)` opens a resource named *s1* with mode *s2* and attributes given in trailing arguments. `open()` recognizes the following resource type:

- file
- socket
- tcp, mode "n" for network
- udp, mode "nu" for UDP network
- http, mode "m" for messaging
- https, mode "m-" (be forgiving with certificate authentication)
- https, mode "m" (authenticated)
- smtp, mode "m"
- pop, mode "m"
- finger, mode "m"
- mailto, mode "m" with email header fields as attribute arguments

---

Todo

a constant todo item is keeping this list up to date

The mode in *s2* can be

- "a" append; write after current contents
- "b" open for both reading and writing (b does not mean binary mode!)
- "c" create a new file and open it
- "d" open a [NG]DBM database
- "g" create a 2D graphics window
- "gl" create a 3D graphics window
- "n" connect to a remote TCP network socket
- "na" accept a connection from a TCP network socket
- "nau" accept a connection from a UDP network socket
- "nl" listen on a TCP network socket
- "nu" connect to a UDP network socket
- "m" connect to a messaging server (HTTP, SMTP, POP, ...)
- "m-" connect to secure HTTPS, unauthenticated certificates allowed
- "o" open an ODBC connection to a (typically SQL) database

- "p" execute a program given by command line s1 and open a pipe to it
- "prw" open an asynchronous pseudo terminal connection
- "r" read
- "t" use text mode, with newlines translated
- "u" use a binary untranslated mode
- "w" write
- "x" DEPRECATED, old X11 mode, use "g"
- "z" libz compressed modifier

```
#
# open.icn, demonstrate the open function
#
procedure main()
    f := open(&file) | stop("Cannot open ", &file, " for read")
    every write(!f)
    close(f)
end
```

Sample run:

```
prompt$ unicon -s open.icn -x
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: October 2016
# Modified: 2016-10-12/01:11-0400
##+
#
# open.icn, demonstrate the open function
#
procedure main()
    f := open(&file) | stop("Cannot open ", &file, " for read")
    every write(!f)
    close(f)
end
```

See also:

close, *read*, *reads*

7.1.173 opencl

`opencl(list)` : *integer*

An experimental Unicon interface to the Open Computing Language. Integrating CPU, GPU and other auxiliary hardware in a single framework with a backing programming language based on `C99`. The specification of OpenCL is royalty free, but there will be vendor specific portions, possibly non-free.

https://en.wikipedia.org/wiki/OpenCL by the Khronos Group.

Attention: Not yet officially part of Unicon release 13.

`opencl(L)` will display information and properties of the given list of devices.

```
#
# opencl.icn, an experimental Unicon interface to Open Computing Language
#
# not yet fully implemented in Unicon, this sample is wrong, and untested
#
procedure main()
    i := opencl(["GPU"])
end
```

Sample run (skipped for now):

7.1.174 oprec

`oprec(x)` : *record*

`oprec(r)` produces a variable reference for the class method vector of *r*.

```
#
# oprec.icn, reference to class method vector
#
link ximage

class sample(a,b,c)
    method one(a)
        write("in method a: ", type(a), ":, ", a)
    end
    initially
        a := 1
        b := 2
        c := 3
        write(a, ", ", b, ", ", c)
end

procedure main()
    c := sample(1,2,3)
    write(type(oprec(c)))
    write(ximage(oprec(c)))

    cmv := oprec(c)

    # too new to know what the first agrument is supposed to be
    # a self reference likely, not sure it that is the same as c
    cmv["initially"](c)
    cmv["one"](c,42)
    cmv["one"](c,"abc")
    cmv["one"](c)
end
```

Sample run:

```
prompt$ unicon -s oprec.icn -x
1, 2, 3
sample__methods
R_sample__methods_1 := sample__methods()
   R_sample__methods_1.one := procedure sample_one
   R_sample__methods_1.initially := procedure sample_initially
1, 2, 3
in method a: integer:, 42
in method a: string:, abc
in method a: null:,
```

7.1.175 ord

`ord(s)` : *integer*

`ord(s)` produces the character ordinal of the one character string *s*. The *ASCII* value of the byte.

```
#
# ord.icn, ordinal value of a one character string
#
procedure main()
    write("ord(A) is ", ord("A"))
    write("ord(\\n) is ", ord("\n"))
end
```

Sample run:

```
prompt$ unicon -s ord.icn -x
ord(A) is 65
ord(\n) is 10
```

7.1.176 OutPort

`OutPort(i1, i2)` : *null* [*MS-DOS*]

`OutPort(i1, i2)` will write *i2* to port *i1*. This is an MS-DOS specific feature of Unicon. *i2* in the range 0-255, a byte value.

```
#
# OutPort.icn, send a value to an MS-DOS port
#
procedure main()
    OutPort(1, 1)
end
```

Sample run (skipped on this GNU/Linux build machine):

See also:

InPort

7.1.177 PaletteChars

PaletteChars (*w*, *s*) → *string*

Returns String

Requires Graphics

Produces a string containing each of the letters in palette *s*. The palettes *c1* through *c6* define different colour encodings of images represented as string data.

```
#
# PaletteChars.icn, demonstrate drawing image strings
#
procedure main()
    &window := open("PaletteChars", "g", "size=85,40", "canvas=hidden")

    # colour palettes are encoded as character data
    write("PaletteChars(\"c1\") is ", PaletteChars("c1"))

    # save image for the document
    WSync()
    WriteImage("../images/PaletteChars.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s PaletteChars.icn -x | par
PaletteChars("c1") is
0123456789?!nNAa#@oOBb$%pPCc&|qQDd,.rREe;:sSFf+-tTGg*/uUHh`'vVIi<>wWJj()
xXKk[]yYLl{}zZMm^=
```

7.1.178 PaletteColor

PaletteColor (*w*, *p*, *s*) → *string*

Returns the colour key *s* in palette *p* from window *w*.

Returns the colour of key *s* in "r,g,b" form.

```
#
# PaletteColor.icn, demonstrate drawing image strings
#
procedure main()
    &window := open("PaletteColor", "g", "size=85,40", "canvas=hidden")

    # colour palettes are mapped using character data keys
    write("PaletteColor(\"c1\", \"9\") is ", PaletteColor("c1", "9"))

    # save image for the document
    WSync()
    WriteImage("../images/PaletteColor.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s PaletteColor.icn -x
PaletteColor("c1", "9") is 65535,49151,32767
```

7.1.179 PaletteKey

PaletteKey (*w*, *p*, *s*) → *string*

Returns the colour key of the closet colour to *s* in palette *p*.

Returns the colour of key *s* in "r,g,b" form.

```
#
# PaletteKey.icn, demonstrate getting colour mapping key
#
procedure main()
    &window := open("PaletteKey", "g", "size=85,40", "canvas=hidden")

    # colour palettes are mapped using character data keys
    write(PaletteKey("c1", "blue"))

    # save image for the document
    WSync()
    WriteImage("../images/PaletteKey.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s PaletteKey.icn -x
J
```

7.1.180 paramnames

`paramnames(CE, i:0)` : *string*

`paramnames(ce, i)` produces the names of the parameters in the procedure activation *i* levels above the current activation in *ce*.

```
#
# paramnames.icn, parameter names from co-expression stack
#
link ximage
procedure main(arglist)
    ce := create 1 to 3
    write(ximage(paramnames(ce)))
end
```

Sample run:

```
prompt$ unicon -s paramnames.icn -x
"arglist"
```

7.1.181 parent

parent(*CE*) → co-expression

`parent(ce)` returns the co-expression that created *ce*. Useful with *load*.

```
#
# parent.icn, co-expression of parent
#
# unicon load-child.icn to get a loadable file first
#
procedure main()
    write("Loading ./load-child from ", image(&current))
    lp := load("load-child")
    @lp
end
```

```
#
# load-child.icn, a demonstration of the parent function
#
procedure main()
    write("in load-child with ", image(&current))
    write("parent is ", image(parent(&current)))
end
```

Sample run:

```
prompt$ unicon -s load-child.icn
```

```
prompt$ unicon -s parent.icn -x
Loading ./load-child from co-expression_1(1)
in load-child with co-expression_1(0)
parent is co-expression_1(1)
```

7.1.182 Pattern

`Pattern(w, s)` : *window*

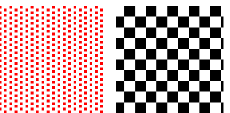
`Pattern(w, s)` selects a stipple pattern `s` for use during draw and fill operations. `s` may be the name of a *system-dependent* pattern or a literal, in the form `"width,bits"`. Patterns are only used when the `fillstyle` attribute is `stippled`, `opaquestippled` or `textured`. `Pattern()` fails if a named pattern is not defined. An error occurs if `s` is a malformed literal.

```
#
# Pattern.icn, demonstrate stippled Pattern fill
#
procedure main()
    &window := open("Pattern", "g",
                    "fillstyle=stippled",
                    "size=85,40", "canvas=hidden")
    # A width 4 pattern, with bit patterns of 2, 8, 2, 8
    Pattern(&window, "4,2,8,2,8")
    Fg(&window, "red")
    FillRectangle(&window, 0, 0, 40, 40)

```

```
    # built in checker board pattern
    Pattern("checkers")
    Fg("black")
    FillRectangle(45, 0, 40, 40)

    # save image for the document
    WSync()
    WriteImage("../images/Pattern.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s Pattern.icn -x
```



7.1.183 Peek

`Peek(A, i)` : *string* [*MS-DOS*]

This is an MS-DOS specific Unicon feature, now outdated.

`Peek(A, i)` builds a string from an "address" *A* (returned from *GetSpace*, memory outside of Unicon control and not garbage collected), of length *i*.

```
#
# Peek.icn, an MS-DOS specific memory region copy to string
#
# This sample is UNTESTED, which means it counts as broken.
#
procedure main()
    mem := GetSpace(32)
    s := Peek(mem, 32)
    write(image(s))
    FreeSpace(mem)
end
```

Sample run (skipped on this GNU/Linux machine):

See also:

Poke, *GetSpace*, *FreeSpace*, *Int86*

7.1.184 Pending

Pending(*w : window='&window'*[, *x, ...*]) → list
: Produce the list of pending events for window *w*, adding optional events *x*, to the end of the list, in a guaranteed order.

`Pending(w)` produces the list of events waiting on window *w*. `Pending(w, x1,...,xn)` adds *x1* through *xn* to the end of *w*'s pending list in the given order.

```
#
# Pending.icn, demonstrate pending event test
#
link enqueue, evmux, ximage
procedure main()
    window := open("Event", "g", "size=20,20", "canvas=hidden")

    # insert an event into the queue, left press, control and shift
    Enqueue(window, &lpress, 11, 14, "cs", 2)
    L := Pending(window)
    write(ximage(L))

    e := Event(window)
    write(image(e))

    # a side effect of the Event function is keywords settings
    write("&x:", &x)
    write("&y:", &y)
    write("&row:", &row)
    write("&col:", &col)
    write("&interval:", &interval)
    write("&control:", &control)
    write("&shift:", &shift)
    write("&meta:", &meta)

    close(window)
end
```

Sample run:

```
prompt$ unicon -s Pending.icn -x
L1 := list(3)
   L1[1] := -1
   L1[2] := 327691
   L1[3] := 131086
-1
&x:11
&y:14
&row:2
&col:2
&interval:2
&control:
&shift:
```

7.1.185 pipe

pipe() → list

`pipe()` creates a pipe and returns a list of two file objects. The first is for reading, the second is for writing.

```
#
# pipe.icn, demonstrate pipe operation
```

```
#
# tectonics:
#
#
procedure main()
    #pipes := pipe()
    #write(read(pipes[1]
    # too new, need better example
end
```

Sample run:

```
prompt$ unicon -s pipe.icn -x
```

7.1.186 Pixel

Pixel(w, ix, iy, iw, ih) : *integer** [*Graphics*]

`Pixel(w, x,y, wid,hgt)` generates pixel contents from a rectangular area within window *w*. Top left corner is *x.y* and pixel colours are generated in the *wid,hgt* rectangle down each column, from left to right. Pixel values are returned as 16bit RGB values, with mutable colours being negative integers, as returned by `NewColor()`.

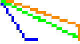
`Pixel()` fails if part of the requested rectangle extends beyond the canvas.

```
#
# Pixel.icn, demonstrate the Pixel value generator
#
procedure main()
    &window := open("Pixel", "g", "size=45,40", "canvas=hidden")

    # Some lines
    Fg("vivid orange")
    DrawLine(1,0, 30,10, 30,15)

    Fg("blue")
    DrawLine(0,0, 10,15, 14,15)

    Fg("green")
    DrawLine(1,1, 29,14)

    # generate the top corner pixel values, down and then to the right
    every write(image(Pixel(0, 0, 4, 4)))

    # save image for the document
    WSync()
    WriteImage("../images/Pixel.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s Pixel.icn -x
"0,0,65535"
"65535,16382,0"
"65535,16382,0"
```

```
"65535,65535,65535"
"65535,65535,65535"
"0,65535,0"
"0,65535,0"
"65535,16382,0"
"65535,65535,65535"
"0,0,65535"
"0,65535,0"
"0,65535,0"
"65535,65535,65535"
"65535,65535,65535"
"0,0,65535"
"65535,65535,65535"
```



7.1.187 PlayAudio

`PlayAudio(s)` : *integer* [*Audio*]

`PlayAudio(s)` will attempt to start an audio thread and play the filename *s*. Accepts .wav and .ogg files when Unicon if built with OpenAL and/or Vorbis support. Returns an integer from 0 to 15, representing the thread stream, or fails if the filename cannot be read, is an invalid format, too many channels are already in play, or there are problems with the sound equipment.

```
#
# PlayAudio.icn, play an audio file
#
procedure main()
    channel := PlayAudio("/home/btiffin/unicon/tests/unicon/handclap.ogg")
    write("Audio channel: ", channel)
    delay(300)
    StopAudio(channel)

    channel := PlayAudio("/home/btiffin/unicon/tests/unicon/alert.wav")
    write("Audio channel: ", channel)
    delay(500)
    StopAudio(channel)
end
```

Sample run (skipped, you probably wouldn't hear it from here anyway):

See also:

StopAudio, *VAttrib*

7.1.188 Poke

`Poke(A, s)` : *null*

This is an outdated MS-DOS specific feature of Unicon.

`Poke(A, s)` will replace the memory at "address" *A*, with the contents of string *s*. The "address" *A* is usually from *GetSpace* outside of Unicon control and garbage collection.

```
#
# Poke.icn, an MS-DOS specific string to memory region copy
#
# This sample is UNTESTED, which means it counts as broken.
#
procedure main()
    s := "Hello, distant past"
    mem := GetSpace(*s)
    Poke(mem, s)
    from := Peek(mem, *s)
    write(image(from))
    FreeSpace(mem)
end
```

Sample run (skipped on this GNU/Linux build machine):

See also:

Peek, *GetSpace*, *FreeSpace*, *Int86*

7.1.189 pop

`pop(L | Message)` : *any?*

`pop(L)` remove an element from the top of the stack *L[1]*, and return the value. `pop(M)` removes and returns the first message in a POP mailbox connection *M*. *The actual delete occurs on close.*

```
#
# pop.icn, demonstrate pop from stack
#
procedure main()
    L := [42, 43, 44]
    write(*L, " ", pop(L), " ", *L)
end
```

Sample run:

```
prompt$ unicon -s pop.icn -x
3 42 2
```

7.1.190 PopMatrix

`PopMatrix``(`*w*`)` → *record*

Returns Display list element record

PopMatrix(w) pops the top matrix from the either the "projection" or "modelview" matrix stack. The function fails if there is only element on the current matrix stack.

```
#
# PopMatrix.icn, demonstrate matrix stack
# Curerntly a work in progress
#
link fullimag
procedure main()
    window := open("PopMatrix", "gl", "bg=black", "buffer=on",
                        "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)
    Refresh(window)
    MatrixMode(window, "projection")
    PushMatrix(window)

    write(fullimage(PopMatrix(window)))

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/PopMatrix.png")
    close(window)
end
```

Sample run:

See also:

PushMatrix, *MatrixMode*

7.1.191 Pos

`Pos(i)` : *null*

A SNOBOL based pattern matching function, positional test.

`Pos(i)` will ensure the scanning position is at *i* or will fail.

Good for ensuring anchored match strategies along with other position verification uses when validating pattern matches.

```
#
# Pos.icn, SNOBOL pattern matching function, test position
#
procedure main()
    # Pattern matching positional test
    "string" ?? Span('pqrstuv') => first || Pos(4) || Tab(0) => rest
    write(image(first), " ", image(rest))

    # enforce anchored pattern matching with unachored Span
    "no match at start of string" ?? Pos(1) || Span('rst') => second
    # second will be unset
    write(image(second))

    # same again, without positional test, Span allowed to scan forward
    "ok, no match at start of string" ?? Span('rst') => third
```

```
    write(image(third))
end
```

Sample run:

```
prompt$ unicon -s Pos.icn -x
"str" "ing"
&null
"t"
```

7.1.192 pos

`pos(i)` : *integer?*

`pos(i)` tests whether *&pos* is at position *i* in *&subject*.

```
#
# &pos, string scanning position
#
# demonstrate how negative position indexes are set to actual
#
procedure main()
    str := &letters
    str ? {
        first := &pos
        &pos := 0
        last := &pos
        &pos := -10
        back10 := &pos
    }
    write("first: ", first, ", last: ", last, ", -10: ", back10)
end
```

Sample run:

```
prompt$ unicon -s pos.icn -x
first: 1, last: 53, -10: 43
```

7.1.193 proc

`proc(any, i:1, CE)` : *procedure*

`proc(s, i)` converts *s* to a procedure, if possible. The parameter *i* is used to resolve ambiguous names by argument count.

- 0, built-in (for when the procedure name is overridden by a user program)
- 1, signature has 1 parameter
- 2, procedure has 2 parameters
- 3, procedure has 3 parameters

`proc` can also resolve procedure names in co-expression *CE*, *i* levels up.

```
#
# proc.icn, convert to procedure if possible
#
procedure main()
     # lookup write builtin
     s := "write"
     p := proc(s, 0) | stop("builtin lookup fail for " || s)

     # and use the default built-in
     p("Hello, world")

     # the global name "write" is overriden by the next procedure
     write("Does nothing now, no output will occur")
end

procedure write(s)
    # sophisticated override
    # ...
end
```

Sample run:

```
prompt$ unicon -s proc.icn -x
Hello, world
```

7.1.194 pull

`pull(L, i:1)` : *any?*

`pull(L)` removes and produces an element from the end of the non-empty list *L*. `pull(L, i)` removes *i* elements, producing the last one removed.

```
#
# pull.icn, remove from end of list and return value
#
procedure main()
    L := [41, 42, 43]
    write(*L, " ", pull(L, 2), " ", *L)
end
```

Sample run:

```
prompt$ unicon -s pull.icn -x
3 42 1
```

7.1.195 push

`push(L, any, ...)` : *list*

`push(L, x1, ..., xn)` pushes elements onto the beginning of *list*, *L*. The order added to the list is the reverse order they are supplied as parameters to the call to `push()`. `push()` returns the list that is passed as the first parameter, with the new elements added.

```
#
# push.icn, push elements to front of list
#
link lists
procedure main()
    L := [1,2,3]
    L2 := push(L, 43, 42, 41)
    write("last parameter is pushed last, head: ", L2[1])

    every L3 := push(L2, 41 to 43)
    write("after 41 to 43 separate, head: ", L3[1])
    write(limage(L3))
end
```

Sample run:

```
prompt$ unicon -s push.icn -x
last parameter is pushed last, head: 41
after 41 to 43 separate, head: 43
[43,42,41,41,42,43,1,2,3]
```

7.1.196 PushMatrix

`PushMatrix`(*w*) → *record*

Returns Display list element record

PushMatrix(w) the current matrix to either the "projection" or "modelview" matrix stack. The function fails if current stack is full, 2 items for "projection" and 32 for "modelview".

```
#
# PushMatrix.icn, demonstrate matrix stack
# Currently a work in progress
#
link fullimag
procedure main()
    window := open("PushMatrix", "gl", "bg=black", "buffer=on",
                   "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)
    Refresh(window)
    MatrixMode(window, "projection")
    PushMatrix(window)

    write(fullimage(PopMatrix(window)))

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/PushMatrix.png")
```

```
    close(window)
end
```

Sample run (skipped):

See also:

PopMatrix, *MatrixMode*

7.1.197 PushRotate

```
#
# PushRotate.icn, demonstrate matrix push with rotatation
# Currently a work in progress
#
link fullimag
procedure main()
    window := open("PushRotate", "gl", "bg=black", "buffer=on",
                    "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)
    Refresh(window)
    MatrixMode(window, "projection")
    PushRotate(window, 90, 0, 1, 1)

    write(fullimage(PopMatrix(window)))

    # save image for the document
    Refresh(window)
    WSync(window)
    WriteImage(window, "../images/PushRotate.png")
    close(window)
end
```

Sample run (not auto generated):

See also:

PushMatrix, *Rotate*

7.1.198 PushScale

`PushScale()` : *type*

Todo

entry for function PushScale

`PushScale()`

Sample run:

7.1.199 PushTranslate

`PushTranslate()` : *type*

Todo

entry for function PushTranslate

`PushTranslate()`

Sample run:

7.1.200 put

`put(L, x1,...,xn)` : *list*

`put(L, x1,..., xn)` puts elements on the end of *list* *L*. Returns the list *L*, with elements added.

```
#
# put.icn, put elements to front of list
#
link lists
procedure main()
    L := [1,2,3]
    L2 := put(L, 43, 42, 41)
    write(*L2, " ", L2[-1])
    write(limage(L3))

    every L3 := put(L2, 41 to 43)
    write(*L3, " ", L3[-1])
    write(limage(L3))

    # L, L2, L3 all reference the same list
    write(*L, " ", *L2, " ", *L3)
end
```

Sample run:

```
prompt$ unicon -s put.icn -x
6 41
9 43
[1,2,3,43,42,41,41,42,43]
9 9 9
```

7.1.201 QueryPointer

`QueryPointer(w)` : *x,y*

`QueryPointer(w)` generates *x* then *y* of the mouse cursor position relative to window *w*. `QueryPointer()` generates *x* then *y* of the mouse position relative to the display screen.

```
#
# QueryPointer.icn, generate x,y position of mouse cursor
#
procedure main()
    w := open("main", "g", "size=150,50", "canvas=hidden")
    writes("mouse cursor relative to window w: ")
    pos := ""
    every pos ||:= QueryPointer(w) || ","
    write(pos[1:-1])

    writes("mouse cursor relative to screen: ")
    pos := ""
    every pos ||:= QueryPointer() || ","
    write(pos[1:-1])
    close(w)
end
```

Sample run:

```
prompt$ unicon -s QueryPointer.icn -x
mouse cursor relative to window w: 0,0
mouse cursor relative to screen: 1321,608
```

7.1.202 Raise

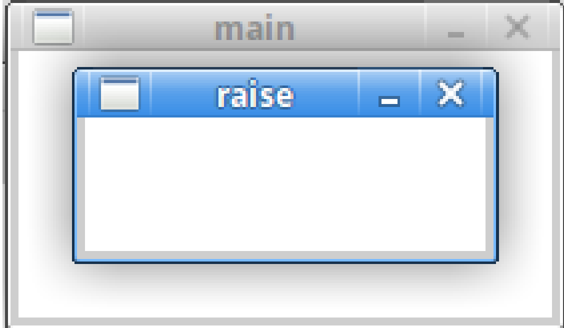
`Raise(w)` : *window* [*Graphics*]

`Raise(w)` raises window *w* to the top of the window stack, making it fully visible. Makes the window the active window.

```
#
# raise.icn, raise a window to the top of the window stack
#
procedure main()
    raise := open("raise", "g", "size=150,50", "canvas=hidden")
    w := open("main", "g", "size=200,100", "canvas=hidden")
    Raise(raise)
    WSync(w, raise)
    close(raise, w)
end
```

Sample run:

```
prompt$ unicon -s Raise.icn -x
```



See also:

Lower

7.1.203 read

`read(f:&input)` : *string?*

`read(f)` reads a line from file *f*. The end of line marker is discarded. Fails on end of file.

```
#
# read.icn, read a record
#
procedure main()
    f := open(&file, "r") | stop("cannot open ", &file, " for read")
    while write(read(f))
    close(f)
end
```

Sample run:

```
prompt$ unicon -s read.icn -x
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: October 2016
# Modified: 2016-10-12/04:29-0400
##+
#
# read.icn, read a record
#
procedure main()
    f := open(&file, "r") | stop("cannot open ", &file, " for read")
    while write(read(f))
    close(f)
end
```

7.1.204 ReadImage

`ReadImage(w, s, x:0, y:0)` : *integer*

`ReadImage(w, s, x, y)` loads an image from file *s* into window (or texture) *w*, at offset *x,y*. Returns an integer 0 for no errors, or a non zero indicating errors occurred. `ReadImage()` fails if *s* could not be read or is an invalid image format.

```
#
# ReadImage.icn, load an image from file into a window (or texture)
#
procedure main()
    w := open("ReadImage", "g", "size=327,420", "canvas=hidden")
    result := ReadImage(w, "../images/uniconart-orange.png")
    DrawString(w, 5, 405, "Result from ReadImage: " || image(result))
    WSync(w)
    WriteImage(w, "../images/ReadImage.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s ReadImage.icn -x
```

Art by Serendel Macphereson

See also:

WriteImage

7.1.205 readlink

readlink (*path : string*) → string

Produces the filename referenced by the symbolic link, *path*.

Parameters **path** – the path to dereference

Returns The linked filename

`readlink(s)` : *string*

`readlink(s)` produces the filename referred to by the symbolic link at path *s*.

```
#
# readlink.icn, demonstrate the POSIX readlink() function
#
procedure main()
    write(readlink("tt.dst"))
end
```

Sample run:

```
prompt$ unicon -s readlink.icn -x
tt.src
```

7.1.206 reads

reads (*f : file='&input', i : integer='1'*) → string

Reads up to *i* characters from file *f*.

Returns string

`reads(f:&input, i:1)` : *string?*

`reads(f, i)` reads up to *i* characters from *f*. It fails on end of file. If *f* is a network connection, *reads()* returns as soon as has input available, even if shorter than *i*. If *i* is `-1`, `reads()` produces the entire file as a string.

```
#
# reads.icn, read a given number of bytes, untranslated.
#
procedure main()
    f := open(&file, "r") | stop("cannot open ", &file, " for read")
    s := reads(f, -1)
    write(&file, " ", *s)
    p := find("procedure main", s)
    writes(s[p:0])
    close(f)
end
```

Sample run:

```
prompt$ unicon -s reads.icn -x
reads.icn 387
procedure main()
    f := open(&file, "r") | stop("cannot open ", &file, " for read")
    s := reads(f, -1)
    write(&file, " ", *s)
    p := find("procedure main", s)
    writes(s[p:0])
    close(f)
end
```

7.1.207 ready

`ready(f:&input, i:0)` : *string?*

`ready(f, i)` reads up to *i* characters from the file *f*. It returns immediately with available data or fails otherwise. If *i* is `0`, `ready()` returns all available input. Not implemented for *window* values.

```
#
# ready.icn, read a given number of bytes, untranslated.
#
link ximage

procedure main()
    f := open(&file, "r") | stop("cannot open ", &file, " for read")
    s := ready(f, 0)
    write(&file, " ", ximage(s))
    /s := "none\n"
    p := find("procedure main", s)
    /p := 1
    writes(s[p:0])
    close(f)
end
```

Sample run:

```
prompt$ unicon -s ready.icn -x
ready.icn &null
none
```

7.1.208 real

`real(any)` : *real?*

`real(x)` converts *x* to a *real*, or fails if the conversion cannot be performed.

```
#
# real.icn, convert to real if possible
#
procedure main()
```

```
    write(real(1))
    write(real("2E10"))
    write(real("123.123"))
    write(real("abc") | "not real")
end
```

Sample run:

```
prompt$ unicon -s real.icn -x
1.0
20000000000.0
123.123
not real
```

7.1.209 receive

`receive(f)` : *record*

`receive(f)` reads a datagram addressed to the port associated with *f*, waiting if necessary. The returned record is: `record posix_message(addr, msg)` containing the address of the sender and the contents of the message.

```
#
# receive.icn, read a datagran from a UDP network
#
link ximage
procedure main()
    f := open(":1025", "nua")
    #while r := receive(f) do {
    #    write(ximage(r))
        # Process the request in r.msg
        # ...
        #    send(r.addr, reply)
    #    break
    #}
    write(ximage(f))
    close(f)
end
```

No sample run, document generator doesn't like to wait.

7.1.210 Refresh

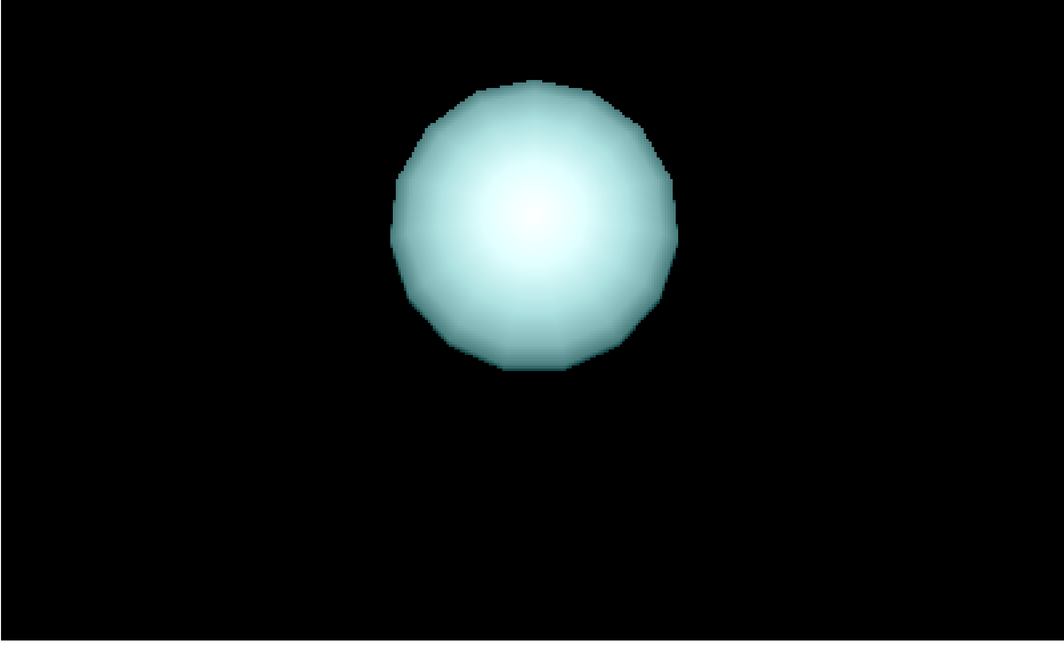
`Refresh(w)` : *window* [*3D Graphics*]

`Refresh(w)` redraws the contents of window *w*. For use when objects have been moved in a 3D scene. Returns the window *w*.

This is similar to the 2D graphic *WSync* function, but causes a redraw of cached 3D graphics, independent of any server side cache that may be in play.

```
#
# Refresh.icn, demonstrate 3D redraw
#
procedure main()
    window := open("Refresh", "gl", "bg=black", "buffer=on",
                    "size=400,240", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A sphere
    DrawSphere(window, 0.0, 0.19, -2.2, 0.3)

    # save image for the document, refresh is required for the WriteImage
    # but there is a bug with Unicon opengl hidden canvas on GNU/Linux
    Refresh(window)
    WriteImage(window, "../images/Refresh.png")
    close(window)
end
```

Sample run (not auto captured due to an issue with hidden canvas in 3D):



See also:

WSync

7.1.211 Rem

`Rem()` : *string*

A SNOBOL pattern match function.

`Rem()` returns the remaining characters from the current position to the end of the subject string. This function won't ever fail.

```
#
# Rem.icn, SNOBOL pattern, return characters from current to end
#
procedure main()
    "string" ?? Tab(4) || Rem() => result || .> curs
     write(image(result), " ", curs)
end
```

Sample run:

```
prompt$ unicon -s Rem.icn -x
"ing" 7
```

7.1.212 remove

`remove(s)` : *?*

`remove(s)` removes the file named *s*, or fails.

```
#
# remove.icn, remove a file
#
procedure main()
    if remove("test.tt") then write("removed test.tt")
    else write("did not remove test.tt")
end
```

Sample run:

```
prompt$ unicon -s remove.icn -x
did not remove test.tt
```

7.1.213 rename

`rename(s, s)` : *?*

`rename(s1, s2)` renames file *s1* to the name *s2*.

```
#
# rename.icn, rename files
#
procedure main()
    if rename("was-here.txt", "now-here.txt") then
        write("renamed file")
    else
        write("rename failed")
end
```

Sample run:

```
prompt$ unicon -s rename.icn -x
rename failed
```

7.1.214 repl

`repl(x, i)` : *x*

`repl(x, i)` concatenates *i* copies of the string *x*. As of Unicon revision 4608, `repl(L, n)` can be used to replicate *n* copies of the *list* *L*.

```
#
# repl.icn, demonstrate element replication
#
link lists
procedure main()
    write(repl("*", 24), " stars ", repl("*", 24))

    r := 0
    every feat := &features do {
        if find("Revision", feat) then {
            feat ? {
                ="Revision"
                tab(many(' '))
                r := integer(tab(0))
            }
            # for Unicon builds after 4608, repl works with lists
            if r > 4607 then {
                L := [1,2,3]
                write(limage(L))
                write(limage(repl(L, 3)))
            }
        }
    }
end
```

Sample run:

```
prompt$ unicon -s repl.icn -x
************************ stars ************************
```

7.1.215 reverse

`reverse(x)` : *x*

`reverse(x)` returns the reverse of the string or list *x*.

```
#
# reverse.icn, reverse a string or list
#
link lists
procedure main()
    L := [1,2,3]
```

```
    write(limage(L))
    write(limage(reverse(L)))

    s := "abc"
    write(s)
    write(reverse(s))
end
```

Sample run:

```
prompt$ unicon -s reverse.icn -x
[1,2,3]
[3,2,1]
abc
cba
```

7.1.216 right

`right(s1, i:1, s2:" ")` : *type*

`right(s1, i, s2)` formats *s1* to be a string of length *i*. If *s1* has more than *i* characters, it is truncated (from the left). If *s1* has less than *i* characters then it is padded to the left with as many copies of *s2* needed to increase the length to *i*. Last copy of *s2* is right side truncated if necessary ("filler" will pad as "fill" for instance).

```
#
# right.icn, fill out a string to length, truncate or pad fill from left
#
procedure main()
    s := "abcdefghij"
    write(":", right(s, 5), ":")
    write(":", right(s, 10), ":")
    write(":", right(s, 20), ":")
    write(":", right(s, 20, "filler"), ":")
end
```

Sample run:

```
prompt$ unicon -s right.icn -x
:fghij:
:abcdefghij:
:          abcdefghij:
:fillerfillabcdefghij:
```

7.1.217 rmdir

`rmdir(s)` : *?*

`rmdir(d)` removes the directory named *d*. `rmdir()` fails if *d* is not empty, or does not exist.

```
#
# rmdir.icn, remove a non-empty directory
#
procedure main()
    if rmdir("dir.tt") then write("removed dir.tt/")
    else write("failed to remove dir.tt/")
end
```

Sample run:

```
prompt$ unicon -s rmdir.icn -x
failed to remove dir.tt/
```

7.1.218 Rotate

`Rotate()` : *type*

Todo

entry for function Rotate

`Rotate()`

Sample run:

7.1.219 Rpos

`Rpos(i)` : *null*

A SNOBOL pattern function. Like *pos* counted back from the end.

`Rpos(i)` will test if the current scanning position is equal to *i* position back from the end of the subject string. Fails if not.

```
#
# Rpos.icn, SNOBOL pattern function, test position counted from end
#
procedure main()
    # Pattern matching positional test, counted back from end
    "string" ?? Span('pqrstuv') => first || Rpos(3) || Tab(0) => rest
    write(image(first), " ", image(rest))
end
```

Sample run:

```
prompt$ unicon -s Rpos.icn -x
"str" "ing"
```

7.1.220 Rtab

`Rtab(i)` : *string*

A SNOBOL inspired pattern matching function.

`Rtab(i)` causes the cursor to be placed *i* positions back from the end of the subject, and returns the characters between the current position and this new position.

```
#
# Rtab.icn, SNOBOL pattern return characters, current to count from end
#
procedure main()
    # match characters from current to new cursor, set back from end
    # Rtab counts backwards from end
    # returns characters from position 1,2 with cursor now at 3
    "string" ?? Rtab(4) => result || .> curs
    write(image(result), " ", curs)
end
```

Sample run:

```
prompt$ unicon -s Rtab.icn -x
"st" 3
```

See also:

tab

7.1.221 rtod

`rtod()` : *real*

`rtod(radians)` given radians as a `Real` value, produces the conversion to degrees, as a `Real` number.

$d^{\circ} = r\frac{180^{\circ}}{\pi}$

```
#
# rtod.icn, demonstrate radians to degrees
#
link numbers

# uses decipos from numbers, align decimal within field
procedure main()
    write("Radians Degrees")
    every r := 0 to 2 * &pi by 0.25 do
        write(decipos(r, 4, 8), decipos(rtod(r), 4, 8))
end
```

Sample run:

```
prompt$ unicon -s rtod.icn -x
Radians Degrees
  0.0      0.0
  0.25    14.3239
  0.5     28.6478
  0.75    42.9718
```

```
  1.0    57.2957
  1.25   71.6197
  1.5    85.9436
  1.75  100.2676
  2.0   114.5915
  2.25  128.9155
  2.5   143.2394
  2.75  157.5633
  3.0   171.8873
  3.25  186.2112
  3.5   200.5352
  3.75  214.8591
  4.0   229.1831
  4.25  243.5070
  4.5   257.8310
  4.75  272.1549
  5.0   286.4788
  5.25  300.8028
  5.5   315.1267
  5.75  329.4507
  6.0   343.7746
  6.25  358.0986
```

7.1.222 runerr

`runerr(i, any)`

`runerr(i, x)` produces a runtime error *i* with value *x*. Program execution is terminated.

```
#
# runerr.icn, terminate a run with an error code and value
#
procedure main()
    x := "abc"
    runerr(102, x)
    write("doesn't get here")
end
```

Sample run:

```
prompt$ unicon -s runerr.icn
```

```
prompt$ ./runerr

Run-time error 102
File runerr.icn; Line 13
numeric expected
offending value: "abc"
Traceback:
   main()
   runerr(102,"abc") from line 13 in runerr.icn
```

7.1.223 save

`save(s)` : *?*

`save(s)` saves the run-time system in file *s*.

```
#
# save.icn, save the run-time system to a file
#
procedure main()
    if &features == "ExecImage" then
        save("save-runtime.tt")
    else
        write("no ExecImage")
end
```

Sample run:

```
prompt$ unicon -s save.icn -x
no ExecImage
```

7.1.224 Scale

`Scale()` : *type*

Todo

entry for function Scale

`Scale()`

Sample run:

7.1.225 seek

`seek(f, any)` : *file?*

`seek(f, i)` seeks to offset *i* in file *f*, if it is possible. If *f* is a regular file, *i* must be an *integer*. If *f* is a database, *i* seeks a position within the current set of selected rows. The position is selected numerically if *i* is convertible to an integer, otherwise *i* must be convertible to a string, and the position is selected associatively by the primary key.

```
#
# seek.icn, seek to position within file, database set
#
procedure main()
    fn := "newfile.txt"
    f := open(fn, "r")
    # skip to position 4, "newfile line 1" to "file line 1"
    seek(f, 4)
    write(read(f))
    close(f)
end
```

Sample run:

```
prompt$ unicon -s seek.icn -x
file line 1
```

7.1.226 select

`select(x1, x2, ?)` : *list*

`select(files?, timeout)` waits for input to become available on any of several file resources, typically network connections or windows. The arguments may be files, or lists of files, ending with an optional integer timeout in milliseconds. It returns a list of the files from the input list that have input waiting.

If the final argument to `select()` is an integer, it acts as a timeout forcing a select return. A timeout of `0` causes `select()` to return immediately with a list of files (if any). If no files are given, `select()` will wait for the timeout. If no timeout is given `select()` waits forever for available input on one of the arguments.

Directories and databases cannot be arguments for `select()`

```
#
# select.icn, demonstrate select (input available) function
#
procedure main()
    res := select(&input, 1)
    write(image(*res), " ", image(res))
end
```

Sample run:

```
prompt$ unicon -s select.icn -x
0 list_2(0)
```

7.1.227 send

`send(s, s)` : *?*

`send(s1, s2)` sends a UDP datagram to the address *s1* (in host:port format) with message contents *s2*.

```
#
# send.icn, send a datagran to a UDP connection
#
link ximage
procedure main()
    #f := open(":1025", "nu")
        # Process the request in r.msg
        # ...
    #r := send(":1025", "test")
    #write(ximage(r))
    #close(f)
end
```

Sample run:

```
prompt$ unicon -s send.icn -x
```

7.1.228 seq

`seq(i:1, i:1)` : *integer**

`seq(i, j)` generates an infinite sequence *i*, *i+j*, *i+2j*, ... *j* may not be `0`.

```
#
# seq.icn, infinite sequence generator
#
procedure main()
    # infinite sequence, with limitation operator
    every write(seq(5, 5)\4)
end
```

Sample run:

```
prompt$ unicon -s seq.icn -x
5
10
15
20
```

7.1.229 serial

`serial(x)` : *integer*

`serial(x)` returns the serial number for structure *x*, if it has one. Serial numbers uniquely identify structure values.

```
#
# serial.icn, serial number of structure
#
procedure main()
    L := [1,2,3]
    S := [1,1,1]
    write("L: ", serial(L))
    write("S: ", serial(S))
    S := [1,2,3]
    write("S: ", serial(S))
    insert(S, 4)
    write("S: ", serial(S))
end
```

Sample run:

```
prompt$ unicon -s serial.icn -x
L: 1
S: 2
S: 3
S: 3
```

7.1.230 set

`set(x, ...)` : *set*

`set()` create a set. Arguments are inserted into the new set, with the exception of lists. `set(L)` creates a set with members taken from the elements of *list* *L*.

```
#
# set.icn, create a new set
#
link fullimag
procedure main()
    S := set()
    write(fullimage(S))
    S := set(1,3,5)
    write(fullimage(S))
    S := set([1,2,3,4,5,5,4,3,2,1])
    write(fullimage(S))
end
```

Sample run:

```
prompt$ unicon -s set.icn -x
set()
set(1,3,5)
set(1,2,3,4,5)
```

7.1.231 setenv

`setenv(s, s)` : *?*

`setenv(s1, s2)` sets environment variable *s1* to the value *s2*.

```
#
# setenv.icn, demonstrate setting an environment variable
#
procedure main()
    # note that children cannot set parent environment strings
    # processes inherit from parents, but can't effect parent space
    if setenv("UNICONTEST", "env value") then
        write(getenv("UNICONTEST"))
end
```

Sample run:

```
prompt$ unicon -s setenv.icn -x
env value
```

See also:

getenv

7.1.232 setgid

`setgid(i)` : *null* [*POSIX*]

`setgid(i)` will attempt to set the current process group id. Usually requires permissions.

```
#
# setgid.icn, change the current group identity
#
link ximage
procedure main()
    # set group to a safe group for demonstration
    write(ximage(setgid(getgr("btiffin").gid)))
    write(getgid())
end
```

Sample run:

```
prompt$ unicon -s setgid.icn -x
&null
btiffin
```

See also:

getgid, *getgr*

7.1.233 setgrent

`setgrent()` : *null* [*POSIX*]

`setgrent()` resets and rewinds the implicit context used by *getgr* to read through the operating system group entities when `getgr()` is given no explicit key value.

```
#
# setgrent.icn, reset and rewind the group file offset used by getgr()
#
link fullimag
procedure main()
    # reset group file reader, returns &null
    write(fullimage(getgr()))
    write(fullimage(getgr()))
    write("reset the group entity context")
    setgrent()
    write(fullimage(getgr()))
end
```

Sample run:

```
prompt$ unicon -s setgrent.icn -x
posix_group("root","x",0,"")
posix_group("daemon","x",1,"")
reset the group entity context
posix_group("root","x",0,"")
```

See also:

getgr

7.1.234 sethostent

`sethostent()` : *type*

Todo

entry for function sethostent

`sethostent()`

Sample run:

7.1.235 setpgrp

`setpgrp()` : *type*

Todo

entry for function setpgrp

`setpgrp()`

Sample run:

7.1.236 setpwent

`setpwent()` : *type*

Todo

entry for function setpwent

`setpwent()`

Sample run:

7.1.237 setservent

`setservent()` : *type*

Todo

entry for function setservent

`setservent()`

Sample run:

7.1.238 setuid

`setuid()` : *type*

Todo

entry for function setuid

`setuid()`

Sample run:

7.1.239 signal

`signal(cv, i:1)` : *?*

`signal(x, y)` signals the condition variable *x*. If *y* is supplied, the condition variable is signalled *y* times. If *y* is 0, a **broadcast** signal is sent, waking up all threads waiting on *x*.

```
#
# signal.icn, send threading signals
#
procedure main()

end
```

Sample run:

```
prompt$ unicon -s signal.icn -x
```

See also:

condvar, *wait*

7.1.240 sin

`sin(r)` : *real*

`sin(r)` returns the sine value of the given angle, *r* (in radians)

```
#
# sin.icn, demonstrate the sine function
#
procedure main()
    write("sin(r): Domain all real repeating within 0 <= r <= 2pi (in radians),␣
↪Range: -1 <= x <= 1")
    every r := 0.0 to &pi * 2 by &pi/4 do {
        write(left("sin(" || r || ")", 24), " radians = ", sin(r))
    }
end
```

Sample run:

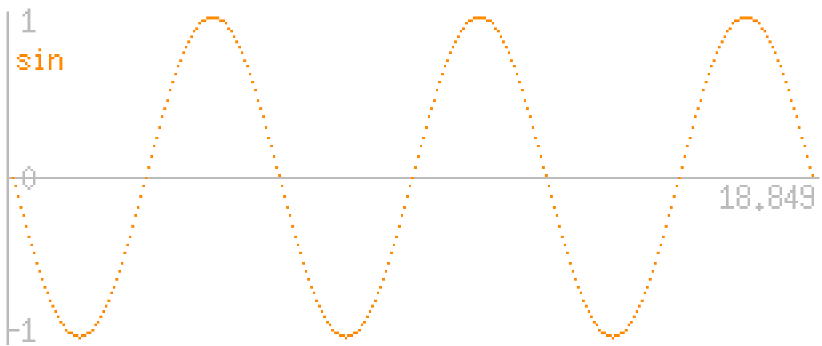
```
prompt$ unicon -s sin.icn -x
sin(r): Domain all real repeating within 0 <= r <= 2pi (in radians), Range: -1 <= x
↪<= 1
sin(0.0)                radians = 0.0
sin(0.7853981633974483) radians = 0.7071067811865475
sin(1.570796326794897)  radians = 1.0
sin(2.356194490192345)  radians = 0.7071067811865476
sin(3.141592653589793)  radians = 1.224646799147353e-16
sin(3.926990816987241)  radians = -0.7071067811865475
sin(4.71238898038469)   radians = -1.0
sin(5.497787143782138)  radians = -0.7071067811865477
sin(6.283185307179586)  radians = -2.449293598294706e-16
```

Graphical plot:

```
#
# plot-function, trigonometric plotting, function from command line
#
$define points 300
$define xoff   4
$define base   64
$define yscale 60
$define xscale 100

invocable "sin", "cos", "tan"

# plot the given function, default to sine
procedure main(args)
    func := map(args[1]) | "sin"
    if not func == ("sin" | "cos" | "tan") then func := "sin"

    # range of pixels for 300 points, y scaled at +/- 60, 4 pixel margins
    &window := open("Plotting", "g", "size=308,128", "canvas=hidden")

    # tan may cause runtime errors
    if func == "tan" then &error := 6

    color := "vivid orange"
    Fg(color)
    write(&window, "\n " || func)
```

```
    Fg("gray")
    DrawLine(2, 64, 306, 64)
    DrawLine(2, 2, 2, 126)
    DrawString(8, 10, "1", 8, 69, "0", 2, 126, "-1")
    DrawString(270, 76, left(points * 2 * &pi / 100, 6))

    Fg(color)
    every x := 0 to points do
        DrawPoint(xoff + x, base + yscale * func((2 * &pi * x) / xscale))

    WSync()
    WriteImage("../images/plot-" || func || ".png")
    close(&window)
end
```

```
prompt$ unicon -s plot-function.icn -x sin
```



See also:

acos, *atan*, *asin*, *cos*, *tan*

7.1.241 sort

`sort(x, i:1)` : *list*

`sort(x)` sorts the structure *x* in ascending order.

If *x* is a table, `sort(x, i)` uses *i* as a sort method indicator. If *i* is `1` or `2`, the table is sorted into a list of lists `[[key, value], ...]`. If *i* is `3` or `4` then the table is sorted into a single list of alternating keys and values. Sorting is by keys for odd values of *i* and element values for even values of *i*.

```
#
# sort.icn, sort a structure
#
link fullimag
procedure main()
    T := table()
    insert(T, "3", "x", "2", "y", "1", "z")
    write(fullimage(sort(T, 1)))
    write(fullimage(sort(T, 2)))
    write(fullimage(sort(T, 3)))
```

```
    write(fullimage(sort(T, 4)))
end
```

Sample run:

```
prompt$ unicon -s sort.icn -x
[["1","z"],<3>["2","y"],<4>["3","x"]]
[["3","x"],<3>["2","y"],<4>["1","z"]]
["1","z","2","y","3","x"]
["3","x","2","y","1","z"]
```

7.1.242 sortf

`sortf(x, i:1)` : *list*

`sortf(x, i)` sorts a list, record or set *x* using field *i* of each elements that has one. Elements that don't have and *i*th field are sorted in standard order, coming before elements that do have an *i*th field.

```
#
# sortf.icn, sort a structure by subfield index
#
link fullimag
procedure main()
    L := [[9,3,6], [4,2,7], [3,9,1], []]
    write("original: ", fullimage(L))
    write("field 1 : ", fullimage(sortf(L, 1)))
    write("field 2 : ", fullimage(sortf(L, 2)))
    write("field 3 : ", fullimage(sortf(L, 3)))
end
```

Sample run:

```
prompt$ unicon -s sortf.icn -x
original: [[9,3,6],[4,2,7],[3,9,1],[]]
field 1 : [[],[3,9,1],[4,2,7],[9,3,6]]
field 2 : [[],[4,2,7],[9,3,6],[3,9,1]]
field 3 : [[],[3,9,1],[9,3,6],[4,2,7]]
```

7.1.243 Span

`Span()` : *type*

Todo

entry for function Span

`Span()`

Sample run:

7.1.244 spawn

`spawn(CE, i, i)` : *thread*

`spawn(ce)` launches co-expression *ce* as an asynchronous thread, executed concurrently with the current co-expression. The two optional integers specify the thread's block and string region allocations. The defaults are 10% of the main thread head size.

```
#
# thread.icn, Demonstrate thread messaging
#
# requires Concurrency build of Unicon
#
procedure main()
    pr := create producerRace()
    cr := create consumerRace(pr)
    pTr := spawn(pr)
    cTr := spawn(cr)
    every wait(pTr | cTr)
    write("racing producer/consumer complete")
    write()

    p := create producer()
    c := create consumer(p)
    pT := spawn(p)
    cT := spawn(c)
    every wait(pT | cT)
    write("main complete")
end

#
# This code can easily trigger incorrect results due to a race condition
#

# send messages to the thread out-box
procedure producerRace()
    write("producer entry")
    every !6@>
end

# receive messages from the out-box of the producer thread
procedure consumerRace(T)
    write("consumer entry")
    while write(<@T)
end

# What follows is the suggested update from Jafar
# It alleviates the race condition where consumerRace
# can complete before the producerRace even starts
#
# an original capture:
#
# JMBPro:proj jafar$ ./thrd
# producer entry
# consumer entry
# racing producer/consumer complete

#
# This is the better code...
```

```
#

# send messages to the thread out-box
procedure producer()
    write("synch producer entry")
    every !6@>
    # produce &null (&null@>)  to signal the end
    @>
end

# receive messages from the out-box of the producer thread
procedure consumer(T)
    write("blocking consumer entry")
    # blocking receive.
    while write(\<<@T)
end
```

Sample run:

```
prompt$ unicon -s spawn.icn -x
consumer entry
producer entry
racing producer/consumer complete

synch producer entry
blocking consumer entry
1
2
3
4
5
6
main complete
```

See also:

create, *thread*, *wait*

7.1.245 sql

`sql(D, s)` : *integer*

`sql(db, query)` executes arbitrary SQL code on *db*. The *query* will be handled by vendor-specific features of the *ODBC* engine in use.

`sql()` is a power function that can leave a database in any state, as specified by the given SQL statement(s). Use care when feeding user entered *query* strings to the `sql()` function.

```
#
# sql.icn, demonstrate SQL statements passed to a ODBC database
#
# tectonics: ~/.odbc.ini setup required for [unicon]
#            assuming the [unicon] ODBC setup, SQLite3
#
procedure main()
    # mode 'o' open, ODBC SQL, default table and connection at defaults
```

```
    db := open("unicon", "o", "", "") | stop("no ODBC for \"unicon\"")

    # Information below was created as part of examples/odbc.icn
    # sql(db, "create table contacts (id integer primary key, name, phone)")

    # make a query
    write("SELECT name, phone")
    sql(db, "SELECT name, phone FROM contacts")

    while row := fetch(db) do {
        write("Contact: ", row.name, ", ", row.phone)
    }

    # make an ordered query
    write()
    write("ORDER BY name")
    sql(db, "SELECT name, phone FROM contacts ORDER BY name")

    while row := fetch(db) do {
        write("Contact: ", row.name, ", ", row.phone)
    }
    close(db)
end
```

Sample run:

```
prompt$ unicon -s sql.icn -x
SELECT name, phone
Contact: brian, 613-555-1212
Contact: jafar, 615-555-1213
Contact: brian, 615-555-1214
Contact: clint, 615-555-1215
Contact: nico, 615-555-1216

ORDER BY name
Contact: brian, 613-555-1212
Contact: brian, 615-555-1214
Contact: clint, 615-555-1215
Contact: jafar, 615-555-1213
Contact: nico, 615-555-1216
```

7.1.246 sqrt

`sqrt(r)` : *real*

`sqrt(r)` produces the square root of *r*. Will raise a runtime error for negative values of *r*.

```
#
# sqrt.icn, demonstrate square root function
#
procedure main()
    every r := 4 | 2 | 1 | 0.5 | 0.25 | 0 | -1 do
        write(r, " ", sqrt(r))
end
```

Sample run (ends with error demonstration):

```
prompt$ unicon -s sqrt.icn -x
4 2.0
2 1.414213562373095
1 1.0
0.5 0.7071067811865476
0.25 0.5
0 0.0

Run-time error 205
File sqrt.icn; Line 13
invalid value
offending value: -1.0
Traceback:
   main()
   sqrt(-1) from line 13 in sqrt.icn
```

7.1.247 stat

`stat(s)` : *record*

`stat(f)` returns a record of filesystem information for file (or path) *f*. See *lstat*.

Return record is:

```
record posix_stat(dev, ino, mode, nlink, gid, rdev, size,
                  atime, mtime, ctime, blksize, blocks, symlink)
```

The `atime`, `ctime`, and `mtime` fields may be formatted with the *ctime()* and *gtime()* functions. `mode` is a string form similar to the output of `ls -l`. `stat()` will fail if the file or path *f* does not exist.

```
#
# stat.icn, File status information, follows symbolic links.
#
link ximage
procedure main()
    every fn := "/usr/bin/cc" | "/usr/bin/gcc" do
        write("stat(", fn, "): ", ximage(stat(fn)))
end
```

Sample run:

```
prompt$ unicon -s stat.icn -x
stat(/usr/bin/cc): R_posix_stat_1 := posix_stat()
   R_posix_stat_1.dev := 2049
   R_posix_stat_1.ino := 4325501
   R_posix_stat_1.mode := "lrwxrwxrwx"
   R_posix_stat_1.nlink := 1
   R_posix_stat_1.uid := "root"
   R_posix_stat_1.gid := "root"
   R_posix_stat_1.rdev := 0
   R_posix_stat_1.size := 20
   R_posix_stat_1.atime := 1572162119
   R_posix_stat_1.mtime := 1452760995
   R_posix_stat_1.ctime := 1452760995
```

```
   R_posix_stat_1.blksize := 4096
   R_posix_stat_1.blocks := 0
   R_posix_stat_1.symlink := "/etc/alternatives/cc"
stat(/usr/bin/gcc): R_posix_stat_2 := posix_stat()
   R_posix_stat_2.dev := 2049
   R_posix_stat_2.ino := 4338962
   R_posix_stat_2.mode := "lrwxrwxrwx"
   R_posix_stat_2.nlink := 1
   R_posix_stat_2.uid := "root"
   R_posix_stat_2.gid := "root"
   R_posix_stat_2.rdev := 0
   R_posix_stat_2.size := 5
   R_posix_stat_2.atime := 1572162052
   R_posix_stat_2.mtime := 1455184062
   R_posix_stat_2.ctime := 1468380583
   R_posix_stat_2.blksize := 4096
   R_posix_stat_2.blocks := 0
   R_posix_stat_2.symlink := "gcc-5"
```

7.1.248 staticnames

`staticnames(CE, i)` : *string**

`staticnames(ce, i)` generates the names of local variables in co-expression *ce*, *i* levels up from the current procedure invocation. The default, level 0, generates names in the currently active procedure inside *ce*.

```
#
# staticames.icn, generate static variable names
#
global var, other
procedure main(arglist)
    static sv
    lv := 1
    var := 1
    every write(staticnames(&main))
end
```

Sample run:

```
prompt$ unicon -s staticnames.icn -x
sv
```

See also:

globalnames, *localnames*

7.1.249 stop

`stop(s|f, ...)`

`stop(args)` halts execution after writing out string arguments, followed by a newline, to *&errout*. If any argument is a `file`, subsequent string arguments are written to that file instead of `&errout`. The program exit status indicates that an error occurred.

```
#
# stop.icn, demonstrate program halt with message
#
procedure main()
    stop(&output, "stdout", &errout, "stderr")
end
```

Sample run:

```
prompt$ unicon -s stop.icn
```

```
prompt$ ./stop ; echo $?
stdout
stderr
1
```

7.1.250 StopAudio

`StopAudio(i)` : *integer* [*Audio*]

`StopAudio(i)` stops the thread playing audio on channel *i*. Returns 1 or triggers an error if *i* is not an integer. Attempting to stop non running channel numbers will be gracefully ignored.

```
#
# StopAudio.icn, terminates an audio stream playback.
#
procedure main()
    channel := PlayAudio("/home/btiffin/unicon/tests/unicon/handclap.ogg")
    write("Audio channel: ", channel)
    # handclap gets 1/3 of a second to complete
    delay(300)
    StopAudio(channel)

    channel := PlayAudio("/home/btiffin/unicon/tests/unicon/alert.wav")
    write("Audio channel: ", channel)
    # alert gets cut off after 1/2 second
    delay(500)
    StopAudio(channel)
end
```

Sample run (skipped, you probably wouldn't hear it anyway):

See also:

PlayAudio, *VAttrib*

7.1.251 string

`string(x)` : *string?*

`string(x)` converts *x* to a *string* and returns the result, or fails if the conversion is not possible.

```
#
# string.icn, demonstrate convert to string
#
procedure main()
    write(image(string(123)))
    write(image(string(123.123)))
    write(image(string('abbracadabra')))
    # this last one fails, no write occurs
    write(image(string([1,2,3])))
end
```

Sample run:

```
prompt$ unicon -s string.icn -x
"123"
"123.123"
"abcdr"
```

7.1.252 structure

`structure(CE)` : *any**

`structure(ce)` generates the values in the block region of *ce*. This heap holds structure types such as lists and tables.

```
#
# structure.icn, demonstrate looking in the heap for structure data
#
link ximage
procedure main()
    T := table(1, 2, 3, 4)
    L := [1,3]
    every write(ximage(structure(&current)))
end
```

Sample run:

```
prompt$ unicon -s structure.icn -x
L-1 := list(0)
L-2 := list(0)
L-3 := list(0)
T1 := table(&null)
   T1[1] := 2
   T1[3] := 4
L1 := list(2)
   L1[1] := 1
   L1[2] := 3
```

See also:

cofail, *fieldnames*, *globalnames*, *keyword*, *localnames*, *paramnames*, *staticnames*, *variable*

7.1.253 Succeed

`Succeed()` : *empty string*

`Succeed()` places a fence on pattern matching, by matching the empty string when going left to right, and forcing the scanner to reverse direction forwards during backtracking.

> **Attention:** This function will almost always cause an endless loop and should NOT be used. Except for wasting CPU when the pattern is known to match on a first pass, or endlessly spinning when the pattern matcher needs to backtrack. Use *Fence* (or possibly *fail*) instead.

```
#
# Succeed.icn, an anti-example, FOR WARNING PURPOSES ONLY
#
procedure main()
    # DON'T DO THIS, it causes an endless loop
    # result := "sstring" ?? Any('rst') || Succeed() || Span('rt') || "ing"

    # Do this instead
    result := "sstring" ?? Any('rst') || Fence() || Span('rt') || "ing"
    write(image(result))

    # the short of it is, DON'T USE Succeed()
end
```

Sample run (skipped, as this pattern function is a no-win trap):

See also:

fail, *Fence*

7.1.254 Swi

`Swi()` : *type*

An unsupported Archimedes specific feature.

`Swi()`

Sample run:

7.1.255 symlink

`symlink(s, s)` : *?* [*POSIX*]

`symlink(src, dst)` creates a symbolic (soft) file system link *dst* pointing to *src*.

```
#
# symlink.icn, demonstrate the POSIX symlink() function
#
procedure main()
    symlink("tt.src", "tt.dst")
end
```

Sample run:

```
prompt$ unicon -s symlink.icn -x
```

7.1.256 sys_errstr

`sys_errstr(i)` : *string*

`sys_errstr(i)` produces the error string corresponding to code *i*, or if not given, a code obtained from *&errno*.

```
#
# sys_errstr.icn, get string from system error code
#
procedure main()
    write(sys_errstr(1))
    write(sys_errstr(2))
    # errno will be 0, which is Success
    write(sys_errstr(&errno))
end
```

Sample run:

```
prompt$ unicon -s sys_errstr.icn -x
Operation not permitted
No such file or directory
Success
```

7.1.257 system

`system(x, f:&input, f:&output, f:&errout, s)` : *integer*

`system(x, f1, f2, f3, waitflag)` executes a program in a separate process. *x* can be a string or a list of strings. For a single string, the command is processed by the platform's command interpreter. For a list, each member is a separate argument, the first is the program and all subsequent parameters are passed as arguments to the command. The `file` arguments are used for standard in, standard out and standard error for the new process. If the *waitflag* is `"nowait"`, then `system()` returns immediately and the result is the new process id. Otherwise Unicon will wait for the new program to finish and the result is a status value.

```
#
# system.icn, execute an external program
#
procedure main()
    r := system("ls -l nonexistent-file system.icn")
    write(r)
end
```

Sample run:

```
prompt$ unicon -s system.icn -x
ls: cannot access 'nonexistent-file': No such file or directory
-rw-rw-r-- 1 btiffin btiffin 255 Oct 14  2016 system.icn
2
```

7.1.258 syswrite

`syswrite(f, s)` : *integer*

`syswrite(f, s)` causes a low level unbuffered write of the string *s* to file *f*.

```
#
# syswrite.icn, demonstrate low level write
#
procedure main()
    r := syswrite(&output, "testing ")
    write(r)
end
```

Sample run:

```
prompt$ unicon -s syswrite.icn -x
testing 8
```

7.1.259 Tab

`Tab(i)` : *string*

A SNOBOL inspired pattern matching function.

`Tab(i)` places the cursor at position *i*, and returns the characters between the current position and the new position.

```
#
# Tab.icn, SNOBOL pattern return characters, current to position
#
procedure main()
    # match characters from current (1) to new cursor, set at 4.
    # returns characters from position 1,2,3 with cursor now at 4.
    "string" ?? Tab(4) => result || .> curs
    write(image(result), " ", curs)
end
```

Sample run:

```
prompt$ unicon -s Tab.icn -x
"str" 4
```

See also:

Rtab

7.1.260 tab

`tab(i:0)` : *string?*

`tab(i)` sets *&pos* to *i* and returns the substring of *&subject* between the old and new positions. `tab(0)` moves the position to the end of the string. This function reverts settings the position to its old value if it is resumed.

```
#
# tab.icn, demonstrate string scanning position change tab function
#
procedure main()
    s := "this is a test"
    s ? out := tab(5)
    write(":", out, ":")
end
```

Sample run:

```
prompt$ unicon -s tab.icn -x
:this:
```

7.1.261 table

`table(k, v, ..., x)` : *table*

`table(x)` creates a table with default value *x*. If *x* is a structure, all references to the default value refers to the same same value, not a separate copy for each key. Given more than one argument, `table(k,v,...,x)` takes alternating keys and values to initialize the table.

```
#
# table.icn, demonstrate creation of table function
#
procedure main()
    # set initial keys and values
    T := table(1, "a", 2, "b", c, "3")
    write(T[1])
    # create with default value
    T := table(42)
    write(T[1])
end
```

Sample run:

```
prompt$ unicon -s table.icn -x
a
42
```

7.1.262 tan

`tan(r)` : *real*

`tan(r)` returns the Tangent of the given angle, *r* (in radians). Tangent is the ratio of the Opposite/Adjacent lines of a right angle triangle. `tan(r)` will cause a runtime error for *r* values that are multiples of *&pi* / 2 radians, the result being imaginary *infinity*.

```
#
# tan.icn, demonstrate the the Tangent function
#
```

```
procedure main()
    write("tan(r): Domain: -2 * &pi <= r <= &pi * 2, Range: all real, imaginary at &
↪pi/2 * i")
    every r := 0.0 to &pi * 2 by &pi/4 do {
        write(left("tan(" || r || ")", 24), " radians = ", tan(r))
    }
end
```

Sample run:

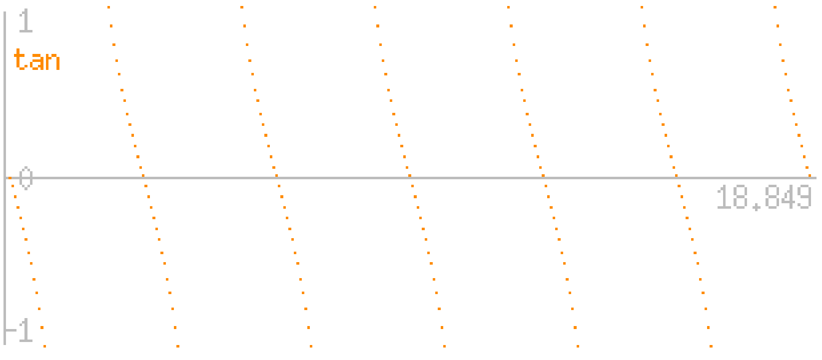
```
prompt$ unicon -s tan.icn -x
tan(r): Domain: -2 * &pi <= r <= &pi * 2, Range: all real, imaginary at &pi/2 * i
tan(0.0)                 radians = 0.0
tan(0.7853981633974483)  radians = 0.9999999999999999
tan(1.570796326794897)   radians = 1.633123935319537e+16
tan(2.356194490192345)   radians = -1.0
tan(3.141592653589793)   radians = -1.224646799147353e-16
tan(3.926990816987241)   radians = 0.9999999999999997
tan(4.71238898038469)    radians = 5443746451065123.0
tan(5.497787143782138)   radians = -1.0
tan(6.283185307179586)   radians = -2.449293598294706e-16
```

Graphical plot:

```
#
# plot-function, trigonometric plotting, function from command line
#
$define points 300
$define xoff   4
$define base   64
$define yscale 60
$define xscale 100

invocable "sin", "cos", "tan"

# plot the given function, default to sine
procedure main(args)
    func := map(args[1]) | "sin"
    if not func == ("sin" | "cos" | "tan") then func := "sin"

    # range of pixels for 300 points, y scaled at +/- 60, 4 pixel margins
    &window := open("Plotting", "g", "size=308,128", "canvas=hidden")

    # tan may cause runtime errors
    if func == "tan" then &error := 6

    color := "vivid orange"
    Fg(color)
    write(&window, "\n " || func)

    Fg("gray")
    DrawLine(2, 64, 306, 64)
    DrawLine(2, 2, 2, 126)
    DrawString(8, 10, "1", 8, 69, "0", 2, 126, "-1")
    DrawString(270, 76, left(points * 2 * &pi / 100, 6))

    Fg(color)
    every x := 0 to points do
```

```
        DrawPoint(xoff + x, base + yscale * func((2 * &pi * x) / xscale))

    WSync()
    WriteImage("../images/plot-" || func || ".png")
    close(&window)
end
```

```
prompt$ unicon -s plot-function.icn -x tan
```



7.1.263 Texcoord

`Texcoord()` : *type*

Todo

entry for function Texcoord

`Texcoord()`

Sample run:

7.1.264 Texture

`Texture()` : *type*

Todo

entry for function Texture

`Texture()`

Sample run:

7.1.265 TextWidth

`TextWidth()` : *type*

Todo

entry for function TextWidth

`TextWidth()`

Sample run:

7.1.266 Translate

`Translate()` : *type*

Todo

entry for function Translate

`Translate()`

Sample run:

7.1.267 trap

`trap(s, p)` : *procedure*

`trap(s, proc)` sets up a signal handler for the signal *s* (by name). The old handler, if any, is returned. If *proc* is null, then handler is reset to a default value.

```
#
# trap.icn, set signal handlers
#
procedure main()
    p := trap("SIGINT", gotint)
    write(image(p))
    p := trap("SIGINT")
    write(image(p))
end

procedure gotint(a)
    write(image(a))
    write("^C captured")
end
```

Sample run:

```
prompt$ unicon -s trap.icn -x
&null
procedure gotint
```

7.1.268 trim

`trim(s, c:'  ', i:-1)` : *string*

`trim(s, c, i)` removes any of the characters in *c* from the ends of *s*. *i* specifies from where:

- `-1` trailing
- `1` leading
- `0` both ends.

```
#
# trim.icn, demonstrate the trim function
#
procedure main()
    s := "  this is a test  "
    write(":", trim(s), ":")
    write(":", trim(s,,1), ":")

    # note, not \t, but "t", trim leading and trailing ts and spaces
    write(":", trim(s,' t',0), ":")
end
```

Sample run:

```
prompt$ unicon -s trim.icn -x
:  this is a test:
:this is a test  :
:his is a tes:
```

7.1.269 truncate

`truncate(f, i)` : *?*

`truncate(f, len)` changes the file *f* which may be a filename or an open file, to be no longer than length *len*. `truncate()` does not work on database, network connection, pipe, or window resources. Files already shorter than *i* will be filled with padding, usually zero bytes.

```
#
# truncate.icn, truncate a file to a given length
#
procedure main()
    if truncate("tt.tt", 256) then
        writes("did") else writes("didn't")
    write(" truncate")
end
```

Sample run:

```
prompt$ unicon -s truncate.icn -x
didn't truncate
```

7.1.270 trylock

`trylock(x)` : *x?*

`trylock(m)` locks the mutex *x* or the mutex associated with the thread-safe object *x*, if it is not already locked.

```
#
# trylock.icn, lock a mutex if not already locked
#
procedure main()
    x := 1
    mtx := mutex()
    if trylock(mtx) then {
        write("locked mtx")
        x := x + 1
        unlock(mtx)
    }
end
```

Sample run:

```
prompt$ unicon -s trylock.icn -x
locked mtx
```

7.1.271 type

`type(x)` : *string*

Returns the type of x.

`type(x)` returns a string representation of the type name of x.

```
#
# type.icn, demonstrate the type() function
#
procedure main()
    # list of expressions to evaluate
    types := [
        "type(1)",
        "type(type(1))",
        "type(\"abc\")",
        "type(L)",
        "type(T)",
        "type(S)",
        "type(R)",
        "type(C)",
        "type(&window)",
        "type(file)"
    ]

    # generate some code to evaluate the expressions
    prog := "generated-type-program.icn"
    tmp := open(prog, "w") | stop("Can't open " || prog || " for writing")
    write(tmp, "record rec(a,b)")
    write(tmp, "procedure main()")
    write(tmp, "    L := []")
```

```
    write(tmp, "    T := table()")
    write(tmp, "    S := set()")
    write(tmp, "    R := rec(1,2)")
    write(tmp, "    C := create 1")
    write(tmp, "    &window := open(\"window\", \"g\", \"canvas=hidden\")")
    write(tmp, "    file := open(&file, \"r\")")
    every t := !types do {
        write(tmp, "    write(right(", image(t), " || \": \"", ",16), ", t, ")")
    }
    write(tmp, "    close(file)")
    write(tmp, "    close(&window)")
    write(tmp, "end")
    close(tmp)

    # pipe in the results
    p := open("unicon -quiet -s -v0 " || prog || " -x", "p")
    while write(read(p))
    close(p)

    # get rid of the generated source and executable
    remove(prog)
    remove(prog[1:-4])
end
```

Sample run:

```
prompt$ unicon -s type.icn -x
       type(1): integer
 type(type(1)): string
   type("abc"): string
       type(L): list
       type(T): table
       type(S): set
       type(R): rec
       type(C): co-expression
 type(&window): window
    type(file): file
```

7.1.272 umask

`umask(integer)` : *integer* [*POSIX*]

`umask(u)` sets the `umask` for the current process to *u*, a nine bit encoding of read, write, execute permissions for user, group and world access. Each bit of `umask` turns off that access, by default, for newly created files. `umask(u)` returns the old value.

```
#
# umask.icn, set default file permission mask
#
procedure main()
    write(umask(0))
end
```

Sample run:

```
prompt$ unicon -s umask.icn -x
2
```

7.1.273 Uncouple

`Uncouple()` : *type*

Todo

entry for function Uncouple

`Uncouple()`

Sample run:

7.1.274 unlock

`unlock(x)` : *x*

`unlock(m)` unlocks the *mutex* *m* or the mutex associated with the thread-safe object *x*

```
#
# unlock.icn, unlock a mutex
#
procedure main()
    x := 1
    mtx := mutex()
    if trylock(mtx) then {
        write("locked mtx")
        x := x + 1
        unlock(mtx)
    }
end
```

Sample run:

```
prompt$ unicon -s unlock.icn -x
locked mtx
```

7.1.275 upto

`upto(c, s:&subject, i:1, i:0)` : *integer**

String scanning function `upto(c, s, i1, i2)` generates the sequence of integer positions in *s* scanning forward for characters in the cset *c* between indexes *i1* and *i2*. `upto()` fails if there is not such position.

```
#
# upto.icn, demonstrate the string scanning upto generator
#
procedure main()
    s := "  this is a test   "
    write(s)
    s ? every i := upto('st') do write(i, " ", &subject[i])
end
```

Sample run:

```
prompt$ unicon -s upto.icn -x
  this is a test
3 t
6 s
9 s
13 t
15 s
16 t
```

7.1.276 utime

`utime(s, i, i)` : *null*

`utime(f, atime, mtime)` sets the access and modification time of filename *f* to *atime* and *mtime* respectively. `ctime` is set to the current time.

```
#
# utime.icn, demonstrate setting file access times
#
procedure main()
    atime := &now
    mtime := &now - 4
    utime("newfile.txt", atime, mtime)
end
```

Sample run:

```
prompt$ unicon -s utime.icn -x
```

7.1.277 variable

`variable(s, CE:¤t, i:0)` : *any?*

`variable(s, ce, i)` returns a reference to the variable name *s* from the co-expression *ce*, *i* levels up from the current procedure frame. Name search is local to *ce* then global to the program that created *ce*.

```
#
# variable.icn, demonstrate reflective variable name lookup
#
procedure main()
```

```
    this := 42
    v := variable("this")
    write(v)

    # assignment made a copy of the integer
    this := 44
    write(v)

    # structures stay as references during assignment
    L := [1,2,3]
    l := variable("L")
    L[2] := 4
    write(l[2])
end
```

Sample run:

```
prompt$ unicon -s variable.icn -x
42
42
4
```

7.1.278 VAttrib

`VAttrib()` : *type*

Todo

entry for function VAttrib

`VAttrib()`

Sample run:

7.1.279 wait

`wait(x)` : *?*

`wait(x)` waits for *x*. If *x* is a thread, `wait()` waits for *x* to finish. If *x* is a condition variable, *condvar*, then `wait()` waits until that variable is signalled from another thread.

```
#
# wait.icn, demonstrate thread and condvar wait
#
procedure main()
    write("waiting ", gettimeofday().usec)
    th := thread(every 1 to 100000)
    wait(th)
    write("no longer waiting ", gettimeofday().usec)
end
```

Sample run:

```
prompt$ unicon -s wait.icn -x
waiting 384876
no longer waiting 390416
```

7.1.280 WAttrib

WAttrib (*w*, *x*, ...) → x

WAttrib(w, s, w2, s2) will change the attribute of window *w* to the value specified in *s* and the attribute of window *w2* to the pair specified in *s2*. **WAttrib(w, s)** will retrieve the given attribute if no "set" operation is specified, "dy" versus "dy=yoffset" in the string *s*.

Todo

this graphics section is woefully incomplete

Attributes include:

Canvas Attributes

- size=wid,hgt
- pos=x,y
- canvas=normal|hidden
- windowlabel=string
- inputmask=string
- pointer=arrow,clock,etc
- pointerx=x
- pointery=y
- display=device (X11)
- depth=integer (# of bits)
- displaywidth=x
- displayheight=y
- image=string

Graphics contexts

- fg=colour
- bg=colour
- font=name
- fheight=integer

- fwidth=integer
- leading=integer (vertical pixels between lines)
- ascent=integer
- descent=integer
- drawop=operation (copy, reverse)
- fillstyle=type (stippled, opaquestippled)
- pattern=pattern (bits,#hex)
- linestyle=style (onoff, doubledash)
- linewidth=integer
- clipx=integer
- clipy=integer
- clipw=integer
- cliph=integer
- dx=integer (coordinate translation)
- dy=integer (coordinate translation)

3D attributes

- dim=
- pick=
- texmode=
- slices=
- rings=
- normode=

Example:

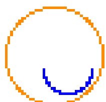
```
#
# WAttrib, demonstrate window attribute control
#
procedure main()
    w := open("WAttrib", "g", "size=40,40", "canvas=hidden")

    write("Default Attributes (given size=40,40)")
    write("-------------------------------------")
    alist := ["fg", "bg", "size", "pos", "canvas", "windowlabel",
              "inputmask", "pointer", "pointerx", "pointery",
              "display", "depth", "displaywidth", "displayheight",
              "font", "fheight", "fwidth",
              "leading", "ascent", "descent", "drawop",
              "fillstyle", "pattern", "linestyle", "linewidth",
              "clipx", "clipy", "clipw", "cliph", "dx", "dy"]
    every a := !sort(alist) do write(left(a, 16), ": ", WAttrib(w, a))

    WAttrib(w, "fg=vivid orange")
    DrawCircle(w, 20, 20, 18)
```

```
    WFlush(w)

    # change the display offsets along with colour
    WAttrib(w, "dx=5", "dy=5", "fg=blue")
    DrawCircle(w, 20, 20, 9, 0.0, &pi)
    WSync(w)

    # save image for the document
    WriteImage(w, "../images/WAttrib.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s WAttrib.icn -x
Default Attributes (given size=40,40)
-------------------------------------
ascent          : 11
bg              : white
canvas          : hidden
cliph           :
clipw           :
clipx           :
clipy           :
depth           : 24
descent         : 2
display         : localhost:10.0
displayheight   : 900
displaywidth    : 1600
drawop          : copy
dx              : 0
dy              : 0
fg              : black
fheight         : 13
fillstyle       : solid
font            : fixed
fwidth          : 6
inputmask       :
leading         : 13
linestyle       : solid
linewidth       : 1
pattern         : black
pointer         : left ptr
pointerx        : -27376
pointery        : 24305
size            : 40,40
windowlabel     : WAttrib
```

Almost the same image as *WFlush*, but `dx` and `dy` are offset a bit on the blue half circle.



See also:

WSync, *Graphics*

7.1.281 WDefault

`WDefault()` : *type*

Todo

entry for function WDefault

`WDefault()`

Sample run:

7.1.282 WFlush

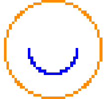
WFlush(*w*) → window

WFlush(w) flushes window *w* output on window systems that buffer text and graphic output, without waiting for all pending events. Window data is automatically flushed during events that block. This function is a no-op on systems that do not buffer graphical output.

```
#
# WFlush, demonstrate windowing buffer flush
#
procedure main()
    w := open("WFlush", "g", "size=40,40", "canvas=hidden")

    Fg(w, "vivid orange")
    DrawCircle(w, 20, 20, 18)
    WFlush(w)

    Fg(w, "blue")
    DrawCircle(w, 20, 20, 9, 0.0, &pi)
    WFlush(w)

    # save image for the document
    WriteImage(w, "../images/WFlush.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s WFlush.icn -x
```



See also:

WSync

7.1.283 where

`where(f)` : *integer*

`where(f)` returns the current offset position in file *f*. `where()` fails for window and network resources. The beginning of a file is offset `1`.

```
#
# where.icn, demonstrate file offset reporting
#
procedure main()
    f := open("newfile.txt", "r")
    read(f)
    write(where(f))
    close(f)
end
```

Sample run:

```
prompt$ unicon -s where.icn -x
16
```

7.1.284 WinAssociate

`WinAssociate(s)` : *string* [*Windows*]

`WinAssociate(s)` returns the application name associated with the string file extension *s*. Requires Unicon for Windows, *non-portable*.

```
#
# WinAssociate.icn, return the application name associated with extension
# Requires Unicon for Windows
#
procedure main()
    write(WinAssociate("test"))
end
```

Sample run: (not supported on the OS that builds this document)

```
prompt$ unicon -s WinAssociate.icn -x

Run-time error 121
File WinAssociate.icn; Line 13
function not supported
Traceback:
   main()
   WinAssociate("test") from line 13 in WinAssociate.icn
```

See Unicon Technical Report 7, http://unicon.org/utr/utr7.html

The Windows only functions are no longer recommended for new Unicon developments. The *VIB* and GUI class library should be used to create cross platform programs.

7.1.285 WinButton

`WinButton(w, s, x,y, wd,ht)` : *string* [*Windows*]

`WinButton(w, s, x, y, wd, ht)` installs a pushbutton with label *s* at *x,y*, *wd* pixels wide and *ht* pixels high, on window *w*. When pressed the button label *s* is placed on the event queue. *Non-portable*.

```
#
# WinButton.icn, install a pushbutton, with label, on window
# only works on supported systems
#
link enqueue, evmux
procedure main()
    window := open("WinButton", "g", "size=90,60", "canvas=hidden")
    WinButton(window, "Button", 5,5, 30,15)
    Enqueue(window, &lpress, 11, 14, "", 2)
    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    close(window)
end
```

Sample run (not supported on the OS that builds this document):

```
prompt$ unicon -s WinButton.icn -x

Run-time error 121
File WinButton.icn; Line 15
function not supported
Traceback:
   main()
   WinButton(window_1:1(WinButton),"Button",5,5,30,15) from line 15 in WinButton.icn
```

See Unicon Technical Report 7, http://unicon.org/utr/utr7.html.

The Windows only functions are no longer recommended for new Unicon developments. The *VIB* and GUI class library can and should be used to create cross platform programs.

See also:

Event

7.1.286 WinColorDialog

`WinColorDialog()` : *type*

Todo

entry for function WinColorDialog

`WinColorDialog()`

Sample run:

7.1.287 WindowContents

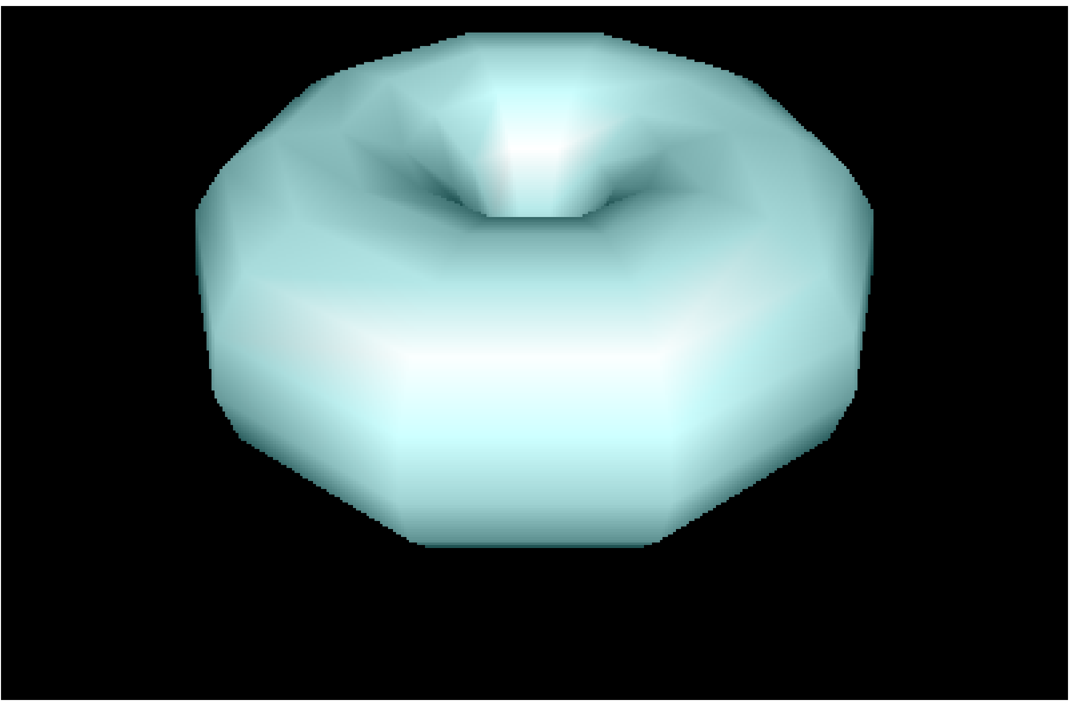
`WindowContents(w)` : *list* [*3D graphics*]

`WindowContents(w)` returns a *List (arrays)* of current 3D window *w* contents.

```
#
# WindowContents.icn, demonstrate WindowContents list
#
link ximage
procedure main()
    window := open("WindowContents", "gl", "bg=black", "buffer=on",
                    "size=400,260", "dim=3")#, "canvas=hidden")
    WAttrib(window,"light0=on, ambient blue-green","fg=specular white")

    # A torus
    DrawTorus(window, 0.0,0.19,-2.2, 0.3,0.4)

    # show the window contents list
    Refresh(window)
    write(ximage(WindowContents(window)))

    # save the image
    WriteImage(window, "../images/WindowContents.png")

    close(window)
end
```

Sample run:

```
prompt$ unicon -s WindowContents.icn -x
L2 := list(3)
   L2[1] := L3 := list(3)
      L3[1] := "dim"
      L3[2] := 336
      L3[3] := 3
   L2[2] := L4 := list(7,65535)
      L4[1] := "Fg"
      L4[2] := 160
      L4[3] := "specular"
   L2[3] := R_gl_torus_1 := gl_torus()
      R_gl_torus_1.name := "DrawTorus"
      R_gl_torus_1.code := 149
      R_gl_torus_1.x := 0.0
      R_gl_torus_1.y := 0.19
      R_gl_torus_1.z := -2.2
      R_gl_torus_1.radius1 := 0.3
      R_gl_torus_1.radius2 := 0.4
```



7.1.288 WinEditRegion

`WinEditRegion(s, s2, x,y, wd,ht)` : *string*

WinEditRegion(w, s, s2, x,y, wd,ht) manipulates a Windows edit region named s. This flexible editor is limited to text that is < 32Kbytes in length. The operation performed depends on argument s2. If argument s2 is omitted, WinEditRegion(s) returns a string containing the current contents of region s. If s2 is supplied and does not start with a !, it is a string to be edited; lines are separated by rn. s2 strings starting with ! are commands:

- WinEditRegion(s, "!clear") clears the current selection.
- WinEditRegion(s, "!copy") copies the current selection.
- WinEditRegion(s, "!cut") cuts the current selection.
- WinEditRegion(s, "!paste") pastes the current selection.
- WinEditRegion(s, "!modified") succeeds if region s has been modified since last assigned.
- WinEditRegion(s, "!setsel") sets the selection using parameters x and y as indices..
- WinEditRegion(s, "!undo") undoes the previous operation, if possible.

Sample run:

7.1.289 WinFontDialog

`WinFontDialog()` : *type*

Todo

entry for function WinFontDialog

`WinFontDialog()`

Sample run:

7.1.290 WinMenuBar

`WinMenuBar()` : *type*

Todo

entry for function WinMenuBar

`WinMenuBar()`

Sample run:

7.1.291 WinOpenDialog

`WinOpenDialog(w, s1, s2, i, s3,j, s4)` : *string*

`WinOpenDialog(w, s1, s2, i, s3,j, s4)` displays a typical open dialog to perform file selection for reading. *s1* is the caption, *s2* the default value and *i* is the text entry field size. *s3* and *j* specify the filter string and its index. *s3* is a string of alternating names and filters, separated and ending in |, of the form `"name1|filter1|name2|filter2|...|"`. *s3* defaults to `"All Files(*.*)|*.*|"`. *j* supplies the default extension index within *s3*; it defaults to first pair in filter string. *s4* is the directory to show when the dialog is opened; it defaults to use Windows version-specific rules. Returns the file name chosen. Fails if the user selects Cancel.

```
#
# WinOpenDialog.icn, open a native Windows file open dialog box.
#
#
procedure main()
    w := open("WinOpenDialog", "g")
    fn := WinSaveDiaglog(w, "Open", "SomeData.dat", 24,
                              "Dat files(*.dat)|*.dat|All (*.*)|*.*|", 1
                              "../data")
    close(w)
end
```

Sample run not available, code untested.

7.1.292 WinPlayMedia

`WinPlayMedia(w, s1,...)` : *null*

`WinPlayMedia(w, s1, s2)` plays media file *s1* followed by *s2* for each argument. Media can be `.wav`, `.mdi`, `.rmi` or if unrecognized the string passed to MCI for supported media files.

Fails on the first media file that cannot be played.

```
#
# WinPlayMedia.icn, play media files with native Windows
#
#
procedure main()
    w := open("WinPlayMedia", "g")
    WinPlayMedia(w, "alert.wav", "relax.mdi")
    close(w)
end
```

Sample run not available, code untested.

Deprecated, see *PlayAudio* for the recommended cross-platform solution.

7.1.293 WinSaveDialog

`WinSaveDialog(w, s1, s2, i, s3,j, s4)` : *string*

`WinSaveDialog(w, s1, s2, i, s3,j, s4)` opens a typical save dialog to perform file selection for writing. *s1* is the caption, *s2* the default value and *i* is the text entry field size. *s3* and *j* specify the filter string and its index. *s3* is a string of alternating names and filters, separated and ending in |, of the form `"name1|filter1|name2|filter2|...|"`. *s3* defaults to `"All Files(*.*)|*.*|"`. *j* supplies the default extension index within *s3*; it defaults to first pair in filter string. *s4* is the directory to show when the dialog is opened; it defaults to use Windows version-specific rules. Returns the file name chosen. Fails if the user selects Cancel.

```
#
# WinSaveDialog.icn, open a native Windows file save dialog box.
#
#
procedure main()
    w := open("WinSaveDialog", "g")
    fn := WinSaveDiaglog(w, "Save to", "NewFile.dat", 24,
                         "Dat files(*.dat)|*.dat|All (*.*)|*.*|", 1
                         "../data")
    close(w)
end
```

Sample run not available, code untested.

7.1.294 WinScrollBar

`WinScrollBar()` : *type*

Todo

entry for function WinScrollBar

`WinScrollBar()`

Sample run:

7.1.295 WinSelectDialog

`WinSelectDialog(w, s1, buttons)` : *string* [*Windows Unicon*]

`WinSelectDialog(w, title, L)` displays a Windows native dialog box on window *w*, labelled *title*, offering a selection from a set of choices in the form of buttons, *[”choice1”, “choice2”]* (as list of strings). *Event()* will return the choice as a string.

```
#
# WinSelectDialog.icn, a Windows native choice selection dialog
#
# Deprecated in favour of the cross-platform Unicon GUI classes
#
procedure main()
    w := open("WinSelectDialog", "g", "size=400,300")
    choice := WinSelectDialog(w, "choices", "Choice A", "Choice B")
    close(w)
end
```

Sample run not available.

Deprecated. The native Windows procedures are now deprecated in favour of the cross-platform Unicon *gui* classes.

7.1.296 write

`write(s|f, ...)` : *string|file*

`write(arglist)` outputs strings, followed by a newline to a file or files. Strings are written to the closest preceding file argument, defaulting to *&output*. A newline is written whenever a non-initial file arguments directs output to a different file, or after the last argument. `write()` returns the last argument.

```
#
# write.icn, demonstrate write (adds newline)
#
procedure main()
    write("first", &errout, "std err", &output, "back to std out")
end
```

Sample run:

```
prompt$ unicon -s write.icn -x
first
std err
back to std out
```

7.1.297 WriteImage

`WriteImage(w, s, x:0, y:0, wid:0, h:0)` : *window* [*Graphics*]

`WriteImage(w, s, x, y, wid, h)` saves an image of dimensions *wid, h* from window *w* (defaults to *&window*), at offset *x, y* to a file named *s*. The default is to save the entire window. The output format is written according the extension, GIF, JPG, BMP, PNG (depending on available library during Unicon build). Platform specific formats, such as XBM or XPM may also be supported. `WriteImage()` fails if *s* cannot be opened for writing.

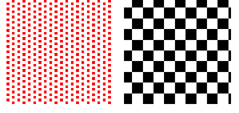
Almost all included graphics are generated during the builds of this documentation and saved with `WriteImage()`, even the "no-image" image.

Only the canvas image is included, window decorations (as provided by desktop window managers) are not part of images saved by `WriteImage()`.

```
#
# WriteImage.icn, demonstrate saving images
#
procedure main()
    &window := open("Pattern", "g",
                    "fillstyle=stippled",
                    "size=85,40", "canvas=hidden")
    # A width 4 pattern, with bit patterns of 2, 8, 2, 8
    Pattern(&window, "4,2,8,2,8")
    Fg(&window, "red")
    FillRectangle(&window, 0, 0, 40, 40)

    # built in checker board pattern
    Pattern("checkers")
    Fg("black")
    FillRectangle(45, 0, 40, 40)

    # save image for the document
    WSync()
    WriteImage("../images/WriteImage.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s WriteImage.icn -x
```



7.1.298 writes

`writes(s|f, ...)` : *string|file*

`writes(arglist)` outputs strings to a file or files. Strings are written to the closest preceding file argument, defaulting to *&output*. `write()` returns the last argument.

```
#
# writes.icn, demonstrate writes
#
procedure main()
    # first to default &output, then a string to &errout, then &output
    # stream buffers for output and error can cause random display order
    writes("first ", &errout, " to std err ", &output, " to std out ")
end
```

Sample run: standard stream buffering might cause *&errout* to display separately from *&output*.

```
prompt$ unicon -s writes.icn -x
first  to std out  to std err
```

7.1.299 WSection

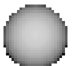
`WSection(w, s)` : *record*

`WSection(w, s)` starts a new window section named *s* on 3D window *w*. Returns a display list section marker record. During window *Refresh*, if the section marker's `skip` field is `1`, the section is skipped. The section name *s* is produced by *&pick* is the graphic object in the block is clicked on when the attribute `"pick=on"` is set. `WSection(w)` marks the end of a named section. `WSection` blocks can be nested.

```
#
# WSection.icn, demonstrate 3D pick object ID during windowing event
#
link enqueue, evmux
procedure main()
    window := open("WSection", "gl", "size=90,60")
    WAttrib(window, "pick=on")

    # mark a named 3D object
    write(image(WSection(window, "sphere")))
    DrawSphere(window, 0.0, 0.19, -2.2, 0.3)
    Refresh(window)
    WSection(window)

    # insert an event into the queue, left press, 2ms interval, on sphere
    #Enqueue(window, &lpress, 45,20, "", 2)

    e := Event(window)
    write(image(e))

    # a side effect of the Event function is keywords settings
    write("&x:", &x)
    write("&y:", &y)
    write("&row:", &row)
    write("&col:", &col)
    every write("&pick:", &pick)

    WriteImage(window, "../images/WSection.png")
    close(window)
end
```

Sample output, having clicked on the small sphere:

```
prompt$ unicon -s WSection -x
record gl_mark_1(7)
-1
&x:43
&y:21
&row:2
&col:8
&pick:sphere
```



7.1.300 WSync

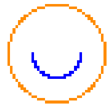
`WSync(w, s)` : *w*

`WSync(w, s)` synchronizes the program with the server attached to window *w* on systems that employ a client-server model. Output to the window is flushed, and `WSync()` waits for a reply from the server indicating that all output has been processed. If *s* is `"yes"`, all pending events on *w* are discarded. `WSync()` is a no-op on windowing systems that do not use a client-server model.

```
#
# WSync.icn, demonstrate window buffer sync
#
procedure main()
    w := open("WSync", "g", "size=40,40", "canvas=hidden")

    Fg(w, "vivid orange")
    DrawCircle(w, 20, 20, 18)
    WSync()

    Fg(w, "blue")
    DrawCircle(w, 20, 20, 9, 0.0, &pi)
    # sync windowing system and discard all pending events
    WSync(w, "yes")

    # save image for the document
    WriteImage(w, "../images/WSync.png")
    close(w)
end
```

Sample run:

```
prompt$ unicon -s WSync.icn -x
```



See also:

Refresh, *WFlush*

CHAPTER EIGHT

KEYWORDS

Keywords in *Icon* and Unicon provide access to wide assortment of environmental data, constants, and feature settings and control. All keywords start with an ampersand symbol &.

History sidetrip: Unicon keywords are an evolution of *SNOBOL* keywords. In SNOBOL, some keywords were also called trapped variables, as accessing the keywords would cause special processing in SNOBOL.

8.1 Unicon Keywords

8.1.1 &allocated

- Read-only
- Generates *integers*

A generator that produces 4 *integers*; the cumulative number of bytes used for the

- heap
- static
- string
- block

memory regions.

```
#
# show initial allocations, create a string, reshow allocations
#
procedure main()
    allocated()
    s := repl(&letters, 1000)
    write("\nAfter string creation")
    allocated()
end
```

```
# Display current memory region allocations
procedure allocated()
    local allocs

    allocs := [] ; every put(allocs, &allocated)

    write("&allocated elements ", *allocs)
    write("---------------------")
    write("Heap   : ", allocs[1])
    write("Static : ", allocs[2])
    write("String : ", allocs[3])
    write("Block  : ", allocs[4])
end
```

Giving:

```
prompt$ unicon -s allocated.icn -x
&allocated elements 4
---------------------
Heap   : 34528
Static : 0
String : 0
Block  : 34528

After string creation
&allocated elements 4
---------------------
Heap   : 86860
Static : 0
String : 52052
Block  : 34808
```

See also:

&collections, *&storage*, *®ions*

8.1.2 &ascii

- Read-only
- Produces a *Cset* that includes the 128 *ASCII* characters.

```
#
# Display some of the &ascii keyword, not all, due to unprintables
#
procedure main()
    write("Size of &ascii: ", *&ascii)
    write("32 bytes starting at offset 65: ", &ascii[65:97])
end
```

Giving:

```
prompt$ unicon -s ascii.icn -x
Size of &ascii: 128
32 bytes starting at offset 65: @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
```

Note that Icon indexing is 1 relative, so the 65th item is byte code 64.

A handy *Cset* keyword, but be wary with displaying this value in full, as it includes unprintable control codes.

See also:

&cset, *&digits*, *&lcase*, *&letters*, *&ucase*, *Cset*

8.1.3 &clock

- Read-only
- Produces a *string* with the current time in `hh:mm:ss` form.

```
#
# &clock sample
#
procedure main()
    write("&clock: ", &clock)
end
```

Giving:

```
prompt$ unicon -s clock.icn -x
&clock: 04:53:34
```

See also:

&date, *&dateline*, *&now*, *&time*

8.1.4 &col

- Read-write
- Produces: an *integer*

`&col` is the mouse horizontal position in text columns, from the most recent *Event()*. If `&col` is assigned, *&x* is set to a corresponding pixel location using the current font of *&window*.

```
#
# col.icn, demonstrate the &col mouse column keyword
#
link enqueue, evmux
procedure main()
    window := open("mouse column", "g", "size=20,20", "canvas=hidden")
    Enqueue(window, &lpress, 11, 14, "", 2)
    w := Active()
    write(image(w))
    e := Event(w, 1)
    write("event at mouse column ", &col)
    write("event at mouse position (", &x, ",", &y, ")")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s col.icn -x
window_1:1(mouse column)
event at mouse column 2
event at mouse position (11,14)
```

See also:

&x, *Event*

8.1.5 &collections

- Read-only
- Generates integers

A generator that produces 4 *integers*; the number of times memory has been reclaimed from the

- heap
- static
- string
- block

memory regions.

```
#
# show collections, create and remove a string, reshow collections
#
procedure main()
    collections()
    s := repl(&letters, 1000)
    s := &null
    collect()
    write("\nAfter string create/remove")
    collections()
end

# Display current memory region allocations
procedure collections()
    local collects

    collects := [] ; every put(collects, &collections)

    write("Collections, ", *collects, " values generated")
    write(repl("-", 30 + **collects))
    write("Heap   : ", collects[1])
    write("Static : ", collects[2])
    write("String : ", collects[3])
    write("Block  : ", collects[4])
end
```

Giving:

```
prompt$ unicon -s collections.icn -x
Collections, 4 values generated
-------------------------------
```

```
Heap   : 0
Static : 0
String : 0
Block  : 0

After string create/remove
Collections, 4 values generated
-------------------------------
Heap   : 1
Static : 0
String : 0
Block  : 0
```

See also:

&allocated, *&storage*, *®ions*

8.1.6 &column

- Read-only
- Returns an integer

Produces the *integer* column, from the program source code, of the current execution point.

```
#
# Display the current source column
#
procedure main()
    write("Executing code from column: ", &column)
end
```

Giving:

```
prompt$ unicon -s column.icn -x
Executing code from column: 10
```

See also:

&line, *&file*, *&level*, *&progname*, *&trace*, *Unicon monitoring*

8.1.7 &control

- Read-only
- Produces: *&null* or failure

null if control key was down on last X event, otherwise a reference to &control fails.

```
#
# control.icn, demonstrate the &control key status keyword
#
link enqueue, evmux
```

```
procedure main()
    window := open("control", "g", "size=20,20", "canvas=hidden")
    # Enqueue an event with a "c" modifier, setting the control key state
    Enqueue(window, &lpress, 11, 14, "c", 2)
    Enqueue(window, &lrelease, 11, 14, "", 2)
    w := Active()
    write(image(w))

    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    write("&control: ", image(&control))

    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    write("&control: ", image(&control))

    close(window)
end
```

Sample run:

```
prompt$ unicon -s control.icn -x
window_1:1(control)
event at mouse position (11,14)
&control: &null
event at mouse position (11,14)
```

See also:

&meta, *&shift*

8.1.8 &cset

- Read-only
- Produces a *cset* that includes all characters.

```
#
# Display some of the &cset keyword, not all, due to unprintables
#
procedure main()
    write("Size of &cset: ", *&cset)
    write("32 bytes starting at offset 41: ", &cset[41:73])
end
```

Giving:

```
prompt$ unicon -s cset.icn -x
Size of &cset: 256
32 bytes starting at offset 41: ()*+,-./0123456789:;<=>?@ABCDEFG
```

Note that Icon indexing is 1 relative, so the 41st item is byte code 40.

A very handy keyword, as subsets of character sequences can easily be made by indexing `&cset`. Includes unprintable character codes.

See also:

&ascii, *&digits*, *&lcase*, *&letters*, *&ucase*, *Cset*

8.1.9 ¤t

- Read-only
- Produces a *co-expression*.

Produces the *co-expression* that is currently executing.

```
#
# &current sample, using image for this unprintable datatype
#
procedure main()
    # create a co-expression
    coex := create {
        write("\t&current in coex: ", image(&current))
        write("\t&source  in coex: ", image(&source))
    }

    # show &current in main
    if &current === &source then
        write("&current in main: ", image(&current))
    else
        write("problem: &current not equal &source")

    # now show &current in the co-expression
    @coex
end
```

Giving:

```
prompt$ unicon -s current.icn -x
&current in main: co-expression_1(1)
        &current in coex: co-expression_2(0)
        &source  in coex: co-expression_1(1)
```

See also:

&source, *create*, *Unicon monitoring*

Side comment: *co-expressions*, are *cool*.

8.1.10 &date

- Read-only
- Produces a *string* with the current date in `yyyy/mm/dd` form.

```
#
# &date sample
#
procedure main()
    write("&date: ", &date)

    # transpose to month/day/year form
    write("M/D/Y: ", map("Mm/Dd/yYxX", "yYxX/Mm/Dd", &date))
end
```

Giving:

```
prompt$ unicon -s date.icn -x
&date: 2019/10/27
M/D/Y: 10/27/2019
```

The sample code highlights a powerful Unicon idiom, for transposing strings around, called `labelling`. It converts from system `yyyy/mm/dd` to `mm/dd/yyyy` format.

If you prefer other forms, the advanced transposition features of function *map* can come in handy.

```
map("Dd/Mm/yYxX", "yYxX/Mm/Dd", &date)
```

Will map the date from yyyy/mm/dd to dd/mm/yyyy form. Note this map is in (transpositions, labels, source) order. The second argument (the labels) can have no duplicate characters, and must be the same size as the third argument for, the transposition type mapping to occur.

But, stick with the default `&date` *format if you can. It's not ambiguous, and it's easily machine sortable*.

See also:

&clock, *&dateline*, *&now*, *&time*

8.1.11 &dateline

- Read-only
- Produces a *string* with the current date in human readable form.

The `&dateline` keyword gives a nice time stamp, with day of the week, date and time (to the minute).

```
#
# &dateline sample
#
procedure main()
    write("&dateline: ", &dateline)
end
```

Giving:

```
prompt$ unicon -s dateline.icn -x
&dateline: Sunday, October 27, 2019  4:53 am
```

See also:

&clock, *&date*, *&now*, *&time*

8.1.12 &digits

- Read-only
- Produces a *cset* that includes the ten decimal digits.

```
#
# Display the &digits keyword cset
#
procedure main()
    write("Size of &digits: ", *&digits)
    write("Cset of &digtis: ", &digits)
end
```

Giving:

```
prompt$ unicon -s digits.icn -x
Size of &digits: 10
Cset of &digtis: 0123456789
```

See also:

&ascii, *&cset*, *&lcase*, *&letters*, *&ucase*, *Cset*

8.1.13 &dump

- Read-Write
- An *integer* that controls the generation of a program dump.

If `&dump` is non-zero when a program halts, all local and global variables are shown, along with their values.

```
#
# Display an end of run storage dump
#
procedure main()
    write("Dump variables on exit")

    a := 1
    b := 1.0
    c := &digits
    d := "Unicon"
    e := &letters || &digits
    f := write
    g := [1,2,3,4]

    &dump := 1
    stop("&dump is ", &dump)
end
```

Sample run (ends with a stop, error code set to 1, the value of `&dump`):

```
prompt$ unicon -s dump.icn -x
Dump variables on exit
&dump is 1

Termination dump:
```

```
co-expression #1(1)
main local identifiers:
   a = 1
   b = 1.0
   c = &digits
   d = "Unicon"
   e = "ABCDEFGHIJKLMNOP..."
   f = function write
   g = list_1 = [1,2,3,4]

global identifiers:
   main = procedure main
   stop = function stop
   write = function write
```

8.1.14 &e

- Read-only
- Produces a *Real*, representing Euler's number, the base of the natural logarithms.

`e` is a marvelous, magical number.

```
#
# &e, Euler's number, the natural logarithm
#
procedure main()
    write("Size  of &e: ", *&e)
    write("Value of &e: ", &e)
end
```

Giving:

```
prompt$ unicon -s e.icn -x
Size  of &e: 17
Value of &e: 2.718281828459045
```

See also:

&phi, *&pi*

8.1.15 &errno

- Read-only
- Produces: *integer*

Variable containing *transient* error number from previous POSIX command.

```
#
# errno.icn, demonstrate the POSIX volatile &errno value
#
procedure main()
    # attempt POSIX operation on non existent file
    write("attempt readlink on non-existent-file")
    readlink("non-existent-file")
    write("&errno: ", &errno)
    write("errno 2 is ENOENT (No such file or directory) in Linux")
end
```

Sample run:

```
prompt$ unicon -s errno.icn -x
attempt readlink on non-existent-file
&errno: 2
errno 2 is ENOENT (No such file or directory) in Linux
```

8.1.16 &error

- Read-Write
- An *integer* that controls how runtime errors are handled.

If `&error` is non-zero then runtime errors are converted into expression failure. `&error` is decremented each time a runtime error occurs. Setting `&error` to -1 (minus one) effectively disables runtime errors indefinitely.

```
#
# Runtime error conversion to failure with &error
#
procedure main()
    &error := 2
    every i := 1 to 3 do {
        write("attempt: ", i, " &error is ", &error)
        a := "1" + []
    }
end
```

Sample run (eventually ends with failure):

```
prompt$ unicon -s error.icn -x
attempt: 1 &error is 2
attempt: 2 &error is 1
attempt: 3 &error is 0

Run-time error 102
File error.icn; Line 15
numeric expected
offending value: list_3 = []
Traceback:
   main()
   {1 + list_3 = []} from line 15 in error.icn
```

Side comment: *Unicon style expression failure, should be in every language.*

See also:

&errornumber, *&errortext*, *&errorvalue*

8.1.17 &errornumber

- Read-only
- Produces an *integer*

`&errornumber` is the number of the last error converted to failure, if any.

```
#
# Runtime error conversion to failure; &errornumber
#
procedure main()
    &error := 2
    write("&errornumber: ", &errornumber)
    every i := 1 to 3 do {
        write("attempt: ", i, " &error is ", &error)
        a := "1" + []
        write("&errornumber: ", &errornumber)
    }
end
```

Sample run (eventually ends with failure code):

```
prompt$ unicon -s errornumber.icn -x
attempt: 1 &error is 2
&errornumber: 102
attempt: 2 &error is 1
&errornumber: 102
attempt: 3 &error is 0

Run-time error 102
File errornumber.icn; Line 16
numeric expected
offending value: list_3 = []
Traceback:
   main()
   {1 + list_3 = []} from line 16 in errornumber.icn
```

On program startup, `&errornumber` is undefined, the first `write` expression fails.

See also:

&error, *&errortext*, *&errorvalue*

8.1.18 &errortext

- Read-only
- Produces a *string*

`&errortext` is the last error message that was converted to failure, if any.

```
#
# Runtime error conversion to failure; &errortext
#
procedure main()
    &error := 2
    write("&errortext: ", &errortext)
    every i := 1 to 3 do {
        write("attempt: ", i, " &error is ", &error)
        a := "1" + []
        write("&errortext: ", &errortext)
    }
end
```

Sample run (eventually ends with failure code):

```
prompt$ unicon -s errortext.icn -x
&errortext:
attempt: 1 &error is 2
&errortext: numeric expected
attempt: 2 &error is 1
&errortext: numeric expected
attempt: 3 &error is 0

Run-time error 102
File errortext.icn; Line 16
numeric expected
offending value: list_3 = []
Traceback:
   main()
   {1 + list_3 = []} from line 16 in errortext.icn
```

On program startup, `&errortext` is undefined, the first `write` expression fails.

See also:

&error, *&errornumber*, *&errorvalue*

8.1.19 &errorvalue

- Read-only
- Produces a value

`&errorvalue` is the value involved in the last error that was converted to failure, if any.

```
#
# Runtime error conversion to failure; &errorvalue
#
procedure main()
    &error := 2
    write("&errorvalue: ", image(&errorvalue))
    every i := 1 to 3 do {
        write("attempt: ", i, " &error is ", &error)
        a := "1" + []
        write("&errorvalue: ", image(&errorvalue))
    }
end
```

Sample run (eventually ends with failure code):

```
prompt$ unicon -s errorvalue.icn -x
attempt: 1 &error is 2
&errorvalue: list_1(0)
attempt: 2 &error is 1
&errorvalue: list_2(0)
attempt: 3 &error is 0

Run-time error 102
File errorvalue.icn; Line 16
numeric expected
offending value: list_3 = []
Traceback:
   main()
   {1 + list_3 = []} from line 16 in errorvalue.icn
```

On program startup, `&errorvalue` is undefined. As this value can be any value, `image` is used to show the type of value.

See also:

&error, *&errornumber*, *&errortext*

8.1.20 &errout

- Read-only
- Produces the current standard error file stream

```
#
# &errout, the standard error stream
#
procedure main()
    write(type(&errout))
    write(image(&errout))
end
```

Giving:

```
prompt$ unicon -s errout.icn -x
file
&errout
```

See also:

&input, *&output*

8.1.21 &eventcode

- Read-only

- Produces: An *integer*

Event code in monitored program, set from last *EvGet*

```
#
# eventcode.icn, demonstrate Execution Monitoring eventcode
# needs 1to4.icn as the monitoring target
#
link evinit, wrap

procedure main()
    wrap()
    EvInit("1to4")
    while EvGet() do write(wrap(left(ord(&eventcode) || ", ",5), 64))
    write(wrap())
end
```

Sample run:

```
prompt$ unicon -s eventcode.icn -x
189, 185, 160, 160, 186, 67,  188, 187, 160, 160, 162, 161,
188, 187, 160, 161, 160, 160, 160, 191, 160, 160, 191, 160,
160, 191, 160, 187, 160, 176, 189, 185, 179, 187, 160, 176,
73,  75,  78,  73,  75,  115, 115, 177, 187, 160, 186, 243,

1
99,  170, 73,  75,  78,  173, 160, 160, 169, 180, 190, 185,
189, 185, 179, 187, 160, 176, 73,  75,  78,  73,  75,  115,

2
115, 177, 187, 160, 186, 243, 99,  170, 73,  75,  78,  173,
160, 160, 169, 180, 190, 185, 189, 185, 179, 187, 160, 176,
73,  75,  78,  73,  75,  115, 115, 177, 187, 160, 186, 243,

3
99,  170, 73,  75,  78,  173, 160, 160, 169, 180, 190, 185,
189, 185, 179, 187, 160, 176, 73,  75,  78,  73,  75,  115,

4
115, 177, 187, 160, 186, 243, 99,  170, 73,  75,  78,  173,
160, 160, 169, 180, 190, 185, 178, 169, 187, 160, 188, 187,
160, 166, 160, 88,
```

See also:

&eventsource, *&eventvalue*, *EvGet*

8.1.22 &eventsource

- Read-only
- Produces: A *co-expression*

Source co-expression of event in monitoring program, set during *EvGet*.

```
#
# eventsource.icn, demonstrate Execution Monitoring eventsource
# needs 1to4.icn as the monitoring target
#
link evinit

procedure main()
    EvInit("1to4")
    EvGet()
    write(image(&eventsource))
    while EvGet()
end
```

Sample run:

```
prompt$ unicon -s eventsource.icn -x
co-expression_1(0)

1

2

3

4
```

See also:

&eventcode, *&eventvalue*, *EvGet*

8.1.23 &eventvalue

- Read-only
- Produces: *any*

Value being processed when *EvGet* returns an execution monitoring event.

```
#
# eventvalue.icn, demonstrate Execution Monitoring eventvalue
# needs 1to4.icn as the monitoring target
#
link evinit, wrap

procedure main()
    wrap()
    EvInit("1to4")
    while EvGet() do write(wrap(image(&eventvalue) || ", ", 64))
    write(wrap())
end
```

Sample run:

```
prompt$ unicon -s eventvalue.icn -x
0, 17474639271449, 98, 61, 17474639271449, procedure main, 11,
23068683, 98, 67, 9, 139797114171300, 12, 11075596, 85, 0, 84,
```

```
69, 77, "\n", 69, 60, 1, 98, 60, 4, 73, 58720268, 45,
function ..., 5, 17474639271478, 1, 46137356, 3, function ||,
"\n", "", "\n", 1, "", 1, 2, "\n1", 33554444, 61,

1
17474639271486, "write+", 61, function write, "\n1", "", "\n1",
"\n1", 70, 53, -1, 0, 1, 17474639271478, 5, 17474639271478, 2,
46137356, 3, function ||, "\n", "", "\n", 2, "", 1, 2, "\n2",
33554444, 61, 17474639271486, "write+", 61, function write,

2
"\n2", "", "\n2", "\n2", 70, 53, -1, 0, 1, 17474639271478, 5,
17474639271478, 3, 46137356, 3, function ||, "\n", "", "\n", 3,
"", 1, 2, "\n3", 33554444, 61, 17474639271486, "write+", 61,

3
function write, "\n3", "", "\n3", "\n3", 70, 53, -1, 0, 1,
17474639271478, 5, 17474639271478, 4, 46137356, 3, function ||,
"\n", "", "\n", 4, "", 1, 2, "\n4", 33554444, 61,

4
17474639271486, "write+", 61, function write, "\n4", "", "\n4",
"\n4", 70, 53, -1, 0, 1, 17474639271478, function ..., -1,
11141132, 69, 13, 2097165, 68, procedure main, 48, 0,
```

See also:

&eventcode, *&eventsource*, *EvGet*

8.1.24 &fail

- Read-only, *but fails when accessed*
- Causes failure, as soon as it is evaluated.

```
#
# &fail, immediate expression failure when evaluated
#
procedure main()
    write(type(&fail))
    write(image(&fail))
end
```

Giving:

```
prompt$ unicon -s fail-keyword.icn -x
```

Which produces no output; both of the expressions, `type` and `image` fail, so `write` is never evaluated.

See also:

Success and Failure, *fail*

8.1.25 &features

- Read-only
- Generates *string* data that indicates the optional, and non-portable features supported by the current `unicon` build.

A reflective aspect of `Unicon` that can be used for purely informational purposes, or to decide at runtime what code fragments are safe to use on the current platform.

```
#
# features.icn, Display the optional and non-portable feature set
#
procedure main()
    every write(&features)

    if &features == "POSIX" then
        write("\nPOSIX code supported with this ", &version)

    f := 0
    every function() do f +:= 1
    write(f, " built in functions")
end
```

Giving:

```
prompt$ unicon -s features.icn -x
UNIX
POSIX
DBM
ASCII
co-expressions
native coswitch
concurrent threads
dynamic loading
environment variables
event monitoring
external functions
keyboard functions
large integers
multiple programs
pattern type
pipes
pseudo terminals
system function
messaging
graphics
3D graphics
X Windows
libz file compression
JPEG images
PNG images
SQL via ODBC
Audio
secure sockets layer encryption
CCompiler gcc 5.5.0
Physical memory: 7808401408 bytes
Revision 6034-743ffd1
Arch x86_64
```

```
CPU cores 4
Binaries at /home/btiffin/unicon-git/bin/

POSIX code supported with this Unicon Version 13.1.  August 19, 2019
302 built in functions
```

The `unicon -feature` command line option will display the `&features` data as well.

See also:

$ifdef, *$ifndef*, *Predefined symbols*, *&version*

8.1.26 &file

- Read-only
- Produces the *string* of the source used to compile the current execution point.

```
#
# &file, Current source file
#
procedure main()
    write(&file)
end
```

Giving:

```
prompt$ unicon -s file.icn -x
file.icn
```

See also:

&column, *&line*, *&level*, *&progname*, *&trace*, *Unicon monitoring*

8.1.27 &host

- Read-only
- Produces a *string* representing the current network hostname.

```
#
# &host, the network host name
#
procedure main()
    write(&host)
end
```

Giving:

```
prompt$ unicon -s host.icn -x
btiffin-CM1745
```

See also:

&version

8.1.28 &input

- Read-only
- Produces a *file* representing the standard input stream.

```
#
# &input, the standard input stream
#
procedure main()
    write(type(&input))
    write(image(&input))
end
```

Giving:

```
prompt$ unicon -s input.icn -x
file
&input
```

See also:

&errout, *&output*

8.1.29 &interval

- Read-only
- Produces: An *integer*

Milliseconds since previous windowing event.

```
#
# interval.icn, demonstrate the &interval event timing
#
link enqueue, evmux
procedure main()
    window := open("interval", "g", "size=20,20", "canvas=hidden")

    # enqueue a press, interval 2ms
    Enqueue(window, &lpress, 11, 14, "", 2)
    # enqueue a release, interval 3ms
    Enqueue(window, &lrelease, 12, 15, "", 3)

    w := Active()
    write(image(w))

    e := Event(w, 1)
    write("event interval ", &interval, " ms")
    e := Event(w, 1)
```

```
    write("event interval ", &interval, " ms")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s interval.icn -x
window_1:1(interval)
event interval 2 ms
event interval 3 ms
```

See also:

Event

8.1.30 &lcase

- Read-only
- Produces a *cset* of the lower case letters, a through z.

```
#
# &lcase keyword, lower case letters Cset
#
procedure main()
    write("Size of &lcase: ", *&lcase)
    write(&lcase)
end
```

Giving:

```
prompt$ unicon -s lcase.icn -x
Size of &lcase: 26
abcdefghijklmnopqrstuvwxyz
```

See also:

&ascii, *&cset*, *&digits*, *&letters*, *&ucase*, *Cset*

8.1.31 &ldrag

- Read-only
- Produces: the *integer* that indicates a left mouse button drag event code.

left button drag event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
```

```
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&mpress*, *&mrelease*, *&mdrag* *&rpress*, *&rrelease*, *&rdrag*

8.1.32 &letters

- Read-only
- Produces a *cset* of all letter, A-Za-z.

```
#
# &letters keyword, upper and lower case letters Cset
#
procedure main()
    write("Size of &letters: ", *&letters)
    write(&letters)
end
```

Giving:

```
prompt$ unicon -s letters.icn -x
Size of &letters: 52
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

See also:

&ascii, *&cset*, *&digits*, *&lcase*, *&ucase*, *Cset*

8.1.33 &level

- Read-only
- Produces the current execution depth, one level for each nested procedure.

```
#
# &level, procedure execution depth
#
procedure main()
    write("main level: ", &level)
    subproc()
end

procedure subproc()
    write("subp level: ", &level)
end
```

Giving:

```
prompt$ unicon -s level.icn -x
main level: 1
subp level: 2
```

See also:

&error, *&trace*

8.1.34 &line

- Read-only
- Produces an *integer*.

Returns the *integer* line, from the program source code, of the current execution point.

```
#
# Display the current source line
#
procedure main()
    write("Executing code from line: ", &line)
end
```

Giving:

```
prompt$ unicon -s line.icn -x
Executing code from line: 12
```

See also:

&column, *&file*, *&level*, *&progname*, *&trace*, *Unicon monitoring*

8.1.35 &lpress

- Read-only
- Produces: the *integer* that indicates a left mouse button press event code.

left button press event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lrelease, *&ldrag*, *&mpress*, *&mrelease*, *&mdrag* *&rpress*, *&rrelease*, *&rdrag*

8.1.36 &lrelease

- Read-only
- Produces: the *integer* that indicates a left mouse button release event code.

left button release event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
```

```
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&ldrag*, *&mpress*, *&mrelease*, *&mdrag* *&rpress*, *&rrelease*, *&rdrag*

8.1.37 &main

- Read-only
- Produces the main *co-expression*

```
#
# &main, main co-expression
#
procedure main()
    if &current === &main then write("&current is &main")
    coex := create(if &source === &main then write("coex: &source is &main"))
    @coex
end
```

Giving:

```
prompt$ unicon -s main.icn -x
&current is &main
coex: &source is &main
```

See also:

¤t, *&source*

8.1.38 &mdrag

- Read-only
- Produces: the *integer* that indicates a middle mouse button drag event code.

middle button drag event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&ldrag*, *&mpress*, *&mrelease*, *&rpress*, *&rrelease*, *&rdrag*

8.1.39 &meta

- Read-only
- Produces: *&null* or fail

null if meta key was down on last X event, otherwise a reference to `&meta` fails.

```
#
# meta.icn, demonstrate the &meta key status keyword
#
link enqueue, evmux
procedure main()
    window := open("meta", "g", "size=20,20", "canvas=hidden")
```

```
    # Enqueue an event with the "m" modifier, setting the meta key state
    Enqueue(window, &lpress, 11, 14, "m", 2)
    Enqueue(window, &lrelease, 11, 14, "", 2)
    w := Active()
    write(image(w))

    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    write("&meta: ", image(&meta))

    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    write("&meta: ", image(&meta))

    close(window)
end
```

Sample run:

```
prompt$ unicon -s meta.icn -x
window_1:1(meta)
event at mouse position (11,14)
&meta: &null
event at mouse position (11,14)
```

See also:

&control, *&shift*

8.1.40 &mpress

- Read-only
- Produces: the *integer* that indicates a middle mouse button press event code.

middle button press event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&ldrag*, *&mrelease*, *&mdrag*, *&rpress*, *&rrelease*, *&rdrag*

8.1.41 &mrelease

- Read-only
- Produces: the *integer* that indicates a middle mouse button release event code.

middle button release event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&ldrag*, *&mrelease*, *&mdrag*, *&rpress*, *&rrelease*, *&rdrag*

8.1.42 &now

- Read-only
- Produces an *integer* representing the current time as a count of seconds since the *Unix epoch* reference standard, which is midnight January 1st, 1970.

```
#
# &now sample
#
procedure main()
    write("&now      :", &now)
    delay(1000)
    write("and &now :", &now, ", 1000 milliseconds(ish) later")
end
```

Giving:

```
prompt$ unicon -s now.icn -x
&now     :1572166418
and &now :1572166419, 1000 milliseconds(ish) later
```

See also:

&clock, *&date*, *&dateline*, *&time*, *y2k*, *Year 2038 problem*

8.1.43 &null

- Read-only
- Represents the null value.

The *null* value is a special case in `Unicon`. Unset variables return &null, and unused arguments test as &null. There are some operators that help manage this special value.

- `\var` will fail if the variable `var` is null
- `/var` will succeed and return the (empty) variable reference if `var` is null which is great for setting optional default values

```
#
# &null, the null value
#

# b will be given a default value, a will remain unset
procedure main()
    \a := "a"
    /b := "b"
    if a === &null then write("a is &null")
    if b === &null then write("b is &null") else write("b is ", image(b))
end
```

Giving:

```
prompt$ unicon -s null.icn -x
a is &null
b is "b"
```

See also:

&fail, *null*

8.1.44 &output

- Read-only
- Produces the standard output stream

```
#
# &output, the standard output stream
#
procedure main()
    write(type(&output))
    write(image(&output))
end
```

Giving:

```
prompt$ unicon -s output.icn -x
file
&output
```

See also:

&errout, *&input*

8.1.45 &phi

- Read-only
- Produces a *real* constant equal to the Golden Ratio

```
#
# &phi, the constant representing the Golden ratio
#
procedure main()
    write(&phi)
end
```

Giving:

```
prompt$ unicon -s phi.icn -x
1.618033988749895
```

See also:

&pi, *&e*

8.1.46 &pi

- Read-only
- Produces a *Real* representing the ratio of the diameter of a circle compared to the radius.

```
#
# &pi, the ratio of the diameter of a circle compared to the radius
#
procedure main()
    write(&pi)
    write("Size of &pi in string form: ", *&pi)
end
```

Giving:

```
prompt$ unicon -s pi.icn -x
3.141592653589793
Size of &pi in string form: 17
```

See also:

&e, *&phi*

8.1.47 &pick

- Read-only
- Generates: *string** [*3D Graphics*]

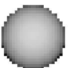
`pick` generates the object IDs selected at point(*&x*, *&y*) from the most recent windowing *Event*; *if* the event was read from a 3D window with attribute "pick=on". Objects need to be registered with *WSection* so Unicon knows which elements to generate for `&pick`.

```
#
# pick.icn, demonstrate 3D pick object ID during windowing event
#
link enqueue, evmux
procedure main()
    window := open("pick", "gl", "size=90,60")
    WAttrib(window, "pick=on")

    # mark a named 3D object
    WSection(window, "sphere")
    DrawSphere(window, 0.0, 0.19, -2.2, 0.3)
    Refresh(window)
    WSection(window)

    # insert an event into the queue, left press, 2ms interval, on sphere
    #Enqueue(window, &lpress, 45,20, "", 2)
```

```
    e := Event(window)
    write(image(e))

    # a side effect of the Event function is keywords settings
    write("&x:", &x)
    write("&y:", &y)
    write("&row:", &row)
    write("&col:", &col)
    every write("&pick:", &pick)

    WriteImage(window, "../images/pick.png")
    close(window)
end
```

Sample run, clicking on the sphere:

```
prompt$ unicon -s pick.icn -x
-1
&x:46
&y:24
&row:2
&col:8
&pick:sphere
```



See also:

&x, *&y*, *Event*, *WSection*

8.1.48 &pos

- Read-write
- Holds the current *string scanning position*, as an *integer*.

&pos is set to 1 at the start of a string scanning expression. &pos can be set to arbitrary values, but will be constrained internally to always be a valid string scanning index. When set to a negative indexing value, it will be reset to a valid positive index value.

```
#
# &pos, string scanning position
#
# demonstrate how negative position indexes are set to actual
#
procedure main()
    str := &letters
    str ? {
        first := &pos
        &pos := 0
```

```
        last := &pos
        &pos := -10
        back10 := &pos
    }
    write("first: ", first, ", last: ", last, ", -10: ", back10)
end
```

Giving:

```
prompt$ unicon -s pos.icn -x
first: 1, last: 53, -10: 43
```

See also:

&subject, *String Scanning*

8.1.49 &progname

- Read-only
- Returns a *string*

Produces the currently executing program name.

```
#
# &progname, Current program name
#
procedure main()
    write(&progname)
end
```

Giving:

```
prompt$ unicon -s progname-sample.icn -x
progname-sample
```

See also:

&column, *&line*, *&file*, *&level*, *&trace*, *Unicon monitoring*

8.1.50 &random

- Read-write
- Produces an *integer* from the internal pseudo-random number generator used by the unary *?* operator.

The `&random` seed is initialized to a different value by Unicon for each program run, but can be set to a known value for reproducible sequences during testing.

```
#
# &random seed, seeds the internal pseudo-random sequencer used by unary ?
#
procedure main()
```

```
    write(&random)
    every 1 to 3 do writes(?&letters)
    write()
    write()
    &random := 1
    every 1 to 3 do writes(?&letters)
    write()
    write(&random)
end
```

Giving:

```
prompt$ unicon -s random.icn -x
20210901
PtW

lwQ
686405327
```

The first value and set of letters will be random (*psuedo-random*) for every run, the seed uniquely initialized at program startup. The second set of characters and the last numeric value will be the same for every run, using a known seed value of `1`.

When testing you can set a known seed to get a predictable, and consistent sequence of random values. Otherwise just let Unicon reset the seed for each run, and the results are almost[1] unpredictable. Don't set a known value for things like games that need random elements, or each play through will be predictable and in the worst case, the same, over and over again.

```
prompt$ unicon -quiet -s random.icn -x
20210912
xfZ

lwQ
686405327
```

8.1.51 &rdrag

- Read-only
- Produces: the *Integer* that indicates a right mouse button drag event

right button drag.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)
```

[1] Psuedo-random number generators are never completely unpredictable, given enough effort. If you need cryptographically secure random values, you will need to augment the default algorithms used in Unicon and mix things up a bit, in order to defeat any bad actors that may try and predict the order and numeric sequences of the psuedo-random numbers. *Even players out to cheat on your game might try and guess at sequencing, so mix it up.*

```
    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&ldrag* *&mpress*, *&mrelease*, *&mdrag* *&rpress*, *&rrelease*

8.1.52 ®ions

- Read-only
- Generates 3 *Integer*s

A generator that produces 3 *Integer*s; the cumulative number of bytes used for the

- static (always zero in Unicon, for backward compatibility)
- string
- block

memory regions.

```
#
# &regions, current static, string and block memory sizes
#
procedure main()
    region := [] ; every put(region, &regions)

    write("Regions ", *region)
    write("---------")
    write("Static : ", region[1])
    write("String : ", region[2])
    write("Block  : ", region[3])
end
```

Giving:

```
prompt$ unicon -s regions.icn -x
Regions 3
---------
Static : 0
String : 139780259
Block  : 139780259
```

See also:

&allocated, *&collections*, *&storage*

8.1.53 &resize

- Read-only
- Produces: *Integer*

Windowing resize event code.

```
#
# resize.icn, display the code representing a resize event
#
procedure main()
    write("&resize: ", &resize)
end
```

Sample run:

```
prompt$ unicon -s resize.icn -x
&resize: -10
```

See also:

Event

8.1.54 &row

- Read-only
- Produces: *Integer* for the effective row of a windowing event

mouse vertical position in text rows.

```
#
# row.icn, demonstrate &row event keyword
#
link enqueue, evmux
procedure main()
    window := open("Event", "g", "size=20,20", "canvas=hidden")

    # insert an event into the queue, left press, control/shift, 2ms
    Enqueue(window, &lpress, 11, 14, "m", 2)
    e := Event(window)
    write(image(e))
```

```
    # a side effect of the Event function is keywords settings
    write("&x:", &x)
    write("&y:", &y)
    write("&row:", &row)
    write("&col:", &col)
    write("&interval:", &interval)
    write("&control:", &control)
    write("&shift:", &shift)
    write("&meta:", &meta)

    close(window)
end
```

Sample run:

```
prompt$ unicon -s row.icn -x
-1
&x:11
&y:14
&row:2
&col:2
&interval:2
&meta:
```

See also:

&col, *&x*, *&y*

8.1.55 &rpress

- Read-only
- Produces: the *Integer* that indicates a right mouse button press event

Right mouse button press event code.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&ldrag* *&mpress*, *&mrelease*, *&mdrag* *&rpress*, *&rdrag*

8.1.56 &rrelease

- Read-only
- Produces: the *Integer* that indicates a right mouse button release event code.

right button release.

```
#
# button-values.icn, display the left, middle and right mouse button
#  event codes for press, release and drag
#
procedure main()
    write("&lpress: ", &lpress)
    write("&mpress: ", &mpress)
    write("&rpress: ", &rpress)

    write("&lrelease: ", &lrelease)
    write("&mrelease: ", &mrelease)
    write("&rrelease: ", &rrelease)

    write("&ldrag: ", &ldrag)
    write("&mdrag: ", &mdrag)
    write("&rdrag: ", &rdrag)
end
```

Sample run:

```
prompt$ unicon -s button-values.icn -x
&lpress: -1
&mpress: -2
&rpress: -3
&lrelease: -4
&mrelease: -5
&rrelease: -6
&ldrag: -7
&mdrag: -8
&rdrag: -9
```

See also:

&lpress, *&lrelease*, *&ldrag* *&mpress*, *&mrelease*, *&mdrag* *&rpress*, *&rdrag*

8.1.57 &shift

- Read-only
- Produces: *&null* or fail

null if shift key was down on last X event, otherwise a reference to `&shift` with fail.

```
#
# shift.icn, demonstrate the &shift key status keyword
#
link enqueue, evmux
procedure main()
    window := open("shift", "g", "size=20,20", "canvas=hidden")
    # Enqueue an event with the "s" modifier, setting the shift key state
    Enqueue(window, &lpress, 11, 14, "s", 2)
    Enqueue(window, &lrelease, 11, 14, "", 2)
    w := Active()
    write(image(w))

    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    write("&shift: ", image(&shift))

    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    write("&shift: ", image(&shift))

    close(window)
end
```

Sample run:

```
prompt$ unicon -s shift.icn -x
window_1:1(shift)
event at mouse position (11,14)
&shift: &null
event at mouse position (11,14)
```

See also:

&control, *&meta*

8.1.58 &source

- Read-only
- Produces the source *Unicon Co-Expressions* of the *current*ly running co-expression.

```
#
# &source, Source co-expression
#
procedure main()
    coex := create(if &source === &main then write("&source is &main"))
    @coex
end
```

Giving:

```
prompt$ unicon -s source.icn -x
&source is &main
```

See also:

¤t, *&main*

8.1.59 &storage

- Read-only
- Generates 3 *Integer*s

A generator that produces 3 *Integer*s; the bytes used for the

- static (always zero in Unicon, for backward compatibility)
- string
- block

memory regions.

```
#
# &storage, current static, string and block memory usage
#
procedure main()
    storage()
    str := repl(&ascii, 1024)
    write("\nAfter string create")
    storage()
end

# Display current memory usages
procedure storage()
    store := [] ; every put(store, &storage)

    write("storage ", *store)
    write("---------")
    write("Static : ", store[1])
    write("String : ", store[2])
    write("Block  : ", store[3])
end
```

Giving:

```
prompt$ unicon -s storage.icn -x
storage 3
---------
Static : 0
String : 0
Block  : 34528

After string create
storage 3
---------
Static : 0
String : 131200
Block  : 34808
```

See also:

&allocated, *&collections*, *®ions*

8.1.60 &subject

- Read-write
- Holds the current *String Scanning* subject, always a *String*.

When `&subject` is explicitly assigned a new string value, *&pos* is automatically set to 1.

```
#
# &subject, string scanning subject
#
procedure main()
    str := &letters
    str ? {
        write(&subject, " at ", &pos)
        move(10)
        write("now at ", &pos)
        &subject := &dateline
        write("now at ", &pos)
    }
end
```

Giving:

```
prompt$ unicon -s subject.icn -x
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz at 1
now at 11
now at 1
```

See also:

&pos, *String Scanning*

8.1.61 &time

- Read-only
- Produces an *Integer* of the number of milliseconds (1/1000ths of a second) of CPU time that have elapsed since program execution began.

```
#
# &time sample
#
procedure main()
    write("&time: ", &time)
end
```

Giving:

```
prompt$ unicon -s time.icn -x
&time: 8
```

This short program starts and completes in less than a tenth of a second under normal circumstances. Most runs of that example will show a relatively small number. Maybe as high as 100 if the system is extremely busy.

See also:

&clock, *&date*, *&dateline*, *&now*,

8.1.62 &trace

- Read-write
- Holds an *Integer* that determines the level of procedure depth for execution tracing.

Set to 0 (default) for no tracing, negative for infinite level tracing or to a desired depth. Unicon examines the environment variable `TRACE` on startup and there is also a `-t` compile time option to turn on tracing.

```
#
# &trace, procedure tracing to given level
#         0 means no tracing, negative is effectively infinite levels
#         Unicon reads environment setting TRACE at startup
#         There is also a -t compiler option, setting trace to ...
#
procedure main()
    write("Initial &trace from -t: ", &trace)
    write("Setting to 3 for this example")

    &trace := 3
    write("main level: ", &level)
    second()
end

procedure second()
    write("second level: ", &level)
    third()
end

procedure third()
    write("third level: ", &level)
```

```
    fourth()
end

procedure fourth()
    write("fourth level: ", &level)
end
```

Giving:

```
prompt$ unicon -s -t trace.icn -x
             :       main()
Initial &trace from -t: -2
Setting to 3 for this example
main level: 1
trace.icn    :   20  | second()
second level: 2
trace.icn    :   25  | | third()
third level: 3
trace.icn    :   30  | | | fourth()
fourth level: 4
```

See also:

Unicon monitoring

8.1.63 &ucase

- Read-only
- Produces

```
#
# &ucase keyword, lower case letters Cset
#
procedure main()
    write("Size of &ucase: ", *&ucase)
    write(&ucase)
end
```

Giving:

```
prompt$ unicon -s ucase.icn -x
Size of &ucase: 26
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

See also:

&ascii, *&cset*, *&digits*, *&lcase*, *&letters*, *Cset*

8.1.64 &version

- Read-only

- Produces

```
#
# &version, Unicon version build information
#
procedure main()
    write(&version)
end
```

Giving:

```
prompt$ unicon -s version.icn -x
Unicon Version 13.1.  August 19, 2019
```

See also:

&features

8.1.65 &window

- Read-only
- Produces: *Window*

A variable containing a default window value. Most graphic functions will default to using `&window` when no initial `window` argument is given.

```
#
# window.icn, demontrate the &window keyword
#
procedure main()
    &window := open("default", "g", "size=90,60", "canvas=hidden")

    # all subsequent graphic functions default to using &window
    write("Colour depth (bits): ", WAttrib("depth"),
          " on device ", WAttrib("display"),
          " with window label ", image(WAttrib("windowlabel")))

    Fg("vivid orange")
    FillRectangle(30, 30, 30, 20)
    Fg("medium cyan")
    FillRectangle(35, 35, 15, 8)

    # normally non graphic functions need to be told the window
    Fg("black")
    write(&window, "&window sample")

    WSync()
    WriteImage("../images/window.png")
    close(&window)
end
```

Sample run:

```
prompt$ unicon -s window.icn -x
Colour depth (bits): 24 on device localhost:10.0 with window label "default"
```

```
&window sample
```

8.1.66 &x

- Read-only
- Produces: *Integer*

Holds the horizontal mouse position. Set by calling *Event*.

```
#
# x.icn, demonstrate the &x mouse position keyword
#
link enqueue, evmux
procedure main()
    window := open("mouse position", "g", "size=20,20", "canvas=hidden")
    Enqueue(window, &lpress, 11, 14, "", 2)
    Enqueue(window, &lrelease, 12, 15, "", 3)
    w := Active()
    write(image(w))
    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s x.icn -x
window_1:1(mouse position)
event at mouse position (11,14)
event at mouse position (12,15)
```

See also:

&col, *&y*

8.1.67 &y

- Read-only
- Produces: *Integer*

Holds the mouse vertical position after *Event* is called.

```
#
# y.icn, demonstrate the &y mouse position keyword
#
```

```
link enqueue, evmux
procedure main()
    window := open("mouse position", "g", "size=20,20", "canvas=hidden")
    Enqueue(window, &lpress, 11, 14, "", 2)
    Enqueue(window, &lrelease, 12, 15, "", 3)
    w := Active()
    write(image(w))
    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    e := Event(w, 1)
    write("event at mouse position (", &x, ",", &y, ")")
    close(window)
end
```

Sample run:

```
prompt$ unicon -s y.icn -x
window_1:1(mouse position)
event at mouse position (11,14)
event at mouse position (12,15)
```

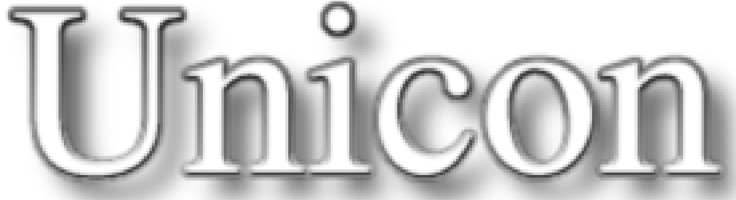
See also:

&x

CHAPTER

NINE

PREPROCESSOR



Unicon features a simple preprocessor, supporting file inclusion, conditional compilation, symbolic constants, source line number (and reported filename) override, and an error messaging directive.

The internal Unicon preprocessor does not support the list of features included in the *IPL* program `ipp.icn`. There is no `$elif` for instance, nor `$if` *constant-expression*. The macro substitution features are also not supported. `ipp` will work with most, if not all, Unicon sources, but it is not the same as the internal Unicon preprocessor. `ipp` may have more features, but the simpler, internal Unicon preprocessor is more accessible, built into the compiler. As a Unicon programmer, you have the option of using `ipp` (it ships with the Unicon source kit along with the *IPL*), but it is probably best to stick with the internal mechanisms.

9.1 Unicon preprocessor

Preprocessor directives are lines that begin with a dollar sign and are processed *before* the translation stage begins.

Please note, that all of the code examples for the Unicon preprocessor are short, and contrived.

9.1.1 $define

`$define` *symbol* [*value*]

Define a preprocessor symbol, for testing with *$ifdef*, *$ifndef* and as a text replacement. Occurrences of *symbol* will be replaced by *value*. Note the values cannot have arguments, the replacement is literal.

```
#
# preproc-define, demonstrate $define preprocessor directive
#
procedure main()
$define STUB 1
$ifndef STUB
    write("skip over some code")
$else
    write("Unicon, cui non; sold, in Italian")
```

```
$endif
end
```

Sample output:

```
prompt$ unicon -E -s preproc-define.icn
#line 0 "/tmp/uni31942190"
#line 0 "preproc-define.icn"










procedure main();




    write("Unicon, cui non; sold, in Italian");

end
```

9.1.2 $else

`$else`

`$else` marks the beginning of an optional, alternate source code block for processing an *$ifdef* or *$ifndef* conditional compile directive sequence. As nesting is allowed, `$else` matches to the closest previous *$ifdef* or *$ifndef*.

See *$ifdef*, *$ifndef* for examples.

9.1.3 $endif

`$endif`

`$endif` marks the end of a conditional compile source block when processing an *$ifdef* or *$ifndef* sequence. As nesting is allowed, `$endif` closes the closest previous *$ifdef*, *$ifndef* sequence.

See *$ifdef*, *$ifndef* for examples.

9.1.4 $error

`$error` *text*

`$error` causes the compiler to output an error with the supplied text as the message.

```
#
# preproc-error, demonstrate $error preprocessor directive
#
procedure main()
```

```
$ifdef _MSDOS
    write("MSDOS available, GetSpace and FreeSpace")
    A := GetSpace(64) | stop("GetSpace failed")
    FreeSpace(A)
$else
$error MSDOS features required for this program
$endif
end
```

Sample output:

```
prompt$ unicon -E -s preproc-error.icn
File preproc-error.icn; Line 17 # $error: MSDOS features required for this program
```

9.1.5 $ifdef

`$ifdef` *symbol*

`$ifdef` tests for a preprocessor symbol and, if defined, will emit source lines up to the next matching *$else* (or *$endif*, when no else directive is present). If not defined, the preprocessor will scan ahead to a matching *$else*, if present, and then output source lines up to the matching *$endif* directive. Nesting is allowed. All `$ifdef` directives must end with an *$endif*.

```
#
# preproc-ifdef, demonstrate $ifdef preprocessor directive
#
procedure main()
$ifdef _POSIX
    write("POSIX available, pid is ", getpid())
$else
    write("No POSIX features with this ", &version)
$endif
end
```

Sample output:

```
prompt$ unicon -E -s preproc-ifdef.icn
#line 0 "/tmp/uni31942190"
#line 0 "preproc-ifdef.icn"











procedure main();

    write("POSIX available, pid is ", getpid());



end
```

```
prompt$ unicon -s preproc-ifdef.icn -x
POSIX available, pid is 18320
```

Attention: Please be advised that this contrived example is not a preferred way to do feature testing in Unicon. The builtin *&features* keyword can be used at runtime for these type of conditional code fragments.

```
if &features == "ms windows" then ...
```

```
procedure main()
  if &features == "POSIX" then
    write("POSIX available, pid is ", getpid())
  else
    write("No POSIX features with this ", &version)
end
```

The runtime penalty is slight, and `.u` code is highly portable. Runtime platform tests are encouraged, preprocessor conditionals (for feature testing) less so.

9.1.6 $ifndef

`$ifndef` *symbol*

`$ifndef` is the opposite of *$ifdef*. The `$ifndef` directive tests for a preprocessor symbol and if *not* defined, will emit source lines up to the next matching *$else* (or *$endif*, when no else directive is present). If the symbol is defined, the preprocessor will scan ahead to a matching *$else*, if present, and then output source lines up to the matching *$endif* directive. Nesting is allowed. All `$ifndef` directives must end with an *$endif*.

```
#
# preproc-ifndef, demonstrate $ifndef preprocessor directive
#
procedure main()
$ifndef _MSDOS
    write("No MSDOS features with this ", &version)
$else
    write("MSDOS available, GetSpace and FreeSpace")
    A := GetSpace(64) | stop("GetSpace failed")
    FreeSpace(A)
$endif
end
```

Sample output:

```
prompt$ unicon -E -s preproc-ifndef.icn
#line 0 "/tmp/uni43851559"
#line 0 "preproc-ifndef.icn"
```

```


procedure main();

    write("No MSDOS features with this ", &version);





end
```

```
prompt$ unicon -s preproc-ifndef.icn -x
No MSDOS features with this Unicon Version 13.1.  August 19, 2019
```

Note: Please be advised that this contrived example is not a preferred way to do feature testing in Unicon. The builtin *&features* keyword can be used at runtime for these type of conditional code fragments. See the *$ifdef* entry for a runtime alternative.

9.1.7 $include

`$include` *filename*

Include another source file.

```
#
# preproc-include, demonstrate $include preprocessor directive
#
$include "preproc-define.icn"
```

Sample output:

```
prompt$ unicon -E -s preproc-include.icn
#line 0 "/tmp/uni31942190"
#line 0 "preproc-include.icn"

#line 10 "preproc-define.icn"
procedure main();




    write("Unicon, cui non; sold, in Italian");

end
```

9.1.8 $line

`$line` *line-number* [*"filename"*]

Override view of current source line (and optionally file). Subsequent lines are treated by the compiler as commencing at the given line number in the current or given filename, *which must be quoted*.

This directive is mainly of use for machine generated sources; for programs that generate (or manipulate) source programs, such as `ulex` or `flex/bison`.

This first sample is a compile time override example:

```
#
# preproc-line, demonstrate $line override preprocessor directive
#
# This feature is mainly for use with machine generated sources
#
procedure main()
$ifdef _MSDOS
    write("MSDOS available, GetSpace and FreeSpace")
    A := GetSpace(64) | stop("GetSpace failed")
    FreeSpace(A)
$else
# error will be reported as coming from line 2 of sample.icn
$line 1 "sample.icn"
$error MSDOS features required for this program
$endif
end
```

Sample output:

```
prompt$ unicon -E -s preproc-line.icn
File sample.icn; Line 2 # $error: MSDOS features required for this program
```

Now to see how it effects the runtime reporting:

```
#
# preproc-line, demonstrate $line override preprocessor directive
#
# This feature is mainly for use with machine generated sources
#
procedure main()
# preprocessor will add blank lines and report runtime error as line 21
$line 20
    a := 1 + []
end
```

Sample preprocessor output:

```
prompt$ unicon -E -s preproc-line-2.icn
#line 0 "/tmp/uni43851559"
#line 0 "preproc-line-2.icn"














procedure main();
```

```
    a := 1 + [];
end
```

There will a runtime error reported when this program runs, with the reported line number influenced by `$line` directive.

```
prompt$ unicon -s preproc-line-2.icn -x

Run-time error 102
File preproc-line-2.icn; Line 21
numeric expected
offending value: list_1 = []
Traceback:
   main()
   {1 + list_1 = []} from line 21 in preproc-line-2.icn
```

One last time; the `$line` directive is of most use with machine generated sources.

9.1.9 $undef

`$undef` *symbol*

Undefine a preprocessor symbol.

```
#
# preproc-undef, demonstrate $undef preprocessor directive
#
procedure main()
$define STUB 1
$ifndef STUB
    write("skip over some code")
$else
    write("Unicon, cui non; sold, in Italian")
$undef STUB
$endif

$ifndef STUB
    write("don't skip over this code")
$else
    write("do skip over this code")
$endif
end
```

Sample output:

```
prompt$ unicon -E -s preproc-undef.icn
#line 0 "/tmp/uni43851559"
#line 0 "preproc-undef.icn"
```

```
procedure main();
    write("Unicon, cui non; sold, in Italian");
    write("don't skip over this code");
end
```

9.1.10 #line

`#line` *line-number filename*

`#line` is an internal (not meant for programmers at the source level) directive keyword that manages post preprocessor line and filename semantics for the compiler proper.

You will see the directive with `unicon -E` output but it is not for general use.

9.2 Predefined symbols

Predefined symbols (for use with *$ifdef*, and *$ifndef*) are provided for each platform, and for any optional features compiled into the running version of Unicon.

Testable symbols include:

Symbol	Feature
_MULTITASKING	*multiple programs*
_X_WINDOW_SYSTEM	*X Windows*
_GRAPHICS	*graphics*
_PIPES	*pipes*
_ARM_FUNCTIONS	*Archimedes extensions*
_MS_WINDOWS	*MS Windows*
_MVS	*MVS*
_MSDOS_386	*MS-DOS/386*
_EVENT_MONITOR	*event monitoring*
_POSIX	*POSIX*
_KEYBOARD_FUNCTIONS	*keyboard functions*
_OS2	*OS/2*

Continued on next page

Table 9.1 – continued from previous page

Symbol	Feature
_MACINTOSH	*Macintosh*
_VMS	*VMS*
_DYNAMIC_LOADING	*dynamic loading*
_EBCDIC	*EBCDIC*
_EXTERNAL_FUNCTIONS	*external functions*
_SYSTEM_FUNCTION	*system function*
_CMS	*CMS*
_MSDOS	*MS-DOS*
_PORT	*PORT*
_V9	*Version 9*
_CONSOLE_WINDOW	*console window*
_DOS_FUNCTIONS	*MS-DOS extensions*
_MESSAGING	*messaging*
_MS_WINDOWS_NT	*MS Windows NT*
_RECORD_IO	*record I/O*
_ACORN	*Acorn Archimedes*
_DBM	*DBM*
_CO_EXPRESSIONS	*co-expressions*
_AMIGA	*Amiga*
_ASCII	*ASCII*
_WIN32	*Win32*
_UNIX	*UNIX*
_PRESENTATION_MGR	*Presentation Manager*
_LARGE_INTEGERS	*large integers*

9.2.1 Substitution symbols

There some predfined symbols that will cause substitution during the preprocessor pass.

Symbol	Replacement
__DATE__	Current date in `YYYY/MM/DD` form
__TIME__	Current time is `hh:mm:ss` form

```
#
# preproc-symbols, demonstrate preprocessor substitution symbols
#
procedure main()
    write(__DATE__)
    write(__TIME__)
end
```

Sample output:

```
prompt$ unicon -E -s preproc-symbols.icn
#line 0 "/tmp/uni43851559"
#line 0 "preproc-symbols.icn"
```

```
procedure main();
    write("2019/10/27");
    write("04:54:16");
end
```

9.3 EBCDIC transliterations

Some legacy keyboards do not include `{`, `}`, `[`, or `]` characters, vital for Unicon programming. The preprocessor will replace:

- `$(` with `{`
- `$)` with `}`
- `$<` with `[`
- `$>` with `]`

This transliteration only applies to EBCDIC system builds.

CHAPTER TEN

DEVELOPMENT TOOLS

10.1 Unicon tools

The main Unicon command set is made up of:

- unicon
- icont
- iconx
- iconc

Todo

complete the list of core tools

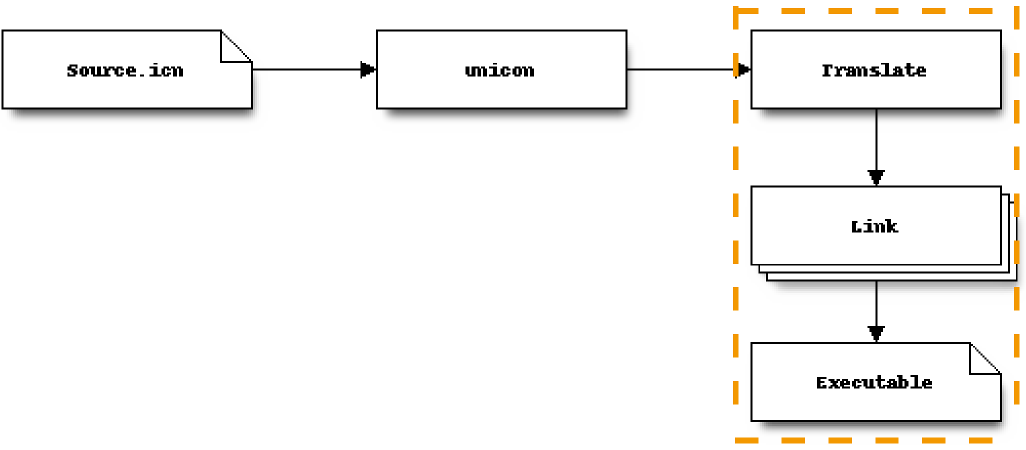
After a source build with Unicon 13, the `bin/` directory includes:

```
prompt$ ls -gGF --time-style=+
total 7156
-rw-rw-r-- 1    5560  dlrgint.o
-rwxrwxr-x 1  349080  iconc*
-rwxrwxr-x 1  171032  icont*
-rwxrwxr-x 1 1205944  iconx*
-rwxrwxr-x 1   48671  ie*
-rwxrwxr-x 1  123864  iyacc*
-rwxrwxr-x 1   28296  libcfunc.so*
-rw-rw-r-- 1   77318  libgdbm.a
-rw-rw-r-- 1   70304  libtp.a
-rw-rw-r-- 1  144424  libucommon.a
-rw-rw-r-- 1    3720  libuconsole.a
-rw-rw-r-- 1  128000  libXpm.a
-rwxrwxr-x 1   10688  patchstr*
-rw-rw-r-- 1 3599918  rt.a
```

```
-rw-rw-r-- 1  159975  rt.db
-rw-rw-r-- 1  322929  rt.h
-rwxrwxr-x 1  246776  rtt*
-rwxrwxr-x 1  263687  udb*
-rwxrwxr-x 1   39298  umake*
-rwxrwxr-x 1  207995  unicon*
-rwxrwxr-x 1   79962  uprof*
```

10.1.1 The unicon command

unicon is the main command line tool for programming with Unicon.



Options include:

```
prompt$ unicon --help
Usage: unicon [-cBCstuEGyZMhK] [-Dsym=val] [-f[adelns]...] [-o ofile]
   [-nofs] [-help] [-version] [-features] [-v i] file... [-x args]
options may be one of:
   -B           : bundle VM (iconx) into executable
   -c           : compile only, do not link
   -C           : generate (optimized) C code executable
   -Dsym[=val]  : define preprocessor symbol
   -e efile     : redirect icont's standard error output to efile
   -E           : preprocess only, do not compile
   -features    : report Unicon features supported in this build
   -fs          : prevent removal of unreferenced declarations
   -G           : generate graphics (wiconx) executable
   -M           : report error message to the authorities
   -o ofile     : generate executable named ofile
   -O           : optimize (under construction)
   -s           : work silently
   -t           : turn on tracing
   -u           : warn of undeclared variables
   -v i         : set diagnostic verbosity level to i
   -version     : report Unicon version
```

```
   -x args     : execute immediately
   -y          : parse (syntax check) only, do not compile
   -Z          : compress icode
   -K           : keep tmpfiles
   -h          : display this information
```

unicon is a wrapper around other tools, mainly the *icont* command. Many of the command lines options are similar, but **unicon** does a better job of explaining the options in the brief help.

Compiled unicon

`C` compiled Unicon, to be more specific, seeing as Unicon already compiles.

unicon -C translates Unicon to C intermediates and then compiles the C code (usually via native assembler source) to native executable for the system in use. Some Unicon features are not available when using **unicon -C**, such as native concurrency at this point in time.

Quiet unicon

Yeah, this is was a little tricky.

Update: as of revision [r4497] *this entire sequence has been simplified*. Just use **unicon -s**.

Left in for anyone running pre [r4497] Unicon.

```
prompt$ unicon version.icn
Parsing version.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O version.icn /tmp/uni15766793
Translating:
version.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  version.u
Linking:
```

Let's try `-quiet` (it's undocumented in `-help`)

```
prompt$ unicon -quiet version.icn
Translating:
version.icn:
  main
No errors
Linking:
```

Umm, not quite, how about `-s`, *work silently*.

```
prompt$ unicon -quiet -s version.icn
```

So close. Now to set verbosity to 0.

```
prompt$ unicon -quiet -s -v0 version.icn
```

Ahh, that'd be the ticket. Quiet, silent, not verbose.

```
-quiet -s -v0
```

```
prompt$ unicon -quiet -s -v0 version.icn -x
Unicon Version 13.1.  August 19, 2019
```

Update as above: Just use **unicon -s**.

The flip side is pretty easy; maximum verbosity.

```
prompt$ unicon -v3 version.icn -x
Parsing version.icn: .
/home/btiffin/unicon-git/bin/icont -c  -v3  -O version.icn /tmp/uni15766793
Translating:
version.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont -v3  version.u -x
Linking:
  bootstrap      344
  header         216
  procedures     152
  records          8
  fields           0
  globals         64
  statics          0
  linenums        80
  strings         23
  total          887
Executing:
Unicon Version 13.1.  August 19, 2019
```

10.1.2 The icont command

icont is the translator portion of the Unicon tool chain.

Options include:

```
prompt$ icont
usage: icont [-cBstuEG] [-f s] [-l logfile] [-o ofile] [-v i] file ... [-x args]
```

icont is a relatively sophisticated compiler, most error messages are quite comprehensive, detailing what it wrong with the source code.

```
#
# Compile time error message example
#
procedure main()
    a ::= "1" + []
end
```

```
prompt$ icont compiletime-error.icn
Translating:
compiletime-error.icn:
compiletime-error.icn:12: # "a": syntax error (91;366)
File compiletime-error.icn; Line 14 # traverse: undefined node type
```

`::=` is not valid assignment syntax.

Like most high level compilers, sometimes the error messages are, *less helpful*:

Todo

ADD SAMPLE

And sometimes, you need to wait until execution time, and triggering an error at run-time.

```
#
# Runtime error message example
#
procedure main()
    a := "1" + []
end
```

Sample run, with error:

```
prompt$ unicon runtime-error.icn -x
Parsing runtime-error.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O runtime-error.icn /tmp/uni16957730
Translating:
runtime-error.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  runtime-error.u -x
Linking:
Executing:

Run-time error 102
File runtime-error.icn; Line 12
numeric expected
offending value: list_1 = []
Traceback:
   main()
   {1 + list_1 = []} from line 12 in runtime-error.icn
```

Integer can't be added to a *List (arrays)* datatype, *in this case, an integer coerced from a string*.

See *Testing Unicon* for some details on testing `Unicon` programs.

10.1.3 The iconx command

`iconx` is the Executor portion of the Unicon tool chain.

`iconx` takes the filename to execute.

The Icon Virtual machine, and **`iconx`**, report any runtime errors, in a relatively comprehensive manner.

```
#
# Runtime error message example
#
procedure main()
    a := "1" + []
end
```

```
prompt$ unicon -c runtime-error.icn
Parsing runtime-error.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O runtime-error.icn /tmp/uni16957730
Translating:
runtime-error.icn:
  main
No errors
```

iconx processes *icode* files, which are binary executable forms created during the link phase from *ucode* files.

```
prompt$ iconx runtime-error

Run-time error 102
File runtime-error.icn; Line 12
numeric expected
offending value: list_1 = []
Traceback:
   main()
   {1 + list_1 = []} from line 12 in runtime-error.icn
```

A segue into `.u` files, *ucode*, and the Icon Virtual Machine.

`.u` files are Icon cross platform machine instructions, as human readable source (this feature helped with the design and debugging of the virtual machine).

For example, the above buggy `runtime-error.icn` was translated to `runtime-error.u`, and is a human readable form of the Icon Virtual Machine language.

```
version        U12.1.00
uid        runtime-error.u1-1480545330-0
impl        local
global        1
        0,000005,main,0

proc main
        local        0,000000,a
        con        0,010000,1,061
        declend
        filen        runtime-error.icn
        line        11
        colm        11
        synt        any
        mark        L1
        pnull
        var        0
        pnull
        str        0
        pnull
        line        12
        colm        16
        synt        any
        llist        0
        line        12
        colm        14
        synt        any
        plus
        line        12
        colm        7
```

```
        synt        any
        asgn
        unmark
lab L1
        pnull
        line        13
        colm        1
        synt        any
        pfail
        end
```

If you look at the first few lines of a **unicon** compiled program, it is actually a script that starts **iconx** with the *ucode* embedded in it.

```
prompt$ unicon runtime-error.icn
Parsing runtime-error.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O runtime-error.icn /tmp/uni16957730
Translating:
runtime-error.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  runtime-error.u
Linking:
```

```
prompt$ cat -v runtime-error | par
#!/bin/sh IXBIN=/home/btiffin/unicon-git/bin/iconx IXLCL=`echo $0 | sed
's=[^/]*$=iconx='`

[ -n "$ICONX" ] && exec "$ICONX" [ -x "$IXLCL" ] && exec  "$0" ${1+"$@"}
"$IXLCL" [ -x "$IXBIN" ] && exec "$IXBIN" exec iconx      "$0" ${1+"$@"}

[executable Icon binary follows]

^L
^@[^A^@^@^@^@^@^@^@^@^@^@^@^@^@M-8^@^@^@^@^@^@M-@^@^@^@^@^@^@M-@^@
^^@^@^@^@^@M-@^@^@^@^@^@^@M-P^@^@^@^@^@^@M-`^@^@^@^@^@^@@^A^@^@^
^@^@^@^@M-`^@^@^@^@^@^@^@M-p^@^@^@^@^@^@I12.U.30/32/64^@^@@M-^CM-F^A^@
^^@^@^@^@^@^@^@^@^@^@^PM-FM-E^A^@^@^@^@M-^@pM-F^A^@^@^@^@M-^@M-@M-E^A^
^@^@^@@M-@M-E^A^@^@^@^@<UM-,M-:!^?^@^@^PM-FM-E^A^@^@^@^@^@^@^@^@^@^@
^^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@@M-^CM-F^A^@^@^@^@@M-^CM-F^A^@^@^@^@^
^@^@^@^@^@^@^@^F^@^@^@^@^@^@X^@^@^@^@^@^@X^@^@^@^@^@^@^@^@^@^@
^^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^D^@^@^@^@^@^@^
^@^@^@^@^@^@^@^A^@^@^@^@^@^@^E^@^@^@^@^@^@b^@^@^@C^@^@^@D^@^@^@^@^
^@^@^@E^@^@^@S^@^@^@^@^@^@^@^@^@^@E^@^@^@M^@^@^@^A^@^@^@^@^@^@^G^@^@
^^@^@^@^@^@E^@^@^@A^@^@^@^@^@^@^@^@^@^@^^^@^@^@^A^@^@^@N^@^@^@E^@^@^@D
^^@^@^@^@^@^@^@^@^@^@^@^@^@^@^F^@^@^@^@^@^@M-0^@^@^@^@^@^@^@^@^D^@^@^@
^^@^@^@^@^@^@^@^@^@^@^@X^@^@^@^@^@^@^@
^^@^@^@^@^@^@^@X^@^@^@^@^@^@^@^K^@`^A^@^@^@^@M-^T^@^@^@^@^@^@^@^L^@^@^B^
^@^@^@^@M-
^^@^@^@^@^@^@^@^L^@M-@^A^@^@^@^@M-$^@^@^@^@^@^@^@^L^@M-`^@^@^@^@^@M-0^@^
^@^@^@^@^@^@^M^@ ^@^@^@^@^@main^@a^@1^@runtime-error.icn^@
```

10.1.4 Coding conventions and style

Unicon has a long history, and *Ralph Griswold* tried to keep a consistent look and feel to both the compiler internal C/rtt sources and the high level Icon programs submitted to the *IPL*. That effort was not 100% successful, and the many hands that have touched on the C code each brought a slightly different style preference. One of the most startling format issue may be the mixing of tab characters and spaces for code indentation. The convention guide (https://www2.cs.arizona.edu/icon/docs/ipd072.htm) requests spaces, but there are tabs in the sources and over time, various text editors have made the code look a little sloppy in terms of nested indentations. Keep to the request, and use spaces, as it avoids this long term churn issue with source code and text editor treatment of tabs.

IPL entries seem to have fared better over the years in terms of consistent look. That is likely due to the more public nature and numbers of interested readers, when compared to the internal source files.

In summary:

- Indentation: Three spaces per level, with no tabs for indentation.
- Line length: Not to exceed 80 characters with tabs expanded.
- Inter-line spacing: At most one empty line as needed for visual clarity. Formfeed (^L) between function declarations (except where there are many short, similar functions). Generally one empty line prior to boxed comments.
- Function declarations: storage/type class on line with function name. No indentation on argument declarations. Function body, including braces and local declarations indented three spaces. Braces enclosing functions on lines by themselves.
- Braces: Beginning brace at end of line for construction involved (except for function declarations as noted above). Ending brace on separate line indented three spaces beyond position of opening construction.
- `if` statements: statement below conditional expression, indented three spaces. If `else` is present, on separate line aligned with corresponding `if`.
- Binary operators: one space on either side.
- Argument lists: one space after each comma.
- Return statements: no parentheses around argument of `return`.
- Casts: no space between right parenthesis surrounding cast and the expression to which it applies.
- `typedefs`: generally all lowercase (a deliberate departure from usual C conventions; typedefs in the Icon source code are viewed as an extension of the C language).
- Comments: Boxed-style per numerous examples. Second and subsequent lines indented one character. "On-line" comments tabbed out as is most readable. No "cute" or illiterate comments. No personal identifications (they cease to be useful with time and produce cumulative clutter).
- Conditional compilation: One empty line before and after `#ifdef/endif` groups. Identifying C-style comment five tabs out on `#else` and `#endif` directives. (*I think by 5 tabs out, Ralph meant column 40*).
- Manifest (defined) constants and macros: For the most part, names should be in mixed upper- and lowercase (a deliberate departure from usual C conventions). Specifically, an uppercase letter identifies a name that has been defined as opposed to a C variable. Mixed case is used for readability.

This convention guide carries over to Unicon sources and *IPL* entries, where appropriate. In particular, IPL entries all start with a consistent comment block, formatted to allow the indexing tool to properly do its job. See the `ipl-skeleton.icn` listed below for this formatting standard.

Note: The examples in this docset all break rule one. I prefer 4 space indents. The bracing rule is also broken, preferring to not indent the closing brace, but to keep it at the same indentation as the starting construct.

Apologies, but that is the way it is.

With that said, reasonable effort is expended to maintain an internal consistency.

The convention guide calls for code formatting ala:

```
while line := read(f) do {
   result := stuff(line)
   morestuff(result)
   }
continuing()
```

This document uses a style of the form:

```
while line := read(f) do {
    result := stuff(line)
    morestuff(result)
}
continuing()
```

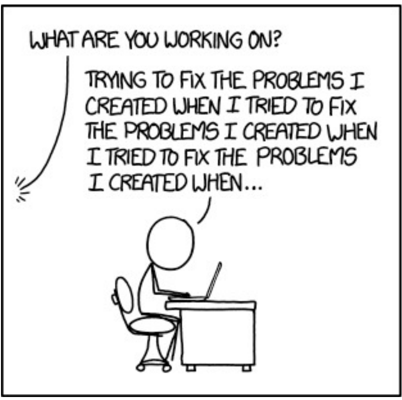
Blame age and long ingrained personal preference for the breach of project preferred convention.

It behooves any Unicon programmer to get used to many forms and coding styles, as for the most part, due to many hands in the pot, you may encounter different formatting in any one particular source file. A key issue is to strive for consistency and to attempt some level of source formatting discipline.

10.2 Supporting tools

Unicon works very well with most of the common software development tools available for various operating systems. This docset focuses on GNU/Linux, but other systems also offer a wide range of useful development aids.

Unicon also ships with a fair number of support tools, custom built for Unicon development.




```
"'What was the original problem you were trying to fix?' 'Well, I noticed one
of the tools I was using had an inefficiency that was wasting my time'"
```

XKCD http://xkcd.com/1739/ by Randall Munroe *CC BY-NC 2.5*

10.2.1 make

make is not a Unicon specific program, but works very well with Unicon development. **make** is also used when building Unicon from source.

See https://www.gnu.org/software/make/manual/make.html for details of the ubiquitous GNU implementation of make.

make normally uses a non visible Tab character (ASCII 9) to prefix action lines. GNU make includes an extension to specify a different prefix character. The example below, and other examples throughout this documentation use a more visible > action statement prefix. This is not supported in all versions of **make**, but it makes copy and paste from web pages a little safer, as it removes any chance of a Tab character being mistaken for spaces.

Some sample recipes:

```
# Unicon programming starter Makefile sample
.RECIPEPREFIX = >

# quick help, first target is default target, "make" or "make help"
.PHONY: help
help:
> @echo "Unicon example Makefile"
> @echo
> @echo "make help\t\tfor this help"
> @echo "make program\t\tto build program.icn"
> @echo "make run\t\tto run executable from program.icn"
> @echo
> @echo "make name\t\tto build and run any given named .icn file"
> @echo "make -B name\t\twill force the run, even if already up to date"
> @echo "make name.u\t\tto build a link file for any given named .icn file"

# make Unicon program, $< is the first prereq
program: program.icn
> unicon $<

run: program
> ./$<

# generic rules
# build and run any given .icn file by typing "make name"
%: %.icn
> unicon -s $^ -x

# create a linkable Unicon file by typing "make name.u"
%.u: %.icn
> unicon -c $^
```

make program will compile `program.icn` if the source is more recent than the target executable, **make run** will run the program (and ensure it is up to date before execution, if sources have changed). **$<** is an automatic **make** variable, set to match the first prerequisite. For the `program:` target, the first (and only) prerequisite is `program.icn`. For the sample `run:` target, the prerequisite is `program`, and **$<** is substituted in to execute **./program**.

make is a fairly powerful tool. There are quite a few features that may not be apparent at first glance. See the GNU make manual referenced above for more details.

A special note on the trick `%: %.icn` and `%.u: %.icn` targets. Those are **make** patterns that match any named Unicon source file, `name.icn`, and compiles and/or runs the program. It is a convenience feature for non-project related make. It allows any Unicon file to be compiled and run by simply typing **make name**. If the source has already been compiled (date of executable is later than the date time of last edit), just use **./name**, or use **make**

-B name to force a new compile and run. To compile a source file for use with *link*, the same trick allows **make name.u**, to match the `name.icn` file, and then run **unicon -c** on the given name.

Again, those are convenience features, normally a Makefile is specific to a project and the targets will include specific instructions for clean up, packaging, installation, etcetera, and will list all prerequisite source files, not singletons as used with the trick filename matching rules.

10.2.2 ui

The Unicon IDE. **ui** ships with Unicon, under the `uni/ide` subdirectory, and is built along with **unicon** and the other command line tools, ready to use with any Unicon installation that supports the 2D graphics facilities. A graphical front end for Unicon development, detailed in Technical Report UTR12a, http://unicon.org/utr/utr12.html

The **Ui** source code provides lots of examples for extending and customizing a graphical user interface generated using *IVIB*, the interface building tool.

Ui supports project management, file management, a contextual help based source code editor, a class browser, along with Compile, Run and Debug layers.

I'll be honest, I develop on the command line, with Vim and associated tools. If you prefer Integrated Development Environments, be sure to fire up **ui**.

Might have to pester Clinton about opening up the font selection list; the four X11 fonts that ship with Unicon are not as beautiful as some that are now available for GNU/Linux.

10.2.3 UDB

The Unicon debugger. A very comprehensive Unicon support tool.

Modelled on `GNU gdb`, **udb** allows breakpoints, Unicon source view, structure display, and many other high and low level debugging features using a command prompt interface.

UDB, by Ziad Al-Sharif, is documented in Unicon Technical Report 10.

http://unicon.org/utr/utr10.html

See *Debugging* for more details.

IVIB

A visual, graphical user interface building tool. Drag and drop user interface elements to generate Unicon code.

10.3 The Icon Virtual Machine

10.3.1 ucode

ucode is source form virtual machine instructions, transportable between all platforms. The translator generates ucode as an intermediate form ready for the link stage. `ucode` was named before Unicon, as part of the Icon project.

Starting small, smaller than Hello in this case.

```
#
# onestatement.icn, a single statement
#
# tectonics:
#     unicon -c onestatement.icn
#
procedure onestatement()
    u := 1
end
```

Producing *ucode* from this source gives a close to minimum set of VM instructions required for the Unicon VM.

Processed via **unicon -c**

```
prompt$ unicon -c onestatement.icn ; sed -i 's/\x0C//' onestatement.u
Parsing onestatement.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O onestatement.icn /tmp/uni15766793
Translating:
onestatement.icn:
  onestatement
No errors
```

Generates a link ready *ucode* file, `onestatement.u`.[1]

```
version        U12.1.00
uid        onestatement.u1-1572166087-0
impl         local
global         1
        0,000005,onestatement,0

proc onestatement
        local         0,000000,u
        con         0,002000,1,1
        declend
        filen         onestatement.icn
        line         14
        colm         11
        synt         any
        mark         L1
        pnull
        var         0
        int         0
        line         15
        colm         7
        synt         any
        asgn
```

[1] Unicon merged *ucode* into a single file. Icon, separates some of the paperwork from the instructions, giving `.u1` and `.u2` files. Having a single `.u` file makes things a little cleaner and more easily transportable.

```
        unmark
lab L1
        pnull
        line        16
        colm        2
        synt        any
        pfail
        end
```

This assigns the value `1` to the local variable, `u`.

The *ucode* file also includes some identification, some procedure paperwork, the statement trace info, and the statement bounding markers, along with the actual low level instructions to assign a `1` to `u`.

The Unicon virtual machine is stack based. An operational stack is used very much like a Forth data stack. Pushing and popping values (which can be encoded or literal) while performing virtual machine level operations that functionally match the high level Unicon source instructions.

The action opcodes here being

```
pnull
var 0
int 0
asgn
```

Push a null descriptor (a place holder for the expression result). Push the id of variable 0 from the declarations (the `u`). Push an integer with id 0 from the constant declarations (a `1` in this case). Then assign; `asgn` pops the value and destination while performing the assignment operation and replaces the initial null descriptor with the result (which in more complex programs would be chained to other parts of an expression).

ucode is the human friendly version of *icode*, the actual byte-code used by the virtual machine. During initial development of Icon, starting back in 1978ish, it was deemed important to have a human readable version of the VM instruction sequencing. This helped (and still helps) verify and debug the Icon source code translator. A nice touch, not really necessary for the virtual machine itself, but of great importance to the people implementing the engine and the translator. And a bonus to developers that can peruse the machine instructions.

As is, this *ucode* requires one more step before being executable code. The *rt* runtime translator at the heart of the Unicon VM always starts with *main*. It gives the engine a starting point, a place to call home that ensures everything is properly initialized. The translation from *ucode* to *icode* happens during the linking phase of the Unicon tool chain. All *link* references are included, and *ucode* files merged and resolved down to *icode*.

Many Unicon programs are singletons, no *link* statements to worry about, but every final step *icode* generation requires a *main*. In this case, the single statement procedure is compiled separately and included in the file listed below via the *link* reserved word.

```
#
# mainstatement.icn, link to a single statement procedure
#
# tectonics:
#     unicon -c onestatement.icn
#     unicon mainstatement.icn
#
link onestatement
procedure main()
    mainstatement()
end
```

That piece of code will link to `onestatement.u` and produce an executable VM image, `mainstatement`. `mainstatement` sits ready to be invoked by the operating system to run the program. A program that invokes

main (implicitly), which invokes *onestatement*, which sets `u` to `1`, then runs down, giving control back to the operating system.

For now, we are more interested in looking at the *ucode*, so **unicon -c** comes into play again.

```
prompt$ unicon -c mainstatement.icn ; sed -i 's/\x0C//' mainstatement.u
Parsing mainstatement.icn: ..
/home/btiffin/unicon-git/bin/icont -c   -O mainstatement.icn /tmp/uni14575856
Translating:
mainstatement.icn:
  main
No errors
```

```
version        U12.1.00
uid        mainstatement.u1-1572166087-0
impl         local
link         onestatement.u
global         1
        0,000005,main,0

proc main
        local        0,000000,mainstatement
        declend
        filen        mainstatement.icn
        line        16
        colm        12
        synt        any
        mark        L1
        var        0
        line        17
        colm        18
        synt        any
        invoke        0
        unmark
lab L1
        pnull
        line        18
        colm        1
        synt        any
        pfail
        end
```

We don't really have an executable yet, the Unicon tool chain stopped after producing the *mainstatement ucode* and did not combine the sources with the runtime engine, nor generate *icode*.

```
prompt$ unicon mainstatement.icn

Parsing mainstatement.icn: ..
/home/btiffin/unicon/bin/icont -c   -O mainstatement.icn /tmp/uni18898220
Translating:
mainstatement.icn:
  main
No errors
/home/btiffin/unicon/bin/icont  mainstatement.u
Linking:
```

Without `-c`, Unicon does the complete pass, including the VM link phase, and creates a new program, ready to run.

```
./mainstatement
```

That pass does nothing visible, but did all kinds of nifty things in the background.

Hello

Now making it a little more complicated is the Hello, world of *ucode*.

```
#
# hellostatement.icn, a single statement, proof of life
#
# tectonics:
#     unicon -c hellostatement.icn
#
procedure hellostatement()
    write("Hello, world")
end
```

Producing *ucode* from this source gives just enough VM instructions for Unicon to create a procedure that displays evidence that a proper installation of Unicon is functional.

Processed via **unicon -c**:

```
prompt$ unicon -c hellostatement.icn ; sed -i 's/\x0C//' hellostatement.u
Parsing hellostatement.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O hellostatement.icn /tmp/uni15766793
Translating:
hellostatement.icn:
  hellostatement
No errors
```

Generates a link ready *ucode* file, `hellostatement.u`.

```
version        U12.1.00
uid        hellostatement.u1-1572166087-0
impl        local
global        1
        0,000005,hellostatement,0

proc hellostatement
        local        0,000000,write
        con        0,010000,12,110,145,154,154,157,054,040,167,157,162,154,144
        declend
        filen        hellostatement.icn
        line        14
        colm        11
        synt        any
        mark        L1
        var        0
        str        0
        line        15
        colm        10
        synt        any
        invoke        1
        unmark
lab L1
        pnull
```

```
        line        16
        colm        1
        synt        any
        pfail
        end
```

Linking that with a `main` that calls the small procedure and we have a Unicon virtual machine program to tell the world that Unicon is functioning properly (and in this case, doubly so; the translator works and the linker works, along with the runtime engine).

```
#
# hellomain.icn, link to a write statement procedure
#
# tectonics:
#     unicon -c hellostatement.icn
#     unicon hellomain.icn
#
link hellostatement
procedure main()
    hellostatement()
end
```

Drum roll...

```
prompt$ unicon -c hellostatement.icn ; unicon hellomain.icn -x ; sed -i 's/\x0C//'
↪hellostatement.u
Parsing hellostatement.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O hellostatement.icn /tmp/uni15766793
Translating:
hellostatement.icn:
  hellostatement
No errors
Parsing hellomain.icn: ..
/home/btiffin/unicon-git/bin/icont -c   -O hellomain.icn /tmp/uni14575856
Translating:
hellomain.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  hellomain.u -x
Linking:
Executing:
Hello, world
```

From Unicon source files, through *ucode*, to linked *icode* and final creation of an executable file. Then automatically invoked (via `-x`) to demonstrate a functional Unicon installation by way of a friendly greeting.

10.3.2 icode

icode is binary form virtual machine instructions, and have an almost one to one correspondence with *ucode* source. Some differences will occur during linkage, the final phase of Unicon translation before executable. The executor, *rt*, is the runtime engine invoked by *The iconx command* that evaluates *icode*. When using GNU/Linux, Unicon create an executable shell script, that assists with search path settings, and then runs the included binary *icode* machine instructions.

The *hellomain* file (reformatted for document capture):

```
prompt$ ./show-icode hellomain
#!/bin/sh
IXBIN=/home/btiffin/unicon-git/bin/iconx
IXLCL=`echo $0 | sed 's=[^/]*$=iconx='`

[ -n "$ICONX" ] && exec "$ICONX" "$0" ${1+"$@"}
[ -x "$IXLCL" ] && exec "$IXLCL" "$0" ${1+"$@"}
[ -x "$IXBIN" ] && exec "$IXBIN" "$0" ${1+"$@"}
exec iconx "$0" ${1+"$@"}

[executable Icon binary follows]

0000: 00 58 02 00 00 00 00 00 00 00 00 00 00 00 00 00  .X..............
0016: 00 28 01 00 00 00 00 00 00 30 01 00 00 00 00 00  .(.......0......
0032: 00 30 01 00 00 00 00 00 00 30 01 00 00 00 00 00  .0.......0......
0048: 00 60 01 00 00 00 00 00 00 90 01 00 00 00 00 00  .`..............
0064: 00 10 02 00 00 00 00 00 00 90 01 00 00 00 00 00  ................
0080: 00 b0 01 00 00 00 00 00 00 49 31 32 2e 55 2e 33  .........I12.U.3
0096: 30 2f 33 32 2f 36 34 00 00 b0 a4 bc 01 00 00 00  0/32/64.........
0112: 00 00 00 00 00 00 00 00 00 20 e6 bb 01 00 00 00  ......... ......
0128: 00 70 a3 bc 01 00 00 00 00 80 e0 bb 01 00 00 00  .p..............
0144: 00 40 e0 bb 01 00 00 00 00 3c d5 2c ed cb 7f 00  .@.......<.,....
0160: 00 20 e6 bb 01 00 00 00 00 00 00 00 00 00 00 00  . ..............
0176: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0192: 00 b0 a4 bc 01 00 00 00 00 b0 a4 bc 01 00 00 00  ................
0208: 00 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00  ................
0224: 00 48 00 00 00 00 00 00 00 48 00 00 00 00 00 00  .H.......H......
0240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0256: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0272: 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0288: 00 62 00 00 00 43 00 00 00 24 00 00 00 00 00 00  .b...C...$......
0304: 00 62 00 00 00 54 00 00 00 01 00 00 00 00 00 00  .b...T..........
0320: 00 62 00 00 00 3d 00 00 00 00 00 00 00 00 00 00  .b...=..........
0336: 00 4e 00 00 00 45 00 00 00 44 00 00 00 00 00 00  .N...E...D......
0352: 00 06 00 00 00 00 00 00 00 48 00 00 00 00 00 00  .........H......
0368: 00 d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0384: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0400: 00 00 00 00 00 00 00 00 00 0e 00 00 00 00 00 00  ................
0416: 00 05 00 00 00 00 00 00 00 62 00 00 00 43 00 00  .........b...C..
0432: 00 3c 00 00 00 00 00 00 00 62 00 00 00 54 00 00  .<.......b...T..
0448: 00 02 00 00 00 00 00 00 00 62 00 00 00 4d 00 00  .........b...M..
0464: 00 0c 00 00 00 00 00 00 00 28 00 00 00 00 00 00  .........(......
0480: 00 62 00 00 00 3d 00 00 00 01 00 00 00 00 00 00  .b...=..........
0496: 00 4e 00 00 00 45 00 00 00 44 00 00 00 00 00 00  .N...E...D......
0512: 00 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00  ................
0528: b0 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00  ................
0544: b0 88 00 00 00 00 00 00 00 06 00 00 00 00 00 00  ................
0560: b0 a7 ff ff ff ff ff ff ff 04 00 00 00 00 00 00  ................
0576: 00 00 00 00 00 00 00 00 00 0e 00 00 00 00 00 00  ................
0592: 00 05 00 00 00 00 00 00 00 05 00 00 00 00 00 00  ................
0608: 00 14 00 00 00 00 00 00 00 48 00 00 00 00 00 00  .........H......
0624: 00 1a 00 00 00 00 00 00 00 d0 00 00 00 00 00 00  ................
0640: 00 35 00 00 00 00 00 00 00 48 00 00 00 00 00 00  .5.......H......
0656: 00 10 00 80 01 00 00 00 00 68 00 00 00 00 00 00  .........h......
0672: 00 11 00 60 02 00 00 00 00 80 00 00 00 00 00 00  ...`............
0688: 00 12 00 20 00 00 00 00 00 d0 00 00 00 00 00 00  ... ............
0704: 00 0e 00 60 01 00 00 00 00 08 01 00 00 00 00 00  ...`............
0720: 00 0f 00 40 01 00 00 00 00 20 01 00 00 00 00 00  ...@..... ......
```

```
0736: 00 10 00 20 00 00 00 00 00 6d 61 69 6e 00 68 65  ... .....main.he
0752: 6c 6c 6f 73 74 61 74 65 6d 65 6e 74 00 77 72 69  llostatement.wri
0768: 74 65 00 68 65 6c 6c 6f 6d 61 69 6e 2e 69 63 6e  te.hellomain.icn
0784: 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 00 68 65  .Hello, world.he
0800: 6c 6c 6f 73 74 61 74 65 6d 65 6e 74 2e 69 63 6e  llostatement.icn
0816: 00                                               .
```

icode support files

icode is based on single byte operation codes.

```
/*
 * Opcode definitions used in icode.
 */

/*
 * Operators. These must be in the same order as in odefs.h.  Not very nice,
 *  but it'll have to do until we think of another way to do this.  (It's
 *  always been thus.)
 */
#define Op_Asgn                  1
#define Op_Bang                  2
#define Op_Cat                  3
#define Op_Compl           4
#define Op_Diff                  5
#define Op_Div                  6
#define Op_Eqv                  7
#define Op_Inter           8
#define Op_Lconcat           9
#define Op_Lexeq          10
#define Op_Lexge          11
#define Op_Lexgt          12
#define Op_Lexle          13
#define Op_Lexlt          14
#define Op_Lexne          15
#define Op_Minus          16
#define Op_Mod                 17
#define Op_Mult                 18
#define Op_Neg                 19
#define Op_Neqv                 20
#define Op_Nonnull          21
#define Op_Null                 22
#define Op_Number          23
#define Op_Numeq          24
#define Op_Numge          25
#define Op_Numgt          26
#define Op_Numle          27
#define Op_Numlt          28
#define Op_Numne          29
#define Op_Plus                 30
#define Op_Power          31
#define Op_Random          32
#define Op_Rasgn          33
#define Op_Rcv                 34
#define Op_RcvBk          35
#define Op_Refresh          36
#define Op_Rswap          37
```

```
#define Op_Sect                  38
#define Op_Snd                   39
#define Op_SndBk         40
#define Op_Size                  41
#define Op_Subsc         42
#define Op_Swap                  43
#define Op_Tabmat         44
#define Op_Toby                  45
#define Op_Unions         46
#define Op_Value         47
/*
 * Other instructions.
 */
#define Op_Bscan         117
#define Op_Ccase         118
#define Op_Chfail         119
#define Op_Coact          120
#define Op_Cofail         48
#define Op_Coret          49
#define Op_Create         50
#define Op_Cset           51
#define Op_Dup            52
#define Op_Efail          53
#define Op_EInit          116
#define Op_Eret           54
#define Op_Escan          55
#define Op_Esusp          56
#define Op_Field          57
#define Op_Goto           58
#define Op_Init           59
#define Op_Int            60
#define Op_Invoke         61
#define Op_Keywd          62
#define Op_Limit          63
#define Op_Line           64
#define Op_Llist          65
#define Op_Lsusp          66
#define Op_Mark           67
#define Op_Pfail          68
#define Op_Pnull          69
#define Op_Pop            70
#define Op_Pret           71
#define Op_Psusp          72
#define Op_Push1          73
#define Op_Pushn1          74
#define Op_Real            75
#define Op_Sdup            76
#define Op_Str             77
#define Op_Unmark          78
#define Op_Var             80
#define Op_Arg                   81
#define Op_Static         82
#define Op_Local         83
#define Op_Global         84
#define Op_Mark0         85
#define Op_Quit                  86
#define Op_Tally         88
#define Op_Apply         89
```

```
/*
 * "Absolute" address operations.  These codes are inserted in the
 * icode at run-time by the interpreter to overwrite operations
 * that initially compute a location relative to locations not known until
 * the icode file is loaded.
 */
#define Op_Acset         90
#define Op_Areal         91
#define Op_Astr                 92
#define Op_Aglobal        93
#define Op_Astatic        94
#define Op_Agoto         95
#define Op_Amark         96


#define Op_Noop                 98


#define Op_Colm                108                /* column number */

/*
 * Declarations and such -- used by the linker but not the run-time system.
 */

#define Op_Proc                101
#define Op_Declend        102
#define Op_End                103
#define Op_Link                104
#define Op_Version        105
#define Op_Con                106
#define Op_Filen        107

/*
 * Global symbol table declarations.
 */
#define Op_Record        105
#define Op_Impl                106
#define Op_Error        107
#define Op_Trace        108
#define Op_Lab                109
#define Op_Invocable        110

/*
 * Extra instructions added for calling Icon from C (used by Posix functions)
 */
#ifdef PosixFns
#define Op_Copyd        111
#define Op_Trapret      112
#define Op_Trapfail     113
#endif                                          /* PosixFns */

#define Op_Synt         114             /* syntax code used by the linker */
#define Op_Uid          115             /* Universal Identifier for .u files */
/* Op_EInit is 116 for now. */
```

Which is manually paired with

```
/*
 * Operator definitions.
 *
 * Fields are:
 *    name
 *    number of arguments
 *    string representation
 *    dereference arguments flag: -1 = don't, 0 = do
 */

OpDef(asgn,2,":=",-1)
OpDef(bang,1,"!",-1)
OpDef(cater,2,"||",0)
OpDef(compl,1,"~",0)
OpDef(diff,2,"--",0)
OpDef(divide,2,"/",0)
OpDef(eqv,2,"===",0)
OpDef(inter,2,"**",0)
OpDef(lconcat,2,"|||",0)
OpDef(lexeq,2,"==",0)
OpDef(lexge,2,">>=",0)
OpDef(lexgt,2,">>",0)
OpDef(lexle,2,"<<=",0)
OpDef(lexlt,2,"<<",0)
OpDef(lexne,2,"~==",0)
OpDef(minus,2,"-",0)
OpDef(mod,2,"%",0)
OpDef(mult,2,"*",0)
OpDef(neg,1,"-",0)
OpDef(neqv,2,"~===",0)
OpDef(nonnull,1,BackSlash,-1)
OpDef(null,1,"/",-1)
OpDef(number,1,"+",0)
OpDef(numeq,2,"=",0)
OpDef(numge,2,">=",0)
OpDef(numgt,2,">",0)
OpDef(numle,2,"<=",0)
OpDef(numlt,2,"<",0)
OpDef(numne,2,"~=",0)
OpDef(plus,2,"+",0)
OpDef(powr,2,"^",0)
OpDef(random,1,"?",-1)
OpDef(rasgn,2,"<-",-1)
OpDef(rcv,2,"@<",0)
OpDef(rcvbk,2,"@<<",0)
OpDef(refresh,1,"^",0)
OpDef(rswap,2,"<->",-1)
OpDef(sect,3,"[:]",-1)
OpDef(snd,2,"@>",0)
OpDef(sndbk,2,"@>>",0)
OpDef(size,1,"*",0)
OpDef(subsc,2,"[]",-1)
OpDef(swap,2,":=:",-1)
OpDef(tabmat,1,"=",0)
OpDef(toby,3,"...",0)
OpDef(union,2,"++",0)
OpDef(value,1,".",0)
/* OpDef(llist,1,"[...]",0) */
```

There is also a set of Unicon application programmer support files, mainly used with Execution Monitoring and lower level debugging programs, available in the *IPL*.

- `ipl/incl/opdefs.icn`, an include file with all opcode numbers
- `ipl/incl/invkdefs.icn`, with definitions for visualizing the symbols
- `ipl/mprocs/opname.icn`, string names for opcodes
- and a few others peppered throughout the IPL.

In the end, Unicon creates an executable shell program that embeds the *icode* byte codes for use by the Unicon executor, *iconx* when producing a virtual machine program image.

The program used above to display the *hellomain* executable is a small Unicon program, *show-icode.icn*. It relies on the fact that the shell scripting is separated from the *ucode* by a form feed character. The shell script lines are displayed by line, and the rest as a hex dump.

```
#
# show-icode.icn, display the parts of a Unicon VM executable
#
link printf
procedure main(argv)
    f := open(argv[1], "r") | stop("Cannot open " || image(argv[1]))
    # the shell control preamble as lines
    while (line := read(f)) ~== "\^L" do write(line)

    # binary icode as hex dump
    # reads(f, -1) won't work here as the file is already reading
    s := ""
    while s ||:= reads(f)
    hexdump(s)

    close(f)
end

#
# display hex codes
#
procedure hexdump(s)
    local c, perline := 0, text := " ", scale := 0, counter := 0

    if *s = 0 then fail
    scale := **s
    scale <:= 4
    printf("%0" || scale || "d: ", counter)
    every c := !s do {
        if (perline +:= 1) > 16 then {
            write(text)
            text := " "
            perline := 1
            printf("%0" || scale || "d: ", counter)
        }
        #writes(map(_doprnt("%02x ", [ord(c)]), &lcase, &ucase))
        printf("%02x ", ord(c))
        text ||:= if 31 < ord(c) < 127 then c else "."
        counter +:= 1
    }
    if perline > 0 then {
        writes(repl(" ", (16 - perline) * 3))
```

```
        write(text)
    }
    else write()
end
```

10.3.3 The Implementation of Icon and Unicon

As part of the Unicon release 13 pass, there has been updates occurring to *The Implementation of Icon and Unicon* book, probably the best place for details on the design of the Unicon virtual machine. Sources for *ib.pdf* ships with the Unicon source kit under `doc/ib`. Plus it's an excellent book regarding how to go about building a programming language and the deep issues that designers face when embarking on the task.

10.4 Editors

Unicon source code is assumed to be ASCII. UFT-8 encoding will work, as long as the characters used stay as single byte code points, in the range 0 to 127.

This means word processor default formats are not suitable for Unicon source. If you like to use a word processor, make sure the source is saved as `Text`.

Literals and data can assume an 8bit range, 0 to 255, but anything beyond value 127 is at the whim of operating environment settings and default handling. Sometimes you will get block looking characters or upside down question marks, sometimes you will get an extended encoding value like line drawing characters. Those non ASCII features are not standard nor cross platform.

There are quite a few editors that support Icon in terms of highlighting, smart indenting and other productivity enhancers; less so with Unicon features added in. Listed below is an updated Vim syntax highlighter.

10.4.1 Vim

The world's best text editor. A text processing tool.

Here is a Vim syntax file, customized for Unicon, building on the `icon.vim` that ships with Vim.

```
" Vim syntax file
" Language: Unicon
" Maintainer:   Brian Tiffin (btiffin@gnu.org)
" URL: https://sourceforge.net/projects/unicon
" Last Change:  2016 Oct 22

" quit when a syntax file was already loaded
if exists("b:current_syntax")
  finish
endif

" Read the Icon syntax to start with
runtime! syntax/icon.vim
unlet b:current_syntax

" Unicon function extensions
```

```
syn keyword uniconFunction Abort Any Arb Arbno array
syn keyword uniconFunction Break Breakx chmod chown
syn keyword uniconFunction chroot classname cofail Color
syn keyword uniconFunction condvar constructor
syn keyword uniconFunction crypt ctime dbcolumns dbdriver
syn keyword uniconFunction dbkeys dblimits dbproduct dbtables display
syn keyword uniconFunction eventmask EvGet EvSend
syn keyword uniconFunction exec Fail fdup Fence fetch fieldnames
syn keyword uniconFunction filepair
syn keyword uniconFunction flock fork
syn keyword uniconFunction getegid geteuid getgid getgr
syn keyword uniconFunction gethost getpgrp getpid getppid getpw
syn keyword uniconFunction getrusage getserv gettimeofday
syn keyword uniconFunction getuid globalnames gtime
syn keyword uniconFunction ioctl istate
syn keyword uniconFunction keyword kill Len link load localnames lock
syn keyword uniconFunction max membernames methodnames
syn keyword uniconFunction methods min mkdir mutex name
syn keyword uniconFunction NotAny Nspan opencl oprec
syn keyword uniconFunction paranames parent pipe
syn keyword uniconFunction Pos proc
syn keyword uniconFunction readlink ready
syn keyword uniconFunction receive Rem rmdir Rpos Rtab
syn keyword uniconFunction select send setenv setgid setgrent
syn keyword uniconFunction sethostent setpgrp setpwent setservent setuid
syn keyword uniconFunction signal Span spawn sql stat staticnames
syn keyword uniconFunction structure Succeed symlink
syn keyword uniconFunction sys_errstr syswrite Tab
syn keyword uniconFunction trap truncate trylock
syn keyword uniconFunction umask unlock utime wait

" Unicon graphics, audio and VOIP
syn keyword uniconGraphics Active Alert
syn keyword uniconGraphics Attrib Bg
syn keyword uniconGraphics Clip Clone Color
syn keyword uniconGraphics ColorValue CopyArea
syn keyword uniconGraphics Couple
syn keyword uniconGraphics DrawArc DrawCircle DrawCube DrawCurve
syn keyword uniconGraphics DrawCylinder DrawDisk DrawImage DrawLine
syn keyword uniconGraphics DrawPoint DrawPolygon DrawRectangle
syn keyword uniconGraphics DrawSegment DrawSphere DrawString DrawTorus
syn keyword uniconGraphics EraseArea Event
syn keyword uniconGraphics Eye Fg
syn keyword uniconGraphics FillArc FillCircle FillPolygon
syn keyword uniconGraphics FillRectangle Font FreeColor
syn keyword uniconGraphics GotoRC GotoXY
syn keyword uniconGraphics IdentityMatrix
syn keyword uniconGraphics Lower MatrixMode
syn keyword uniconGraphics MultMatrix
syn keyword uniconGraphics NewColor Normals
syn keyword uniconGraphics PaletteChars PaletteColor PaletteKey
syn keyword uniconGraphics Pattern Pending
syn keyword uniconGraphics Pixel PlayAudio PopMatrix
syn keyword uniconGraphics PushMatrix PushRotate PushScale PushTranslate
syn keyword uniconGraphics QueryPointer Raise ReadImage
syn keyword uniconGraphics Refresh Rotate
syn keyword uniconGraphics Scale
syn keyword uniconGraphics StopAudio
```

```
syn keyword uniconGraphics Texcoord Texture
syn keyword uniconGraphics TextWidth Translate
syn keyword uniconGraphics Uncouple
syn keyword uniconGraphics VAttrib
syn keyword uniconGraphics WAttrib WDefault WFlush
syn keyword uniconGraphics WindowContents
syn keyword uniconGraphics WriteImage WSection WSync

" Unicon system specific
syn keyword uniconSpecific FreeSpace GetSpace InPort Int86
syn keyword uniconSpecific OutPort Peek Poke Swi
syn keyword uniconSpecific WinAssociate WinButton WinColorDialog
syn keyword uniconSpecific WinEditRegion WinFontDialog WinMenuBar
syn keyword uniconSpecific WinOpenDialog WinPlayMedia WinSaveDialog
syn keyword uniconSpecific WinScrollBar WinSelectDialog

" Unicon and Icon Graphic Keywords
syn match uniconKeyword "&col"
syn match uniconKeyword "&column"
syn match uniconKeyword "&control"
syn match uniconKeyword "&errno"
syn match uniconKeyword "&eventcode"
syn match uniconKeyword "&eventsource"
syn match uniconKeyword "&eventvalue"
syn match uniconKeyword "&interval"
syn match uniconKeyword "&ldrag"
syn match uniconKeyword "&lpress"
syn match uniconKeyword "&lrelease"
syn match uniconKeyword "&mdrag"
syn match uniconKeyword "&meta"
syn match uniconKeyword "&mpress"
syn match uniconKeyword "&mrelease"
syn match uniconKeyword "&pick"
syn match uniconKeyword "&now"
syn match uniconKeyword "&rdrag"
syn match uniconKeyword "&resize"
syn match uniconKeyword "&row"
syn match uniconKeyword "&rpress"
syn match uniconKeyword "&rrelease"
syn match uniconKeyword "&shift"
syn match uniconKeyword "&window"
syn match uniconKeyword "&x"
syn match uniconKeyword "&y"

" New reserved words
syn keyword uniconReserved critical import initially invocable method
syn keyword uniconReserved package thread

" Storage class reserved words
syn keyword uniconStorageClass  abstract class

" Define the highlighting colour groups
hi def link uniconStorageClass  StorageClass
hi def link uniconFunction      Statement
hi def link uniconGraphics      Special
hi def link uniconSpecific      SpecialComment
hi def link uniconReserved      Label
hi def link uniconKeyword       Operator
```

```
let b:current_syntax = "unicon"
```

Place that file in `$HOME/.vim/syntax/unicon.vim`. Then add

```
" Unicon - see ~/.vim/ftdetect/unicon.vim
autocmd BufRead,BufNewFile *.icn     set filetype=unicon
```

to your `~/.vimrc` Vim startup file.

Or copy this short file type detection file

```
" Unicon file detection override Icon
" Modified: 2016-10-24/05:19-0400

autocmd BufRead,BufNewFile *.icn     set filetype=unicon
```

to `$HOME/.vim/ftdetect/unicon.vim`

Please note the filename is also `unicon.vim` *but in the* `~/.vim/ftdetect/` *directory. It is renamed here as* `unicon-ftdetect.vim` *to avoid a name conflict within the documentation directory structure.*

After saving the syntax file and changes to the file type auto detect system, whenever you open or edit a file with a `.icn` extension, it will be highlighted for Unicon syntax. *As well as working with older Icon sources.*

Once you add the syntax file, you can choose Unicon highlighting for any Vim buffer by typing the `ex` colon command, `set filetype=unicon`.

As hinted at in *Documentation*, you can also add automatic templates for new Unicon source files by adding the following to your `~./vimrc` startup file.

```
autocmd BufNewFile  *.icn      0r ~/lang/unicon/header.icn
autocmd BufNewFile  *.opt      0r ~/lang/unicon/header-options.icn
    \ |0file|file <afile>:s?\.opt?\.icn?|filetype detect
```

Change the `~/lang/unicon/header` part to a local site installation filename. It will pre-populate new `.icn` files with text from the template header.

The second Vim `autocmd` listed above allows for a more sophisticated skeleton for programs what will have command line option handling. Use with **`vim filename.opt`**. It loads a more extensive template (header-options), then renames the buffer to `filename.icn`. The `.opt` shortcut relies on having the `unicon.vim` file type detection code properly installed in `~/.vim/ftdetect`.

At time of writing, my custom templates look like:

`header.icn`

```
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: October 2016
# Modified:
##+
#
# program.icn, description
#
# tectonics:
#
#
```

```
procedure main()

end
```

I use `header.icn` quite often, and keep the `Date:` line up to date with the current month.

`header-options.icn`

```
##-
# Author: Brian Tiffin
# Licensed under the GNU LGPL, version 3 or greater
#
# Date:
# Modified:
##+
#
# program.icn, description
#
# tectonics:
#
$define VERSION 0.1

link options

procedure main(argv)
    opts := options(argv, "-h! -v! -source!", optError)
    if \opts["h"] then return showHelp()
    if \opts["v"] then return showVersion()
    if \opts["source"] then return showSource()

end

#
# show help, version and source info
#
procedure showVersion()
    write(&errout, &progname, " ", VERSION, " ", __DATE__)
end

procedure showHelp()
    showVersion()
    write(&errout, "Usage:")
    write(&errout, "\t-h\tshow this help")
    write(&errout, "\t-v\tshow version")
    write(&errout, "\t-source\tlist source code")
end

procedure showSource()
    local f
    f := open(&file, "r") | stop("Source file ", &file, " unavailable")
    every write(!f)
    close(f)
end

#
# options error
#
procedure optError(s)
```

```
    write(&errout, s)
    stop("Try ", &progname, " -h for more information")
end
```

Personalize to taste, and save a little typing when you start new Unicon files. Plus the added benefit of a consistent look to all of your programs (which are, by their very nature, the world's best programs).

There is also the Icon Programming Library skeleton, developed by *Ralph Griswold* a long time ago, which is not a bad choice for consistent Unicon programming.

```
############################################################################
#
#        File:
#
#        Subject:  Program
#
#        Author:
#
#        Date:
#
############################################################################
#
#  This file is in the public domain.
#
############################################################################
#
#
#
############################################################################
#
#  Requires:
#
############################################################################
#
#  Links:
#
############################################################################

procedure main()

end
```

A well supported, de-facto standard in the Icon world.

10.4.2 Emacs

The world's other best text editor. An operating system that handles text.

Robert Parlett created a Unicon major more for Emacs:

http://www.zenadsl6357.zen.co.uk/unicon/

10.4.3 Evil

The Extensible Vi Layer for Emacs, Evil adds features of the world's best text editor to the world's other best text editor.

https://www.emacswiki.org/emacs/Evil

This is *thee* editor combination. Emacs (including OrgMode) with Vim key bindings and modal editing. Sweet. `Evil` is a very complete, robust and well done Vim emulator, augmenting the powers inherent in Emacs.

CHAPTER ELEVEN

STRING PROCESSING

11.1 Unicon String Processing

11.1.1 String Scanning

`Unicon` string scanning is a very powerful computational feature of the language. With roots dating back to *SNOBOL*, string scanning was added to *Icon* by the team lead by *Ralph Griswold* as a modernization of the pattern matching features of `SNOBOL`. `Unicon` has gone full circle and starting with the release 13 beta builds, `SNOBOL` *Patterns* are actually part of the system again. And to add to the already rich options, *Regular expressions* are also available now. A trifecta.

Scanning

String scanning syntax is based on the scanning operator *? (string scan)*.

```
expr1 ? expr2
```

Where the result of *expr*1 sets a scanning environment, *&subject* and *&pos* specifying the subject characters and current position of the scanning cursor. *expr*2 can be any valid Unicon expression, usually a sequence, and can range from simple to very complex selection and side effect operations.

A "simple" scan. Find Waldo.

```
#
# simple-scan.icn, A simple string scanning example
#
procedure main()
    s := "Where's Waldo?"
    s ? p := find("Waldo")
    if \p then write("Found starting at position ", p)
end
```

Sample run:

```
prompt$ unicon -s simple-scan.icn -x
Found starting at position 9
```

A middling scan. Find name from command line (or Waldo).

```
#
# middling-scan.icn, A slightly less simple string scanning example
#
procedure main(arglist)
    who := \arglist[1] | "Waldo"
    "Where's " || map(who) ? {
        write("Subject: ", image(&subject))
        write("target : ", image(who))
        p := find(map(who))
    }
    if \p then write("Found ", who, " starting at position ", p)
end
```

Sample run:

```
prompt$ unicon -s middling-scan.icn -x
Subject: "Where's waldo"
target : "Waldo"
Found Waldo starting at position 9
```

Todo

add string scanning samples

See: *Patterns* for a more on scanning.

CHAPTER

TWELVE

PATTERNS

12.1 Unicon Pattern data

12.1.1 SNOBOL patterns

Unicon version 13 alpha has *SNOBOL* inspired pattern matching. New functions and operators were added to emulate the very powerful, and well studied `SNOBOL` pattern matching features. This augments `String scanning` quite nicely. These features introduce a new datatype, `pattern`.

Details are in Technical Report UTR18a, http://unicon.org/utr/utr18.pdf.

SNOBOL is still relevant to many developers and SNOBOL4 implementations have been made freely available, thanks in large part to Catspaw Inc.

There is also a very comprehensive tutorial hosted at http://www.snobol4.org.

Chapter 4 of the tutorial is about Pattern Matching.

http://www.snobol4.org/docs/burks/tutorial/ch4.htm

This is a conversion (with some changes to add a test pass, and outputting results) of the small program listed in section 4.7 of that page:

```
#
# snobols.icn, SNOBOL like patterns
#
procedure main()

   # From http://www.snobol4.org/docs/burks/tutorial/ch4.htm
   # SNOBOL code
   # (('B' | 'F' | 'N') . FIRST 'EA' ('R' | 'T') . LAST) . WORD
   #
   # matches 'BEAR', 'FEAR', 'NEAR', 'BEAT', 'FEAT', or 'NEAT',
   # assigning the first letter matched to FIRST,
   # the last letter to LAST, and the entire result to WORD.

```

```
   # Unicon version, with test strings, and addition of cursor
   # position capture. BEAD expected to fail.
   every str := !["BEAR", "FEAR", "NEAR",
                  "BEAT", "FEAT", "NEAT", "BEAD"] do {
       writes("subject: ", str, " ")
       if str ?? .> p1 || (("B" .| "F" .| "N") -> first || "EA" ||
                 .> p2 || ("R" .| "T") -> last) -> word then
           write("first: ", first, ";", p1, ", last: ", last, ";", p2,
                 ", word: ", word)
       else
           write("did not match")
   }
end
```

```
subject: BEAR first: B;1, last: R;4, word: BEAR
subject: FEAR first: F;1, last: R;4, word: FEAR
subject: NEAR first: N;1, last: R;4, word: NEAR
subject: BEAT first: B;1, last: T;4, word: BEAT
subject: FEAT first: F;1, last: T;4, word: FEAT
subject: NEAT first: N;1, last: T;4, word: NEAT
subject: BEAD did not match
```

Clinton Jeffery, along with Sudarshan Gaikaiwari and John Goettsche carefully designed this feature set to be an almost one to one correspondence to `SNOBOL` patterns. It provides a highly viable path for porting old, beloved, `SNOBOL` programs to Unicon.

Unicon currently lacks the full `eval` *potential of SNOBOL but ameliorates that downside, somewhat, by allowing invocation of functions and methods along with variable and field references inside patterns.*

Internals

To see a little bit of how the implementation actually works, let's take a look at the preprocessor output. *The listing below has extra blank lines squeezed out,* `cat -s`*, and is reformatted,* `fmt`*. This is only for human curiousity and the listing below is not the version sent to the compiler.*

```
prompt$ unicon -s -E snobols.icn | cat -s | fmt
#line 0 "/tmp/uni17224850" #line 0 "snobols.icn"

procedure main();

   every str := !["BEAR", "FEAR", "NEAR",
                  "BEAT", "FEAT", "NEAT", "BEAD"] do {
       writes("subject: ", str, " "); if( "" ? pattern_match(
       str,pattern_setcur(        "p1",p1 ) || pattern_assign_onmatch(
       ( pattern_assign_onmatch( ( pattern_alternate(
       pattern_alternate( "B",   "F" ) ,    "N" ) ),    "first",first )
       ||"EA"||pattern_setcur(
                    "p2",p2 ) || pattern_assign_onmatch( (
                    pattern_alternate( "R",   "T" ) ),    "last",last )
                    ),"word",word ) ))  then
           write("first: ", first, ";", p1, ", last: ", last, ";", p2,
                 ", word: ", word)
       else
           write("did not match")
   };
end
```

Nice. The SNOBOL operators are actually a new class of functions.

I talked with Clinton about this, and for now, those functions are for compiler internal use only. Much smarter, and cleaner, to use the operators.

12.1.2 Regular expressions

When SNOBOL patterns were added to Unicon, regular expression features were also added. This means Unicon has the power of *String Scanning*, *SNOBOL patterns* and `regular expressions` available. And all three features can be freely mixed in string manipulation expressions. *Raising the bar.*

Regular expression literals are surrounded by angle brackets, not quotes. Pattern matching uses a *??* operator. As of early Unicon release 13, regular expressions are limited to `basic` regex patterns.

```
#
# hello-regex
#
procedure main()
    L := ["Hello?", "Hello, world", "helloworld", "Hello World!", "World"]
    every write(!L ?? <[hH]ello","?[ \t]*[wW]orld"!"?>)
end
```

Displays a message when the subject includes some form of `Hello, world`. In the example, the first and last elements of the string list do not match. The regular expression looks for Hello with or without a capital H, an optional comma, any number of spaces or tabs (including zero), followed by World (or world), with an optional exclamation mark.

```
Hello, world
helloworld
Hello World!
```

12.1.3 Pattern operators

- `??` - a variant form of string scanning, s ?? p matching a pattern, not a general Unicon expression as with *?* scanning. Unanchored.
- `=p` - anchored match of pattern, p.
- `.|` - a pattern alternation. Accepts Unicon expressions as an operand.
- `->` - conditional assignment.
- `=>` - immediate assignment, (regardless of an actual successful match result).
- `.>` - cursor position assignment.
- <r> - a regular expression literal is surrounded in angle brackets (chevrons).

12.1.4 Regex syntax

Regular expressions can include the following components

- `r` - ordinary symbol that matches to r.
- `r1 r2` - juxtaposition is concatenation.
- `r1 | r2` - regular expression alternate (not a generator).

- `r*` - match zero or more occurrences of r.
- `r+` - match one or more occurrences of r.
- `r?` - match zero or one occurrences of r.
- `r{n}` - braces surround an integer count, match n occurrences of r.
- `"lit"` - match the literal string, with the usual escapes allowed.
- `'lit'` - cset literal matching any one character of the set, escapes allowed.
- `[chars]` - cset literal with dash range syntax.
- `.` - match any character *except newline*.
- `(r)` - parentheses are used for grouping.

CHAPTER

THIRTEEN

OBJECTS

13.1 Unicon Objects and Classes

The object oriented features of Unicon stem from an early *Icon* preprocessor, called *IDOL*. Unicon supports object oriented design and development, but is not a purely object oriented language. Unlike, for instance, Ruby where *everything is an object*, Unicon is still very much an *everything is an expression* language. Native types are native types, and objects are a design and development assistive technology, not a *core* element of the Unicon programming language.

Objects and classes add another aspect to the multi-paradigm dimensions of Unicon programming.

Unicon supports *class* declarations, and *method* definitions within an inherited hierarchy of classes. *class* defined words create instances of objects that allow method calls in an object oriented fashion.

The Unicon view of object oriented programming *(and there are many different points of view regarding OO)* starts with encapsulation, inheritance, and polymorphism.

Note: Objects and classes are very much a *programming in the large* design and development feature. Many of the examples in this document will be small, contrived, and may belittle the powerful potentials of object oriented design and programming. Try and overlook the small, and think big when applying *class* elements to your programs. On the flip side, as a cautionary warning, don't try and shoehorn a small problem into objects when the procedural elements of Unicon would be more appropriate.

13.1.1 SOLID

There is an object oriented set of principles, with a mnemonic acronym, SOLID.

- **S**ingle responsibility
- **O**pen-closed
- **L**iskov substitution

- **I**nterface segregation
- **D**ependency inversion

These principles are outlined at SOLID: Wikipedia

The Unicon implementation of objects and classes can be applied to uphold these principles of design. *But like many general purpose, flexible, programming languages, they are guidelines to be applied when they are of benefit and suit working habits.*

Todo

map out Unicon examples of each SOLID principle

13.1.2 IDOL

The Icon Derived Object Language, by *Clinton Jeffery*, circa 1990.

See https://www2.cs.arizona.edu/icon/ftp/doc/tr90_10.pdf for the initial Idol technical report from January of 1990.

Todo

Much to do regarding the Objects chapter.

CHAPTER

FOURTEEN

GRAPHICS

14.1 Unicon graphics

Unicon has graphics built in, *well optionally built in, if the operating system supports X11, Win32 and/or OpenGL.*

In the case of Unicon, graphics doesn't just mean colours drawn on a canvas. Unicon includes an entire Graphical User Interface engine as well. Graphics come in 2D and 3D forms.

That means you can draw, and you can work with widgets and event driven programming without changing language or tool.

Much of the callback management normally associated with event driven programming is handled by Unicon for many of the graphic operations. The design of the graphics features of Unicon allow for both procedural and event programming in a seamless way. Using graphics does not dictate an event driven only style of programming in Unicon.

14.2 Colours

Unicon uses a pretty nifty colour naming system, as English phrases.

There are base colour names, with attributes for lightness, saturation and transparency levels.

```
colour := "medium strong bluish green"
```

Is parsed down to an RGBA (Red Green Blue Alpha) colour value. If the phrase is not understood, Unicon passes the name down to the operating system for another round of lookup. For X11 that means all the Xorg colour names are also valid.

Jafar Al-Gharaibeh was nice enough to extract the pertinent bits from `unicon/src/rwindow.r`

```
static colrname colortable[] = {      /* known colors */
   /* color        ish-form     hue  lgt  sat */
   { "black",     "blackish",     0,   0,   0 },
   { "blue",      "bluish",     240,  50, 100 },
   { "brown",     "brownish",    30,  25, 100 },
   { "cyan",      "cyanish",    180,  50, 100 },
   { "gray",      "grayish",      0,  50,   0 },
   { "green",     "greenish",   120,  50, 100 },
   { "grey",      "greyish",      0,  50,   0 },
   { "magenta",   "magentaish", 300,  50, 100 },
   { "orange",    "orangish",    15,  50, 100 },
   { "pink",      "pinkish",    345,  75, 100 },
   { "purple",    "purplish",   270,  50, 100 },
   { "red",       "reddish",      0,  50, 100 },
   { "violet",    "violetish",  270,  75, 100 },
   { "white",     "whitish",      0, 100,   0 },
   { "yellow",    "yellowish",   60,  50, 100 },
   };

static colrmod lighttable[] = {          /* lightness modifiers */
   { "dark",        0 },
   { "deep",        0 },      /* = very dark (see code) */
   { "light",     100 },
   { "medium",     50 },
   { "pale",      100 },      /* = very light (see code) */
   };

static colrmod sattable[] = {            /* saturation levels */
   { "moderate",  50 },
   { "strong",    75 },
   { "vivid",    100 },
   { "weak",      25 },
   };

static colrmod transptable[] = {         /* transparency levels */
   { "dull",   75 },              /* alias for subtranslucent */
   { "opaque",  100 },
   { "subtranslucent",  75 },
   { "subtransparent",  25 },
   { "translucent",  50 },
   { "transparent",  5 },
   };
```

Along with all those combinations, there are the system names, for things like Teal, and Cornflower, and Dark Khaki, when using X11.

https://en.wikipedia.org/wiki/X11_color_names

With a full list (many hundreds of named colours) in the source code at

https://cgit.freedesktop.org/xorg/xserver/tree/os/oscolor.c

If you look closely, there are more than double the 50 Shades of Grey.

Other specific graphic subsystems will have their own list of available names.

14.2.1 Unicon colour scheme

I've asked about an official Unicon project colour scheme, and I'm not sure if there has been an actual choice made. *I've even asked to put* `unicon` *in the list of known colours, for potential branding purposes (and cool code samples). That may never happen though, but it's worth asking.*

From the project home page, *from September 2016*, the current guess is a form of orange (although the logo on that page uses a professionally laid out gradient, by Serendel Macphereson).

Best (bad)[1] guess (vivid orange) shown below.

```
#
# colour-sample.icn, a best (bad) guess at a Unicon project colour.
#
procedure main()
    &window := open("colour", "g", "size=70,45", "canvas=hidden")
    colour := "vivid orange"
    Fg(colour)
    FillRectangle(5, 5, 60, 35)
    WSync()
    WriteImage("../images/colour-sample.png")
    close(&window)
end
```

```
prompt$ unicon -s colour-sample.icn -x
```

14.3 Drawing

Points, lines, rectangles, circles, spheres, torus and more, are all part and parcel of Unicon graphics. As is text. The *write* statement can write to a graphical window as readily as it can write to standard output and to files.

14.4 Events

Event driven programming was never easier or more adaptable than with Unicon.

14.5 Attributes

Unicon windows use a set of attributes to control look, feel and certain features of graphic handling. These attributes can be set during *open* and with the function *WAttrib*.

The `src/runtime/rwindow.r` source file lists the following supported attributes.

[1] Do not mistake this author for a graphic designer. There are few design skills in these fingers and little artistic flair hiding behind the eyes.

```
stringint attribs[] = {
   { 0,                NUMATTRIBS},
   {"ascent",          A_ASCENT},
   {"bg",              A_BG},
   {"buffer",          A_BUFFERMODE},
   {"canvas",          A_CANVAS},
   {"ceol",            A_CEOL},
   {"cliph",           A_CLIPH},
   {"clipw",           A_CLIPW},
   {"clipx",           A_CLIPX},
   {"clipy",           A_CLIPY},
   {"col",             A_COL},
   {"columns",         A_COLUMNS},
   {"cursor",          A_CURSOR},
   {"depth",           A_DEPTH},
   {"descent",         A_DESCENT},
   {"dim",             A_DIM},
   {"display",         A_DISPLAY},
   {"displayheight",    A_DISPLAYHEIGHT},
   {"displaywidth", A_DISPLAYWIDTH},
   {"drawop",          A_DRAWOP},
   {"dx",              A_DX},
   {"dy",              A_DY},
   {"echo",            A_ECHO},
   {"eye",             A_EYE},
   {"eyedir",          A_EYEDIR},
   {"eyepos",          A_EYEPOS},
   {"eyeup",           A_EYEUP},
   {"fg",              A_FG},
   {"fheight",         A_FHEIGHT},
   {"fillstyle",       A_FILLSTYLE},
   {"font",            A_FONT},
   {"fovangle",        A_FOV},
   {"fwidth",          A_FWIDTH},
   {"gamma",           A_GAMMA},
   {"geometry",        A_GEOMETRY},
   {"glrenderer",      A_GLRENDERER},
   {"glvendor",        A_GLVENDOR},
   {"glversion",       A_GLVERSION},
   {"height",          A_HEIGHT},
   {"iconic",          A_ICONIC},
   {"iconimage",       A_ICONIMAGE},
   {"iconlabel",       A_ICONLABEL},
   {"iconpos",         A_ICONPOS},
   {"image",           A_IMAGE},
   {"inputmask",       A_INPUTMASK},
   {"label",           A_LABEL},
   {"leading",         A_LEADING},
   {"light",           A_LIGHT},
   {"light0",          A_LIGHT0},
   {"light1",          A_LIGHT1},
   {"light2",          A_LIGHT2},
   {"light3",          A_LIGHT3},
   {"light4",          A_LIGHT4},
   {"light5",          A_LIGHT5},
   {"light6",          A_LIGHT6},
   {"light7",          A_LIGHT7},
   {"lines",           A_LINES},
```

```
    {"linestyle",     A_LINESTYLE},
    {"linewidth",     A_LINEWIDTH},
    {"meshmode",      A_MESHMODE},
    {"normode",       A_NORMODE},
    {"pattern",       A_PATTERN},
    {"pick",          A_PICK},
    {"pointer",       A_POINTER},
    {"pointercol",    A_POINTERCOL},
    {"pointerrow",    A_POINTERROW},
    {"pointerx",      A_POINTERX},
    {"pointery",      A_POINTERY},
    {"pos",           A_POS},
    {"posx",          A_POSX},
    {"posy",          A_POSY},
    {"resize",        A_RESIZE},
    {"reverse",       A_REVERSE},
    {"rgbmode",       A_RGBMODE},
    {"rings",         A_RINGS},
    {"row",           A_ROW},
    {"rows",          A_ROWS},
    {"selection",     A_SELECTION},
    {"size",          A_SIZE},
    {"slices",        A_SLICES},
    {"texcoord",      A_TEXCOORD},
    {"texmode",       A_TEXMODE},
    {"texture",       A_TEXTURE},
    {"titlebar",      A_TITLEBAR},
    {"visual",        A_VISUAL},
    {"width",         A_WIDTH},
    {"windowlabel",   A_WINDOWLABEL},
    {"x",             A_X},
    {"y",             A_Y},
};
```

14.6 Vidgets

For lack of a better term for Graphical User Interface (*GUI*) elements, Unicon includes a rich set of ready to go widgets. Some few built in, many as part of the *IPL* as `vidgets` and with the Unicon gui classes.

```
#
# vidget-button.icn, button demo
#
link evmux
link button
link enqueue

procedure main()
    &window := open("vidget", "g", "size=70,45", "canvas=hidden")
    b := button(&window, "Hello", hello, -3, 10, 10, 50, 25)
    WriteImage("../images/vidget-button.png")

    Enqueue(&window, &lpress, 11, 14, "", 2)
    Enqueue(&window, &lrelease, 11, 14, "", 3)
    evhandle(&window)
```

```
end

procedure hello()
    write("Hello, button")
end
```

Hello

The sample simulates a mouse click, invoking the `hello` procedure.

```
prompt$ unicon -s vidget-button.icn -x
Hello, button
```

14.6.1 On names

Icon was developed long before modern graphical desktops were mainstream. There is still no ubiquitous term for graphic elements, but `widget` seems to be well understood. When Clint Jeffery was working with Ralph Griswold on the early graphics features of Icon, even the term widget was not in common use. The Icon (now Unicon) team went with a portmanteau of Visual Gadget, `vidget`. Things change, but history stays the same.

Unicon classes allow for a modernized approach to *GUI* development.

14.7 Unicon GUI

Todo

samples of Robert Parlett's GUI classes

14.8 Plot coordinate pairs

A nice example of Unicon graphics can be found at *Rosetta Code* under the `Plot coordinate pairs` task. Duplicated here from a copy taken in August of 2016 *with some slight modifications to captured image name, and not waiting for a mouse click to end the run.*

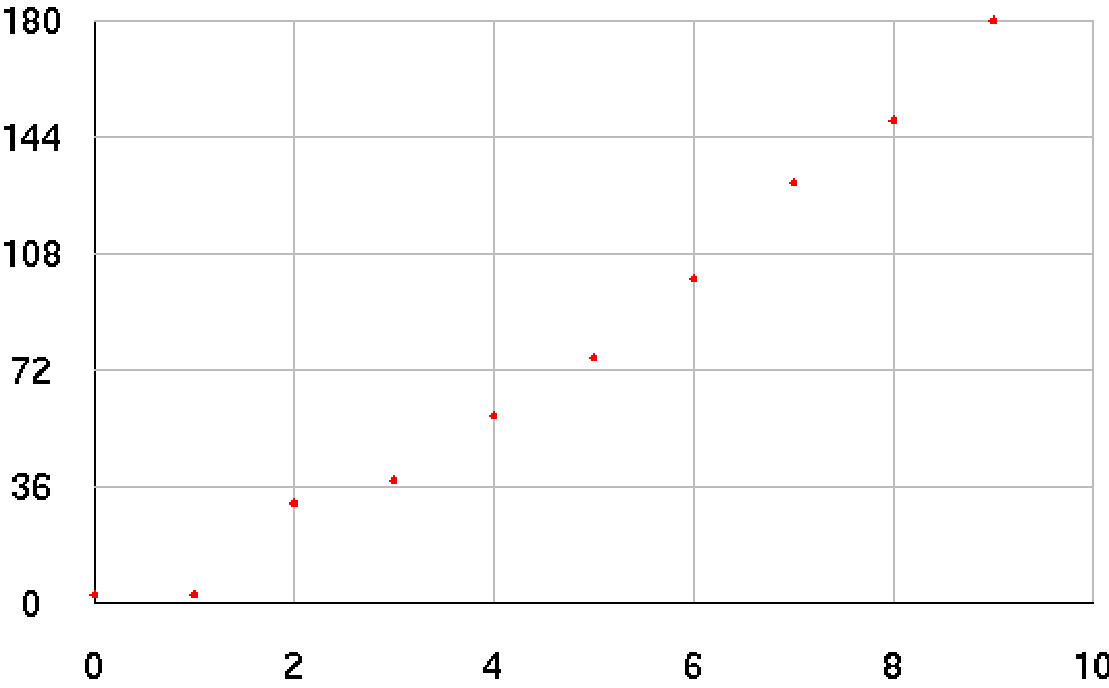
```
#
# From http://rosettacode.org/wiki/Plot_coordinate_pairs#Icon_and_Unicon
#
link printf,numbers

procedure main()
x := [0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]
y := [2.7, 2.8, 31.4, 38.1, 58.0, 76.2, 100.5, 130.0, 149.3, 180.0]
Plot(x,y,600,400)
end
```

```
$define POINTR 2                        # Point Radius
$define POINTC "red"                    # Point Colour
$define GRIDC  "grey"                   # grid colour
$define AXISC  "black"                  # axis/label colour
$define BORDER 60                       # per side border
$define TICKS  5.                       # grid ticks per axis
$define AXISFH 20                       # font height for axis labels

procedure Plot(x,y,cw,ch)

   /cw := 700                            # default dimensions
   /ch := 400
   uw := cw-BORDER*2                     # usable dimensions
   uh := ch-BORDER*2

   wparms  := ["Plot","g", "canvas=hidden",
               sprintf("size=%d,%d",cw,ch),
               "bg=white"]              # base window parms

   dx := sprintf("dx=%d",BORDER)        # grid origin
   dy := sprintf("dy=%d",BORDER)

   &window := open!wparms | stop("Unable to open window")
   X := scale(x,uw)                     # scale data to usable space
   Y := scale(y,uh,"invert")

   WAttrib(dx,dy)                       # set origin=grid & draw grid
   every x := (X.tickfrom to X.tickto by X.tick) * X.tickscale do {
      if x = 0 then Fg(AXISC) else Fg(GRIDC)
      DrawLine(x,Y.tickfrom*Y.tickscale,x,Y.tickto*Y.tickscale)
      }
   every y := (Y.tickfrom to Y.tickto by Y.tick) * Y.tickscale do {
      if y = uh then Fg(AXISC) else Fg(GRIDC)
      DrawLine(X.tickfrom*X.tickscale,y,X.tickto*X.tickscale,y)
      }

   Fg(POINTC)                           # draw data points ....
   every i := 1 to *X.scaled do
      FillCircle(X.scaled[i],Y.scaled[i],POINTR)

   Fg(AXISC)                            # label grid
   WAttrib(dx,"dy=0")                   # label X axis
   Font(sprintf("Helvetica,%d",AXISFH))
   ytxt := ch-BORDER+1+(WAttrib("ascent") - WAttrib("descent"))/2

   every x := X.tickscale * (xv := X.tickfrom to X.tickto by X.tick) do
      DrawString(x - TextWidth(xv)/2, ytxt + integer(AXISFH*1.5),xv)

   WAttrib("dx=0",dy)                   # label Y axis
   every y := Y.tickscale * (yv := Y.tickfrom to Y.tickto by Y.tick) do
      DrawString(BORDER/2 - TextWidth(yv)/2, ytxt - BORDER - y,yv)

   WriteImage("../images/PlotPairs.png")          # save image

   WAttrib("dx=0","dy=0")               # close off nicely
   Font("Helvetica,10")
   #DrawString(10,ch-5,"click to exit")
```

```
   #until Event() == &lpress            # wait for left mouse button
   close(&window)
end

record scaledata(low,high,range,pix,raw,scaled,tick,tickfrom,tickto,tickscale)

procedure scale(data,pix,opts[])
   P := scaledata( pmin := min!data, pmax := max!data,
                   prange := real(pmax-pmin), pix,
                   data,q :=[])

   /ticks := TICKS
   P.tick := ceil(prange/(10^(k:=floor(log(prange,10))))*(10^k)/ticks)
   P.tickfrom  := P.tick*floor(pmin/P.tick)
   P.tickto    := P.tick*ceil(pmax/P.tick)
   P.tickscale := real(pix)/(P.tickto-P.tickfrom)
   every put(q,integer((!data-P.tickfrom)*P.tickscale))
   if !opts == "invert" then            # invert is for y
      every q[i := 1 to *q] := pix - q[i]
   return P
end
```

A nice feature of Unicon is the *WriteImage* graphics function. It saves a canvas image in GIF, JPG, BMP or PNG format (depending on given filename extension, including system specific types like XBM, XPM). That handy function is used to generate `PlotPairs.png` (and most of the other images) during builds of this documentation.

```
prompt$ unicon -s plotpairs.icn -x
```



CHAPTER

FIFTEEN

DATABASE

15.1 Unicon databases

Unicon supports a couple of different database types.

- DBM
- ODBC

These are optional dependencies during compiler build. DBM requires a DBM engine, GDBM for instance, and ODBC requires a Open Data Base Connection layer. A third option is using the native *table* datatype (or other aggregate data structure) to handle memory based database management

15.1.1 Tables

table data can be an easy way to handle ad-hoc database problems.

The *IPL* entries for `xencode` and `xdecode` can then be used to add persistence.

```
#
# table-db, a small database in a table
#
link xcodes

procedure main()
    db := table()
    db["key"] := "valuable data"
    db["alternate"] := ["more Unicon treasure", "in a list"]

    dbf := open("table-db.dat", "w")
    xencode(db, dbf)
    close(dbf)

    dbf := open("table-db.dat")
    newdb := xdecode(dbf)
```

```
    close(dbf)

    write("newdb[\"key\"] is ", newdb["key"])
end
```

```
newdb["key"] is valuable data
```

Note: The `xencode` and `xdecode` procedures come in a couple of different flavours; as `link xcode` or `link xcodes`. `xcodes` makes handling *record* definitions a little easier, and provides for *file* and *procedure* structures that may not be present in the decoding program. Try and use `xcodes` for most developments.

15.1.2 DBM

When supported, *open* mode "d" (and mode "dr" for read-only) will open DBM database resources. Once opened, the resource is treated as a persistent *table* datatype. `datum := dbm[s]` will retrieve data for key `s`, and `dbm[s] := "some data"` will attempt to insert or update the DBM information on disk. `insert`, `delete` and `fetch` built-in functions can also be used. *Update and insert are blocked for mode "dr" read-only data stores.*

```
#
# dbm, database sample
#
procedure main()
    db := open("dbm.dat", "d")
    db["key"] := "valuable data"
    db["alternate"] := "more Unicon treasure"
    close(db)

    dbf := open("dbm.dat", "dr")
    write(dbf["key"])
    write(dbf["alternate"])
    close(dbf)
end
```

```
valuable data
more Unicon treasure
```

The example above will create `dbm.dat`, the user visible data file, and also some internal files; `dbm.dat.dir` and `dbm.dat.pag`.

DBM information is converted to *string* form when written to disk. Unlike memory tables, `1` and `"1"` are the same key in DBM mode. Use `xencode` (and `xdecode`) if you need to differentiate between string and other datatypes for DBM keys and values.

15.1.3 ODBC

Unicon includes *SQL* features, when built with `ODBC` support.

Documented in Unicon Technical Report, UTR1, http://unicon.org/utr/utr1/utr1.htm by Federico Balbi and *Clint*.

Requirements

- ODBC, `unixodbc` package (for instance)

- SQLiteODBC, `libsqliteodbc` package (or other ODBC driver)
- datasource definintion, `~/.odbc.ini`

```
#
# odbc.icn, ODBC trial
#
link ximage

procedure main()
    # mode 'o' open, ODBC SQL
    db := open("unicon", "o", "", "") | stop("no odbc for \"unicon\"")

    # Display some ODBC driver information
    write("dbproduct:")
    write(ximage(dbproduct(db)))

    write("\ndbdriver:")
    write(ximage(dbdriver(db)))

    write("\ndblimits:")
    write(ximage(dblimits(db)))

    # create a sample table
    sql(db, "drop table if exists contacts")
    sql(db, "create table contacts (id integer primary key, name, phone)")

    # insert some records, with and without transaction control
    sql(db, "insert into contacts (name, phone) _
             values ('brian', '613-555-1212')")

    sql(db, "BEGIN; insert into contacts (name, phone) _
             values ('jafar', '615-555-1213'); _
             COMMIT")

    sql(db, "insert into contacts (name, phone) _
             values ('brian', '615-555-1214')")

    sql(db, "BEGIN; insert into contacts (name, phone) _
             values ('clint', '615-555-1215'); _
             COMMIT")

    sql(db, "insert into contacts (name, phone) _
             values ('nico', '615-555-1216')")

    # display ODBC view of table schema
    write("\ndbtables:")
    every write(ximage(dbtables(db)))

    tables := dbtables(db)
    write("\ndbcolumns.", tables[1].name, ":")
    every write(ximage(dbcolumns(db, tables[1].name)))

    # query a few phone numbers
    write("\nPhone numbers for brian:")
    sql(db, "select id, phone from contacts where name='brian'")
    while rec := fetch(db) do write(rec.id, ": ", rec.phone)

    write("\nPhone numbers for jafar:")
```

```
    sql(db, "select id, phone from contacts where name='jafar'")
    while rec := fetch(db) do write(rec.id, ": ", rec.phone)

    # query with an intrinsic function
    write("\nRecord count:")
    sql(db, "select count(*) from contacts")
    rec := fetch(db)
    write(ximage(rec))
    write(rec["count(*)"])

    # query with an intrinsic function given alias
    writes("\nRecord count (as counter): ")
    sql(db, "select count(*) as counter from contacts")
    rec := fetch(db)
    write(rec.counter)

    # close off the resource
    close(db)
end
```

With a unicon DSN (data source name) configuration of:

```
[unicon]
Description=Unicon ODBC sample
Driver=SQLite3
Database=/home/btiffin/lang/unicon/databases/unicon.db
Timeout=2000
```

Giving:

```
dbproduct:
R__1 := ()
   R__1.name := "SQLite"
   R__1.ver := "3.9.2"

dbdriver:
R__1 := ()
   R__1.name := "sqlite3odbc.so"
   R__1.ver := "0.9992"
   R__1.odbcver := "03.00"
   R__1.connections := 0
   R__1.statements := ""
   R__1.dsn := "unicon"

dblimits:
R__1 := ()
   R__1.maxbinlitlen := 0
   R__1.maxcharlitlen := 0
   R__1.maxcolnamelen := 255
   R__1.maxgroupbycols := 0
   R__1.maxorderbycols := 0
   R__1.maxindexcols := 0
   R__1.maxselectcols := 0
   R__1.maxtblcols := 0
   R__1.maxcursnamelen := 255
   R__1.maxindexsize := 0
   R__1.maxownnamelen := 255
   R__1.maxprocnamelen := 0
```

```
   R__1.maxqualnamelen := 255
   R__1.maxrowsize := 0
   R__1.maxrowsizelong := "N"
   R__1.maxstmtlen := 16384
   R__1.maxtblnamelen := 255
   R__1.maxselecttbls := 0
   R__1.maxusernamelen := 16

dbtables:
L1 := list(1)
   L1[1] := R__1 := ()
      R__1.qualifier := ""
      R__1.owner := ""
      R__1.name := "contacts"
      R__1.type := ""
      R__1.remarks := ""

dbcolumns.contacts:
L6 := list(3)
   L6[1] := R__1 := ()
      R__1.catalog := ""
      R__1.schema := ""
      R__1.tablename := "contacts"
      R__1.colname := "id"
      R__1.datatype := 4
      R__1.typename := "integer"
      R__1.colsize := 9
      R__1.buflen := 10
      R__1.decdigits := 10
      R__1.numprecradix := 0
      R__1.nullable := 1
      R__1.remarks := ""
   L6[2] := R__2 := ()
      R__2.catalog := ""
      R__2.schema := ""
      R__2.tablename := "contacts"
      R__2.colname := "name"
      R__2.datatype := 12
      R__2.typename := ""
      R__2.colsize := 0
      R__2.buflen := 255
      R__2.decdigits := 10
      R__2.numprecradix := 0
      R__2.nullable := 1
      R__2.remarks := ""
   L6[3] := R__3 := ()
      R__3.catalog := ""
      R__3.schema := ""
      R__3.tablename := "contacts"
      R__3.colname := "phone"
      R__3.datatype := 12
      R__3.typename := ""
      R__3.colsize := 0
      R__3.buflen := 255
      R__3.decdigits := 10
      R__3.numprecradix := 0
      R__3.nullable := 1
      R__3.remarks := ""
```

```
Phone numbers for brian:
1: 613-555-1212
3: 615-555-1214

Phone numbers for jafar:
2: 615-555-1213

Record count:
R__1 := ()
   R__1.count(*) := 5
5

Record count (as counter): 5
```

That same code will work with MariaDB, PostgreSQL, Oracle, or any of the many other ODBC drivers that are available for most operating systems and database engines.

CHAPTER

SIXTEEN

NETWORKING

16.1 Unicon Networking

Ahh, the network. Unicon fully supports client server computing using standard words and phrases.

Note: *Update: Nov 16 2016*

Unicon has been setup to run on SourceForge. CGI and other browser friendly demonstrations will be posted. For instance:

Simple CGI form echo

16.1.1 Network mode

The *open* emphatic is used to initiate networking subsystems.

Network mode `n` is the network client mode of *open*.

Network mode `na` is network *accept* (blocking server) mode of *open*.

Network mode `nl` is network *listen* (non-blocking server) mode of *open*.

Protocols include:

- TCP Transmission Control Protocol
- UDP User Datagram Protocol (add `u` to the *open* mode; for example "nua" for a blocking UDP server)

16.1.2 Message mode

Message mode `m` is a special network mode of *open*. This is the web as seen from Unicon. Support for http and https. Use `m-` for less secure but functional unauthenticated https, until local certificates are configured.

HTTP

HyperText Transport Protocol. A text based protocol used in much of the modern web.

Unicon handles HTTP by using *open* with a mode of `m` (or `ms` if only HTTP headers are needed, and no page data).

```
#
# http.icn
#
procedure main()
    # pull in the page
    url := "http://example.com"
    web := open(url, "m") | stop("Can't open " || url)
    page := ""
    while page ||:= read(web) || "\n"

    # pull out any title
    page ? {
        tab(find("<title")) & tab(find(">")) & move(1) &
        title := tab(find("</title"))
    }
    close(web)

    # display the title, if any, and then the page
    write("Page title: ", \title)
    write(page)
end
```

Giving:

```
Page title: Example Domain
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open␣
↪Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
```

```
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

HTTPS

Secure (encrypted) HTTP. The text based HTTP protocol wrapped in encrypted packets. These packets use TLS and/or SSL encryption technology, a certificate based key pairing. An initial *usually verified* certificate is hosted by the server and temporary tokens are delivered to the client (browser) for single pair communications. This avoids clear text messaging, a rather easy thing to eavesdrop on, and or change in transit.

As of early Unicon 13 alpha releases, HTTPS is not quite ready. The encryption works, once setup. It is the setup that is not quite ready. It is not the smooth setup that will eventually be part of Unicon builds. There is internal certificate management issues that may require end user intervention. It will be advertised as ready when a basic build of Unicon does not require any extraordinary effort on behalf of a Unicon programmer. For example, the sample below required an externally set `SSL_CERT_DIR` environment variable before Jafar fixed the search path settings in SVN revision 4474 of the Unicon code base.

Note: *Leaving the sample in, as it doesn't "hurt", but this override is no longer required (at least for Ubuntu based Unicon built from source).*

```
#
# https.icn, read and show a secure webpage, extract title
#
procedure main()
    local url, web, page, title

    # pull in the page
    url := "https://example.com"
    web := open(url, "m-") | stop("Can't open " || url)
    page := ""
    while page ||:= read(web) || "\n"

    # pull out any title
    page ? {
        tab(find("<title")) & tab(find(">")) & move(1) &
        title := tab(find("</title"))
    }
```

```
    close(web)

    # display the title, if any, and then the page
    write("Page title: ", \title)
    write(page)
end
```

```
prompt$ SSL_CERT_DIR=/usr/lib/ssl/certs unicon https-full.icn -x
Parsing https-full.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O https-full.icn /tmp/uni15588305
Translating:
https-full.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  https-full.u -x
Linking:
Executing:
Page title: Example Domain
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open␣
↪Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
```

```
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

Without the external certificate finder, the program does did not function properly.

Now, the environment setting is not necessary.

```
prompt$ unicon https-full.icn -x
Parsing https-full.icn: .
/home/btiffin/unicon-git/bin/icont -c   -O https-full.icn /tmp/uni34111072
Translating:
https-full.icn:
  main
No errors
/home/btiffin/unicon-git/bin/icont  https-full.u -x
Linking:
Executing:
Page title: Example Domain
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open␣
↪Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
```

```
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

16.1.3 Verify

Messaging mode (*open* mode "m") with `https` adds a new, optional, - flag, for skipping certificate verification. The default is to try and validate a certificate. `"m"` is more secure than `"m-"`, but does require some external setup for certificate pairing. Otherwise the system will block access with:

```
cannot verify peer's certificate
```

When validation fails, *open* will fail.

16.1.4 More HTTPS

The Unicon project is on SourceForge. And the Allura system that makes up most of the forge user interface supports a rich API.

Apache Allura public API

To get the download counts for Unicon, using Unicon:

```
#
# https-json.icn, Try HTTPS with a SourceForge JSON query
#
# tectonics:
#     unicon -s https-json.icn -x | jq .downloads[-4:-1]
#
procedure main()
    uri := "https://sourceforge.net/projects/unicon/files/stats/json" ||
           "?start_date=2010-10-01&end_date=2110-10-01"
    web := open(uri, "m-") | stop("Can't open ", uri)
    showHeaders(web)

    size := 0 < integer(web["Content-Length"]) | stop("can't get the file size")
    data := reads(web, size) | stop("failed to reads ", size, " bytes")
    write(data)
    close(web)
end

# write HTTPS response headers to standard error
procedure showHeaders(m)
    local k
    every k := key(\m) do write(&errout, left(k||":", 32), m[k])
end
```

That program will pull down a JSON response from SourceForge. The captured command output asks `jq` (JSON query) for the last entry from the `downloads` list from the data displayed by Unicon. *Web server response headers are also displayed from Unicon, directed to standard error to not interfere with the JSON data passed to the* `jq` *command. The capture only includes a limited range of the header lines from the output, hence the ellipsis.*

```
prompt$ unicon -s https-json.icn -x | jq .downloads[-4:-1]
Status-Code:                        200
Reason-Phrase:                      OK
Server:                             nginx/1.14.0 (Ubuntu)
Date:                               Sun, 27 Oct 2019 08:53:46 GMT
...
[
  [
    "2019-07-01 00:00:00",
    74
  ],
  [
    "2019-08-01 00:00:00",
    54
  ],
  [
    "2019-09-01 00:00:00",
    47
  ]
]
```

SMTP

Simple Mail Transfer Protocol

Todo

add a mail sending sample

POP

Post Office Protocol

You've got mail.

Todo

add a mail reader sample

16.2 CGI

There is a Technical Report (UTR4a 2011/02/04) `Writing CGI and PHP Scripts in Icon and Unicon` that details and lists working samples of `Common Gateway Interface` programming with Unicon. Methods for integrating Unicon with `PHP` is also explained.

http://unicon.org/utr/utr4.html

A CGI support layer is built along with `unicon`, so programmers only need to `link cgi` to take advantage of CGI features.

There is a small sample now hosted on SourceForge, to show off a very simple Unicon CGI program. `simple-cgi.icn` can be tried using a form posted at:

http://btiffin.users.sourceforge.net/form.html

form-cgi.html

```
<!-- Simple Unicon CGI example form -->
<HTML>
<HEAD>
<title>A Unicon HTML Form Example</title>
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 12;
        padding: 4;
        font-family: "Open Sans", "Helvetica Neue", Helvetica,
                      Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 50px;
        background-color: #fff;
        border-radius: 1em;
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        body {
            background-color: #fff;
        }
        div {
            width: auto;
            margin: 0 auto;
            border-radius: 0;
            padding: 1em;
        }
    }
    </style>
</HEAD>
<BODY>
<h1> A <tt>cgi.icn</tt> Demonstration</h1>
<form method="GET" action="/cgi-bin/simple.cgi">
    1. Name: <input type="text" name="name" size=25> <p>
    2. Age: <input type="text" name="age" size=3>  Years <p>
    3. Quest:
        <input type="checkbox" name="fame">Fame</input>
        <input type="checkbox" name="fortune">Fortune</input>
        <input type="checkbox" name="grail">Grail</input><p>
    4. Favorite Color:
        <select name="color">
            <option>Red
            <option>Green
```

```
            <option>Blue
            <option selected>Don't Know (Aaagh!)
        </select><p>
    Comments:<br>
        <textarea rows=5 cols=60 name="comments"></textarea><p>
        <input type="submit" value="Submit Data">
        <input type="reset" value="Reset Form">
</form>
</BODY>
</HTML>
```

That form invokes:

simple-cgi.icn

```
#
# simple-cgi.icn
# tectonics:
#    unicon -B simple-cgi.icn
#    mv simple ../cgi-bin/simple.cgi
#
link cgi
procedure cgimain()
    # set defaults for both CGI and AJAX usage
    if /cgi["name"] | cgi["name"] === "" then cgi["name"] := "Guest"
    if /cgi["age"] | cgi["age"] === "" then cgi["age"] := "no"
    if /cgi["comments"] then cgi["comments"] := ""
    if /cgi["word"] then cgi["word"] := ""

    # remove any potentially dangerous characters
    cgi["name"] := map(cgi["name"], "<>&%", "....")
    cgi["age"] := map(cgi["age"], "<>&%", "....")
    cgi["comments"] := map(cgi["comments"], "<>&%", "....")
    cgi["word"] := map(cgi["word"], "<>&%", "....")

    # output for the web
    cgiEcho("Hello, ", cgi["name"], "!")
    cgiEcho("Are you really ", cgi["age"], " years old?")
    cgiEcho("You seek: ", cgi["fame"]==="on" & "fame")
    cgiEcho("You seek: ", cgi["fortune"]==="on" & "fortune")
    cgiEcho("You seek: ", cgi["grail"]==="on" & "grail")
    cgiEcho("Your favorite color is: ", cgi["color"])
    cgiEcho("Your comments: ", cgi["comments"])
    cgiEcho("")
    cgiEcho("Your AJAX word: ", cgi["word"])
    cgiEcho("")
    cgiEcho("<a href=\"/demos/\">Home</a> / " ||
        "<a href=\"/demos/simple-form.html\">Back to HTML form</a> / " ||
        "<a href=\"/demos/simple-ajax.html\">Back to AJAX form</a>")
end
```

Unicon was built on a SourceForge hosted CentOS server using a Developer Web Services shell. Details of how this was set up is documented in a blog entry, *SourceForge*. There will be more examples posted. An index page will be placed at

http://btiffin.users.sourceforge.net/demos/index.html

16.2.1 CGI 1.1

The full CGI 1.1 specification is referenced in RFC3875,

https://tools.ietf.org/html/rfc3875

16.3 AJAX

The `simple.cgi` demo can also be put to work with an AJAX interface.

```
<!-- simple-ajax.html, Unicon AJAX example -->
<html>
<head>
<title>Simple AJAX Example with Unicon</title>
<script language="Javascript">
function xmlhttpPost(strURL) {
    var xmlHttpReq = false;
    var self = this;
    // Mozilla/Safari
    if (window.XMLHttpRequest) {
        self.xmlHttpReq = new XMLHttpRequest();
    }
    // IE
    else if (window.ActiveXObject) {
        self.xmlHttpReq = new ActiveXObject("Microsoft.XMLHTTP");
    }
    self.xmlHttpReq.open('POST', strURL, true);
    self.xmlHttpReq.setRequestHeader('Content-Type',
        'application/x-www-form-urlencoded');
    self.xmlHttpReq.onreadystatechange = function() {
        if (self.xmlHttpReq.readyState == 4) {
            updatepage(self.xmlHttpReq.responseText);
        }
    }
    self.xmlHttpReq.send(getquerystring());
}

<!-- pass the form query -->
function getquerystring() {
    var form     = document.forms['form1'];
    var word = form.word.value;
    var name = form.name.value;
    var age = form.age.value;
    qstr = 'word=' + escape(word) + '&name=' + escape(name) +
           '&age=' + escape(age);
    return qstr;
}

<!-- change the result DOM div -->
function updatepage(str){
    document.getElementById("result").innerHTML = str;
}
</script>
</head>
<body>
An asynchronous Javascript to Unicon example.<br>
Pressing <b>Go</b> will cause an AJAX call to the server,
```

```
and CGI results will appear below
<form name="form1">
  <p>Name: <input name="name" type="text" size="25"></p>
  <p> Age: <input name="age" type="text" size="2"></p>
  <p>word: <input name="word" type="text">
  <input value="Go" type="button"
      onclick='javascript:xmlhttpPost("/cgi-bin/simple.cgi")'></p>
  <div id="result"></div>
</form>
</body>
</html>
```

That AJAX example invokes the same server side `simple-cgi.icn` as the CGI form. The server side program is not a CGI application in this case, and there is not the same set of environment variables that a web server will set up for CGI programs, but the example program accounts for this to produce AJAX ready output.

16.4 PHP

PHP (initially PHP/FI, Personal Home Page/Forms Interpreter, now know as PHP: Hypertext Preprocessor) is a widely used server side scripting language. PHP was never really meant to be a programming language, but has attained a lead position as a tool of choice for many web applications.

> "I don't know how to stop it, there was never any intent to write a programming language [. . .] I have absolutely no idea how to write a programming language, I just kept adding the next logical step on the way."
>
> —Rasmus Lerdorf

Unicon can leverage that unstoppable project. PHP is almost always available along side any web server install, and is a feature rich and well supported web development environment.

Todo

add Unicon, PHP integration sample

CHAPTER

SEVENTEEN

THREADING

17.1 Unicon threading

Unicon supports concurrent processing, tasks, and threads.

Threading is documented in Unicon Technical Report 14, by *Jafar Al-Gharaibeh* and *Clinton Jeffery*.

http://unicon.org/utr/utr14.pdf

Unicon threads build on *Unicon Co-Expressions*. Normally co-expressions are synchronous, thread co-expressions are asynchronous.

17.1.1 Thread creation

Unicon threads can be created with the reserved word *thread* or the function *spawn*. The `spawn()` function turns a previously created synchronous co-expression into an asynchronous threaded co-expression. The `thread` reserved word creates a thread and starts it running.

`thread expr` is equivalent to:

```
T := spawn(create expr))
@T
```

Both `thread` and `spawn()` return a reference to the new thread.

17.1.2 Hello, threads

This example creates running threads, but there is no synchronization with main. The main procedure may end, terminating the run, before the threads have a chance to complete, (or it may not, due to the nature of asynchronous threading).

```
#
# threading.icn, Hello, threads, not sychronized
#
procedure main()
    every t := !10 do thread write("Hello, world; I am thread: ", t)
    write("main: complete")
end
```

Initial test:

```
Hello, world; I am thread: 1
Hello, world; I am thread: 2
Hello, world; I am thread: 3
Hello, world; I am thread: 5
Hello, world; I am thread: 6
Hello, world; I am thread: 7
Hello, world; I am thread: 8
Hello, world; I am thread: 9
main: complete
```

That sample run may or may not allow all ten threads to finish before the main program completes and returns to the operating system.

This next example uses *wait* to ensure that all the threads complete before main terminates.

```
#
# threading-synch.icn, Hello, threads, sychronized with wait
#
procedure main()
    Lt := []
    every t := !10 do put(Lt, thread write("Hello, world; I am thread: ", t))
    every wait(!Lt)
    write("main: complete")
end
```

Waited run:

```
Hello, world; I am thread: 1
Hello, world; I am thread: 2
Hello, world; I am thread: 3
Hello, world; I am thread: 4
Hello, world; I am thread: 5
Hello, world; I am thread: 6
Hello, world; I am thread: 7
Hello, world; I am thread: 8
Hello, world; I am thread: 9
Hello, world; I am thread: 10
main: complete
```

Using *wait* is the recommended way of dealing with thread execution to ensure the threads are given a chance to complete. Keeping a list structure of thread references is a handy and reliable way of managing this idiom.

Attention: Threaded programming is hard. It adds an extra dimension to processing that requires special care and attention to detail. Most operations in a computer program are multiple step, non atomic operations. Even something as simple as incrementing a variable is split at the machine level as fetch, increment, store. Threads may enter the operation at any point in time, and two competing threads may conflict between the fetching and

incrementing steps, causing erroneous results. Guards must be put in place to ensure that every section of code is allowed to complete each step before another thread starts in on the operation. Unicon has kept this in mind and *critical* section management is part and parcel of safe, reliable thread programming.

17.2 Multi-tasking

Clinton Jeffery has enhanced the virtual machine to allow multi-tasking. Multiple programs can be loaded into the VM and invoked as *Unicon Co-Expressions*s. See *load*.

The poor person's expensive `uval` ***compile at runtime*** function uses *load* to load a different task to perform the equivalent of an `eval` function. Not perfect, but satisfactory for many applications that need to evaluate arbitrary Unicon programs or program fragments. The *unitest* testing framework uses `uval` to compile and test arbitrary code fragments from text source.

The multi-tasking features started life as a method to allow execution monitoring and control. Comes in handy for many things, and will likely play a large part of any future *evaluate at runtime* form of an `eval` function.

CHAPTER EIGHTEEN

FEATURES

18.1 Unicon features

Unicon is a feature rich environment. This chapter deals with some of the more *miscellaneous* components included in Unicon that are not detailed in other sections.

18.1.1 Keyboard functions

Unicon supports keyboard scanning outside normal standard IO operation support. This allows for pending key tests, unbuffered input (no Enter key required) and other niceties.

`kbhit()` checks if there is a keyboard character waiting to be read. `getch()` reads a key (and will wait if needed) without echo to screen. `getche()` reads a key (and will wait if needed) with echo to screen.

```
#
# keyboarding.icn, demonstrate keyboard functions
#
procedure main()
    if kbhit() then write(ch := getch())
    write("keyscan: ", image(ch))
end
```

Sample run:

```
prompt$ unicon -s keyboarding.icn -x
keyscan: &null
```

18.1.2 Pseudo terminals

Qutaiba Mahmoud added support for pseudo terminals in Unicon when attaining his Master's Degree in Computer Science, outlined in a report hosted along with the Unicon Technical Reports, at

http://www2.cs.uidaho.edu/~jeffery/unicon/reports/mahmoud.pdf

Basically, a file stream is opened with mode `"prw"`, which gives a Unicon master terminal program asynchronous read and write access to another interactive process.

This feature is similar to the capabilities provided in the famous *Expect Tcl/Tk* extension by Don Libes.

Todo

add pty sample

18.1.3 libz compression

Compression can be used in Unicon file operations, and is also supported in the compiler for compressed *icode* generation, if `libz` is available during Unicon build.

`unicon -Z program.icn` will produce a compressed `icode` file. It will be automatically uncompressed at runtime, *assuming support is included in the current VM when invoked*.

Programmers can also use this feature with the `"z"` mode modifier of the *open* function. Compressed data is not line oriented, so use *reads* and *writes*.

```
#
# gzio.icn, Demonstrate libz compression
#
link printf
procedure main()
    if not find("libz", &features) then stop("no libz compression")

    # compress some text, libz adds a little overhead
    filename := "gzio.txt.gz"
    f := open(filename, "wz") | stop("cannot write ", filename)
    writes(f, "First line\n")
    every 1 to 4 do writes(f, &ucase || &lcase || "\n")
    writes(f, "Last line\n")
    close(f)

    # image of compressed file
    f := open(filename, "r")
    data := reads(f, -1)
    close(f)

    write("Compressed data, size=", *data)
    hexdump(data)

    # read and uncompress the data
    write("\nUncompressed data")
    f := open(filename, "rz")
    while writes(reads(f))
    close(f)
```

```
end

#
# display hex codes
#
procedure hexdump(s)
    local c, perline := 0
    every c := !s do {
        if (perline +:= 1) > 16 then write() & perline := 1
        writes(map(_doprnt("%02x ", [ord(c)]), &lcase, &ucase))
    }
    write()
end
```

Sample run:

```
prompt$ unicon -s gzio.icn -x
Compressed data, size=89
1F 8B 08 00 00 00 00 00 00 03 73 CB 2C 2A 2E 51
C8 C9 CC 4B E5 72 74 72 76 71 75 73 F7 F0 F4 F2
F6 F1 F5 F3 0F 08 0C 0A 0E 09 0D 0B 8F 88 8C 4A
4C 4A 4E 49 4D 4B CF C8 CC CA CE C9 CD CB 2F 28
04 EA 29 2D 2B AF A8 AC 1A EC 9A 7C 12 61 BE 03
00 22 21 2D 35 E9 00 00 00

Uncompressed data
First line
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
Last line
```

CHAPTER

NINETEEN

DOCUMENTATION

19.1 Documenting Unicon programs

19.1.1 Unicon headers

Ralph Griswold was fond of full intent documentation at the top of his Icon programs. Unicon can continue that tradition. Comment blocks at the top of a source file, with a little discipline to allow for machine readable processing, is a cheap and effective way to document small programs, and to help keep them organized. The *IPL* has a sample of the skeleton file that can be used to start a new Unicon source file. This author recommends taking a copy of that, customizing it to suit your preferences and use it for each and every program you write, large or small.

```
############################################################################
#
#        File:
#
#        Subject:  Program
#
#        Author:
#
#        Date:
#
############################################################################
#
#   This file is in the public domain.
#
############################################################################
#
#
#
############################################################################
#
#   Requires:
#
############################################################################
```

```
#
#   Links:
#
############################################################################

procedure main()

end
```

The empty section in the middle if meant for the program synopsis and explanation. All the programs in the *IPL* follow this convention.

Personal

For most of the files in this doc set, a different template is used[1].

```
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: October 2016
# Modified: 2019-10-20/19:15-0400 btiffin
##+
#
# filename.icn, purpose
#
# tectonics
#
procedure main()

end
```

For the document generator, a special feature of docutils literal include is used, where each listing is included with the option:

```
:start-after: ##+
```

The listings in the book aren't burdened with repetitive rights information, but when downloaded, the copyright and license (or dedication) is included so everyone clearly knows how the source can be used.

See *tectonics* for the definition of the word, meaning *from source build instructions*, in this docset.

vim autoload

With *Vim*, adding these skeletons to new Unicon files is as easy as adding one line to a `.vimrc` file.

```
" Auto load Unicon template
autocmd BufNewFile  *.icn      0r ~/lang/icon/skeleton.icn
```

Now, every time you start an empty `.icn` file in `vim` it'll be preloaded with your personal template. The `0r` vim command (*read at 0*) is followed by a site-local filename; customize the `~/lang/icon/` part to match your setup.

If you don't want the skeleton for a particular file, tapping `u` for undo, at the start of an edit, will clear the template.

Another sequence in `Vim` that is very handy, auto modification stamping:

[1] That might be a mistake. These headers will not produce valid results from the *IPL* indexing program, iplweb.icn.

```
" Auto update modified time stamp
"    Modified (with colon) must occur in the first 16 lines,
"    32 chars of data before Modified tag remembered
"    modify strftime to suit
function! LastModified()
    if &modified
        let save_cursor = getpos(".")
        let n = min([16, line("$")])
        keepjumps exe '1,' . n . 's#^\(.\{,32}Modified:\).*#\1'
            \ . strftime(" %Y-%m-%d/%H:%M%z") . '#e'
        call histdel('search', -1)
        call setpos('.', save_cursor)
    endif
endfunction
au BufWritePre * call LastModified()
```

Add that to `~/.vimrc`.

Now, on file write, a top `Modified:` line will get a nice timestamp.

The routine only looks at the first 16 lines of text. The function also limits the scan for the `Modified:` tag within the first 32 characters of each line. You can customize all three of these preferences by changing the literals in the *VimL* function definition.

19.1.2 unidoc

There is an extensive API style documentation tool that ships with the Unicon source tree, `unidoc`, by Robert Parlett. Nice. From your source kit, see `uni/unidoc/docs/html/index.html` for a quick, yet detailed, view of some of the *package*d utilities that ship with Unicon.

19.2 Unicon Technical Reports

There is an abundance of technical documentation for Unicon and Icon. Unicon programmers are fortunate that the core reference materials are in the Public Domain, or use liberal usage and redistribution licensing. User guides and deep implementation documentation are freely available. Extension and enhancement materials in the form of Technical Reports are also a click away, and can be redistributed with applications.

http://unicon.sourceforge.net/reports.html

CHAPTER

TWENTY

TESTING

20.1 Testing Unicon

Unit testing is an important part of software development. The more tests, the better. These can be ad-hoc, and that ad-hoc form of testing is probably still the most prevalent form of testing. In this case, *ad-hoc* meaning that a developer writes some code, then runs the program to see if the new feature functions as expected. This is important to mentally ensure that the current work being done is acceptable, catches typos and the like, but is not very reproducible and ends up being a linear time burden. Each testing pass takes time, or more explicitly, the programmer has to devote some time to testing.

Wisdom dictates that it might be smarter to automate some of the testing burden. Spend a little bit of extra time during development to create reproducible test cases. Written once, these tests can be evaluated many times, building up a level of continual confidence in the software and retesting older work to ensure new work does not break previous assumptions or expectations.

20.1.1 Unit testing

There are a few de facto standard unit testing frameworks. TAP, xUnit, to name two. There are orders of magnitude more engines that run tests and produce an output compatible with these frameworks. cUnit, check, jUnit, to name just a few from the long list. Some programming languages have unit testing baked into the design of the syntax; the `D` programming language for instance.

```
class Sum
{
    int add(int x, int y) { return x + y; }

    unittest
    {
        Sum sum = new Sum;
        assert(sum.add(3,4) == 7);
        assert(sum.add(-2,0) == -2);
```

```
    }
}
```

The D compilers support a `-unittest` command option that set up special compiles for running the `unittest` blocks.

20.1.2 unitest

Now, Unicon gets *unitest*, an engine that can assist with Unicon unit testing. Note the one *t*, uni-test. Yet another unit testing engine. `unitest` follows the xUnit framework specification by default[1].

```
#
# unitest.icn, Unicon unit testing
#
link fullimag, lists

# test suite container, and testresults aggregate
record testcontrol(testname, speaktest, looplimit, xmlout, testcases)
record testresults(control, trials, skips,
                   errors, pass, fails, breaks)
global testlabel

procedure testsuite(testname, speak, looplimit, xmlout)
    local control, suite
    testlabel := create ("test-" || seq())
    control := testcontrol(testname, speak, looplimit, xmlout, [])
    suite := testresults(control, 0, 0, 0, 0, 0, 0)
    return suite
end

#
# single result testing
#
procedure test(suite, code, arglist, result, output, error)
    suite.trials +:= 1
    put(suite.control.testcases, [@testlabel, 0, &null])

    if suite.control.speaktest > 0 then {
        write(repl("#", 18), " Test: ", right(suite.trials, 4), " ",
              repl("#", 18))
        writes(image(code))
        if \arglist then write("!", fullimage(arglist)) else write()
        if \result then write("Expecting: ", result)
    }

    case type(code) of {
        "string"    : task := uval(code)
        "procedure" : task := create code!arglist
        default     : testFailure("Unknown code type: " || type(code))
    }

    if \task then {
        suite.control.testcases[*suite.control.testcases][2] :=
            gettimeofday()
        # fetch a result
```

[1] Fibbing, release 0.6 of unitest.icn is not yet xUnit compatible.

```
        r := @task
        if \result then
            if r === result then pass(suite) else fails(suite)
        else pass(suite)
        suite.control.testcases[suite.trials][2] :=
            bigtime(suite.control.testcases[suite.trials][2])
    } else errors(suite)

    if suite.control.speaktest > 0 then {
        if \result then write("Received: ", type(r), ", ", image(r))
        write("Trials: ", suite.trials, " Errors: ", suite.errors,
              "  Pass: ", suite.pass, " Fail: ", suite.fails)
        write(repl("#", 48), "\n")
    }
end

#
# record a pass
#
procedure pass(suite)
    suite.pass +:= 1
end

#
# record a fail
#
procedure fails(suite)
    suite.fails +:= 1
    suite.control.testcases[suite.trials][3] := 1
end

#
# record an error
#
procedure errors(suite)
    suite.errors +:= 1
    suite.control.testcases[suite.trials][3] := 2
end

#
# report, summary and possibly XML
#
procedure testreport(suite)
    write("Trials: ", suite.trials, " Errors: ", suite.errors,
          "  Pass: ", suite.pass, " Fail: ", suite.fails)
    write()

    if suite.control.xmlout > 0 then {
        write("<?xml version=\"1.0\" encoding=\"UTF-8\"?>")
        write("<testsuite name=\"", suite.control.testname,
              "\" tests=\"", suite.trials,
              "\" errors=\"", suite.errors, "\" failures=\"", suite.fails,
              "\" skip=\"", suite.skips, "\">")
        every testcase := !suite.control.testcases do {
            write("    <testcase classname=\"", &progname,
                  "\" name=\"", testcase[1], "\" time=\"", testcase[2], "\">")
            if \testcase[3] = 1 then {
                write("        <failure type=\"unitest\"> unitest failure </failure>")
```

```
            }
            if \testcase[3] = 2 then {
                write("        <error type=\"unicon\" message=\"code error\">")
                write("            CodeError: code problem")
                write("        </error>")
            }
            write("    </testcase>")
        }
        write("</testsuite>")
    }
end

#
# Multiple result testing
#
procedure tests(suite, code, arglist, result, output, error)
    suite.trials +:= 1
    put(suite.control.testcases, [@testlabel, 0, &null])
    if suite.control.speaktest > 0 then {
        write(repl("#", 8), " Generator test: ", right(suite.trials, 4),
              " ", repl("#", 18))
        writes(image(code))
        if \arglist then write("!", fullimage(arglist)) else write()
        if \result then write("Expecting: ", limage(result))
    }

    case type(code) of {
        "string"    : task := uvalGenerator(code)
        "procedure" : task := create code!arglist
        default     : testFailure("Unknown code type: " || type(code))
    }

    resultList := list()
    loops := 0;
    if \task then {
        suite.control.testcases[suite.trials][2] :=
            gettimeofday()
        # fetch a result list
        while put(resultList, @task) do {
            loops +:= 1
            if loops > suite.control.looplimit > 0 then {
                suite.breaks +:= 1
                pull(resultList)
                break &null
                # should limiter breaks ever count as a pass?  todo
            }
        }
        if \result then
            if lequiv(resultList, result) then pass(suite) else fails(suite)
        else pass(suite)
        suite.control.testcases[suite.trials][2] :=
            bigtime(suite.control.testcases[suite.trials][2])
    } else errors(suite)

    if suite.control.speaktest > 0 then {
        if \result then write("Received: ", limage(resultList))
        write("Trials: ", suite.trials, "  Errors: ", suite.errors,
              "  Limits: ", suite.breaks,
```

```
              "  Pass: ", suite.pass, " Fail: ", suite.fails)
        write(repl("#", 48), "\n")
    }
end

#
# timer calculation
#
procedure bigtime(timer)
    secs := gettimeofday().sec - timer.sec
    usecs := gettimeofday().usec - timer.usec
    return secs * 1000000 + usecs
end

#
# usage failure
#
procedure testFailure(s)
    write(&errout, s)
end

#
# uval.icn, an eval function
#
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: September 2016
# Modified: 2016-09-17/14:48-0400
#
$define base "/tmp/child-xyzzy"

link ximage

#
# try an evaluation
#
procedure uval(code)
    program := "# temporary file for unitest eval, purge at will\n_
        procedure main()\n" || code || "\nreturn\nend"
    return eval(program)
end

#
# try a generator
#
procedure uvalGenerator(code)
    program := "# temporary file for unitest eval, purge at will\n_
        procedure main()\n" || code || "\nend"
    return eval(program)
end

#
# eval, given string (either code or filename with isfile)
#
procedure eval(s, isfile)
    local f, codefile, code, coex, status, child, result
```

```
    if \isfile then {
        f := open(s, "r") | fail
        code ||:= every(!read(f))
    } else code := s

    # compile and load the code
    codefile := open(base || ".icn", "w") | fail
    write(codefile, code)
    close(codefile)

    status := system("unicon -s -o " || base  || " " ||
                     base || ".icn 2>/dev/null")

    # task can have io redirection here for stdout compares...
    if \status then coex := load(base)

    remove(base || ".icn")
    remove(base)
    return coex
end
```

This is currently a work in progress. Support for other framework integrations and extended capabilities are in the work plan.

`unitest` can be used in two ways. In source, which requires a couple of Unicon preprocessor lines to define two different `main` functions depending on a compile time define. Or it can be used to *load* one or more secondary program in the Unicon multi-tasking virtual machine space and act as a monitoring command and control utility.

Todo

monitoring test mode not yet ready for prime time. Nor is xUnit compatibility actually finished for that matter.

In both of these modes, unit testing can be by on the fly expression compiles, given strings, or more conventional *procedure* testing of the module under test.

Character escapes

Any expression strings passed to `test` include a burden of backslash escaping on the part of the test writer. To test:

```
generator(arg)\1
```

the string needs to be passed as:

```
test("generator(arg)\\1")
```

That is due to Unicon string compilation, certain characters need to be protected when inside string literals. For the most part test writers will need to protect:

```
' quote
" double quote
\ backslash
```

An example of *in source test expressions*:

```
#
# tested.icn, Unit testing, in source
#

$ifndef UNITEST
#
# unit testing example
#
procedure main()
    write("Compile with -DUNITEST for the demonstration")
end

$else
link unitest
#
# unit test trial
#
procedure main()
    speaktest := 0
    xmlout := 1
    looplimit := 100
    suite := testsuite("unitest", speaktest, looplimit, xmlout)

    test(suite, "1 + 2")
    test(suite, "xyzzy ::=+ 1")
    test(suite, "delay(1000)")
    test(suite, "return 1 + 2",, 3)
    test(suite, "return 1 + 2",, 0)
    test(suite, "return write(1 + 2)",, 3, "3\n")
    test(suite, internal, [21], 42)

    testreport(suite)

    speaktest := 1
    # set up for loop limit break testing
    looplimit := 3
    suite := testsuite("unitest generators", speaktest, looplimit, xmlout)

    tests(suite, "suspend 1 to 3",, [1,2,3])
    tests(suite, "syntaxerror 1 to 3",, [1,2,3])
    tests(suite, "suspend seq()\\4",, [1,2,3,4])
    tests(suite, generator, [1,3], [1,2,3])
    tests(suite, generator, [1,2], [1,2,3])

    testreport(suite)
end
$endif

#
# some internal procedure tests
# todo: need some way of handling the arguments
#
procedure internal(v)
    return v * 2
end

procedure generator(low, high)
    suspend low to high
```

```
end
```

Example testing pass:

```
prompt$ unicon -s -c unitest.icn
```

```
prompt$ unicon -s tested.icn -x
Compile with -DUNITEST for the demonstration
```

```
prompt$ unicon -s -DUNITEST tested.icn -x
3
Trials: 7 Errors: 1  Pass: 5 Fail: 1

<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="unitest" tests="7" errors="1" failures="1" skip="0">
    <testcase classname="tested" name="test-1" time="21">
    </testcase>
    <testcase classname="tested" name="test-2" time="0">
        <error type="unicon" message="code error">
            CodeError: code problem
        </error>
    </testcase>
    <testcase classname="tested" name="test-3" time="1001066">
    </testcase>
    <testcase classname="tested" name="test-4" time="14">
    </testcase>
    <testcase classname="tested" name="test-5" time="20">
        <failure type="unitest"> unitest failure </failure
    </testcase>
    <testcase classname="tested" name="test-6" time="24">
    </testcase>
    <testcase classname="tested" name="test-7" time="7">
    </testcase>
</testsuite>
######## Generator test:    1 ##################
"suspend 1 to 3"
Expecting: [1,2,3]
Received: [1,2,3]
Trials: 1  Errors: 0  Limits: 0  Pass: 1 Fail: 0
################################################

######## Generator test:    2 ##################
"syntaxerror 1 to 3"
Expecting: [1,2,3]
Received: []
Trials: 2  Errors: 1  Limits: 0  Pass: 1 Fail: 0
################################################

######## Generator test:    3 ##################
"suspend seq()\\4"
Expecting: [1,2,3,4]
Received: [1,2,3]
Trials: 3  Errors: 1  Limits: 1  Pass: 1 Fail: 1
################################################

######## Generator test:    4 ##################
procedure generator![1,3]
```

```
Expecting: [1,2,3]
Received: [1,2,3]
Trials: 4  Errors: 1  Limits: 1  Pass: 2 Fail: 1
################################################

######## Generator test:    5 ##################
procedure generator![1,2]
Expecting: [1,2,3]
Received: [1,2]
Trials: 5  Errors: 1  Limits: 1  Pass: 2 Fail: 2
################################################

Trials: 5 Errors: 1  Pass: 2 Fail: 2

<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="unitest generators" tests="5" errors="1" failures="2" skip="0">
    <testcase classname="tested" name="test-1" time="37">
    </testcase>
    <testcase classname="tested" name="test-2" time="0">
        <error type="unicon" message="code error">
            CodeError: code problem
        </error>
    </testcase>
    <testcase classname="tested" name="test-3" time="17">
        <failure type="unitest"> unitest failure </failure
    </testcase>
    <testcase classname="tested" name="test-4" time="14">
    </testcase>
    <testcase classname="tested" name="test-5" time="10">
        <failure type="unitest"> unitest failure </failure
    </testcase>
</testsuite>
```

Test Assisted Development

The author of `unitest` is not actually a practitioner of the more formal *Test Driven Development*, `TDD` method, but follows a slightly looser model of *test assisted development*, `TAD`.

The order of write a test first, then the code, is not a modus operandi in test assisted development. The goal is to write code and then verify it works with various test cases. Let the implementation occur during coding, not as a side effect of how it will pass or fail various tests.

Using `unitest` does not preclude `TDD`, but it is not a strict requirement or expectation.

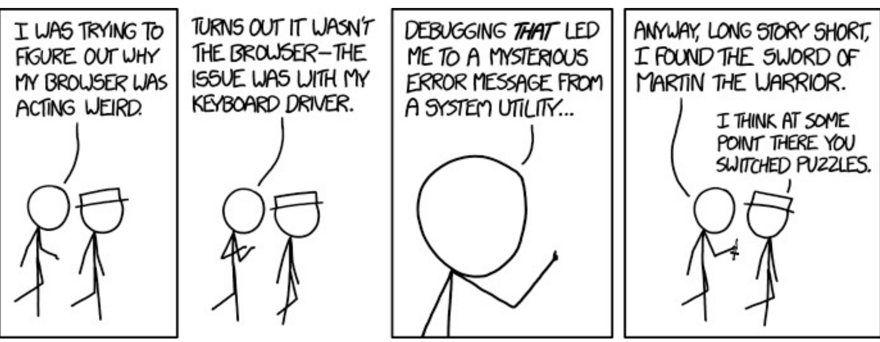
CHAPTER

TWENTYONE

DEBUGGING

21.1 Debugging Unicon

Unicon development is, in large part, an academic research effort. With later releases, Unicon is adding features ready for industrial and commercial use, but the seeds are in academia. This bodes well for debugging features, with facilities available to quantize and analyze program correctness, performance and aid in visualization of program structure and runtime characteristics.



XKCD http://xkcd.com/1722/ by Randall Munroe *CC BY-NC 2.5*

21.1.1 Trace

Unicon has tracing features baked in. A command line switch

```
unicon -t program.icn
```

turns on tracing. There is also a check for an environment variable `TRACE`, during program startup, and the keyword *&trace* to provide fine tuned control over when source level tracing is used.

21.1.2 UDB

Ziad A. Al-Sharif wrote a dissertation on an Extensible Debugging Architecture, and one of the aspects of the paper was a production ready Unicon Debugger, UDB.

The dissertation is in Technical Report (UTR10a 2009/01/08) `Debugging with UDB`.

http://unicon.org/utr/utr10.html

CHAPTER

TWENTYTWO

EXECUTION MONITORING

22.1 Unicon monitoring

Execution monitoring is a profiling aid for `Unicon` programs.

Relying on multi threading, a monitoring control program loads other programs and receives (possibly filtering) events.

TP Target Program

EM Execution Monitor

In the Unicon source distribution, in `unicon/tests/special` there is a basic monitoring test. Monitoring is usually setup with a TP (target program) and an EM (execution monitor). That sample has `1to10.icn` a simple generator as the target program, and `monitor.icn` the sample EM.

For the capture, `1to10.icn` is changed to `1to4.icn`, to decrease the listing size.

```
#
# program counts to 4 for use by monitoring test
#
procedure main()
    every write("\n" || (1 to 4))
end
```

```
#
# if you can't do this much monitoring, the monitoring facilities do not work
#
link evinit

procedure main()
  if not (&features == "dynamic loading") then runerr(121)
  if not (&features == "event monitoring") then runerr(121)
  EvInit("1to4")
  while EvGet() do {
      write(ord(&eventcode), ": ", image(&eventvalue))
```

```
  }
end
```

```
# Make the monitor sample
.RECIPEPREFIX = >

monitor: monitor.icn 1to4
> unicon monitor
> ./monitor

1to4: 1to4.icn
> unicon 1to4
```

```
prompt$ make monitor
make[1]: Entering directory '/home/btiffin/wip/writing/unicon/examples'
make[1]: 'monitor' is up to date.
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/examples'
```

Hmm, codes. What all the monitoring event numbers mean are held in the *IPL*, in the mprocs, mprogs, mincl directories. `mprocs/evnames.icn` is a good clue, but they are indexed by enumeration names, which is in `mincl/evdefs.icn` (which is mostly octal character code). Execution monitoring with Unicon is very much a tool chain that requires computer assistance to use effectively at the capture and dump level. Higher level programs that provide visualization is a much more human friendly interface to execution monitoring.

But just for instance, when the first column is `80:` (octal 120), that matches an `E_Fret` event code. Makes sense as the second column for those events are the 1, 2, 3, 4 values returned from the `1to4` program.

Let's change the EM to be a little more friendly as a sample.

```
#
# if you can't do this much monitoring, the monitoring facilities do not work
#
link evinit, evnames

procedure main()
  if not (&features == "dynamic loading") then runerr(121)
  if not (&features == "event monitoring") then runerr(121)
  EvInit("1to4")
  while e := EvGet() do {
      write(right(ord(&eventcode), 3), ": ", right(image(&eventvalue), 16),
            ": ", evnames(e))
  }
end
```

```
prompt$ make monitor-names
make[1]: Entering directory '/home/btiffin/wip/writing/unicon/examples'
make[1]: 'monitor-names' is up to date.
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/examples'
```

Ahh, that's a little easier on the eyes and brain. And a sweet little view of what's going on. I wonder how that maps to some ucode?

```
make[1]: Entering directory '/home/btiffin/wip/writing/unicon/examples'
make[1]: 'monitor-names.u' is up to date.
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/examples'
```

```
version        U12.1.00
uid        monitor-names.u1-1476161329-0
impl        local
link        evinit.u
link        evnames.u
global        1
        0,000005,main,0
^L
proc main
        local        0,000000,runerr
        local        1,000000,EvInit
        local        2,000000,e
        local        3,000000,EvGet
        local        4,000000,write
        local        5,000000,right
        local        6,000000,ord
        local        7,000000,image
        local        8,000000,evnames
        con        0,010000,15,144,171,156,141,155,151,143,040,154,157,141,144,151,
↪156,147
        con        1,002000,3,121
        con        2,010000,16,145,166,145,156,164,040,155,157,156,151,164,157,162,
↪151,156,147
        con        3,010000,4,061,164,157,064
        con        4,002000,1,3
        con        5,010000,2,072,040
        con        6,002000,2,16
        declend
        filen        monitor-names.icn
        line        13
        colm        12
        synt        any
        mark        L1
        line        14
        colm        3
        synt        if
        mark0
        mark        L2
        pnull
        line        14
        colm        11
        synt        any
        keywd        features
        str        0
        line        14
        colm        21
        synt        any
        lexeq
        unmark
        efail
lab L2
        pnull
        unmark
        var        0
        int        1
        line        14
        colm        54
        synt        any
```

```
        invoke        1
        line        14
        colm        3
        synt        endif
        unmark
lab L1
        mark        L3
        line        15
        colm        3
        synt        if
        mark0
        mark        L4
        pnull
        line        15
        colm        11
        synt        any
        keywd        features
        str        2
        line        15
        colm        21
        synt        any
        lexeq
        unmark
        efail
lab L4
        pnull
        unmark
        var        0
        int        1
        line        15
        colm        55
        synt        any
        invoke        1
        line        15
        colm        3
        synt        endif
        unmark
lab L3
        mark        L5
        var        1
        str        3
        line        16
        colm        9
        synt        any
        invoke        1
        unmark
lab L5
        mark        L6
lab L7
        line        17
        colm        3
        synt        while
        mark0
        pnull
        var        2
        var        3
        line        17
        colm        19
```

```
        synt         any
        invoke         0
        line         17
        colm         11
        synt         any
        asgn
        unmark
        mark         L7
        var        4
        var        5
        var        6
        line         18
        colm         23
        synt         any
        keywd         eventcode
        line         18
        colm         22
        synt         any
        invoke         1
        int        4
        line         18
        colm         18
        synt         any
        invoke         2
        str        5
        var        5
        var        7
        line         18
        colm         58
        synt         any
        keywd         eventvalue
        line         18
        colm         57
        synt         any
        invoke         1
        int        6
        line         18
        colm         51
        synt         any
        invoke         2
        str        5
        var        8
        var        2
        line         19
        colm         26
        synt         any
        invoke         1
        line         18
        colm         12
        synt         any
        invoke         5
lab L8
        unmark
        goto         L7
lab L9
        line         17
        colm         3
        synt         endwhile
```

```
        unmark
lab L6
        pnull
        line        21
        colm        1
        synt        any
        pfail
        end
```

And, no, not really any quick grok matching .u to the monitor; that listing means relatively little, at this point in studying Unicon, in relation to the execution events. .u with the VM, and this particular monitor, are at different layers, and there is no one to one mapping. That's ok, lesson learned. Reading .u is pretty nice though, the `line` and `colm` mnemonics make it pretty easy to mentally map source to VM and let the brain know where things are in the code for each operation.

22.1.1 Visualization

And Event Monitoring starts to shine. Take your eyes off the code and look at some pictures. Let the running programming paint its own picture.

Todo

add some plots

CHAPTER

TWENTYTHREE

PERFORMANCE

23.1 Unicon performance

Unicon is a very high level language. The runtime engine, with generators, co-expressions, threading, and the assortment of other features means that most operations need to include a fair number of conditional tests to verify correctness. While this is some overhead, Unicon still performs at a very respectable level. The C compilation mode can help when performance is a priority, and *loadfunc* is always available when C or Assembly level speed is necessary for critical operations.

Unicon, interpreting *icode* files, ranges from 20 to 40 times slower than optimized C code in simple loops and straight up computations. On par with similar code in Python. Compiled Unicon (via `unicon -C`) is probably about 10 times slower than similar C when timing a tight numeric computation loop. This is mainly due to the overhead that is required for the very high level language features of Unicon, mentioned above.

As always, those are somewhat unfair comparisons. The tasks that Unicon can be applied to and the development time required to get correct solutions can easily outweigh a few seconds of runtime per program pass. Saving a week of effort can mean that many thousands of program runs are required before a developer hits a break even point. Five eight hours days total up 144,000 seconds. If the delta for a program run between C and Unicon is 5 seconds per run, you'd need to run a program over 28,000 times to make up for a one week difference in development time. Critical routines can leverage loadable functions when needed. All these factors have to be weighed when discussing performance issues.

With all that in mind, it's still nice to have an overall baseline for daily tasks, to help decide when to bother with *loadfunc* or which language is the right tool for the task at hand[1].

Note: This section is unashamedly biased towards Unicon. It's the point of the exercise. All comparisons assume that point of view and initial bias.

[1] Or more explicitly; *not the wrong tool for the task at hand*. Most general purpose programming languages are capable of providing a workable solution to any computing problem, but sometimes the problem specific advantages in one language make it an obvious choice for mixing with Unicon for performance and or development time benefits.

23.2 Summing integers

Given a simple program, creating a sum of numbers in a tight loop. Comparing `Unicon` with `Python`, and `C`.

Other scripting and compiled languages are included for general interest sake. This lists simple code running 16.8 million iterations while tallying a sum in each language.

Note: The representative timings below each listing are approximate, and results will vary from run to run. There is a fixed number included with each the listing to account for those times when the document generation may have occurred while the build system was blocked or busy at the point of timing run capture. Tested running Xubuntu with an AMD A10-5700 quadcore APU chipset. Different hardware would have different base values, but should have equivalent relative timings.

23.2.1 Unicon

Unicon, *tightloop.icn*

```
#
# tightloop trial, sum of values from 0 to 16777216
#
procedure main()
    total := 0
    every i := 0 to 2^24 do total +:= i
    write(total)
end
```

Representative timing: 2.02 seconds, 0.55 seconds (-C compiled)

unicon (icode)

```
prompt$ time -p unicon -s tightloop.icn -x
140737496743936
real 2.23
user 2.21
sys 0.01
```

unicon -C

```
prompt$ unicon -s -o tightloop-uc -C tightloop.icn
```

```
prompt$ time -p ./tightloop-uc
140737496743936
real 0.62
user 0.62
sys 0.00
```

23.2.2 Python

Python, *tightloop-py.py*

```
# Sum of values from 0 to 16777216
total = 0
n = 2**24
```

```
for i in range(n, 1, -1):
    total += i
print(total)
```

Representative timing: 2.06 seconds

```
prompt$ time -p python tightloop-py.py
140737496743935
real 2.14
user 1.91
sys 0.22
```

23.2.3 C

C, *tightloop-c.c*

```
/* sum of values from 0 to 16777216 */
#include <stdio.h>

int
main(int argc, char** argv)
{
    int i;
    int n;
    unsigned long total;

    total = 0;
    n = 1 << 24;
    for (i = n; i > 0; i--)
        total += i;
    printf("%lu\n", total);
    return 0;
}
```

Representative timing: 0.05 seconds

```
prompt$ gcc -o tightloop-c tightloop-c.c
```

```
prompt$ time -p ./tightloop-c
140737496743936
real 0.05
user 0.05
sys 0.00
```

23.2.4 Ada

Ada, *tightloopada.adb*

```
-- Sum of values from 0 to 16777216
with Ada.Long_Long_Integer_Text_IO;

procedure TightLoopAda is
    total : Long_Long_Integer;
begin
```

```
    total := 0;
    for i in 0 .. 2 ** 24 loop
        total := total + Long_Long_Integer(i);
    end loop;
    Ada.Long_Long_Integer_Text_IO.Put(total);
end TightLoopAda;
```

Representative timing: 0.06 seconds

GNAT Ada, 5.4.0

```
prompt$ gnatmake tightloopada.adb
gnatmake: "tightloopada" up to date.
```

```
prompt$ time -p ./tightloopada
     140737496743936
real 0.05
user 0.05
sys 0.00
```

23.2.5 ALGOL

ALGOL, *tightloop-algol.a68*[2]

```
# Sum of values from 0 to 16777216 #
BEGIN
    INT i := 0;
    LONG INT total := 0;
    FOR i FROM 0 BY 1 TO 2^24 DO
        total +:= i
    OD;
    print ((total))
END
```

Representative timing: 5.91 seconds

```
prompt$ a68g tightloop-algol
                    +140737496743936
real 5.91
user 5.86
sys 0.03
```

23.2.6 Assembler

Assembler, *tightloop-assembler.s*

```
# Sum of integers from 0 to 16777216

           .data
aslong:    .asciz  "%ld\n"
```

[2] Due to the length of the timing trials, some results are not automatically captured during documentation generation. Results for those runs were captured separately and copied into the document source. All other program trials are evaluated during each Sphinx build of this document *(results will vary slightly from release to release)*.

```
          .text
          .globl main
main:
           push %rbp                  # need to preserve base pointer
           movq %rsp, %rbp            # local C stack, frame size 0

           movq $0, %rax              # clear out any high bits
           movl $1, %eax              # eax counts down from
           shll $24, %eax             # 2^24
           movq $0, %rbx              # rbx is the running total

top:       addq %rax, %rbx            # add in current value, 64 bit
           decl %eax                  # decrement counter (as 32 bit)
           jnz top                    # if counter not 0, then loop again

done:      movq %rbx, %rsi            # store sum in rsi for printf arg 2
           lea aslong(%rip), %rdi     # load format string address
           call printf                # output formatted value

           movl $0, %eax              # shell result code
           leave
           ret
```

Representative timing: 0.01 seconds

```
prompt$ gcc -o tightloop-assembler tightloop-assembler.s
```

```
prompt$ time -p ./tightloop-assembler
140737496743936
real 0.01
user 0.01
sys 0.00
```

23.2.7 BASIC

BASIC, *tightloop-basic.bac*

```
REM Sum of values from 0 to 16777216
total = 0
FOR i = 0 TO 1<<24
    total = total + i
NEXT
PRINT total
```

Representative timing: 0.05 seconds

```
prompt$ bacon -y tightloop-basic.bac >/dev/null
```

```
prompt$ time -p ./tightloop-basic
140737496743936
real 0.05
user 0.05
sys 0.00
```

23.2.8 C (baseline)

See above, Unicon, C and Python are the ballpark for this comparison.

23.2.9 COBOL

COBOL, *tightloop-cobol.cob*

```
      *> Sum of values from 0 to 16777216
       identification division.
       program-id. tightloop-cob.

       data division.
       working-storage section.
       01 total          usage binary-double value 0.
       01 counter        usage binary-long.
       01 upper          usage binary-long.

       procedure division.
       compute upper = 2**24
       perform varying counter from 0 by 1 until counter > upper
           add counter to total
       end-perform
       display total
       goback.
       end program tightloop-cob.
```

Representative timing: 0.06 seconds

GnuCOBOL 2.0-rc3

```
prompt$ cobc -x tightloop-cobol.cob
```

```
prompt$ time -p ./tightloop-cobol
+00000140737496743936
real 0.07
user 0.07
sys 0.00
```

23.2.10 D

D, *tightloop-d.d*

```
/* Sum of values from 0 to 16777216 */
module tightloop;
import std.stdio;

void
main(string[] args)
{
    long total = 0;
    for (int n = 0; n <= 1<<24; n++) total += n;
    writeln(total);
}
```

Representative timing: 0.05 seconds

gdc

```
prompt$ gdc tightloop-d.d -o tightloop-d
```

```
prompt$ time -p ./tightloop-d
140737496743936
real 0.05
user 0.05
sys 0.00
```

23.2.11 ECMAScript

ECMAScript, *tightloop-js.js*

```
/* Sum of values from 0 to 16777216 */
var total = 0;
for (var i = 0; i <= Math.pow(2,24); i++) total += i;

// Account for gjs, Seed, Duktape, Jsi
try { print(total); } catch(e) {
    try { console.log(total); } catch(e) {
        try { puts(total); } catch(e) {}
    }
}
```

Representative timing: 0.83 seconds (node.js), 10.95 (gjs), 63.37 (duktape)

nodejs

```
prompt$ time -p nodejs tightloop-js.js
140737496743936
real 0.78
user 0.78
sys 0.00
```

gjs[2]

```
prompt$ time -p gjs tightloop-js.js
140737496743936
real 10.96
user 10.95
sys 0.00
```

Duktape[2]

```
prompt$ time -p duktape tightloop-js.js
140737496743936
real 63.37
user 63.36
sys 0.00
```

23.2.12 Elixir

Elixir, *tightloop-elixir.ex*

```
# Sum of values from 0 to 16777216
Code.compiler_options(ignore_module_conflict: true)
defmodule Tightloop do
    def sum() do
        limit = :math.pow(2, 24) |> round
        IO.puts Enum.sum(0..limit)
    end
end
Tightloop.sum()
```

Representative timing: 2.03 seconds

elixirc 1.1.0-dev

```
prompt$ time -p elixirc tightloop-elixir.ex
140737496743936
real 2.06
user 2.05
sys 0.12
```

23.2.13 Forth

Forth, *tightloop-ficl.fr*

```
( Sum of values from 0 to 16777216)
variable total
: tightloop ( -- )  1 24 lshift 1+ 0 do  i total +!  loop ;
0 total !  tightloop  total ? cr
bye
```

Representative timing: 0.52 seconds (Ficl), 0.12 seconds (Gforth)

ficl

```
prompt$ time -p ficl tightloop-ficl.fr
140737496743936
real 0.51
user 0.51
sys 0.00
```

gforth

```
prompt$ time -p gforth tightloop-ficl.fr
140737496743936
real 0.12
user 0.12
sys 0.00
```

23.2.14 Fortran

Fortran, *tightloop-fortran.f*

```
! sum of values from 0 to 16777216
program tightloop
    use iso_fortran_env
```

```
    implicit none

    integer :: i
    integer(kind=int64) :: total

    total = 0
    do i=0,2**24
        total = total + i
    end do
    print *,total
end program tightloop
```

Representative timing: 0.06 seconds

gfortran

```
prompt$ gfortran -o tightloop-fortran -ffree-form tightloop-fortran.f
```

```
prompt$ time -p ./tightloop-fortran
      140737496743936
real 0.06
user 0.06
sys 0.00
```

23.2.15 Groovy

Groovy, *tightloop-groovy.groovy*

```
/* Sum of value from 0 to 16777216 */

public class TightloopGroovy {
    public static void main(String[] args) {
        long total = 0;
        for (int i = 0; i <= 1<<24; i++) {
            total += i
        }
        println(total)
    }
}
```

Representative timing: 0.47 seconds (will use multiple cores)

groovyc 1.8.6, OpenJDK 8

```
prompt$ groovyc tightloop-groovy.groovy
```

```
prompt$ time -p java -cp ".:/usr/share/groovy/lib/*" TightloopGroovy
140737496743936
real 0.65
user 1.27
sys 0.07
```

23.2.16 Java

Java, *tightloopjava.java*

```
/* Sum of values from 0 to 16777216 */
public class tightloopjava {
    public static void main(String[] args) {
        long total = 0;

        for (int n = 0; n <= Math.pow(2, 24); n++) {
            total += n;
        }
        System.out.println(total);
    }

}
```

Representative timing: 0.11 seconds

OpenJDK javac

```
prompt$ javac tightloopjava.java
```

```
prompt$ time -p java -cp . tightloopjava
140737496743936
real 0.67
user 0.79
sys 0.02
```

23.2.17 Lua

Lua, *tightloop-lua.lua*

```
-- Sum of values from 0 to 16777216
total = 0
for n=0,2^24,1 do
    total = total + n
end
print(string.format("%d", total))
```

Representative timing: 0.73 seconds

lua

```
prompt$ time -p lua tightloop-lua.lua
140737496743936
real 0.72
user 0.72
sys 0.00
```

23.2.18 Neko

Neko, *tightloop-neko.neko*

```
// Sum of values from 0 to 16777216
var i = 0;
var total = 0.0;
var limit = 1 << 24;
while i <= limit {
```

```
    total += i;
    i += 1;
}
$print(total, "\n");
```

Representative timing: 0.89 seconds

nekoc, neko

```
prompt$ nekoc tightloop-neko.neko
```

```
prompt$ time -p neko tightloop-neko
140737496743936
real 0.96
user 0.99
sys 0.22
```

23.2.19 Nickle

Nickle, *tightloop-nickle.c5*

```
/* Sum of values from 0 to 16777216 */
int total = 0;
int n = 1 << 24;

for (int i = n; i > 0; i--) {
    total += i;
}
printf("%g\n", total);
```

4.85 seconds representative

Nickle 2.77[2]

```
prompt$ time -p nickle tightloop-nickle.c5
140737496743936
real 4.85
user 4.83
sys 0.01
```

23.2.20 Nim

Nim, *tightloopNim.nim*

```
# Sum of values from 0 to 16777216
var total = 0
for i in countup(0, 1 shl 24):
    total += i
echo total
```

Representative timing: 0.31 seconds

nim

```
prompt$ nim compile --verbosity:0 --hints:off tightloopNim.nim
```

```
prompt$ time -p ./tightloopNim
140737496743936
real 0.30
user 0.30
sys 0.00
```

23.2.21 Perl

Perl, *tightloop-perl.pl*

```
# sum of values from 0 to 16777216
my $total = 0;
for (my $n = 0; $n <= 2 ** 24; $n++) {
    $total += $n;
}
print "$total\n";
```

Representative timing: 1.29 seconds

perl 5.22

```
prompt$ time -p perl tightloop-perl.pl
140737496743936
real 1.40
user 1.39
sys 0.00
```

23.2.22 PHP

PHP, *tightloop-php.php*

```
<?php
# Sum of values from 0 to 16777216
$total = 0;
for ($i = 0; $i <= 1 << 24; $i++) {
    $total += $i;
}
echo $total.PHP_EOL;
?>
```

Representative timing: 0.39 seconds

PHP 7.0.15, see *PHP*.

```
prompt$ time -p php tightloop-php.php
140737496743936
real 0.40
user 0.39
sys 0.00
```

23.2.23 Python

See above. Unicon, C and Python are the ballpark for this comparison.

23.2.24 REBOL

REBOL, *tightloop-rebol.r*

```
; Sum of values from 0 to 16777216
REBOL []
total: 0
for n 0 to integer! 2 ** 24 1 [total: total + n]
print total
```

2.22 seconds representative

r3

```
prompt$ time -p r3 tightloop-rebol.r3
140737496743936
real 2.75
user 2.17
sys 0.00
```

23.2.25 REXX

REXX, *tightloop-rexx.rex*

```
/* Sum of integers from 0 to 16777216 */
parse version host . .
parse value host with 6 which +6
if which = "Regina" then
    numeric digits 16

total = 0
do n=0 to 2 ** 24
    total = total + n
end
say total
```

2.38 seconds representative (oorexx), 4.48 (regina)

oorexx 4.2

```
prompt$ time -p /usr/bin/rexx tightloop-rexx.rex
140737496743936
real 2.51
user 2.49
sys 0.01
```

regina 3.9, slowed by NUMERIC DIGITS 16 for clean display[2]

```
prompt$ time -p rexx tightloop-rexx.rex
140737496743936
real 4.84
```

```
user 4.84
sys 0.00
```

23.2.26 Ruby

Ruby, *tightloop-ruby.rb*

```
# Sum of values from 0 to 16777216
total = 0
for i in 0..2**24
    total += i
end
puts(total)
```

Representative timing: 1.16 seconds

ruby 2.3

```
prompt$ time -p ruby tightloop-ruby.rb
140737496743936
real 1.23
user 1.22
sys 0.00
```

23.2.27 Rust

Rust, *tightloop-rust.rs*

```
// sum of values from 0 to 16777216
fn main() {
    let mut total: u64 = 0;
    for i in 0..1<<24 {
        total += i;
    }
    println!("{}", total);
}
```

Representative timing: 0.44 seconds

rustc 1.16.0

```
prompt$ /home/btiffin/.cargo/bin/rustc tightloop-rust.rs
```

```
prompt$ time -p ./tightloop-rust
140737479966720
real 0.37
user 0.37
sys 0.00
```

23.2.28 Scheme

Scheme, *tightloop-guile.scm*

```
; sum of values from 0 to 16777216
(define (sum a b)
  (do ((i a (+ i 1))
       (result 0 (+ result i)))
      ((> i b) result)))

(display (sum 0 (expt 2 24)))
(newline)
```

Representative timing: 0.85 seconds

guile 2.0.11

```
prompt$ time -p guile -q tightloop-guile.scm
140737496743936
real 0.85
user 0.85
sys 0.00
```

23.2.29 Shell

Shell, *tightloop-sh.sh*

```
# Sum of integers from 0 to 16777216
i=0
total=0
while [ $i -le $((2**24)) ]; do
    let total=total+i
    let i=i+1
done
echo $total
```

Representative timing: 281.29 seconds

bash 4.3.46[2]

```
prompt$ time -p source tightloop-sh.sh
140737496743936
real 281.29
user 280.08
sys 1.11
```

23.2.30 S-Lang

S-Lang, *tightloop-slang.sl*

```
% Sum of values from 0 to 16777216
variable total = 0L;
variable i;
for (i = 0; i <= 1<<24; i++)
    total += i;
message(string(total));
```

Representative timing: 4.92 seconds

slsh 0.9.1 with S-Lang 2.3[2]

```
prompt$ time -p slsh tightloop-slang.sl
140737496743936
real 4.92
user 4.92
sys 0.00
```

23.2.31 Smalltalk

Smalltalk, *tightloop-smalltalk.st*

```
"sum of values from 0 to 16777216"
| total |
total := 0
0 to: (1 bitShift: 24) do: [:n | total := total + n]
(total) printNl
```

Representative timing: 4.60 seconds

GNU Smalltalk 3.2.5[2]

```
prompt$ time -p gst tightloop-smalltalk.st
140737496743936
real 4.56
user 4.55
sys 0.00
```

23.2.32 SNOBOL

SNOBOL, *tightloop-snobol.sno*

```
* Sum of values from 0 to 16777216
        total = 0
        n = 0
loop    total = total + n
        n = lt(n, 2 ** 24) n + 1                    :s(loop)
        output = total
end
```

Representative timing: 5.83 seconds

snobol4 CSNOBOL4B 2.0[2]

```
prompt$ time -p snobol4 tightloop-snobol.sno
140737496743936
real 5.83
user 5.82
sys 0.00
```

23.2.33 Tcl

Tcl, *tightloop-tcl.tcl*

```
# Sum of values from 0 to 16777216
set total 0
for {set i 0} {$i <= 2**24} {incr i} {
    incr total $i
}
puts "$total"
```

Representative timing: 4.59 seconds (jimsh), 17.69 seconds (tclsh)

jimsh 0.76[2]

```
prompt$ time -p jimsh tightloop-tcl.tcl
140737496743936
real 4.59
user 4.59
sys 0.00
```

tclsh 8.6[2]

```
prompt$ time -p tclsh tightloop-tcl.tcl
140737496743936
real 17.69
user 17.67
sys 0.00
```

23.2.34 Vala

Vala, *tightloop-vala.vala*

```
/* Sum of values from 0 to 16777216 */
int main(string[] args) {
    long total=0;
    for (var i=1; i <= 1<<24; i++) total += i;
    stdout.printf("%ld\n", total);
    return 0;
}
```

Representative timing: 0.16 seconds

valac 0.34.2

```
prompt$ valac tightloop-vala.vala
```

```
prompt$ time -p ./tightloop-vala
140737496743936
real 0.10
user 0.10
sys 0.00
```

23.2.35 Genie

Vala/Genie, *tightloop-genie.gs*

```
/* Sum of values from 0 to 16777216 */
[indent=4]
init
    total:long = 0
    for var i = 1 to (1<<24)
        total += i
    print("%ld", total)
```

Representative timing: 0.16 seconds

valac 0.34.2

```
prompt$ valac tightloop-genie.gs
```

```
prompt$ time -p ./tightloop-genie
140737496743936
real 0.10
user 0.10
sys 0.00
```

23.2.36 Unicon loadfunc

A quick test for speeding up Icon

Unicon loadfunc, *tightloop-loadfunc.icn*

```
#
# tightloop trial, sum of values from 0 to 16777216
#
procedure main()
    faster := loadfunc("./tightloop-cfunc.so", "tightloop")
    total := faster(2^24)
    write(total)
end
```

Representative timing: 0.05 seconds

C loadfunc for Unicon, *tightloop-cfunc.c*

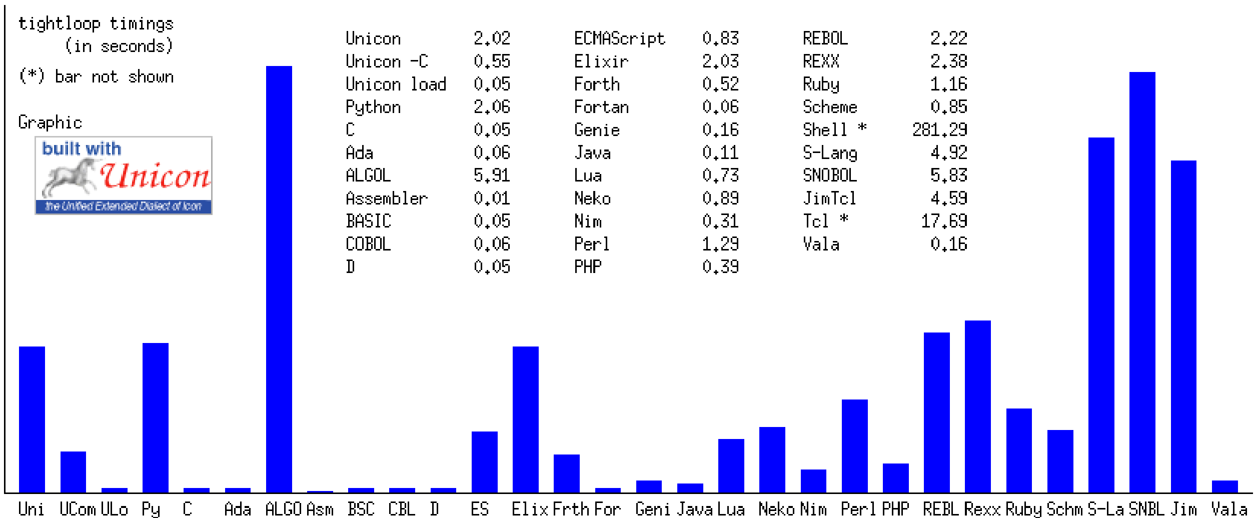
```
/* sum of values from 0 to integer in argv[1] */

#include "../icall.h"

int
tightloop(int argc, descriptor argv[])
{
    int i;
    unsigned long total;

    ArgInteger(1);
    total = 0;
    for (i = 0; i <= IntegerVal(argv[1]); i++) total += i;
    RetInteger(total);
}
```

unicon with loadfunc

```
prompt$ gcc -o tightloop-cfunc.so -O3 -shared -fpic tightloop-cfunc.c
```

```
prompt$ time -p unicon -s tightloop-loadfunc.icn -x
140737496743936
real 0.05
user 0.03
sys 0.00
```

23.2.37 Summary



In the image above, bars for Bash shell and Tclsh are not included. ECMAScript value is from Node.js (V8).[3]

There was no attempt to optimize any code. Compile times are only included when that is the *normal* development cycle. Etcetera. Routines can always be optimized, tightened, and fretted over. The point of this little exercise is mainly for rough guidance (perhaps with healthy doses of confirmation, self-serving, halo effect, academic, and/or experimenters bias[8]).[4]

While there may be favouritism leaking through this summary, to the best of my knowledge and belief there is no deliberate shilling. As this is free documentation in support of free software, I can attest to no funding, bribery or insider trading bias as well. Smiley. I will also attest to the belief that **Unicon is awesome**. You should use it for as many projects as possible, and not feel any regret or self-doubt while doing so. *Double smiley.*

With that out of the way, here is a recap:

Unicon translated to *icode* is nearly equivalent to *Python* and *Elixir* in terms of the timing (variants occur between runs, but within a few tenths or hundredths of a second difference, up and down).

C wins, orders of magnitude faster than the scripted language trials and marginally faster than most of the other compiled trials.

A later addition of `gcc -O3` compiles and an Assembler sample run faster, but that counts as fretting[4].

[3] The bar chart graphic was generated with a small Unicon program.

[8] With biased opinions comes cognitive filtering. While writing the various `tightloop` programs, I wanted Unicon to perform well in the timing trials. That cognitive bias may have influenced how the results were gathered and reported here. Not disappointed with the outcomes, but `C` sure sets a high bar when it comes to integer arithmetic.

[4] *Ok, I eventually fretted.* Added `loadfunc()` to show off mixed programming for speed with *Unicon*. Also added -O3 gcc timing and an assembler sample that both clock in well under the baseline timing.

The *GnuCOBOL*, *gfortran*, *GNAT/Ada* and *BaCon* programs fare well, only a negligible fraction slower than the baseline *C*. Both *GnuCOBOL* and *BaCon* use *C* intermediates on the way to a native compile. *gfortran* uses the same base compiler technology as *C* in these tests (GCC). *Unicon* can load any of these modules when pressed for time.

D also fares well, with sub tenth of a second timing.

Java and *Gforth* test at a third as fast as C, admirable, neck and neck with *Vala* and *Genie*. *Nim*, *PHP* and *Rust* clock in shortly after those timings.

Unicon compiled via `-C` completes roughly 4 times faster than the *icode* virtual machine interpreter version, at about 1/10th C speed.

Ruby, *Perl* and *Neko*, are faster than interpreted *Unicon* for this sample.

Elixir clocks in next and like *Python*, pretty close to the bar set by interpreted *Unicon*.

REBOL and *REXX* clock in just a little slower than the *Unicon* mark.

Python and *Elixir* perform this loop with similar timings to interpreted *Unicon*, all within 2% of each others elapsed time. Widening the circle a little bit, *REXX* and *REBOL* become comparable as well. *Revealing a bit of the author's bias, let's call these* **the close competition** *while discussing raw performance. I have a different (overlapping) set of languages in mind when it comes to overall competitive development strategies*[7].

Tcl takes about twice as long as *Unicon* when using the *JimTcl* interpreter, and approaching 9 times slower with a full *Tcl* interpreter. *Gjs* ended up timing in between the two *Tcl* engines.

ALGOL, *S-Lang*, *Smalltalk* and *SNOBOL* also took about twice as long as *Unicon* when running this trial.

Duktape ran at second to last place, just over a minute.

bash was unsurprisingly the slowest of the bunch, approaching a 5 minute run time for the 16.8 million iterations.

Native compiled *Unicon* timing is on par with *Ficl*, and a little faster than *Lua*, *node.js*, and then *Guile*, but still about 10 times slower than the compiled *C* code. (*Unicon* includes automatic detection of native integer overflow, and will seamlessly use large integer support routines when needed. Those tests will account for some of the timing differences in this particular benchmark when compared to straight up *C*).

Once again, these numbers are gross estimates, no time spent fretting over how the code was run, or worrying about different implementations, just using the tools as *normal* (for this author, when not fretting[4]).

Unicon stays reasonably competitive, all things considered.

[7] Not to leave you hanging; I put C, C++, C#, Erlang, Go, Java, Lua, Node.js, Perl, Python, PHP, Red, and Ruby firmly in the Unicon competitive arena. Bash/Powershell and Javascript count as auxiliary tools, that will almost always be mixed in with project developments. Same can be said for things like HTML/CSS and SQL, tools that will almost always be put to use, but don't quite count as "main" development systems. For Erlang, I also count Elixir, and for Java that includes Scala, Groovy and the like. I live in GNU/Linux land, so my list doesn't include Swift or VB.NET etc, your short list may differ. I also never quite took to the Lisp-y languages so Scheme and Closure don't take up many brain cycles during decision making. And finally, I'm a huge COBOL nerd, and when you need practical programming, COBOL should always be part of the development efforts.

Table 23.1: Timings

Scale	Languages
<1x	Assembler, -O3 C
1x	C, Ada, BASIC, COBOL, D, Fortran, **Unicon loadfunc**
3x	Java, GForth, Vala, Genie
6x	Nim, PHP, Rust
10x	**Unicon -C**
15x	Ficl, Lua, Scheme
20x	Neko, Node.js
25x	Perl, Ruby
40x	**Unicon**, Elixir, Python, REBOL, REXX
100x	Algol, JimTcl, S-Lang, Smalltalk, SNOBOL
200x	Gjs
350x	Tcl
1200x	Duktape
5600x	Shell

The Unicon perspective

With Unicon in large (or small) scale developments, if there is a need for speed, you can always write critical sections in another compiled language and load the routines into Unicon space. The overhead for this is minimal in terms of both source code wrapper requirements and runtime data conversions[5]. The number of systems that can produce compatible loadable objects is fairly extensive; C, C++, COBOL, Fortran, Vala/Genie, Ada, Pascal, Nim, BASIC, Assembler, to name but a few. This all means that in terms of performance, Unicon is never a bad choice for small, medium or large projects. The leg up on development time may outweigh a lot of the other complex considerations.

A note on development time

With a program this short and simple minded, development time differences are negligible. Each sample takes a couple of minutes to write, test and document[6].

That would change considerably as problems scaled up in size. Not just in terms of lines of code needed, but also the average time to get each line written, tested and verified correct.

General know how with each environment would soon start to influence productivity and correctness, and bring up a host of other issues. A lot can be said for domain expertise with the language at hand. Without expertise, development time may extend; referencing materials, searching the ecosystem for library imports, becoming comfortable with idioms, with testing and with debugging.

The less lines that need to be written to solve tasks can add up to major benefits when comparing languages in non-trivial development efforts.

---

[5] There are a few examples of how much (little) code is required to wrap compiled code in Unicon in this docset. See *libsoldout markdown* for one example, wrapping a C library to perform Markdown to HTML processing in about 30 lines of mostly boilerplate source. Other entries in the *Programs* chapter exercise and demonstrate the *loadfunc* feature that allows these mixed language integrations.

[6] (Except for the *BaCon* trial). That example stole a few extra minutes for debugging, out of the blue, many days after the initial code writing and verification pass. It turns out `BaCon` generated a temporary `Makefile` during its translation to C phase, and I had to track down the mystery when an unrelated `Makefile` sample kept disappearing during generation of new versions of this document. That led to moving all the performance samples to a separate sub-directory to avoid the problem, and any others that may occur in the future when dealing with that many programming environments all at the same time and in the same space. *BaCon was fixed the same day as the inconvenience report.*

23.3 Development time

Even though execution time benchmarking is hard to quantify in an accurate way (there are always issues unaccounted for, or secrets yet to uncover) and is fraught with biased opinion[8], judging development time is magnitudes harder. Sometimes lines pour out of fingers and code works better than expected. Sometimes an off by one error can take an entire afternoon to uncover and flush productivity down the toilet.

In general, for complex data processing issues, very high level languages beat high level languages which beat low level languages when it comes to complete solution development time. It's not a hard and fast rule, but in general.

Unicon counts as a very high level language. There are features baked into Unicon that ease a lot of the day to day burdens faced by many developers. Easy to wield data structures, memory managed by the system and very high level code control mechanisms can all work together to increase productivity and decrease development time. *In general*. For some tasks, `Ruby` may be the better choice, for others tasks `Python` or `C` or `Tcl`, or some language you have never heard of may be the wisest course. Each with a strength, and each having skilled practitioners that can write code faster in that language than in any other.

Within the whole mix, `Unicon` provides a language well suited to productive, timely development with good levels of performance. Other languages can be mixed with `Unicon` when appropriate, including loadable routines that can be as close to the hardware as hand and machine optimized assembler can manage.

If the primary factor is development time, Unicon offers an extremely competitive environment. The feature set leaves very few domains of application left wanting.

23.3.1 Downsides

Unicon has a few places that can expose hard to notice construction problems.

Goal-directed evaluation can spawn exponential backtracking problems when two or more expressions are involved. Some expression syntax can end up doing a lot more work in the background than it would seem at a glance. Bounded expressions (or lack thereof) can cause some head scratching at times.

There are tools in place with Unicon to help find these issues, but nothing will ever beat experience, and experience comes from writing code, lots of code.

Luckily, Unicon is at home when programming in the small as it is with middling and large scale efforts[9]. The *class*, *method*, and *package* features, along with the older *link* directive make for a programming environment that begs for application. There are a lot of problem domains that Unicon can be applied to, and that can all help gaining experience.

[9] Having no actual experience beyond middling sized projects, I asked the forum if anyone has worked on a large Unicon system with over 100,000 lines of code. Here are a couple of the responses:

The biggest project I worked on using Unicon was CVE (http://cve.sourceforge.net/) not sure if we broke the 100,000 LOC [*mark*] though. I don't see any reason why you can't write large projects using Unicon. With the ability to write very concise code in Unicon, I'd argue it is even easier to go big.

–Jafar

The two largest known Unicon projects are SSEUS at the National Library of Medicine, and mKE by Richard McCullough of Context Knowledge Systems, not necessarily in that order. They are in the 50-100,000 lines range. CVE approaches that ballpark when client and server are bundled together.

Ralph Griswold used to brag that Icon programs were often 10x smaller than corresponding programs in mainstream languages, so this language designed for programs mostly under 1,000 lines is applicable for a far wider range of software problems than it sounds. While Icon was designed for programming in the small, its size limits have gradually been eliminated. Unicon has further improved scalability in multiple aspects of the implementation, both the compiler/linker and the runtime system. In addition, Unicon was explicitly intended to further support larger scale software systems, and that is why classes and packages were added.

Clint

Those quotes don't answer all the questions, like what maintainers go through, or how long it takes new contributors to get up to speed, but as anecdotes, I now feel quite comfortable telling anyone and everyone that Unicon is up to the task of supporting large scale development and deployments, *along with the small*.

Due to some of the extraordinary features of Unicon, it can be applied to very complex problems[10]. Complex problems always shout out for elegant solutions, and that lure can lead to some false positives with initial Unicon programs. It can take practice to know when an edge case is not properly handled, or when a data dependent bug sits waiting for the right combination of inputs to cause problems. Rely on Unicon, but keep a healthy level of skepticism when starting out. *This is advice from an author that is just starting out, so keep that in mind. Read the other Technical Reports, articles, and Unicon books; as this document is very much entry level to intermediate Unicon.* Wrap expressions in small test heads and throw weird data at your code. Experiment. Turn any potential Unicon downsides into opportunities.

23.4 Unicon Benchmark Suite

There is a Technical Report (UTR16a) 2014-06-09, by Shea Newton and Clinton Jeffery detailing `A Unicon Benchmark Suite`

http://unicon.org/utr/utr16.pdf and sourced in the Unicon source tree under `doc/utr/utr16.tex`.

The results table shows that `unicon` runs of the benchmarks range from

- `345.2x` (`n-body` calculation) to
- `1.6x` for the `thread-ring` trial, compared to the C code baseline timings.

`uniconc` compiled versions range from

- `57.9x` (`n-body`) to
- `0.6x` (`regex-dna`) of the C baseline.

Uniconc increasing the `n-body` run speed by a factor of `6` compared to the *icode* interpreter. The `regex-dna` trial actually ran faster with Uniconc than in C. Take a look at the TR for full details.

With another caveat; runtime vs development time. There should be a column in any benchmark result sheet that quantifies the development time to get correctly functioning programs. I'd wager that Unicon would shine very bright in that column.

23.4.1 run-benchmark

You can run your own test pass in the `tests/bench` sub-directory of the source tree.

---

[10] I once overheard a C++ engineer with a problem distilling a Grady/Booch style Rapid Application Development system output into a usable API for the project at hand. (I was programming a Forth system that was being upgraded as part of the C++ project, working in the same office space (the big rewrite flopped eventually)).

There was a Tcl/Tk prototype ready for user screening but no easy way of getting smooth data flow across systems, due to the utterly complex cloud diagrams to C++ class data interactions. 1995 timeframe.

Icon, a one day development effort, decoding the RAD binaries and text, creating templated sources in C++, Forth, Tcl/Tk (all of it simplified, as a prototype) with surprisingly accurate (to those experiencing first exposure to Icon) data sub fielding in drop downs linked to recursive sub lists. *Ok, two days, and an all nighter; from the middle of one working day until near the end of the next, to a working demo.* Source code all generated by an Icon translator from RAD to programming languages, data access names, structures, all synced for inter-system transfer. RAD to C++, Forth, Tcl/Tk; thanks to Icon.

I had overheard and then bragged that their Grady/Booch project plan wasn't too big or complex for Icon version 6; then had to deliver. There was motivation with nerd points at stake. Icon shone, but no one really noticed. Management was more impressed by having (prototype) GUI screens with data connectivity for the big project than how it was developed overnight, leveraging goal directed evaluation, string scanning, and utterly flexible Icon data structures.

And yes, further use uncovered some interesting hot spots when it came to performance. Learning done the hard way. Implicit back tracking can lead to unnecessary cycles if some care is not shown when nesting conditionals; for instance, the choice of early truth detection in loops can mean the difference between impressing the team, or losing nerd cred on bets and brags.

It is worth spending some time with Unicon profiling, and memory map visualization. http://www2.cs.arizona.edu/icon/docs/docs.htm

Get used to the feel of common case decision order for tests across a set of fields in an application. It takes practise when trying to shave branches off a decision tree (or to avoid computing completely new forests needlessly). There will likely be some hard to figure out fails when gaining wisdom in Unicon, but the language can handle extremely complicated data mashing, with flair; and should be used so, repeatedly.

```
prompt$ cd tests/bench
prompt$ make
prompt$ ./run-benchmark
```

A local pass came up looking like

```
prompt$ make
...
./generate
generating input files............done!

prompt$ ./run-benchmark

Times reported below reflect averages over three executions.
Expect 2-20 minutes for suite to run to completion.

Word Size  Main Memory  C Compiler  clock    OS
64 bit     7.272 GB     gcc 5.4.0   3.4 GHz  UNIX

CPU
4x AMD A10-5700 APU with Radeon(tm) HD Graphics

                                        Elapsed time h:m:s        |Concurrent |
benchmark                            |Sequential|   |Concurrent|   |Performance|
concord concord.dat                      3.213           N/A
deal 50000                               2.469           N/A
ipxref ipxref.dat                        1.345           N/A
queens 12                                3.554           N/A
rsg rsg.dat                              2.815           N/A
binary-trees 14                          5.172          7.736         0.668x
chameneos-redux 65000                      N/A          5.706
fannkuch 9                               3.601           N/A
fasta 250000                             3.174           N/A
k-nucleotide 150-thou.dat                5.153           N/A
mandelbrot 750                           13.877          7.662         1.811x
meteor-contest 600                       4.793           N/A
n-body 100000                            4.326           N/A
pidigits 7000                            2.903           N/A
regex-dna 700-thou.dat                   4.231          3.452         1.225x
reverse-complement 15-mil.dat            4.828           N/A
spectral-norm 300                        3.469           N/A
thread-ring 700000                         N/A          6.460
```

To compare (and take part) visit http://unicon.org/bench

If there are no results for your particular machine type and chipset on the Accumulated Results chart, *Clinton Jeffery* collects summaries with information on how to report them at

http://unicon.org/bench/resform.html

The `makefile` creates some fairly large test files, so you'll probably want to clean up after running the benchmark pass.

```
prompt$ make clean
prompt$ rm *-thou.dat *-mil.dat ipxref.dat
```

Unfortunately, `make clean` does not remove the benchmarking data files.

CHAPTER TWENTYFOUR

ICON PROGRAM LIBRARY

24.1 IPL

The Icon Program Library. A large collection of Icon programs/procedures and supporting data. Covers core utilities, like HTTP support in Icon, fun things, like games, and a plethora of short and long examples and samples.

https://www2.cs.arizona.edu/icon/library/ipl.htm

The IPL is Public Domain source code, and is included with the Unicon distribution. Still useful for Unicon development, many of the more critical utilities have been superseded by features built into Unicon proper. HTTP support, for instance is just an `"m"` class Messaging option when using the *open* statement. One of the beauties of Unicon, is Icon with Networking (one, of the beauties).

24.1.1 Exploring the IPL

Getting used to the IPL can be a little daunting at first, there are hundreds of programs, supporting data files and thousands of helpful procedures to take advantage of.

The Programming with Unicon book details the contents of the IPL in Appendix B.

The Arizona site link above also helps with figuring out all the useful little tidbits available.

Learning what is where in the IPL is similar to getting used to the support libraries available with many languages, like `PHP` and `Python`, or `libc` for that matter. When facing a problem, it can be worthwhile perusing the library. The procedures contained within can save time, effort and increase the odds of having correct code as the entries have been vetted by experts, and most edge cases are accounted for.

Aside from reading, and searching, and exploring, there is no easy way to get used to large libraries. It takes time and experience to build up the knowledge.

Start early. When first learning Unicon, it is well worth the extra minutes to look through the IPL directories. Early exposure might trigger a memory later on, when the task at hand requires an unfamiliar solution. Someone else may have already solved, or partially solved, the problem.

24.1.2 Useful procedures

This list of handy IPL procs is based on a personal bias. Your "top-ten" list will likely be different, based on your own needs and priorities.

Some of these IPL entries include multiple functions. Do not worry about bloat. The *ucode* linker will only include the procedures that are actually used from files with multiple procedures in the bytecode output.

core

Ralph Griswold carefully vetted two of the early Icon library amalgams. core.icn and graphics.icn.

For Unicon builds in late 2019, core.icn looks like:

```
############################################################################
#
#        File:     core.icn
#
#        Subject:  Procedures for general application
#
#        Author:   Gregg M. Townsend
#
#        Date:     August 4, 2000
#
############################################################################
#
#   This file is in the public domain.
#
############################################################################
#
#  Links to core modules of the basic part of the library, as defined
#  in the Icon Language book (3/e, p.179) and Graphics book (p.47).
#
############################################################################
#
#  Links: convert, datetime, factors, io, lists, math, numbers,
#     random, records, scan, sets, sort, strings, tables
#
############################################################################

link convert
link datetime
link factors
link io
link lists
link math
link numbers
link random
link records
link scan
link sets
link sort
link strings
link tables
```

Use this as an early entry point for IPL exploration. It's a fair amount of reading, with the variety of functions included in the linked files, but worth the time to get to know. Along with graphics.icn, if you lean to procedural based GUI programming.

```
############################################################################
#
#       File:     graphics.icn
#
#       Subject:  Procedures for graphics
#
#       Author:   Gregg M. Townsend
#
#       Date:     August 4, 2000
#
############################################################################
#
#   This file is in the public domain.
#
############################################################################
#
#  Links to core subset of graphics procedures.
#
############################################################################
#
#  Links: bevel, color, dialog, enqueue, gpxop, gpxlib,
#     vidgets, window, wopen
#
############################################################################

link bevel
link color
link dialog
link enqueue
link gpxop
link gpxlib
link vidgets        # basic set needed by Dialog() and Vset()
link window
link wopen
```

This is one of the GUIs this author grew up on. Graphic programming was done in Icon, or Tcl/Tk. Ahh, vbuttons.

Some of the entries in core and graphics are mentioned in more detail in this ode to the IPL. Along with the *Unicon Class Library*, the IPL is well worth getting used to. The IPL is kinda huge, and mind boggling at first, then it isn't, and you'll get to know where to look.

options

This is *thee* way to handle Unicon command line options. Easy to use and defacto standard with Unicon programming. *Most programs should support version and help display. Everyone needs a little help from time to time.*

```
procedure main(argv)
    opts := options(argv, "-h! -v!", errorproc)
    \opts["h"] & show_help() & return
    \opts["v"] & show_version() & return

    write("Non option arguments:")
    every write("\t", !argv)
end

procedure errorproc(s)
    write("Error in provided options:")
```

```
    write(s)
end

...
```

Option tags are user defined and can be single letter, `-h`, or long form, `-help` (for instance), with modifiers of:

```
!  no value required
:  string value required
+  integer value required
.  real value required
```

If the optstring spec is omitted, any single letter is assumed valid with no follow up data.

The null and non-null test operators can make for some very concise option handling source code. The returned table is key-value with the option (if present) as key, and any required follow up data as the value.

The optional `errorproc` defaults to displaying a message and stopping when invalid arguments are provided. The procedure is called with one argument: a string describing the error that occurred. After `errproc()` is called, `options()` immediately returns the outcome of `errproc()`, without processing further arguments.

An argument of `--` causes `options()` to stop parsing options, and leaves all subsequent command line data in the `argv` argument list.

The `argv`[1] list provided to `main()` will have all options and associated arguments removed from the list after the call to `options()`.

Unlike the GNU standard, long option names are single dash, not double dash when programming with the `options()` procedure.

ximage

Unicon has *image* built in, but only displays a type summary for most of the aggregate structures. `ximage()` is a routine, by Rob Alexander, that produces a *string* image of *x*, but also includes all elements of structured data, indented to aid in visualizing nested structures.

wrap

The `wrap()` procedure is a very handy way of displaying long lists of built up data. Initialize with `wrap()` (if required following a previous use) then `write(wrap(data, width))` inside loops, ending with a final `write(wrap())` to finish off any buffered data. `width` defaults to 80 character positions. `ipl/procs/wrap.icn`, `link wrap`.

```
#
# ipl-wrap.icn, demonstrate short string accumulated wrap
#
link wrap

# wrap printable ASCII with a width of 16 characters per line
procedure main()
    wrap()
    every write(wrap(!&ascii[33:-1], 16))
    write(wrap())
end
```

[1] `argv` is just an identifier name, use anything you are comfortable with, *but remember that args is a built in function, so is not a recommended choice.* `argv` is provided to `main` as a *List (arrays)* of strings in the same order as given on the command line.

```
prompt$ unicon -s ipl-wrap.icn -x
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

---

strings

The `strings.icn` entry in the IPL encapsulates a lot of idiomatic Unicon in small functions. `ipl/procs/strings.icn`, `link strings`.

- cat: concatenate strings
- charcnt: character count
- collate: string collation
- comb: character combinations
- compress: character compression
- coprefix: find common prefix of strings
- cosuffix: find common suffix of strings
- csort: lexically ordered characters
- decollate: string decollation
- deletec: delete characters
- deletep: delete by position
- deletes: delete string
- diffcnt: number of different characters
- extend: extend string
- fchars: characters in order of frequency
- interleave: interleave strings
- ispal: test for palindrome
- maxlen: maximum string length
- meander: meandering strings
- multicoll: collate strings in list
- ochars: first appearance unique characters
- odd_even: odd-even numerical string
- palins: palindromes
- permutes: generate string permutations
- pretrim: pre-trim string
- reflect: string reflection

- replace: string replacement
- replacem: multiple string replacement
- replc: replicate characters
- rotate: string rotation
- schars: lexical unique characters
- scramble: scramble string
- selectp: select characters
- slugs: generate s in chunks of size <= n
- starseq: closure sequence
- strcnt: substring count
- substrings: generate substrings
- transpose: transpose characters
- words: generate words from string

It is very much worth knowing about the many helper functions in `strings.icn`, and the source code gives good hints on idiomatic Unicon.

lists

List manipulation routines. `ipl/procs/lists.icn`, `link lists`.

- file2lst create list from lines in file
- imag2lst convert limage() output to list
- l_Bscan begin list scanning
- l_Escan end list scanning
- l_any any() for list scanning
- l_bal bal() for list scanning
- l_find find() for list scanning
- l_many many() for list scanning
- l_match match() for list scanning
- l_move move() for list scanning
- l_pos pos() for list scanning
- l_tab tab() for list scanning
- l_upto upto() for list scanning
- llayer interleave lists with layering
- lcompact compact sequence
- lclose close open palindrome
- lcomb list combinations
- ldecollate list decollation

- ldelete delete specified list elements
- ldupl list term duplication
- lequiv compare lists for equivalence
- levate elevate values
- lextend list extension
- lfliph list horizontal flip (reversal)
- lflipv list vertical flip
- limage list image
- lcollate generalized list collation
- lconstant test list for all terms equal
- lindex generate indices for items matching x
- linterl list interleaving
- llpad list padding at left
- lrunup list run up
- lrundown list run up
- lltrim list left trimming
- lmap list mapping
- lresidue list residue
- lpalin list palindrome
- lpermute list permutations
- lreflect list reflection
- lremvals remove values from list
- lrepl list replication
- lreverse list reverse
- lrotate list rotation
- lrpad list right padding
- lrtrim list right trimming
- lshift shift list terms
- lst2str convert list to string
- lswap list element swap
- lunique keep only unique list elements
- lmaxlen size of largest list entry
- lminlen size of smallest list entry
- sortkeys extract keys from sorted list
- sortvalues extract values from sorted list
- str2lst list from string

A few examples from this feature rich IPL entry:

```
#
# ipl-lists.icn, Demonstrate some of the lists utilities
#
link lists
procedure main()
    L := [1,2,3,4,5,4,3,2,1]
    # unadorned image
    write(limage(L))
    # list repl
    write(limage(lrepl(L[1:3], 3)))
    # list interweave
    write(limage(linterl(L, [6,7])))
    # list unique
    write(limage(lunique(L)))
end
```

```
prompt$ unicon -s ipl-lists.icn -x
[1,2,3,4,5,4,3,2,1]
[1,2,1,2,1,2]
[1,6,2,7,3,6,4,7,5,6,4,7,3,6,2,7,1,6]
[1,2,3,4,5]
```

---

numbers

Another multiple procedure IPL entry. The `numbers.icn` entry in the IPL encapsulates a lot of numeric Unicon in small functions. `ipl/procs/numbers.icn`, `link numbers`.

- adp additive digital persistence
- adr additive digital root
- amean arithmetic mean
- ceil ceiling
- commas insert commas in number
- decimal decimal expansion of rational
- decipos position decimal point
- digred sum digits of integer repeated to one digit
- digroot digital root
- digprod product of digits
- digsum sum of digits
- distseq generate low to high nonsequentially
- div real division
- fix format real number
- floor floor
- frn format real number

- gcd greatest common divisor
- gcdl greatest common divisor of list
- gmean geometric mean
- hmean harmonic mean
- large detect large integers
- lcm least common multiple
- lcml least common multiple of list
- mantissa mantissa (fractional part)
- max maximum value
- mdp multiplicative digital persistence
- mdr multiplicative digital root
- min minimum value
- mod1 modulus for 1-based integers
- npalins palindromic numbers
- qmean quadratic mean
- residue residue for j-based integers
- roman convert integer to Roman numeral
- round round real
- sigma synonym for digroot()
- sign sign
- spell spell integer in American English
- sum sum of numbers
- trunc truncate real
- unroman convert Roman numeral to integer

For example, `decipos` can come in handy for aligning display lists.

```
#
# dtor.icn, demonstrate degrees to radians
#
link numbers

# uses decipos from numbers, align decimal within field
procedure main()
    write("Degrees Radians")
    every r := 0.0 to 360.0 by 45.0 do
        write(decipos(r, 4, 8), decipos(dtor(r), 2, 20))
end
```

```
prompt$ unicon -s dtor.icn -x
Degrees Radians
  0.0   0.0
 45.0   0.7853981633974483
 90.0   1.570796326794897
```

```
135.0   2.356194490192345
180.0   3.141592653589793
225.0   3.926990816987241
270.0   4.71238898038469
315.0   5.497787143782138
360.0   6.283185307179586
```

Lisp

The IPL contains a small, yet fairly complete, Lisp interpreter. `ipl/progs/lisp.icn`.

```
#
# lisping.icn, Demonstrate the IPL Lisp interpreter
#
link lisp, fullimag
procedure main()
    initialize()
    preload()
    s := "(print (quote (1 2 3)))"
    every l := bstol(Map(s)) do { PRINT(result := [EVAL([l])]) }
    write("first result: ", fullimage(result))

    s := "(setq a (quote (1 2 3)))"
    every l := bstol(Map(s)) do { PRINT(result := [EVAL([l])]) }
    write("second result: ", fullimage(result))
end
```

The code above just prints a list.

Build some *icode* for lisping.

```
prompt$ unicon -s -c -DNOMAIN lisp.icn
```

And evaluate some Lisp

```
prompt$ unicon -s lisping.icn -x
(1 2 3)
NIL
first result: [[]]
(1 2 3)
second result: [["1","2","3"]]
```

24.2 Programming Corner

The Programming Corner was a feature of the Icon Newsletter which was published from 1979 through 2000, 60 issues.

https://www2.cs.arizona.edu/icon/inl/inl.htm

The Corner was a short piece of wisdom, or a puzzler, or other educational item, usually shining a light on core idioms for the language. All of this wisdom applies to Unicon as well.

https://www2.cs.arizona.edu/icon/progcorn.htm

This is information dating back to 1979, when the core language features in Icon were being worked out. There was also a desire to separate SNOBOL language features from Icon, and many of the articles attempt to explain how SNOBOL idioms may or may not apply to Icon/Unicon and how to watch out for the *SNOBOL Syndrome*.

One of the first Programming Corner items, was about the idiom shift from SNOBOL to Icon control flow.

The SNOBOL fragment `(P1 P2) | P3` may translate to Unicon as `(e1 & e2) | e3` or more clearly in most cases as `if e1 then e2 else e3`.

The Unicon way avoids some mutual evaluations, some potential backtracking and probably reads better to most programmers.

Todo

fill in more wisdoms

24.2.1 Newsletter Catalog

Thanks to David Gamey, there is a contents page for the Icon Newsletter issues.

```
INL1 Icon Newsletter #1
INL1 Ratfor implementation available for CDC 6000/Cyber and Decsystem-10
INL2 Icon Newsletter #2 - August 4, 1979
INL2 1. Version 1.3 of Icon
INL2 2. Feedback from Users
INL2 3. Implementation Issues
INL2 4. Portability Issues
INL2 5. An Implementation of Icon in C
INL2 6. Language Issues (and the SNOBOL4 syndrome)
INL2 7. Documentation
INL2 Acknowledgements
INL3 Icon Newsletter #3 - February 22, 1980
INL3 Perspective on Icon
INL3 Version 2 of Icon
INL3 Feedback
INL3 Portable Icon
INL3 The C Implementation of Icon
INL3 Programming Corner - scanning tab and move
INL3 Distribution Request Forms - Version 2.0 and Portable Icon
INL4 Icon Newsletter #4 - June 3, 1980
INL4 Icon for UNIX
INL4 Features of Version 3
INL4 Future Directions
INL4 Programming Corner - goal-directed evaluation and limiting backtracking
INL4 Whimsy
INL4 Publications
INL4 Document Request
INL4 Request for Version 3 of Icon
INL5 Icon Newsletter #5 -December 31, 1980
INL5 1. Version 2 of Icon
INL5 1.1 Status
INL5 1.2 Existing Implementations
INL5 1.3 Corrections to the Portable Implementation
INL5 2. Version 3 of Icon
INL5 3. Current Research
INL5 3.1 Generators and Control Structures
INL5 3.2 Generators in C
```

```
INL5 3.3 Pattern Matching in Icon
INL5 4. Other Icon Documents
INL5 5. Programming Corner - puzzles and posed questions
INL5 Acknowledgements
INL5 Request for Icon Documents
INL5 Portable Icon Distribution Request; Version 2.0
INL5 UNIX Icon Distribution Request; Version 3.2
INL6 Icon Newsletter #6 - May I. 1981
INL6 I. Portable Icon
INL6 1.1 Implementations
INL6 1.2 Word-Size Limitations
INL6 1.3 Updated Corrections to Version 2
INL6 2. The UNIX Implementation of Icon
INL6 2.1 Version 3
INL6 2.2 Version 4
INL6 3. Current Research
INL6 3.1 Sequences and Expression Evaluation
INL6 3.2 Models of String Pattern Matching
INL6 3.3 Generators in C
INL6 4. Programming Corner
INL6 4.1 An Idiom
INL6 4.2 Solutions to Questions Posed in Newsletter #5
INL6 Request for Icon Documents
INL7 Icon Newsletter #7 - August 4, 1981
INL7 1. Version 4 of Icon
INL7 2. Other Implementation News
INL7 3. Programming Corner - self reproducing program
INL7 References
INL7 Request for Icon Documents
INL 7 UNIX Icon Distribution Request; Version 4
INL8 Icon Newsletter #8 - November 30, 1981
INL8 1. Cg
INL8 2. Version 5 of Icon
INL8 3. Other Implementation News
INL8 3.1 Version 2.1 Implementation for PRIME Computers
INL8 3.2 Icon for the ONYX C8002
INL8 4. Icon Book
INL8 5. Programming Corner
INL8 Request for Icon Documents
INL8 Request for Cg/Version 5 Icon
INL9 Icon Newsletter #9 - August 22, 1982
INL9 Version 2 Implementation Information
INL9 Version 5 Implementation Information
INL9 Transporting the C Implementation of Icon
INL9 1. The Icon Book
INL9 Research Related to Icon
INL9 Icon Documents
INL9 Programming Corner - odd shuffle
INL9 Request for Icon Documents
INL9 Request for Version 2 Icon for the IBM 360/370 and VAX
INL10 Icon Newsletter #10 - November 8, 1982
INL10 Version 5 Icon for the VAX Operating under UNIX*
INL10 Version 5 Icon for the VAX Operating under VMS
INL10 Porting the C Implementation of Icon
INL10 The Icon Book
INL10 Electronic Mail
INL10 Programming Corner - result sequences
INL10 Request for Version 5 Icon for VAX/UNIX
```

```
INL11 Icon Newsletter #11 - March 8, 1983
INL11 Icon Book
INL11 Version 5 Implementation Information
INL11 Icon Program Library
INL11 Icon Documents
INL11 Programming Corner
INL11 Contacting the Icon Project
INL11 Request for Icon Documents
INL11 Request for Release 5g of Icon for UNIX
INL12 Icon Newsletter #12 - July 14, 1983
INL12 The Icon Compiler Versus the Interpreter
INL12 Icon Program Library
INL12 Transporting Icon to UNIX Environments
INL12 Version 5 of Icon for the Onyx C8002
INL12 Version 5.8 of Icon
INL12 Status of Version 5 of Icon for VAX/VMS
INL12 Icon Discussion Group
INL12 Version 2 Versus Version 5 of Icon
INL12 Recent Icon Documents
INL12 Programming Corner - N-Queens, N-Rooks, Scanning entire files, random
numbers
INL12 Request for Icon Documents
INL12 Request for Version 5.8 of Icon for UNIX
INL13 Icon Newsletter #13 - August 31, 1983
INL13 Version 5 of Icon for VMS
INL13 Programming Corner: Random Numbers
INL14 Icon Newsletter #14 - January 17, 1984
INL14 Survey
INL14 Programming Corner: Answers; Returning More than One Value from a
Procedure; Initial Assigned Values in Tables; Matching Expressions; Problems
with Dereferencing; Syntactic Pitfalls; Trivia Corner
INL14 Recent Icon Documents
INL14 Request for Icon Documents
INL15 Icon Newsletter #15 - June 7, 1984
INL15 1. Results from the Questionnaire
INL15 2. Version 5 Implementation News
INL15 3. Bugs in Version 5 VAX/VMS Icon
INL15 4. Work in Progress: Version 5.9, Sets in Icon; Production Icon;
Generators in Object-Oriented Languages
INL15 5. Use of Icon in Computer Science Courses
INL15 6. Portability of Version 5
INL15 7. Programming Corner: Assignment to Subscripted Strings; Trivia Corner;
Pitfalls;
INL15 8. Electronic Mail
INL15 9. The Icon Mailing List
INL15 10. New Documents
INL15 Request for Icon Documents
INL16 Icon Newsletter #16 - November 12, 1984
INL16 1. Implementations of Icon
INL16 Version 5.9 of UNIX Icon
INL16 Other Implementation News
INL16 2. Bug in Version 5 of Icon
INL16 3. Record Field References
INL16 4. Comments on Teaching Icon and the Icon Book
INL16 5. Rebus
INL16 6. Programming Corner
INL16 Chosing Programming Techniques in Icon
INL16 Different Ways of Looking at Things
```

```
INL16 7. Electronic Mail
INL16 8. New Documents
INL16 Request for Icon Documents
INL16 Version 5.9 UNIX Icon Distribution Request
INL16 Version 5.8 Eunice Icon Distribution Request
INL17 Icon Newsletter #17 - March 1, 1985
INL17 1. Icon Distribution Policy
INL17 2. Implementation News
INL17 Icon for the Sun Workstation
INL17 Icon for the AT&T 3B20
INL17 Version 5.9 of Icon for VAX/VMS
INL17 3. Contributions from Users
INL17 The Programming Language (Pascal-Icon)
INL17 Generators in Smalltalk
INL17 Logicon: An Integration of Prolog into Icon
INL17 An Application for Linguistic Analysis
INL17 4. Use of Icon in Computer Science Courses
INL17 5. Programming Corner: Anagramming;
INL17 Logicon: an Integration of Prolog into Icon
INL17 1. Integration of Prolog into Icon
INL17 1.1 Prolog Term Representation
INL17 1.2 External Interface
INL17 Entering Relations
INL17 2. Uses of Logicon
INL17 Bibliography
INL17 Version 5.9 UNIX Icon Distribution Request
INL17 Request for Version 5.9 of Icon for the AT&T 3B20
INL17 Request for Version 5.9 of Icon for VAX/VMS
INL18 Icon Newsletter #18 - April 23, 1985
INL18 1. Implementation News
INL18 Icon for the IBM PC under PC/IX
INL18 Icon for AT&T 3B2/3B5 and for UNIX/PC
INL18 Corrected Request Form for Version 5.9 VAX/VMS Icon
INL18 Another Implementation Project
INL18 2. An Icon Machine?
INL18 3. Programming Corner
INL18 4. New Documents
INL18 Request for Version 5.9 of Icon for VAX/VMS
INL18 Request for Version 5.9 of Icon for PC/IX
INL18 Request for Icon Documents
INL19 Icon Newsletter #19 - September 25, 1985
INL19 1. Implementation News
INL19 Version 5.9 of Icon for MS-DOS Systems
INL19 Version 5.9 of Icon for the WICAT
INL19 Version 5.10 of Icon for UNIX Systems
INL19 Version 5.10 of Icon for the UNIX-PC
INL19 The IIT Implementation of Icon
INL19 2. Implementation Book
INL19 3. Contribution from a User
INL19 4. New Documents
INL19 Request for Icon Documents
INL19 Request for Version 5.9 of Icon for MS-DOS
INL19 Request for Version 5.10 of Icon for UNIX
INL20 Icon Newsletter #20 - January 24, 1986
INL20 1. Implementation News
INL20 Version 5.10 of Icon for AT&T 3B Computers
INL20 Version 6 of Icon
INL20 2. The Icon Program Library for DOS Systems
```

```
INL20 3. Icon Workshop Anyone?
INL20 4. Availability of the Icon Book
INL20 5. New Payment Policy for Icon Materials
INL20 6. Programming Corner: String Scanning; A Programming Idiom; Teasers;
Other Exercises
INL20 7. New Documents
INL20 Acknowledgement
INL20 Icon Workshop
INL20 Request for Icon Documents
INL20 Request for Version 5.9 of Icon for DOS
INL21 Icon Newsletter #21 - June 10, 1986
INL21 1. Welcome, New Readers!
INL21 2. Implementation News
INL21 Version 6 of Icon
INL21 Icon for DOS
INL21 Version 5.10 on the Sun Workstation
INL21 3. Payment for Icon Materials
INL21 4. Status of the Icon Workshop
INL21 5. From our Mail
INL21 6. Programming Corner
INL21 Correction
INL21 Solutions to Previous Problems
INL21 Request for Icon Documents
INL21 Request for Version 6.0 of Icon for UNIX Systems
INL21 Request for Version 6.0 of Icon for VAX/VMS
INL21 Request for Version 5.9 of Icon for DOS
INL22 Icon Newsletter #22 - October 21,1986
INL22 1. Newsletter Subscriptions
INL22 2. Implementation News
INL22 MS-DOS
INL22 UNIX
INL22 Implementations in Progress
INL22 3. The Implementation Book
INL22 4. Electronic Access
INL22 Program Material via FTP
INL22 Electronic Bulletin Board at the University of Arizona
INL22 BIX
INL22 5. From our Mail
INL22 6. Programming Corner
INL22 A Simple Calculator
INL22 Acknowledgements
INL22 Ordering Information
INL22 Order Form
INL23 The Icon Newsletter - Number 23 - February 3,1987
INL23 Questionnaires
INL23 Status of the Newsletter
INL23 Implementation News
INL23 Icon for the Macintosh
INL23 Icon for the Atari ST
INL23 MS-DOS Icon
INL23 Porting Icon to Other Computers
INL23 The Implementation Book
INL23 Access to the Icon Project
INL23 Program Material via FTP
INL23 Electronic Bulletin Board
INL23 BIX
INL23 From our Mail
INL23 Programming Corner
```

```
INL23 A Program to Deal and Display Bridge Hands
INL23 Processing Command-Line Arguments
INL23 Upcoming in the Newsletter
INL23 Ordering Information
INL23 Order Form
INL24 The Icon Newsletter - Number 24 - June 13,1987
INL24 Welcome to New Subscribers
INL24 Tabulation of Questionnaires
INL24 Applications of Icon
INL24 Prototyping at AT&T Information Systems
INL24 Test Generation at Tartan Laboratories
INL24 Report from a Conference
INL24 Implementation News
INL24 New Implementations
INL24 Summary of Existing Implementations
INL24 Stand-Alone Icon for the Mac
INL24 Documents Related to Icon
INL24 Implementation Documentation
INL24 Technical Reports
INL24 TR 87-2 A Recursive Interpreter for Icon
INL24 TR 87-6 Programming in Icon; Part II - Programming with Co-Expressions
INL24 IPD29 Supplementary Information for the Implementation of Version 6 of
Icon, by Ralph E. Griswold.
INL24 Back Issues of the Icon Newsletter
INL24 From our Mail
INL24 What is the Icon Project?
INL24 Research in Progress
INL24 Implementing Generators and Goal-Directed Evaluation
INL24 Type Inference
INL24 Pattern Matching in Real Time
INL24 A New Language?
INL24 The Icon Program Library-
INL24 Programming Corner
INL24 Correction
INL24 Processing Command-Line Options
INL24 Efficient Programming in Icon
INL24 Puzzles and Such
INL24 Upcoming in the Newsletter
INL24 Our Army
INL24 A SNOBOL's Chance
INL24 Ordering Icon Material
INL24 What's Available
INL24 Program Material
INL24 Documentation
INL24 Order Form
INL25 Odds and Ends
INL25 The Icon Newsletter - No. 25 - November 1,1987
INL25 Odds and Ends
INL25 Subscriptions to the Newsletter
INL25 Use of Our Mailing List
INL25 Contacting the Icon Project
INL25 Reporting Problems
INL25 Implementation News
INL25 An Extension Interpreter for Icon
INL25 From Our Mail
INL25 Icon BBS at Arizona
INL25 Ordering Icon Books Outside the United States
INL25 A Brief History of Icon
```

```
INL25 The Icon Project (continued)
INL25 Language Corner
INL25 Failure
INL25 Programming Corner
INL25 Pattern Words
INL25 Environment Variables
INL25 Benchmarking Icon Expressions
INL25 Queens Never Die
INL25 Icon Electronic Clip-Art "Contest"
INL25 New Documents
INL25 IPD18, Benchmarking Icon Expressions
INL25 IPD41, Tabulating Expression Activity in Icon
INL25 Upcoming in the Newsletter
INL25 Ordering Icon Material
INL25 What's Available
INL25 Program Material
INL25 Icon for Personal Computers
INL25 Icon for Porting:
INL25 Documentation
INL25 Order Form
INL26 The Icon Newsletter - No. 26 - March 1,1988
INL26 Odds and Ends
INL26 The Icon Extension Interpreter
INL26 Icon Clip-Art Contest
INL26 Implementation News
INL26 Version 7 of Icon is Released
INL26 Update Policy
INL26 Status of the Icon Program Library
INL26 Icon for Prime Computers
INL26 ICEBOL3 in April
INL26 Revision of the Icon Language Book
INL26 Feedback
INL26 From Our Mail
INL26 In Support of Icon
INL26 A Brief History of Icon - Continued
INL26 Language Corner
INL26 The Null Value
INL26 Programming Corner
INL26 Timing Expressions
INL26 Storage Allocation
INL26 Clip-Art Credits
INL26 Ordering Icon Material
INL26 What's Available
INL26 Program Material
INL26 Order Form
INL27 The Icon Newsletter - No. 27 - June 11,1988
INL27 Logo!
INL27 New Icon Implementations
INL27 Commercial Support for Icon?
INL27 ICEBOL3
INL27 A Brief History of Icon - Concluded
INL27 From Our Mail
INL27 Bugs
INL27 Leap-Year Woes
INL27 String-Allocation "Botch"
INL27 Language Corner
INL27 Failure and Errors
INL27 Inside Icon
```

```
INL27 Clip-Art Credits
INL27 Ordering Icon Material
INL27 What's Available
INL27 Program Material
INL27 Order Form
INL28 The Icon Newsletter - No. 28 - October 15,1988
INL28 For New Readers
INL28 Odds and Ends
INL28 Correction
INL28 We're Flattered
INL28 Icon Workshop
INL28 Icon "Clip Art"
INL28 Implementation News
INL28 MS-DOS Icon for 386 PCs
INL28 MS-DOS Icon under Turbo C
INL28 XENIX V/386 Icon
INL28 Icon for the IBM 370 Architecture
INL28 Amiga Icon
INL28 Implementation Updates
INL28 From Our Mail
INL28 A Contribution from Users
INL28 Inside Icon
INL28 Bugs
INL28 Icon Benchmarks
INL28 Documents Related to Icon
INL28 Quick Reference Sheets for Icon
INL28 Clip Art Credits
INL28 Quick Reference Sheets for Icon
INL28 Icon Programming Language Reference Sheet
INL28 Ordering Icon Material
INL28 Order Form
INL29 The Icon Newsletter - No. 29 - February 14,1989
INL29 Implementation News
INL29 Icon for OS/2
INL29 Implementation Updates
INL29 Letter from an "Old Icon Hand"
INL29 A Contribution from Users (cont'd)
INL29 Run-Time Record Definition
INL29 From Our Mail
INL29 Programming Corner
INL29 Table Keys
INL29 Unique Values in a List
INL29 Data Backtracking
INL29 Demography
INL29 What's in the Works
INL29 Faculty Positions
INL29 ICEBOL4 in October
INL29 Art Credits
INL29 Ordering Icon Material
INL29 What's Available
INL29 Program Material
INL29 Documentation
INL29 Order Form
INL30 The Icon Newsletter - No. 30 - June 4,1989
INL30 Credit Card Orders
INL30 Icon Program Library
INL30 Implementation News
INL30 Version 7.5
```

```
INL30 Communicating with the Icon Project
INL30 Getting Material Electronically
INL30 Network File Transfer
INL30 Electronic Bulletin Board
INL30 Electronic Mail
INL30 Electronic Newsgroup
INL30 From Our Mail
INL30 The New Features
INL30 Using Object Icon
INL30 Programming Corner
INL30 Two-Way Tables
INL30 The ProIcon Group Announces First Language Release
INL30 Graphics Credits
INL30 Ordering Icon Material
INL31 The Icon Newsletter - No. 31 - September 15,1989
INL31 Price Increases
INL31 Implementation News
INL31 Icon for the IBM 370
INL31 Other Implementation News
INL31 Source Updates for MS-DOS
INL31 A Word of Thanks
INL31 Geographical Distribution of Newsletter Subscriptions
INL31 Improving the Performance of Sets and Tables in Icon
INL31 Introduction
INL31 Algorithm Overview
INL31 Hashing
INL31 Segment Handling
INL31 Changes to Operations
INL31 Reorganization and Element Generation
INL31 Performance Measurements
INL31 Conclusions
INL31 Icon Benchmarks
INL31 Benchmarks for Version 7.5 of Icon
INL31 Icon Version Numbering
INL31 Graphic Credit
INL31 Programming Corner
INL31 ProIcon Licenses
INL31 Ordering Icon Material
INL32 The Icon Newsletter - No. 32 - January 15,1990
INL32 Address Change
INL32 From Our Mail
INL32 ICEBOL4
INL32 Object-Oriented Icon
INL32 Motivation
INL32 Classes
INL32 Objects
INL32 Object Invocation
INL32 Inheritance
INL32 Multiple Inheritance
INL32 Invoking Superclass Operations INL32
INL32 Public Fields
INL32 Miscellany
INL32 Running Idol
INL32 Getting a Copy
INL32 Bugs
INL32 Language Corner
INL32 Returning from Procedures
INL32 Generators
```

```
INL32 Graphic Credits
INL32 Programming Corner
INL32 Correction
INL32 Cset Operations
INL32 Black Holes
INL32 Records
INL32 Idiomatic Icon
INL32 Trivia Corner
INL32 Ordering Icon Material
INL33 The Icon Newsletter - No. 33 - May 15,1990
INL33 Version 8 of Icon is Here!
INL33 New Language Features
INL33 Implementation Changes
INL33 Memory Monitoring
INL33 Available Implementations
INL33 The Icon Program Library
INL33 Thank You!
INL33 Books, Books
INL33 Second Edition of the Icon Language Book
INL33 Another Icon Book
INL33 Newsletter Change - Important
INL33 From Our Mail
INL33 Icon Documentation
INL33 Getting Icon Material By FTP
INL33 SNOBOL4
INL33 A Compiler for Icon
INL33 Overview
INL33 Compiler Organization
INL33 Programming Corner
INL33 Trivia
INL33 Scope
INL33 Graphic Credits
INL33 Ordering Icon Material
INL34 The Icon Newsletter - No. 34 - October 15,1990
INL34 Version 8 of Icon
INL34 Corrections to the Second Edition of the Icon Book
INL34 Newsletters
INL34 Book Prices
INL34 ICEBOL5
INL34 Second Icon Workshop
INL34 ProIcon Version 2.0
INL34 Update on the Icon Optimizing Compiler
INL34 The Icon Program Library
INL34 Subscribing to Library Updates
INL34 Contributions to the Icon Program Library
INL34 Example Library Program
INL34 Icon Program Library (continued)
INL34 FTP Access to Icon Executable Files
INL34 From Our Mail
INL34 An Oral History of Icon
INL34 An Icon Programming Environment
INL34 Bug in Version 8 of Icon
INL34 Ordering Icon Material
INL35 The Icon Newsletter - No. 35 - March 1,1991
INL35 Newsletters
INL35 The Icon Program Library
INL35 Icon in Your Pocket
INL35 Bug in Version 8 of MS-DOS Icon
```

```
INL35 Update on the Icon Optimizing Compiler
INL35 New Company Formed to Provide Commercial Support for Icon
INL35 ICEBOL5
INL35 ProIcon Version 2.0
INL35 Programming Corner
INL35 Syntax
INL35 A Prime Number Generator
INL35 From Our Mail
INL35 SNOBOL4 Corner
INL35 The End of an Era?
INL35 Ordering Icon Material
INL36 The Icon Newsletter - No. 36 - July 1,1991
INL36 For New Readers
INL36 The Icon Compiler
INL36 Advantages of the Compiler
INL36 Disadvantages of the Compiler
INL36 Using the Icon Compiler
INL36 Getting the Icon Compiler
INL36 Looking Ahead
INL36 Icon News Group
INL36 The Icon Analyst
INL36 X-Window Facilities for Icon
INL36 Getting Icon Material Via FTP
INL36 ICEBOL5
INL36 Icon from ISI
INL36 ISIcon Does Modules
INL36 Programming Corner
INL36 From Our Mail
INL36 SNOBOL4 Corner
INL36 Ordering Icon Material
INL37 The Icon Newsletter - No. 37 - November 1,1991
INL37 For New Readers
INL37 What's Going on with Icon?
INL37 Icon Compiler Documentation
INL37 Geographical Distribution of Subscriptions
INL37 Executable Files for the Interpreter
INL37 ISI Icon Release
INL37 ProIcon 2.0 Upgrade
INL37 Smaller Icode Files for UNIX
INL37 Programming Corner
INL37 From Our Mail
INL37 Programming Language Archives
INL37 Faculty Positions
INL37 Thanks from the Editors
INL37 Icon Seminar
INL37 Icon Program Library Update
INL37 Graphic Credits
INL37 Tweening
INL37 A Puzzle
INL37 Public-Domain 386 MS-DOS Icon
INL37 Ordering Icon Material
INL38 The Icon Newsletter - No. 38 - March 1,1992
INL38 Newsletter Subscriptions
INL38 Version 8.5 of Icon
INL38 New Macintosh Icon
INL38 More on UNIX Icode Files
INL38 ICEBOL6
INL38 Icon Applications
```

```
INL38 RBBS at the Icon Project
INL38 A Puzzle
INL38 Producing the Newsletter
INL38 ISIcon Release 1.0
INL38 Icon Program Library Updates
INL38 ProIcon Spring Sale
INL38 Graphic Credits
INL38 From Our Mail
INL38 Ordering Icon Material
INL38 Newsletter Subscription Renewal Form
INL39 The Icon Newsletter - No. 39 - August 15,1992
INL39 Feedback
INL39 New Implementations
INL39 The Icon Optimizing Compiler
INL39 Changes in Icon Distribution
INL39 Icon Via FTP
INL39 An Icon Debugger
INL39 Icon Auto-Stereogram
INL39 From Our Mail
INL39 ICEBOL6
INL39 Graphic Credits
INL39 Icon on CD-ROM
INL39 Icon Class Projects
INL39 Ordering Icon Material
INL40 The Icon Newsletter - No. 40 - December 21,1992
INL40 Reflections
INL40 New Implementations
INL40 HOPL-II
INL40 Icon Graphics
INL40 TEXT Technology
INL40 ICEBOL6
INL40 Acknowledgments
INL40 Icon Class Projects
INL40 Programming Corner
INL40 Icon on the NEC 9801
INL40 Ordering Icon Material
INL41 The Icon Newsletter - No. 41 - March 15,1993
INL41 New Icon Program Library
INL41 Supporting the Icon Project
INL41 In the Works
INL41 Moving?
INL41 FTP Files by Electronic Mail
INL41 ProIcon Price Reduction
INL41 Noun Stem Generation of Finnish
INL41 Programming Corner
INL41 Third Icon Workshop
INL41 Ordering Icon Material
INL42 The Icon Newsletter - No. 42 - July 15,1993
INL42 Version 8.10 of Icon
INL42 Exploring Natural Language Syntax
INL42 Programming Corner
INL42 Supporting the Icon Project
INL42 Icon Analyst Back Issues
INL42 From Our Mail
INL42 Ordering Icon Material
INL43 The Icon Newsletter - No. 43 - November 15,1993
INL43 Implementation News
INL43 An Icon-Based Parser Generator
```

```
INL43 Icon for Humanists Out of Print
INL43 Holiday Closing
INL43 From Our Mail
INL43 Icon in the Classroom
INL43 Graphics Credits
INL43 Ordering Icon Material
INL44 The Icon Newsletter - No. 44 - March 15, 1994
INL44 ProIcon Now in Public Domain
INL44 Version 9 of Icon
INL44 SNOBOL4 Corner
INL44 Graphics Programming Book
INL44 Thank You
INL44 Uploading Files
INL44 Language Archives
INL44 Icon Mugs
INL44 From Our Mail
INL44 Frequently Asked Questions
INL44 Ordering Icon Material
INL45 The Icon Newsletter - No. 45 - August 15, 1994
INL45 Version 9 of Icon
INL45 Update Subscriptions
INL45 The Icon Analyst
INL45 BYTE Article on Icon
INL45 Icon RBBS
INL45 FTP Files by Electronic Mail
INL45 Graphics Credits
INL45 Stereograms
INL45 Language Archives
INL45 Access to CBI Archives
INL45 IClip
INL45 Phasing Out 5.25" Floppies
INL45 From Our Mail
INL45 Ordering Icon Material
INL46 The Icon Newsletter - No. 46 - December 29, 1994
INL46 Version 9 of Icon
INL46 ProIcon Manual Reprint
INL46 Icon RBBS Discontinued
INL46 Getting Books About Icon
INL46 Vidgets
INL46 Graphics Book
INL46 Icon for Software Engineering
INL46 Credits
INL46 From Our Mail
INL46 Ordering Icon Material
INL47 The Icon Newsletter - No. 47 - June 15, 1995
INL47 New Area Code
INL47 Status Report
INL47 Icon Analyst
INL47 Diskette Distribution
INL47 Icon Program Library
INL47 Contributing Articles
INL47 Graphics Programming Course
INL48 The Icon Newsletter - No. 48 - November 28, 1995
INL48 Icon Documentation on the Web
INL48 Icon Newsletter Subscriptions
INL48 Status Report
INL48 News and Notes
INL48 Technical Support
```

```
INL48 Thank you
INL48 View from the Icon Project
INL49 The Icon Newsletter - No. 49 - April 1, 1996
INL49 New Editor
INL49 Icon Program Library
INL49 Icon on the Web
INL49 Programming Languages Book
INL49 Graphics Programming Course
INL50 The Icon Newsletter - No. 50 - August 1, 1996
INL50 Third Edition of the Icon Book
INL50 New Implementations of Icon
INL50 Icon for Chinese Computing
INL50 Teaching Icon
INL50 Book Sale!
INL50 Web Links
INL50 From Our Mail
INL51 The Icon Newsletter - No. 51 - December 1, 1996
INL51 Third Edition of the Icon Book
INL51 Graphics Programming Book
INL51 Version 9.3 of Icon
INL51 Version 9.3 of the Program Library
INL51 New MS-DOS Implementation
INL51 Icon in Java
INL51 Teaching Icon
INL51 Web Links
INL51 Chicon
INL52 The Icon Newsletter - No. 52 - April 1, 1997
INL52 Mail-Order Program Material
INL52 Teaching Icon
INL52 Web Links
INL52 Native Interface for Windows
INL52 Programming Language Handbook
INL52 From Our Mail
INL52 Knowledge Explorer
INL53 The Icon Newsletter - No. 53 - August 1, 1997
INL53 Icon in Java
INL53 Icon Documentation in Japanese
INL53 Handbook of Programming Languages
INL53 Icon Analyst Promotional Offer
INL53 Program Visualization Course
INL54 The Icon Newsletter - No. 54 - December 1, 1997
INL54 The Curse of DOS
INL54 New Icon Program Library Release
INL54 Status of the Graphics Book
INL54 Unicon
INL54 Chinese Icon Book
INL54 From Our Mail
INL54 Updated Icon Web Site
INL54 Icon Mirror Site
INL54 Using Icon to Spin the Web
INL54 An Introduction to Wi
INL55 The Icon Newsletter - No. 55 - April 1, 1998
INL55 Book Sale
INL55 Version 9.3.1 of Icon
INL55 Windows Icon
INL55 Graphics Programming Book
INL55 Help Wanted
INL55 Icon Documentation in Japanese
```

```
INL55 OS/2 Icon 9.3 with Graphics
INL55 Handbook of Programming Languages ..
INL56 The Icon Newsletter - No. 56 - August 1, 1998
INL56 Jcon
INL56 Graphics Programming Book
INL56 SL5 Re-Implementation Project
INL56 Macintosh Icon Implementation
INL56 Work in Progress
INL56 Thanks to Our Publisher
INL56 From Our Mail
INL57 The Icon Newsletter - No. 57 - December 1, 1998
INL57 Version 2 of Jcon
INL57 More Icon Books Coming
INL57 Database Connectivity
INL57 Whither Idol?
INL57 Chinese Book on Icon
INL57 Icon Club at Yahoo
INL57 On-Line Icon Book
INL57 Icon CD-ROM Sale
INL58 The Icon Newsletter - No. 58 - June 1, 1999
INL58 Newsletter
INL58 Early Icon Analysts On-Line
INL58 News from Clint Jeffery
INL58 Version 9.3.2 of Icon
INL58 Icon for BeOS
INL58 Graphics Programming Course
INL59 The Icon Newsletter - No. 59 - December 1, 1999
INL59 Icon Newsletter to Cease Publication
INL59 More Early Icon Analysts On-Line
INL59 Version 9.3.2 of Icon Released
INL59 Jcon News
INL59 New DOS Version of Icon
INL59 Minicon
INL60 The Icon Newsletter - No. 60 - June 1, 2000
INL60 Sixty and Out
INL60 USGS Map Viewer
INL60 Back Issues of the Icon Newsletter
INL60 Messaging Language Extension
INL60 Icon Program Library
```

Again, these can all be found at:

https://www2.cs.arizona.edu/icon/inl/inl.htm

CHAPTER

TWENTYFIVE

UNICON CLASS LIBRARY

25.1 UCL

The Unicon Class Library. A collection of Unicon programs, procedures, classes, methods, and supporting data. Includes the Unicon GUI classes by Robert Parlett, JSON support by Gigi Young, development tools, and many more useful extensions to core Unicon.

The library is also very useful for learning Unicon, with lots of well written Unicon source code.

The UCL source code included with the Unicon distribution is licensed under the Lesser GNU General Public License.

Like the *IPL*, knowing how to best leverage the Unicon Class Library can take some time. There are hundreds of supporting classes to take advantage of.

Find all the sources in the uni/lib directory of the Unicon source tree. Most of these are precompiled with an installation, easily included in applications with a simple *link* expression.

25.1.1 JSON

A contribution by Gigi Young and *Clinton Jeffery*, provides access to JSON handling.

Use `link json` to include the features. There are high level (thread safe) functions of `jtou()` and `utoj()`. These convert JSON to Unicon and Unicon to JSON data structures, respectively.

A technical report for Unicon JSON, UTR20, is hosted at http://unicon.org/utr/utr20.pdf

Quick sample, from the technical report:

```
#
# json-hello.icn, demonstrate the json.icn Unicon Class Library contribution.
# By Gigi Young and Clinton Jeffery.
#
link json
procedure main()
   # JSON string to Unicon table
```

```
   T := jtou("{\"To\": \"world\", \"Say\": \"Hello, \"}")
   write(T["Say"], T["To"])

   # Unicon table to JSON string
   TU := table()
   TU["one"] := 1
   TU["two"] := 2
   TU["list"] := [1,2,3]
   TU["table"] := table()
   write("Unicon table in JSON: ", utoj(TU))
end
```

```
prompt$ unicon -s json-trial.icn -x
Hello, world
Unicon table in JSON: {"two":2,"table":{},"one":1,"list":[1,2,3]}
```

CHAPTER

TWENTYSIX

USE CASE SCENARIOS

26.1 Scenarios

Discussing some of the obvious and less obvious use cases where Unicon is at home in providing solutions.

Unicon is a very high level general purpose programming language. In theory, being Turing Complete, any computational problem can be solved with Unicon code. There are areas where Unicon shines bright, and solutions fit the design goals, and areas where Unicon might be better off in a supporting role.

Of some of the more general programming use cases:

- Command line
- Desktop
- Web
- Mobile
- Back end services

Of these five main areas, Unicon would be a good first choice in all but the Mobile space. The Unicon ecosystem provides excellent support for command line, desktop, web and backroom service programming. As of version 13, programming with Unicon in the Mobile space would take extra effort to fit comfortably.

26.1.1 Experience

- on boarding
- educational materials
- community support
- contributing

This author is biased, and old(er), so this section of the document may seem more glowey than usual.

Getting started with Unicon is a very pleasant experience. From ready to go pre-built binaries for many platforms and a smooth source kit build, the on boarding experience with Unicon fares well relative to other programming environments. Educational materials are plentiful and free. The community is small, but very open to help and welcoming to new programmers. Contributions are appreciated and the principals will assist with efforts for inclusion in the product when appropriate.

26.1.2 Processing

Getting more specific:

- Text processing
- Numeric processing
- Data processing
- Transaction processing
- Image processing
- Sound processing

For these areas, Unicon would be a good first choice in all roles; but numeric processing may benefit most by using *Multilanguage* programming support for heavy number crunching. Unicon is quite good with numbers, large and small, but perhaps not the best choice when large scale numeric processing performance is the priority. Having said that, Unicon is very good at codifying extremely complex algorithms, so programming in Unicon can be highly beneficial when prototyping numeric recipes.

Where Unicon will shine is text processing. Manipulation and analysis of strings of text are a Unicon strength. I/O speeds are also highly competitive.

General data processing is a good fit for Unicon. The advanced data structure and I/O support will rarely be found lacking when it comes to sophisticated data processing tasks. Unicon lends itself very well to applications that can benefit from *concise complexity*. Solutions that seem out of practical reach when programming in other languages may benefit from goal-directed evaluations, implicit backtracking and associative data stores. Unicon syntax and semantics can place solutions within reach. *The overall design of core Icon with Unicon usability enhancements may actually increase the amount of complexity allowed in a system before overwhelming human understanding or codebase maintainability*.

Transaction processing is another area where Unicon can fill the role nicely. Ease of networking and concurrent programming support are builtin. There may not be many off-the-shelf Unicon transaction processing frameworks available, but rolling a custom solution is well within the strengths of Unicon programming.

Image processing is well supported as part of the venerable Icon cross-platform *Graphics* facilities, and Unicon 3D graphic extensions.

Sound file processing is also supported in Unicon, but there may be custom work required for some features that programmers find lacking, relative to other resource manipulation programs.

26.1.3 Development

From some of the more conceptual areas:

- prototyping
- testing
- debugging

- deployment
- documentation
- maintenance

Unicon should be a first choice in any prototype development, full-stop. Testing and deployment features may be less supported than in some other environments, but Unicon is no slouch in these areas either. Due to the *write less code* paradigm inherent in the very high level feature sets of Unicon and the extensive builtin monitoring facilities, debugging and maintenance task burdens are likely on par with, or helpfully superior to, most other programming environments. In terms of documentation, Unicon hits about the middle; some programming environments offer superior documentation systems, some offer a lesser experience.

26.1.4 Devices

Unicon is not overly rich in low level system programming features. Writing device drivers would not be a point of Unicon strength. Programs that control device drivers, directly via *loadfunc* loaded C code, or just as a presentation layer, can be coded in Unicon without fuss. Building system level hardware access would definitely require a lot of fussing using Unicon only source codes. Building C support layers, and integrating into the Unicon core would be middling to high levels of fussy.

26.1.5 Sciences

Unicon offers up all the computational facilities a scientist or engineer may require. At this point though, the amount of ready built library support may be lacking in comparison to other programming tools.

26.1.6 Mathematics

Eminently suitable.

26.1.7 Finance

Unlimited precision integers can be used to good effect for fixed point decimal calculations. Banks like decimal arithmetic, and Unicon can help keep track of dollars and cents, given a few support routines that take advantage of unlimited precision integers.

CHAPTER

TWENTYSEVEN

PROGRAMS

27.1 Sample programs and integrations

For lack of a better chapter name, this part of the docset is miscellaneous sample programs.

27.1.1 S-Lang

An example of embedding an S-Lang interpreter. S-Lang programs, as Unicon strings, are evaluated, and the last S-Lang result is passed back to Unicon.

Allowed return types:

- *Integer*
- *Real numbers*
- *String*
- *List (arrays)* (from S-Lang array, single dimension, cast to `double`)

S-Lang, by John Davis. http://www.jedsoft.org/slang/

Note: The `mkRlist` function in `ipl/cfuncs/icall.h` had the wrong prototype prior to Revision 4501 of the Unicon sources. Was `int x[]`, needs to be `double x[]`.

```
-word mkRlist(int x[], int n);
+word mkRlist(double x[], int n);
```

Already fixed, thanks to *Jafar Al-Gharaibeh*.

Note: Also be aware that some of the memory management in `slang.c` may be erroneous. Not for production use if you see this note.

Here is the `slang` *loadfunc* C function:

```
/*
 Embed an S-Lang interpreter in a Unicon loadfunc extension
 tectonics: gcc -o slang.so -shared -fpic slang.c -lslang
*/

#include <stdio.h>
#include <slang.h>
#include "icall.h"

/*
 slangEval, run S-Lang code or load filename
 Init S-Lang if necessary
 Then load from or evaluate a string argv[1]
 The last result stacked by S-Lang is returned to Unicon
 Integer, Double, String and Array as List values allowed
*/
int
slangEval(int argc, descriptor *argv, int fromfile)
{
    static int slang_loaded = 0;

    int tos;
    int i, iv;
    double r;
    char *s, *slast = NULL;
    /* Limit to single dimension arrays for this version */
    listblock *list;
    SLang_Array_Type *at;
    SLindex_Type ind;

    /* load slang, and all intrinsics */
    if (!slang_loaded) {
        if (-1 == SLang_init_all()) {
            /* Program malfunction */
#ifdef DEBUG
            fprintf(stderr, "Can't initialize S-Lang\n");
#endif
            Error(500);
        } else {
            slang_loaded = 1;
        }
    }

    /* ensure argv[1] is a string */
    ArgString(1)

    if (fromfile) {
        /* evaluate filename in argv[1] */
        if (-1 == SLang_load_file(StringVal(argv[1]))) {
            SLang_restart(1);
            SLang_set_error(0);

            /* report invalid procedure type error to Unicon */
            Error(178);
        }
     } else {
        /* evaluate argv[1] */
```

```
        if (-1 == SLang_load_string(StringVal(argv[1]))) {
            /* Reset S-Lang to allow later code attempts */
            SLang_restart(1);
            SLang_set_error(0);

            /* report invalid procedure type error to Unicon */
            Error(178);
        }
    }

    /* Unicon result will be last S-Lang expression */
    tos = SLang_peek_at_stack();
    switch (tos) {
        case SLANG_INT_TYPE:
            /* return an integer to Unicon */
            SLang_pop_integer(&i);
            RetInteger(i);
            break;
        case SLANG_DOUBLE_TYPE:
            /* return a real to Unicon */
            SLang_pop_double(&r);
            RetReal(r);
            break;
        case SLANG_STRING_TYPE:
            /* return an allocated string to Unicon */
            /* memory allocation strategy; previous string is freed */
            if (slast)  SLfree(slast);
            SLpop_string(&s);
            slast = s;
            RetString(s);
            break;
        case SLANG_ARRAY_TYPE:
            /* return an array as a Unicon list */
            if (-1 == SLang_pop_array_of_type(&at, SLANG_DOUBLE_TYPE)) {
                /* report malfuntion */
                Error(500);
            }
#ifdef DEBUG
            if (at->num_dims != 1) {
                /* warn about flattening array */
                fprintf(stderr, "S-Lang array flattened to one dimension\n");
            }
#endif
            double *doubles = malloc(sizeof(double) * at->num_elements);
            for (i = 0; i < at->num_elements; i++) {
                (void) SLang_get_array_element(at, &i, &r);
                doubles[i] = r;
            }
            /*
             mkRlist was defined as (int [], n) now (double [], n)
            */
            list = mkRlist(doubles, at->num_elements);

            /* clean up the temporary array*/
            free(doubles);

            RetList(list);
            break;
```

```
        default:
#ifdef DEBUG
            fprintf(stderr, "Unsupported S-Lang datatype %d\n", tos);
#endif
            /* report invalid value error to Unicon */
            Error(205);
    }
    return 0;
}

/*
 input string is a filename
 Usage from Unicon
     slangfile = loadfunc("./slang.so", "slangFile")
     x := slangfile("slang.sl")
*/
int
slangFile(int argc, descriptor *argv)
{
    int result;
    result = slangEval(argc, argv, 1);
    return result;
}

/*
 input string is S-Lang code
 Usage from Unicon
     slang = loadfunc("./slang.so", "slang")
     x := slang("S-Lang statements;")
*/
int
slang(int argc, descriptor *argv)
{
    int result;
    result = slangEval(argc, argv, 0);
    return result;
}
```

A Unicon test head:

```
#
# slang.icn, load a S-Lang interpreter, and evaluate some statements
#
# tectonics: gcc -o slang.so -shared -fpic slang.c -lslang
link ximage
procedure main()
    # embed the interpreter
    slang := loadfunc("./slang.so", "slang")

    # return a computed variable, sum of list
    code := "variable slsum = sum([0,1,2,3,4,5,6,7,8,9]);_
             slsum;"
    result := slang(code)
    write("Unicon sum: ", result)

    # return value is from S-Lang printf (bytes written)
    code := "printf(\"S-Lang: %f\\n\", slsum);"
    write("Unicon printf length: ", slang(code))
```

```
# S-Lang IO mix in
code := "printf(\"S-Lang: %s = %f and %s = %f\\n\",_
       \"hypot([3,4])\", hypot([3,4]),_
       \"sumsq([3,4])\", sumsq([3,4]));"
write("Unicon printf length: ", slang(code))

# 3D vector length
code := "variable A = [3,4,5]; hypot(A);"
write("Unicon hypot([3,4,5]): ", slang(code))

# try some strings, last one created will stay allocated
code := "\"abc\";"
write("Unicon abc: ", slang(code))
code := "\"def\";"
write("Unicon def: ", slang(code))

# Pass an array, returned as a list of Real
code := "[1, 2.2, 3, [4, 5, [6, 7], 8], 9.9];"
write("Unicon from ", code)
L := slang(code)
writes("Unicon (array flattened) ")
every i := !L do writes(i, " ")
write()

# Cummulative summation
code := "cumsum([1.1, 2.2, 3.3, 4.4]);"
L := slang(code)
writes("Unicon from ", code, ": ")
every i := !L do writes(i, " ")
write()

# try a small S-Lang program
code := "variable t, i; t = 0; for (i = 0; i < 10; i++) t += i; t;"
write("Unicon from ", code, ": ", slang(code))


# Exercise S-Lang load file
write()
write("Unicon run code from file slang.sl")
slangfile := loadfunc("./slang.so", "slangFile")
file := "slang.sl"

# show the file
cf := open(file, "r") | write("No ", file, " for test")
write("####")
while write(read(cf))
close(cf)
write("####")

# run the file
L := slangfile(file)
writes("Unicon from ", file, ": ")
every i := !L do writes(i, " ")
write()


# convert an error to failure
```

```
    write()
    write("Unicon convert S-Lang error to failure")
    &error := 1
    code := "[1, 2, \"abc\"];"
    write("Unicon trying: ", code)
    slang(code)
    write("Unicon S-Lang &errornumber: ", &errornumber)

    # and an abend
    write()
    write("Unicon abend on S-Lang divide by zero")
    code := "1/0"
    slang(code)
end
```

And a flying carpet run to see how things go:

```
prompt$ gcc -o slang.so -shared -fpic slang.c -lslang
```

Sample run ends in a purposeful error demonstration:

```
prompt$ unicon -s slang.icn -x
Unicon sum: 45.0
S-Lang: 45.000000
Unicon printf length: 18
S-Lang: hypot([3,4]) = 5.000000 and sumsq([3,4]) = 25.000000
Unicon printf length: 61
Unicon hypot([3,4,5]): 7.071067811865476
Unicon abc: abc
Unicon def: def
Unicon from [1, 2.2, 3, [4, 5, [6, 7], 8], 9.9];
Unicon (array flattened) 1.0 2.2 3.0 4.0 5.0 6.0 7.0 8.0 9.9
Unicon from cumsum([1.1, 2.2, 3.3, 4.4]);: 1.1 3.3 6.6 11.0
Unicon from variable t, i; t = 0; for (i = 0; i < 10; i++) t += i; t;: 45

Unicon run code from file slang.sl
####
%
% slang.sl, S-Lang file loaded from Unicon
%
% Unicon test program expects a list result
%
% Date: August 2016
% Modified: 2016-08-30/10:15-0400
%
define factorial();   % declare, for recursion

define factorial(n)
{
    if (n < 2) return 1;
    return n * factorial(n - 1);
}

variable list=[factorial(7),factorial(8),factorial(9)];
list;
####
Unicon from slang.sl: 5040.0 40320.0 362880.0
```

```
Unicon convert S-Lang error to failure
Unicon trying: [1, 2, "abc"];
Unable to typecast Integer_Type to String_Type
***string***:1:<top-level>:Type Mismatch
Unicon S-Lang &errornumber: 178

Unicon abend on S-Lang divide by zero
Divide by Zero
***string***:1:<top-level>:Divide by Zero

Run-time error 178
File slang.icn; Line 95

Traceback:
   main()
   slang("1/0") from line 95 in slang.icn
```

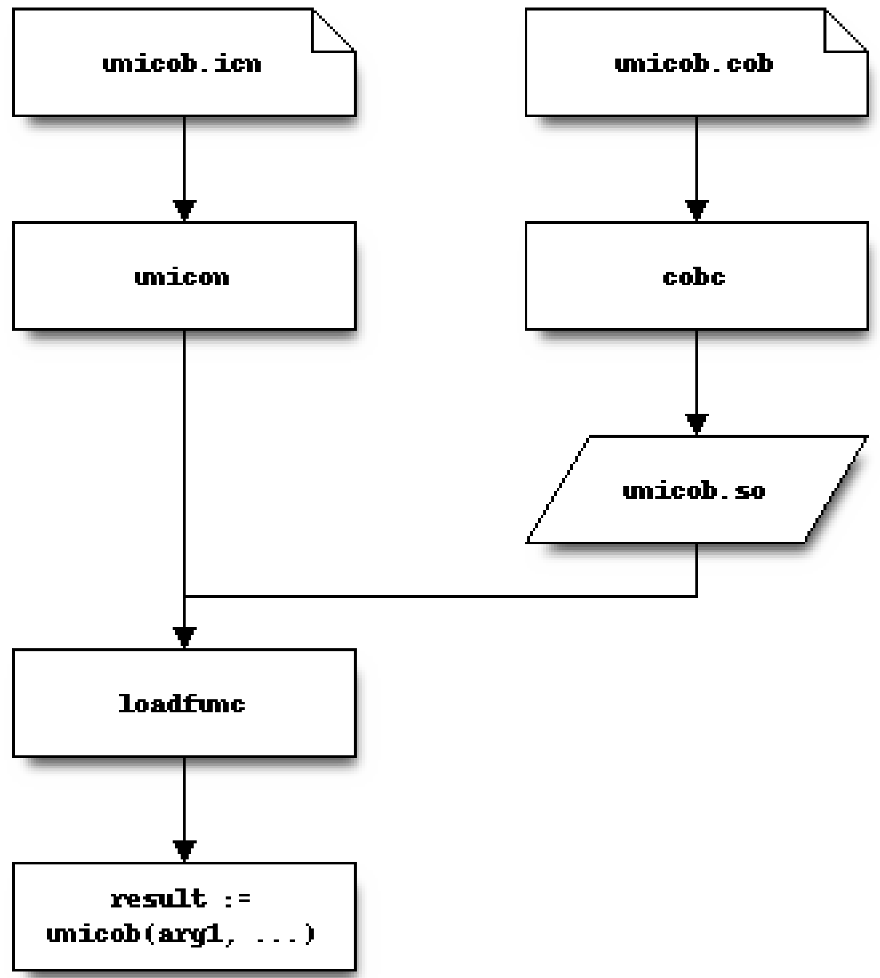
And Unicon can use S-Lang scripts whenever necessary.

27.1.2 COBOL

An example of embedding a COBOL module. First pass is simply seeing if integers make into the COBOL runtime.

GnuCOBOL is a free software COBOL compiler; part of the GNU project, copyright Free Software Foundation. https://sourceforge.net/projects/open-cobol/

GnuCOBOL



Note: This is first step trial code

The loaded COBOL function, `unicob`:

```
      *> Unicon interfacing with COBOL
       identification division.
       program-id. unicob.

      *> tectonics: cobc -m -fimplicit-init unicob.cob
      *> In Unicon: unicob := loadfunc("./unicob.so", "unicob")

       environment division.
       configuration section.
       repository.
           function all intrinsic.
```

```
       data division.
       working-storage section.
       01 actual usage binary-long.
       01 arguments based.
          05 args occurs 1 to 10 times depending on actual.
             10 dword usage binary-double unsigned.
             10 vword usage binary-double.
       linkage section.
       01 argc usage binary-long.
       01 argv usage pointer.

       procedure division using by value argc, argv.
       sample-main.
       display argc, ", ", argv

      *> if there is a null argv, report program malfunction
       if argv equal null then
           move 500 to return-code
           goback
       end-if

      *> argc needs one extra allocation to account for zeroth
       add 1 to argc giving actual
       set address of arguments to argv

      *> Let's see some integers
       perform varying tally from 1 by 1 until tally > actual
           display dword(tally), ", ", vword(tally)
       end-perform

      *> initial trickery to get a result to Unicon
       move dword(2) to dword(1)
       compute vword(1) = vword(2) * 6

      *> the "C" function returns 0 on success
       move 0 to return-code
       goback.
       end program unicob.
```

A test head:

```
#
# unicob.icn, load a COBOL module and show some integers
#
# tectonics: cobc -m -fimplicit-init unicob.cob
procedure main()
    # embed some COBOL
    unicob := loadfunc("./unicob.so", "unicob")
    result := unicob(7, 8, 9)
    write("unicob completed with ", result)
end
```

And a flying carpet run to see how things go:

```
prompt$ cobc -m -w -fimplicit-init unicob-v1.cob
```

```
prompt$ unicon -s unicob-v1.icn -x
+0000000004 arguments
```

```
&null       :
integer     : +00000000000000000007
integer     : +00000000000000000008
integer     : +00000000000000000009
unicob completed with 42
```

Seems to work ok. *Nerd dancing ensues, with a couple of "Oh, yeah, uh huh"s thrown in.*

Step 2

This is still fairly experimental code. A little bit of `icall.h` ported in, with support of more datatypes than simple integers.

```
      *> Unicon interfacing with COBOL
       identification division.
       program-id. unicob.

      *> tectonics: cobc -m -fimplicit-init unicob.cob
      *> In Unicon: unicob := loadfunc("./unicob.so", "unicob")
      *>   result := unicob(integer, real, or strin, ...)

       environment division.
       configuration section.
       repository.
           function all intrinsic.

       data division.
       working-storage section.
       01 actual usage binary-long.

       01 arguments based.
          05 args occurs 1 to 96 times depending on actual.
             10 dword usage binary-double unsigned.
             10 vword usage binary-double.

       01 unicon-int usage binary-c-long based.
       01 unicon-real usage float-long based.
       01 unicon-string usage pointer based.
       01 cobol-buffer pic x(8192) based.
       01 cobol-string pic x(8192).

      *> If DESCRIPTOR-DOUBLE not found in environment, default to set
       >>DEFINE DESCRIPTOR-DOUBLE PARAMETER
       >>IF DESCRIPTOR-DOUBLE IS NOT DEFINED
       >>DEFINE DESCRIPTOR-DOUBLE 1
       >>END-IF

       >>IF P64 IS SET
       01 FLAG-NOT-STRING constant as H"8000000000000000".
       01 FLAG-VARIABLE   constant as H"4000000000000000".
       01 FLAG-POINTER    constant as H"2000000000000000".
       01 FLAG-TYPECODE   constant as H"1000000000000000".
       01 DESCRIPTOR-TYPE constant as H"A000000000000000".
       01 DESCRIPTOR-NULL constant as H"A000000000000000".
       01 DESCRIPTOR-INT  constant as H"A000000000000001".
       >>IF DESCRIPTOR-DOUBLE IS DEFINED
```

```
       01 DESCRIPTOR-REAL constant as H"A000000000000003".
       >>ELSE
       01 DESCRIPTOR-REAL constant as H"B000000000000003".
       >>END-IF
       >>ELSE  *> not 64 bit
       01 FLAG-NOT-STRING constant as H"80000000".
       01 FLAG-VARIABLE   constant as H"40000000".
       01 FLAG-POINTER    constant as H"20000000".
       01 FLAG-TYPECODE   constant as H"10000000".
       01 DESCRIPTOR-TYPE constant as H"A0000000".
       01 DESCRIPTOR-NULL constant as H"A0000000".
       01 DESCRIPTOR-INT  constant as H"A0000001".
       >>IF DESCRIPTOR-DOUBLE IS DEFINED
       01 DESCRIPTOR-REAL constant as H"A0000003".
       >>ELSE
       01 DESCRIPTOR-REAL constant as H"B0000003".
       >>END-IF
       >>END-IF

      *> take argc int, argv array pointer
       linkage section.
       01 argc usage binary-long.
       01 argv usage pointer.

       procedure division using by value argc, argv.
       unicob-main.

      *> if there is a null argv, report program malfunction
       if argv equal null or argc less than 1 then
           move 500 to return-code
           goback
       end-if

      *> argc needs one extra allocation to account for zeroth
       add 1 to argc giving actual
       set address of arguments to argv
       display actual " arguments"
       display space

      *> Let's see the arguments (including current &null result slot)
       perform varying tally from 1 by 1 until tally > actual
           *> display "Arg: " tally " = " dword(tally), ", ", vword(tally)
           evaluate dword(tally)
              when equal DESCRIPTOR-NULL
                  display "&null      :"
              when equal DESCRIPTOR-INT
                  perform show-integer
              when equal DESCRIPTOR-REAL
                  perform show-real
              when less than FLAG-NOT-STRING
                  perform show-string
              when other
                  display "unsupported: type code is " dword(tally)
           end-evaluate
       end-perform

      *> send back the universal answer
       move DESCRIPTOR-INT to dword(1)
```

```
       move 42 to vword(1)

      *> the loadfunc function returns 0 on success
       move 0 to return-code
       goback.

      *> ****************
       show-integer.
       call "cnv_c_int" using args(tally) args(tally)
       display "integer    : " vword(tally)
       .

       show-real.
       call "cnv_c_dbl" using args(tally) args(tally)
       set address of unicon-real to address of vword(tally)
       display "float-long : " unicon-real
       .

       show-string.
       call "cnv_c_str" using args(tally) args(tally)
       set address of unicon-string to address of vword(tally)
       set address of cobol-buffer to unicon-string
       string cobol-buffer delimited by low-value into cobol-string
       *> The length is in dword(tally)
       display 'string     : "' trim(cobol-string) '"'
       .

       end program unicob.
```

Adding to the test head:

```
#
# unicob.icn, load a COBOL module and show some integers
#
# tectonics: cobc -m -fimplicit-init unicob.cob
procedure main()
    # embed some COBOL
    unicob := loadfunc("./unicob.so", "unicob")
    result := unicob(7, 8, 9, &phi, [], "Unicon and COBOL, together at last")
    write()
    write("Unicon     : unicob completed with ", result)
end
```

And a fly by to check out the new datatype support:

```
prompt$ cobc -m -w -fimplicit-init unicob.cob
```

```
prompt$ unicon -s unicob.icn -x
+0000000007 arguments

&null      :
integer    : +0000000000000000007
integer    : +0000000000000000008
integer    : +0000000000000000009
float-long : 1.618033988749895
unsupported: type code is 12682136550675316744
string     : "Unicon and COBOL, together at last"
```

```
Unicon      : unicob completed with 42
```

So, yeah, Unicon and COBOL; might come in handy.

There are a lot more details about GnuCOBOL at http://open-cobol.sourceforge.net/faq/index.html

---

27.1.3 Duktape

A Javascript engine. Another exploratory trial.

Duktape is hosted at http://duktape.org You will need the `.c` and `.h` files from the `src/` directory from the distribution. This test uses version 1.5.1. http://duktape.org/duktape-1.5.1.tar.xz

With Duktape, you simply include the `.c` files in a build. In this case, `uniduk.so` is built with `uniduk.c` and `dukctape.c`.

The C sample for `loadfunc`. `uniduk-v1.c`.

```
/*
 uniduk-v1.c, first trial, integrate a Javascript engine in Unicon
 tectonics: gcc -std=c99 -o uniduk.so -shared -fpic uniduk-v1.c duktape.c -lm
*/

#include <stdio.h>
#include "duktape.h"
#include "icall.h"

int
uniduk(int argc, descriptor *argv)
{

    duk_context *ctx = duk_create_heap_default();
    duk_eval_string(ctx, argv[1].vword.sptr);
    duk_destroy_heap(ctx);
    argv[0].dword = D_Integer;
    argv[0].vword.integr = 42;
    return 0;
}
```

The sample Unicon file to load and test the engine, `uniduk-v1.icn`.

```
#
# uniduk.icn, load the Duktape ECMAScript engine
#
# tectonics: gcc -std=c99 -o uniduk.so -shared -fpic uniduk.c duktape.c
procedure main()
    # embed some Duktape
    uniduk := loadfunc("./uniduk.so", "uniduk")
    result := uniduk("print('Hello, world');")
    write("Unicon: uniduk completed with ", result)
end
```

And a test run

```
prompt$ gcc -std=c99 -o uniduk.so -shared -fpic uniduk-v1.c duktape.c
```

```
prompt$ unicon -s uniduk-v1.icn -x
Hello, world
Unicon: uniduk completed with 42
```

And Duktape Javascript step 1 has been taken.

Second step

Todo

extend this further to handle more datatypes

The extended C sample for `loadfunc`. `uniduk.c`.

```
/*
 uniduk.c, integrate a Javascript engine in Unicon
 tectonics: gcc -std=c99 -o uniduk.so -shared -fpic uniduk.c duktape.c -lm
*/

#include <stdio.h>
#include "duktape.h"
#include "icall.h"

/*
 dukeval, evaluate a Javascript string or from file
*/
static duk_context *unictx;
static int duk_loaded = 0;

int
dukeval(int argc, descriptor *argv, int fromfile)
{
    /* Need a string argument */
    if (argc < 1) Error(103);

    if (!duk_loaded) {
        unictx = duk_create_heap_default();
        /* if bad init, report program malfunction */
        if (!unictx)  Error(500);
        duk_loaded = 1;
    }

    /* argument is either a filename or code string */
    ArgString(1);

    if (fromfile) {
        duk_eval_file(unictx, StringVal(argv[1]));
    } else {
        duk_eval_string(unictx, StringVal(argv[1]));
    }
    duk_int_t typ = duk_get_type(unictx, -1);
    switch (typ) {
        case DUK_TYPE_NONE:
```

```
        case DUK_TYPE_UNDEFINED:
            RetNull();
            break;
        case DUK_TYPE_NULL:
            duk_pop(unictx);
            RetNull();
            break;
        case DUK_TYPE_NUMBER:
        case DUK_TYPE_BOOLEAN:
            RetReal(duk_get_number(unictx, -1));
            break;
        case DUK_TYPE_STRING:
            RetConstString((char *)duk_get_string(unictx, -1));
            break;
        default:
            fprintf(stderr, "Unsupported Duk type: %d\n", typ);
            Error(178);
            break;
    }
    return 0;
}

/*
 uniduk, a Unicon loadfunc function
 Usage from Unicon
    uniduk := loadfunc("./uniduk.so", "uniduk")
    result := uniduk("print('Hello'); var r = 7 * 6;")
*/
int
uniduk(int argc, descriptor *argv)
{
    int result;
    result = dukeval(argc, argv, 0);
    return result;
}

/*
 unidukFile, load Duktape code from file
 Usage from Unicon
    unidukfile := loadfunc("./uniduk.so", "unidukFile")
    result := unidukfile("uniduk.js")
*/
int
unidukFile(int argc, descriptor *argv)
{
    int result;
    result = dukeval(argc, argv, 1);
    return result;
}

/*
 unidukDone, a Unicon loadfunc function for Duktape rundown
 Usage from Unicon
    unidukdone := loadfunc("./uniduk.so", "unidukDone")
    result := unidukDone()
 Unicon result is &null, by nature of not being set
*/
int
```

```
unidukDone(int argc, descriptor *argv)
{
    duk_destroy_heap(unictx);
    duk_loaded = 0;
    return 0;
}
```

The sample Unicon file to load and test the engine, `uniduk.icn`.

```
#
# uniduk.icn, load the Duktape ECMAScript engine
#
# tectonics: gcc -std=c99 -o uniduk.so -shared -fpic uniduk.c duktape.c
#
procedure main()
    # embed some Duktape ECMAScript
    uniduk := loadfunc("./uniduk.so", "uniduk")
    unidukfile := loadfunc("./uniduk.so", "unidukFile")
    unidukdone := loadfunc("./uniduk.so", "unidukDone")

    # numbers
    code := "1 + 2;"
    write("Attempt: ", code)
    result := uniduk(code)
    write("Unicon: uniduk completed with ", result)

    # no result, but side effect
    code := "print('Duktape print'); var r = 7 * 6;"
    write("Attempt: ", code)
    result := uniduk(code)
    write("Unicon: uniduk completed with ", result)

    # var r, number, set from previous script
    code := "r;"
    write("Attempt: ", code)
    result := uniduk(code)
    write("Unicon: uniduk completed with ", result)

    # string
    code := "'abc';"
    write("Attempt: ", code)
    result := uniduk(code)
    write("Unicon: uniduk completed with ", result)

    # JSON (Duktape custom JX format, readable JSON)
    code := "var obj = {foo: 0/0, bar: [1, undefined, 3]};_
             Duktape.enc('jx', obj);"
    write("Attempt: ", code)
    result := uniduk(code)
    write("Unicon: uniduk completed with ", result)

    # evaluate a test script from file
    filename := "uniduk.js"
    write("Attempt: ", filename)
    result := unidukfile(filename)

    # close up
    write("Unicon: Unload Duktape")
```

```
    uniduk done()
end
```

```
// fib.js
function fib(n) {
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    return fib(n-1) + fib(n-2);
}

function test() {
    var res = [];
    for (i = 0; i < 20; i++) {
        res.push(fib(i));
    }
    print(res.join(' '));
}

test();
```

And a test run

```
prompt$ gcc -std=c99 -o uniduk.so -shared -fpic uniduk.c duktape.c
```

```
prompt$ unicon -s uniduk.icn -x
Attempt: 1 + 2;
Unicon: uniduk completed with 3.0
Attempt: print('Duktape print'); var r = 7 * 6;
Duktape print
Unicon: uniduk completed with
Attempt: r;
Unicon: uniduk completed with 42.0
Attempt: 'abc';
Unicon: uniduk completed with abc
Attempt: var obj = {foo: 0/0, bar: [1, undefined, 3]};Duktape.enc('jx', obj);
Unicon: uniduk completed with {foo:NaN,bar:[1,undefined,3]}
Attempt: uniduk.js
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
Unicon: Unload Duktape
```

And Duktape Javascript step 2 has been taken. *Can you feel the nerd dancing? Running man with a wicked loud*[1] *"Ice, Ice Baby" playing in the background?*

To be a little more confident, here is an initial stress test, calling out the Viking, `valgrind`.

```
prompt$ valgrind ./uniduk
==18468== Memcheck, a memory error detector
==18468== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==18468== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==18468== Command: ./uniduk
==18468==
==18469== Warning: invalid file descriptor -1 in syscall close()
==18470==
==18470== HEAP SUMMARY:
==18470==     in use at exit: 10,182 bytes in 59 blocks
==18470==   total heap usage: 66 allocs, 7 frees, 10,894 bytes allocated
```

[1] Turned up wayyy past 4 on the dial, maybe even a 6.

```
==18470==
==18470== LEAK SUMMARY:
==18470==    definitely lost: 0 bytes in 0 blocks
==18470==    indirectly lost: 0 bytes in 0 blocks
==18470==      possibly lost: 0 bytes in 0 blocks
==18470==    still reachable: 10,182 bytes in 59 blocks
==18470==         suppressed: 0 bytes in 0 blocks
==18470== Rerun with --leak-check=full to see details of leaked memory
==18470==
==18470== For counts of detected and suppressed errors, rerun with: -v
==18470== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==18469==
==18469== HEAP SUMMARY:
==18469==     in use at exit: 2,017 bytes in 59 blocks
==18469==   total heap usage: 64 allocs, 5 frees, 2,201 bytes allocated
==18469==
==18469== LEAK SUMMARY:
==18469==    definitely lost: 0 bytes in 0 blocks
==18469==    indirectly lost: 0 bytes in 0 blocks
==18469==      possibly lost: 0 bytes in 0 blocks
==18469==    still reachable: 2,017 bytes in 59 blocks
==18469==         suppressed: 0 bytes in 0 blocks
==18469== Rerun with --leak-check=full to see details of leaked memory
==18469==
==18469== For counts of detected and suppressed errors, rerun with: -v
==18469== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Attempt: 1 + 2;
Unicon: uniduk completed with 3.0
Attempt: print('Duktape print'); var r = 7 * 6;
Duktape print
Unicon: uniduk completed with
Attempt: r;
Unicon: uniduk completed with 42.0
Attempt: 'abc';
Unicon: uniduk completed with abc
Attempt: var obj = {foo: 0/0, bar: [1, undefined, 3]};Duktape.enc('jx', obj);
Unicon: uniduk completed with {foo:NaN,bar:[1,undefined,3]}
Attempt: uniduk.js
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
Unicon: Unload Duktape
```

There used to be a warning about invalid fd passed to the `close` syscall. Turns out, *by tracking with* `strace`, it was actually Unicon start up doing that.[2]. *Didn't effect program outcome, but after mentioning it to the principals it was fixed.*[3].

The rest is all good, 0 leaked RAM. The still reachable being non-zero is a common thing in most processes; exit was called while the Unicon engine was still in play, so there is valid runtime memory used for statics (like message strings) and a little bit of allocation for buffers. The important numbers for this test pass are

```
...
==nnnnn== LEAK SUMMARY:
==nnnnn==    definitely lost: 0 bytes in 0 blocks
==nnnnn==    indirectly lost: 0 bytes in 0 blocks
==nnnnn==      possibly lost: 0 bytes in 0 blocks
```

[2] Tried this with a bare bones Unicon program, single write of a string. `valgrind` still reported the invalid -1 to the close syscall.

[3] Mentioned the -1 being passed to `close` during *ucode* invocation. This was caused by a lower level `sh` edge case interaction in handling the way `ucode` is attached to an invocation script. Harmless, and not to be entirely blamed on Unicon, but it was fixed anyway. Every bug reported to the Unicon team has been fixed while writing this book.

```
...
==nnnnn== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
...
```

Those are what you want to see from a Viking[4] report.

Process numbers will vary by machine and run

Duktape license obligation

```
Copyright (c) 2013-2016 by Duktape authors (see AUTHORS.rst)

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

27.1.4 mruby

Integrate the *mruby* (Mini Ruby) library with Unicon.

First pass, see if things gel:

```
/*
 uniruby-v1.c loadfunc an mruby interpreter in Unicon

 tectonics: gcc -o unirbuy.so -shared -fpic uniruby-v1.c \
                /usr/lib/libmruby.a -lm
*/
#include <stdio.h>
#include <stdlib.h>

#include <mruby.h>
#include <mruby/compile.h>

#include "icall.h"

int
uniruby(int argc, descriptor *argv)
```

[4] `valgrind` is a Norse name, pronounced to rhyme with `grinned`, not `grind`. Go vikings.

```
{
    /* start up an mruby engine */
    mrb_state *mrb = mrb_open();
    if (!mrb) Error(500);

    /* Need a string of code parameter */
    if (argc < 1) Error(103);
    ArgString(1);

    /* run the Ruby code, and return a universal answer */
    mrb_load_string(mrb, StringVal(argv[1]));
    RetInteger(42);
}
```

The sample Unicon file to load and test the engine, `uniruby-v1.icn`.

```
#
# uniruby-v1.icn, integrate mruby in Unicon
#
# tectonics: gcc -o uniruby.so -shared -fpic uniruby-v1.c \
#                 /usr/lib/libmruby.a -lm
procedure main()
    uniruby := loadfunc("./uniruby.so", "uniruby")

    code := "p 'Hello, world'"
    write("Attempt: ", code)
    result := uniruby(code)
    write("Unicon result: ", result)
end
```

And a test run

```
prompt$ gcc -o uniruby.so -shared -fpic uniruby-v1.c /usr/lib/libmruby.a -lm
```

```
prompt$ unicon -s uniruby-v1.icn -x
Attempt: p 'Hello, world'
"Hello, world"
Unicon result: 42
```

mruby license obligation

```
Copyright (c) 2016 mruby developers

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
```

```
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

27.1.5 ficl

The Forth Inspired Command Language.

http://ficl.sourceforge.net/

This sample embeds a Forth interpreter using ficl-4.1.0 as a shared library.

The initial trial, `unificl-v1`

```
/*
 unificl-v1.c a Forth interpreter in Unicon with loadfunc

 tectonics:
     gcc -o unificl.so -shared -fpic unificl-v1.c -lficl

*/
#include <stdio.h>
#include <stdlib.h>

#include "ficl.h"
#include "icall.h"

/* Global variables to track VM across calls */
int unificlLoaded = 0;
ficlVm *unificlVm = NULL;
ficlSystem *unificlSystem = NULL;

/* Unicon calling ficl */
int
unificl(int argc, descriptor *argv)
{
    int returnValue = 0;
    char buffer[256];

    /* Need a string of code parameter */
    if (argc < 1) Error(103);
    ArgString(1);

    /* start up the ficl VM */
    if (!unificlLoaded) {
        unificlSystem = ficlSystemCreate(NULL);
        ficlSystemCompileExtras(unificlSystem);
        unificlVm = ficlSystemCreateVm(unificlSystem);
        returnValue = ficlVmEvaluate(unificlVm,
                                     ".ver .( " __DATE__ " ) cr quit");
        unificlLoaded = 1;
    }

    /* Run the Forth code and get an integer status */
```

```
    returnValue = ficlVmEvaluate(unificlVm, StringVal(argv[1]));

    /* return to Unicon */
    RetInteger(returnValue);
}

/* run down the ficl VM, return a 0 to Unicon */
int
unificlRundown(int argc, descriptor *argv)
{
    ficlSystemDestroy(unificlSystem);
    unificlVm = NULL;
    unificlSystem = NULL;
    unificlLoaded = 0;
    RetInteger(0);
}
```

A sample Unicon file to load and test the engine, `unificl-v1.icn`.

```
#
# unificl.icn, Forth scripting with ficl
#
# tectonics:
#    gcc -o unificl.so -shared -fpic unificl.c -lficl
#
procedure main()
    unificl := loadfunc("./unificl.so", "unificl")
    unificlRundown := loadfunc("./unificl.so", "unificlRundown")

    # say hello, and leave a number on the stack
    code := "cr .( Hello, world) cr 123454321"
    write("\nEvaluate: ", image(code), "\n")
    result := unificl(code)
    write("Unicon result: ", result)

    # display the left over number from previous invocation
    code := ". cr"
    write("\nEvaluate: ", image(code), "\n")
    result := unificl(code)
    write("Unicon result: ", result)

    # rundown the ficl system
    unificlRundown()

    # start a fresh copy
    code := ": unificl-test  6 7 * . cr ;  unificl-test"
    write("\nEvaluate: ", image(code), "\n")
    result := unificl(code)
    write("Unicon result: ", result)

    # display the default ficl word list
    code := "words"
    write("\nEvaluate: ", image(code), "\n")
    result := unificl(code)
    write("Unicon result: ", result)

    # and a test with an error
    code := "nonsense forth code"
```

```
    write("\nEvaluate: ", image(code))
    write("Expect ficl error", "\n")
    result := unificl(code)
    write("Unicon result: ", result)
end
```

And a test run (using an uninstalled copy of `ficl`, so the Makefile includes C compiler `-L`, `-I` options and `LD_LIBRARY_PATH` runtime settings).

Forth scripting inside Unicon. *If you look closely, that ficl word-list display includes the test definition of unificl-test, along with the ficl core, and default extension words.*

Note that `ficl` result code `-257` is the normal exit status. Defined as

```
/* hungry - normal exit */
#define FICL_VM_STATUS_OUT_OF_TEXT  (-257)
```

That means the text was successfully interpreted and the engine is ready for more.

`-260` is defined as

```
/* interpreter found an error */
#define FICL_VM_STATUS_ERROR_EXIT   (-260)
```

Second step

And now for some real integration.

```
/*
 unificl-v1.c a Forth interpreter in Unicon with loadfunc

 tectonics:
     gcc -o unificl.so -shared -fpic unificl-v1.c -lficl

*/
#include <stdio.h>
#include <stdlib.h>

#include "ficl.h"
#include "icall.h"

/* Global variables to track VM across calls */
int unificlLoaded = 0;
ficlVm *unificlVm = NULL;
ficlSystem *unificlSystem = NULL;

/* Unicon calling ficl */
int
unificl(int argc, descriptor *argv)
{
    int returnValue = 0;
    char buffer[256];

    /* Need a string of code parameter */
    if (argc < 1) Error(103);
    ArgString(1);
```

```
    /* start up the ficl VM */
    if (!unificlLoaded) {
        unificlSystem = ficlSystemCreate(NULL);
        ficlSystemCompileExtras(unificlSystem);
        unificlVm = ficlSystemCreateVm(unificlSystem);
        //returnValue = ficlVmEvaluate(unificlVm,
        //                              ".ver .( " __DATE__ " ) cr quit");
        unificlLoaded = 1;
    }

    /* Run the Forth code and get an integer status */
    returnValue = ficlVmEvaluate(unificlVm, StringVal(argv[1]));

    /* return to Unicon */
    RetInteger(returnValue);
}

/* run down the ficl VM, return a 0 to Unicon */
int
unificlRundown(int argc, descriptor *argv)
{
    ficlSystemDestroy(unificlSystem);
    unificlVm = NULL;
    unificlSystem = NULL;
    unificlLoaded = 0;
    RetInteger(0);
}

/* Return the stack */
int
unificlStack(int argc, descriptor *argv)
{
    int depth;
    int i;
    listblock *list;

    if (!unificlLoaded) {
        Error(117); /* report engine not loaded, missing main procedure */
    }

    depth = ficlStackDepth(unificlVm->dataStack);
    int *integers = malloc(sizeof(int) * depth);
    for (i = 0; i < depth; i++) {
        integers[i] = ficlStackFetch(unificlVm->dataStack, i).i;
    }
    list = mkIlist(integers, depth);

    /* return to Unicon */
    free(integers);
    RetList(list);
}

/* Return the floating point stack */
int
unificlFloatStack(int argc, descriptor *argv)
{
    int depth;
    int i;
```

```
    listblock *list;

    if (!unificlLoaded) {
        Error(117); /* report engine not loaded, missing main procedure */
    }

    depth = ficlStackDepth(unificlVm->floatStack);
    double *doubles = malloc(sizeof(double) * depth);
    for (i = 0; i < depth; i++) {
        doubles[i] = ficlStackFetch(unificlVm->floatStack, i).f;
    }
    list = mkRlist(doubles, depth);

    /* return to Unicon */
    free(doubles);
    RetList(list);
}
```

A sample Unicon file to load and test the updated engine, `unificl.icn`.

```
#
# unificl.icn, Forth scripting with ficl
#
# tectonics:
#    gcc -o unificl.so -shared -fpic unificl.c -lficl
#
link fullimag
procedure main()
    unificl := loadfunc("./unificl.so", "unificl")
    unificlStack := loadfunc("./unificl.so", "unificlStack")
    unificlFloatStack := loadfunc("./unificl.so", "unificlFloatStack")
    unificlRundown := loadfunc("./unificl.so", "unificlRundown")

    # say hello, and leave a number on the stack
    code := ".( Hello, world) cr 123454321"
    write("\nUnicon evaluate: ", image(code))
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))

    # display the left over number from previous invocation
    code := ". cr"
    write("\nUnicon evaluate: ", image(code))
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))

    # rundown the ficl system
    unificlRundown()

    # start a fresh copy, and leave some numbers on the data stack
    code := ": unificl-test  6 7 * dup 1+ dup 1+ ; unificl-test"
    write("\nUnicon evaluate: ", image(code))
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))

    # try the floating point stack
    code := ": unificl-float  1e 4.2e ; unificl-float"
    write("\nUnicon evaluate: ", image(code))
    result := unificl(code)
```

```
    write("Unicon ficl (", result, "): ", fullimage(unificlFloatStack()))

    # try addresses, add the Xt of the sample definition to the stack
    code := "' unificl-test"
    write("\nUnicon evaluate: ", image(code))
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))

    # execute that Xt
    code := "execute"
    write("\nUnicon evaluate: ", image(code))
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))


    # and a test with an error
    code := "nonsense forth code"
    write("\nUnicon evaluate: ", image(code))
    write("Unicon expect ficl error")
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))

    # and a test with a crash
    code := "0 ?"
    write("\nUnicon evaluate: ", image(code))
    write("Unicon expect ficl segfault")
    result := unificl(code)
    write("Unicon ficl (", result, "): ", fullimage(unificlStack()))
end
```

Sample run ends in a purposeful error, Unicon trapping a Ficl segfault:

```
prompt$ make -B unificl
make[1]: Entering directory '/home/btiffin/wip/writing/unicon/programs'
gcc -o unificl.so -shared -fpic unificl.c -lficl -lm
unicon -s unificl.icn -x

Unicon evaluate: ".( Hello, world) cr 123454321"
Hello, world
Unicon ficl (-257): [123454321]

Unicon evaluate: ". cr"
123454321
Unicon ficl (-257): []

Unicon evaluate: ": unificl-test  6 7 * dup 1+ dup 1+ ; unificl-test"
Unicon ficl (-257): [44,43,42]

Unicon evaluate: ": unificl-float  1e 4.2e ; unificl-float"
Unicon ficl (-257): [4.199999809265137,1.0]

Unicon evaluate: "' unificl-test"
Unicon ficl (-257): [40163192,44,43,42]

Unicon evaluate: "execute"
Unicon ficl (-257): [44,43,42,44,43,42]

Unicon evaluate: "nonsense forth code"
```

```
Unicon expect ficl error
nonsense not found
Unicon ficl (-260): []

Unicon evaluate: "0 ?"
Unicon expect ficl segfault

Run-time error 302
File unificl.icn; Line 72
memory violation
Traceback:
   main()
   &null("0 ?") from line 72 in unificl.icn
Makefile:47: recipe for target 'unificl' failed
make[1]: *** [unificl] Error 1
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/programs'
```

The `unificl` engine can evaluate Forth source, and Unicon can snag the stack and the floating point stack as needed, as a list. That returned list is ready for Unicon style stack functions, *pop* will pop what would be the top of the `ficl` data stack. *Separate structures, the ficl stack is the ficl stack, and Unicon gets a copy as a list.*

As a side bonus, no effort was required to have Unicon catch (and report) the purposeful segfault in the last `ficl` test of `0 ?` (an attempt to read address 0).

FICL License obligation

```
FICL LICENSE

Copyright © 1997-2001 John Sadler (john_sadler@alum.mit.edu)
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

27.1.6 Lua

Lua scripts embedded in Unicon.

First pass, see if things gel:

```
/*
 unilua-v1.c loadfunc a Lua interpreter in Unicon

 tectonics: gcc -o unilua-v1.so -shared -fpic unilua-v1.c \
                -I/usr/include/lua5.3 -llua5.3
*/
#include <stdio.h>
#include <string.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

#include "icall.h"

int unilua (int argc, descriptor argv[]) {
  char buff[256];
  int error;

  char *unibuf;

#ifdef LUA50
  lua_State *L = lua_open();   /* opens Lua */
  if (!L) {
      Error(500);
  }
  luaopen_base(L);             /* opens the basic library */
  luaopen_table(L);            /* opens the table library */
  luaopen_io(L);               /* opens the I/O library */
  luaopen_string(L);           /* opens the string lib. */
  luaopen_math(L);             /* opens the math lib. */
#else
  lua_State *L = luaL_newstate();
  if (!L) {
      Error(500);
  }
  luaL_openlibs(L);
#endif

  /* ensure argv[1] is a string */
  ArgString(1);

  /* evaluate some Lua */
  unibuf = StringVal(argv[1]);
  error = luaL_loadbuffer(L, unibuf, strlen(unibuf), "line") ||
          lua_pcall(L, 0, 0, 0);
  if (error) {
      fprintf(stderr, "%s", lua_tostring(L, -1));
      lua_pop(L, 1);  /* pop error message from the stack */
      Error(107);
  }

  lua_close(L);
  RetInteger(42);
```

```
  return 0;
}
```

The sample Unicon file to load and test the engine, `unilua-v1.icn`.

```
#
# unilua-v1.icn, Initial trial of Lua integration
#
# tectonics:
#    gcc -o unilua-v1.so -shared -fpic unilua-v1.c \
#        -I/usr/include/lua5.3 -llua5.3
#
procedure main()
    unilua := loadfunc("./unilua-v1.so", "unilua")

    code := "print(\"Hello, world\")"
    result := unilua(code)
    write("Unicon: ", result)
end
```

The *make* recipes:

```
# Lua in Unicon
# alpha test
unilua-v1.so: unilua-v1.c
> gcc -o unilua-v1.so -shared -fpic unilua-v1.c \
      -I/usr/include/lua5.3 -llua5.3

unilua-v1: unilua-v1.so
> unicon -s unilua-v1.icn -x

# unilua
unilua.so: unilua.c
> gcc -o unilua.so -shared -fpic unilua.c \
      -I/usr/include/lua5.3 -llua5.3

unilua: unilua.so
> unicon -s unilua.icn -x
```

And the alpha test run:

```
prompt$ make -B --no-print-directory unilua-v1.so
gcc -o unilua-v1.so -shared -fpic unilua-v1.c -I/usr/include/lua5.3 -llua5.3
```

```
prompt$ unicon -s unilua-v1.icn -x
Hello, world
Unicon: 42
```

Second step

Lua state is held in a persistent variable, remembered across calls. A new `luaclose` function is supported.

```
/*
 unilua.c loadfunc a Lua interpreter in Unicon

 tectonics: gcc -o unilua.so -shared -fpic unilua.c \
```

```
                -I/usr/include/lua5.3 -llua5.3
*/
#include <stdio.h>
#include <string.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

#include "icall.h"

lua_State *uniLuaState;

/*
 unilua: execute Lua code from Unicon string
*/
int
unilua (int argc, descriptor argv[])
{
    int error;

    char *unibuf;
    int luaType;

    if (!uniLuaState) {
#ifdef LUA50
        uniLuaState = lua_open();     /* opens Lua */
        if (!uniLuaState) {
            Error(500);
        }
        luaopen_base(uniLuaState);    /* opens the basic library */
        luaopen_table(uniLuaState);   /* opens the table library */
        luaopen_io(uniLuaState);      /* opens the I/O library */
        luaopen_string(uniLuaState);  /* opens the string lib. */
        luaopen_math(uniLuaState);    /* opens the math lib. */
#else
        uniLuaState = luaL_newstate();
        if (!uniLuaState) {
            Error(500);
        }
        luaL_openlibs(uniLuaState);
#endif
    }

    /* ensure argv[1] is a string */
    ArgString(1);

    /* evaluate some Lua */
    unibuf = StringVal(argv[1]);
    error = luaL_loadbuffer(uniLuaState, unibuf, strlen(unibuf), "line") ||
            luaL_dostring(uniLuaState, unibuf);
    if (error) {
        fprintf(stderr, "%s", lua_tostring(uniLuaState, -1));
        lua_pop(uniLuaState, 1);  /* pop error message from the stack */
        Error(107);
    }

    luaType = lua_type(uniLuaState, -1);
    switch (luaType) {
```

```
        case LUA_TSTRING:
            RetString((char *)lua_tostring(uniLuaState, -1));
            break;
        case LUA_TNUMBER:
            if (lua_isinteger(uniLuaState, -1)) {
                RetInteger(lua_tointeger(uniLuaState, -1));
            } else {
                RetReal(lua_tonumber(uniLuaState, -1));
            }
            break;
        default:
            RetString((char *)lua_typename(uniLuaState, luaType));
            break;
    }
    return 0;
}

/*
 Close Lua state
*/
int
uniluaClose(int argc, descriptor argv[])
{
    lua_close(uniLuaState);
    RetNull();
    return 0;
}
```

Another sample Unicon file to load and test the engine, `unilua.icn`.

```
#
# unilua.icn, Lua integration demonstration
#
# tectonics:
#     gcc -o unilua.so -share -fpic unilua.c \
#         -I/usr/include/lua5.3 -llua5.3
#
procedure main()
    unilua := loadfunc("./unilua.so", "unilua")
    luaclose := loadfunc("./unilua.so", "uniluaClose")

    code := "return \"Running \" .. _VERSION"
    result := unilua(code)
    write("Unicon: ", result)

    luaclose()
end
```

And the second test run:

```
prompt$ make -B unilua.so
make[1]: Entering directory '/home/btiffin/wip/writing/unicon/programs'
gcc -o unilua.so -shared -fpic unilua.c -I/usr/include/lua5.3 -llua5.3
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/programs'
```

```
prompt$ unicon -s unilua.icn -x
Unicon: Running Lua 5.3
```

A final step will be returning all Lua stack items to Unicon during each call, and perhaps exposing a few more Lua internal API features.

Lua license obligation

```
Copyright © 1994-2016 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

27.1.7 Fortran

Calling FORTRAN programs from Unicon.

The first step passes no arguments, but uses various Fortran source forms; FORTRAN-66, FORTRAN-77. A Fortran-90 (free format) program takes an integer and returns a result to Unicon.

```
C
C FORTRAN 66 FORM
C
      WRITE (6,7)
    7 FORMAT(13H HELLO, WORLD)
      END
```

```
*
* Fortran 77 form
*
      PROGRAM HELLO
      PRINT *,'Hello, world'
      END
```

This next Fortran-90 source accepts an integer argument and returns the square of the Unicon value (using Fortran `subroutine` call frame expectations which has no return value, all parameters passed by reference) plus the cube of the Unicon number (using Fortran `function` call frame expectations). The data marshalling to and from Fortran uses a small layer of C, but that could be pure Fortran if the data structures from `icall.h` were ported to Fortran friendly data definitions.

```
!
! Fortran 90 form
!
! Compute the square of n, result in m
subroutine squareto(n,m)
    m = n*n
    return
end

! Compute the cube of n, return value
integer function cube(n)
    cube = n*n*n
    return
end
```

A little bit of C as an intermedia data marshalling layer:

```
/*
 unifortran.c loadfunc some Fortran functions in Unicon

 tectonics:
     gfortran -o fortran.o -fpic fortran.f
     gcc -o fortran.so -shared -fpic unifortran.c fortran.o
*/
#include <stdio.h>
#include <string.h>

#include "icall.h"

/* Fortran is pass by reference */
int squareto_(int *, int *);
int cube_(int *);

/*
 unifortran: execute Fortran functions
*/
int
unifortran (int argc, descriptor argv[])
{
    int n, m;
    if (argc != 1) Error(104);

    /* ensure argv[1] is an integer */
    ArgInteger(1);
    n = IntegerVal(argv[1]);

    /* first call the subroutine, data comes back in second argument */
    squareto_(&n, &m);

    /* invoke Fortran function, argument by address, add to previous */
    m += cube_(&n);
    RetInteger(m);
}
```

The make recipes:

```
# gfortran modules
fortran-66.so: fortran-66.f
```

```
> gfortran -o fortran-66.so -shared -fpic fortran-66.f

fortran-77.so: fortran-77.f
> gfortran -o fortran-77.so -shared -fpic fortran-77.f

fortran.so: fortran.f unifortran.c
> gfortran -ffree-form -c -fpic fortran.f
> gcc -o fortran.so -shared -fpic unifortran.c fortran.o

fortran: fortran.icn fortran-66.so fortran-77.so fortran.so
> unicon -s $< -x
```

A Unicon test file:

```
#
# fortran.icn, invoke some gfortran programs and functions
#
# tectonics:
#    gfortran -o fortran-66.so -shared -fpic fortran-66.f
#    gfortran -o fortran-77.so -shared -fpic fortran-77.f
#    gfortran -ffree-form -c -fpic fortran.f
#    gcc -o fortran.so -shared -fpic unifortran.c fortran.o
#
procedure main()
    # load and invoke an old form Fortran main module
    # this could just as well be an open pipe or system call
    # but this is an alpha level proof of mechanism
    fortran66 := loadfunc("./fortran-66.so", "main")
    fortran66()

    # load and invoke another Fortran main module
    # a simple demonstration of variant forms of Fortran source
    fortran77 := loadfunc("./fortran-77.so", "main")
    fortran77()

    # load a Fortran module, and pass arguments
    # any realistic use of Fortran would build on this type of interface
    # or, would require fortran-ization of the C macros in icall.h
    fortran := loadfunc("./fortran.so", "unifortran")
    result := fortran(5)
    write("Subroutine square(5, result) + cube(5) from Fortran: ", result)
end
```

The alpha trial serves multiple purposes in this case. There is a simple goal of trying various forms of Fortran source; FORTRAN 66, FORTRAN 77, and more modern Fortran syntax.

There is also a proof of technology test to see if `main` modules can be loaded with *loadfunc*.

A third purpose is ensuring that C interstitial code plays well between Fortran and Unicon, when passing parameters and retrieving results.

```
prompt$ make --no-print-directory -B fortran
gfortran -o fortran-66.so -shared -fpic fortran-66.f
gfortran -o fortran-77.so -shared -fpic fortran-77.f
gfortran -ffree-form -c -fpic fortran.f
gcc -o fortran.so -shared -fpic unifortran.c fortran.o
unicon -s fortran.icn -x
 HELLO, WORLD
```

```
 Hello, world
Subroutine square(5, result) + cube(5) from Fortran: 150
```

27.1.8 Assembler

Calling assembly programs from Unicon.

Assembler is no different than C when it comes to the binary objects produced for the operating system. Assembler is a step on the way to native binary for many C compilers, *GCC* in particular.

A very similar Unicon *loadfunc* setup, identical actually:

```
#
# uniasm.icn, load an assembler object file
#
# tectonics:
#    gcc -S -fpic uniasm.c
#    gcc -o uniasm.so -shared -fpic uniasm.s
#
procedure main()
    # load the uniasm module
    uniasm := loadfunc("./uniasm.so", "uniasm")

    # pass a 42, and get back the length of an output message
    result := uniasm(42)
    write("Unicon: ", result)
end
```

A fairly sophisticated looking piece of x86_64 assembler source:

```
        .file        "uniasm.c"
        .section        .rodata
.LC0:
        .string        "uniasm: %ld\n"
        .text
        .globl        uniasm
        .type        uniasm, @function
uniasm:
.LFB2:
        .cfi_startproc
        pushq        %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        subq        $16, %rsp
        movl        %edi, -4(%rbp)
        movq        %rsi, -16(%rbp)
        cmpl        $0, -4(%rbp)
        jg        .L2
        movl        $101, %eax
        jmp        .L3
.L2:
        movq        -16(%rbp), %rax
        leaq        16(%rax), %rdx
```

```
        movq        -16(%rbp), %rax
        addq        $16, %rax
        movq        %rdx, %rsi
        movq        %rax, %rdi
        call        cnv_int@PLT
        testl        %eax, %eax
        jne        .L4
        movq        -16(%rbp), %rcx
        movq        -16(%rbp), %rax
        movq        24(%rax), %rdx
        movq        16(%rax), %rax
        movq        %rax, (%rcx)
        movq        %rdx, 8(%rcx)
        movl        $101, %eax
        jmp        .L3
.L4:
        movq        -16(%rbp), %rax
        movabsq        $-6917529027641081855, %rcx
        movq        %rcx, (%rax)
        movq        -16(%rbp), %rax
        addq        $16, %rax
        movq        8(%rax), %rax
        movq        %rax, %rsi
        leaq        .LC0(%rip), %rdi
        movl        $0, %eax
        call        printf@PLT
        movslq        %eax, %rdx
        movq        -16(%rbp), %rax
        movq        %rdx, 8(%rax)
        movl        $0, %eax
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size        uniasm, .-uniasm
        .ident        "GCC: (Ubuntu 5.5.0-12ubuntu1~16.04) 5.5.0 20171010"
        .section        .note.GNU-stack,"",@progbits
```

Which is computer generated output from a much simpler looking C source file:

```
/* uniasm.c, used to produce uniasm.s */
#include <stdio.h>
#include "icall.h"

int
uniasm(int argc, descriptor argv[])
{
    /* Expect an integer argument from Unicon */
    ArgInteger(1);

    /* print a message with arg, and return the number of bytes written */
    RetInteger(printf("uniasm: %ld\n", IntegerVal(argv[1])));
}
```

```
prompt$ gcc -S -fpic uniasm.c
```

That assembly can be used just like a C file when it comes to creating the shared objects required by *loadfunc*.

The `-fpic` option is required along with `gcc -S` to generate assembly code that can be relocated, for use in a dynamic shared object file.

```
prompt$ gcc -o uniasm.so -shared -fpic uniasm.s
```

Note the `.s` on that command line, not a `.c` file.

And running that from Unicon:

```
prompt$ unicon -s uniasm.icn -x
uniasm: 42
Unicon: 11
```

Although this example was generated assembly, the `.s` source code could be used as a basis for hand edited files, all the Unicon *loadfunc* requirements, and associated macros, properly expanded into working assembler.

27.1.9 vedis

`vedis`, an embedded Redis clone by Symisc Systems. Using a Redis style data store from Unicon.

http://vedis.symisc.net/

The Unicon setup uses `pathload` from IPL file `io.icn`.

```
#
# univedis-v1.icn, Embed a Redis clone, vedis by Symisc
#
# tectonics:
#    gcc -o univedis-v1.so -shared -fpic univedis-v1.c vedis.c
#
link io
procedure main()
    lib := "univedis-v1.so"
    VedisOpen := pathload(lib, "VedisOpen")
    Vedis := pathload(lib, "Vedis")
    VedisClose := pathload(lib, "VedisClose")

    handle := VedisOpen(":mem:")

    Vedis(handle, "SET message 'Hello, world'")
    result := Vedis(handle, "GET message")
    write(result)

    VedisClose(handle)
end
```

The vedis source is an SQLite style amalgamation bundle. Just include `vedis.c` in a build.

```
# vedis (Embedded Redis clone)
univedis-v1.so: univedis-v1.c
> gcc -o univedis-v1.so -shared -fpic univedis-v1.c vedis.c \
  -Wno-unused

univedis-v1: univedis-v1.so univedis-v1.icn
> unicon -s univedis-v1.icn -x
```

The initial trial is a simple vedis example:

```
/*
 univedis-v1.c, trial for vedis embedding in Unicon

 tectonics:
     gcc -o univedis-v1.so -shared -fpic univedis-v1.c vedis.c
*/

#include <stdio.h>
#include "vedis.h"
#include "icall.h"

/*
 open a vedis data store (":mem:" for in-memory)
*/
int
VedisOpen(int argc, descriptor argv[])
{
    int rc;
    vedis *vp;

    ArgString(1)

    rc = vedis_open(&vp, StringVal(argv[1]));
    if (rc != VEDIS_OK) Error(500);

    RetInteger((long)vp);
}

/*
 close a vedis connection
*/
int
VedisClose(int argc, descriptor argv[])
{
    int rc;
    vedis *vp;

    /* argv[1] is vedis handle */
    ArgInteger(1);
    rc = vedis_close((vedis *)IntegerVal(argv[1]));
    RetInteger(rc);
}

/*
 execute a vedis command
*/
int
Vedis(int argc, descriptor argv[])
{
    int rc;
    vedis *vp;
    vedis_value *rp;
    const char *result;

    /* argv[1] is vedis handle */
    ArgInteger(1);
    /* argv[2] is vedis command as string - single result */
```

```
    ArgString(2);

    vp = (vedis *)IntegerVal(argv[1]);
    rc = vedis_exec(vp, StringVal(argv[2]), -1);
    vedis_exec_result(vp, &rp);
    result = vedis_value_to_string(rp, 0);
    RetString((char *)result);
}
```

And a sample run:

```
prompt$ make -B --no-print-directory univedis-v1
gcc -o univedis-v1.so -shared -fpic univedis-v1.c vedis.c \
  -Wno-unused
unicon -s univedis-v1.icn -x
Hello, world
```

There are some 70 `Redis` type commands in the `vedis` engine.

vedis license obligation

```
/*
 * Copyright (C) 2013 Symisc Systems, S.U.A.R.L [M.I.A.G Mrad Chems Eddine
↪<chm@symisc.net>].
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Redistributions in any form must be accompanied by information on
 *    how to obtain complete source code for the Vedis engine and any
 *    accompanying software that uses the Vedis engine software.
 *    The source code must either be included in the distribution
 *    or be available for no more than the cost of distribution plus
 *    a nominal fee, and must be freely redistributable under reasonable
 *    conditions. For an executable file, complete source code means
 *    the source code for all modules it contains.It does not include
 *    source code for modules or files that typically accompany the major
 *    components of the operating system on which the executable file runs.
 *
 * THIS SOFTWARE IS PROVIDED BY SYMISC SYSTEMS ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT, ARE DISCLAIMED.  IN NO EVENT SHALL SYMISC SYSTEMS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

27.1.10 libcox

`libcox` a cross platform system command evaluation library by Symisc Systems.

http://libcox.symisc.net/

Another Unicon *loadfunc* sample.

```
#
# unicox.icn, Embed libcox system utilities by Symisc Systems
#
# tectonics:
#    gcc -o unicox.so -shared -fpic unicox.c libcox.c
#
link io
procedure main()
    lib := "unicox.so"
    unicox := pathload(lib, "unicox")
    unicoxClose := pathload(lib, "unicoxClose")

    # Fetch the libcox supported commands
    result := unicox("CMD_LIST")
    write("\nCMD_LIST\n",result)

    # list the .rst file names from the given directory
    result := unicox("glob *.txt '%s'", "..")
    write("\nglob *.txt from ..\n",result)

    # shut down the command engine
    unicoxClose() | stop("Error shutting down libcox")
end
```

The libcox source is an SQLite style amalgamation bundle. Just include `libcox.c` in a build.

```
# libcox (cross platform POSIX type commands)
unicox.so: unicox.c
> gcc -o unicox.so -shared -fpic unicox.c libcox.c
> @echo

unicox: unicox.so unicox.icn
> unicon -s unicox.icn -x
```

The loadable:

```
/*
 unicox.c, trial for libcox embedding in Unicon

 tectonics:
     gcc -o unicox.so -shared -fpic unicox.c libcox.c
*/
#include <stdio.h>
#include "libcox.h"
#include "icall.h"
```

```
static libcox *libcoxHandle;
static libcox_value *libcoxResult;

int
unicox(int argc, descriptor argv[])
{
    const char *libcoxValue;
    int rc;

    /* handle is remembered across calls */
    if (!libcoxHandle) {
        rc = libcox_init(&libcoxHandle);
        if (rc != LIBCOX_OK) {
            Error(500);
        }
    }

    /* Unicon passes a command string (and possibly one argument) */
    ArgString(1);

    if (argc > 1) {
        ArgString(2);
    }

    /* last result left alone, freed before converting a new value */
    if (libcoxResult) {
        libcox_exec_result_destroy(libcoxHandle, libcoxResult);
    }

    /* Evaluate the command, with no, or one argument */
    if (argc > 1) {
        rc = libcox_exec_fmt(libcoxHandle, &libcoxResult,
                             StringVal(argv[1]), StringVal(argv[2]));
    } else {
        rc = libcox_exec(libcoxHandle, &libcoxResult,
                         StringVal(argv[1]), -1);
    }

    if (rc != LIBCOX_OK) {
        Error(107);
    }

    libcoxValue = libcox_value_to_string(libcoxResult, 0);
    RetString((char *)libcoxValue);
}

int
unicoxClose(int argc, descriptor argv[])
{
    if (libcoxHandle) {
        libcox_release(libcoxHandle);
        RetNull();
    } else {
        Fail;
    }
}
```

The initial trial includes the `libcox` **`CMD_LIST`** and a sample file expansion **`glob`** from a different working direc-

tory.

```
prompt$ make -B --no-print-directory unicox | par
gcc -o unicox.so -shared -fpic unicox.c libcox.c

unicon -s unicox.icn -x

CMD_LIST
["glob","list","ls","mmap","cat","CMD_LIST","time","microtime","getdate"
,"gettimeofday","date","strftime","gmdate","localtime","idate","mktime",
"base64_decode","base64_encode","urldecode","urlencode","size_format","s
trrev","strrchr","strripos","strrpos","stripos","strpos","stristr","strs
tr","bin2hex","strtoupper","strtolower","rtrim","ltrim","trim","explode"
,"implode","strncasecmp","strcasecmp","strncmp","strcmp","strlen","html_
decode","html_escape","chunk_split","substr_count","substr_compare","sub
str","base_convert","baseconvert","octdec","bindec","hexdec","decbin","d
ecoct","dechex","round","os","osname","uname","umask","slink","symlink",
"lnk","link","fnmatch","strglob","pathinfo","basename","dirname","touch"
,"file_type","filetype","dt","disk_total_space","df","disk_free_space","
chgrp","chown","chmod","delete","remove","rm","unlink","usleep","sleep",
"chroot","lstat","stat","tmpdir","temp_dir","tmp_dir","fileexists","file
_exists","filemtime","file_mtime","filectime","file_ctime","fileatime","
file_atime","filesize","file_size","isexec","is_exec","is_executable","i
swr","is_wr","is_writable","isrd","is_rd","is_readable","isfile","is_fil
e","islnk","is_lnk","islink","is_link","isdir","is_dir","getgid","getuid
","gid","uid","getusername","username","getpid","pid","random","rand","g
etenv","fullpath","full_path","real_path","realpath","rename","set_env",
"setenv","putenv","env","echo","mkdir","rmdir","getcwd","cwd","pwd","chd
ir","cd"]

glob *.txt from .. ["gpl-3.0.txt","preamble.txt","lgpl-3.0.txt"]
```

With `libcox.c` version 1.7, there are over 145 commands available. Set to work across multiple platforms; GNU/Linux and Windows at a minimum.

libcox license obligation

```
/*
 * Symisc libcox: Cross Platform Utilities & System Calls.
 * Copyright (C) 2014, 2015 Symisc Systems http://libcox.net/
 * Version 1.7
 * For additional information on licensing, redistribution of this file,
 * and for a DISCLAIMER OF ALL WARRANTIES please contact Symisc Systems via:
 *       licensing@symisc.net
 *       contact@symisc.net
 * or visit:
 *      http://libcox.net/
 */
/*
 * Copyright (C) 2014, 2015 Symisc Systems, S.U.A.R.L [M.I.A.G Mrad Chems Eddine
↪<chm@symisc.net>].
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
```

```
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY SYMISC SYSTEMS ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT, ARE DISCLAIMED.  IN NO EVENT SHALL SYMISC SYSTEMS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

27.1.11 PH7

`PH7` is an embeddable PHP engine from Symisc Systems.

Calling PHP programs from Unicon with PH7.

http://ph7.symisc.net/

A very similar Unicon setup:

```
#
# uniph7-v1.icn, trial run for PH7 integration
#
# tectonics:
#    gcc -o uniph7-v1.so -shared -fpic uniph7-v1.c ph7.c
#
procedure main()
    ph7 := loadfunc("./uniph7-v1.so", "uniph7")
    phpProg := "<?php _
        echo PHP_EOL.'Welcome, '.get_current_user().PHP_EOL;_
        echo 'System time is: '.date('Y-m-d H:i:s').PHP_EOL;_
        echo 'Running: '.substr(php_uname(),0,54).'...'.PHP_EOL;_
    ?>"
    ph7(phpProg)
end
```

The PH7 source is an SQLite style amalgamation bundle. Just include `ph7.c` in a build.

```
# PH7 (Embedded PHP)
uniph7-v1.so: uniph7-v1.c
> gcc -o uniph7-v1.so -shared -fpic uniph7-v1.c ph7.c \
  -Wno-unused -Wno-sign-compare

uniph7-v1: uniph7-v1.so uniph7-v1.icn
> unicon -s uniph7-v1.icn -x
```

The initial trial is a simple change to the PH7 example, `ph7_intro.c`.

```
--- programs/ph7_intro.c
+++ programs/uniph7-v1.c
@@ -29,11 +29,9 @@
   *        and you are running Microsoft Windows 7 localhost 6.1 build 7600 x86
   *
   */
-#define PHP_PROG "<?php "\
-                 "echo 'Welcome guest'.PHP_EOL;"\
-                 "echo 'Current system time is: '.date('Y-m-d H:i:s').PHP_EOL;"\
-                 "echo 'and you are running '.php_uname().PHP_EOL;"\
-                 "?>"
+
+/* PHP_PROG passed from Unicon */
+
 /* Make sure you have the latest release of the PH7 engine
   * from:
   *  http://ph7.symisc.net/downloads.html
@@ -42,6 +40,10 @@
 #include <stdlib.h>
 /* Make sure this header file is available.*/
 #include "ph7.h"
+
+/* Unicon loadfunc */
+#include "icall.h"
+
 /*
   * Display an error message and exit.
   */
@@ -78,7 +80,7 @@
 /*
   * Main program: Compile and execute the PHP program defined above.
   */
-int main(void)
+int uniph7(int argc, descriptor argv[])
 {
         ph7 *pEngine; /* PH7 engine */
         ph7_vm *pVm;  /* Compiled PHP program */
@@ -92,13 +94,17 @@
                   */
                 Fatal("Error while allocating a new PH7 engine instance");
         }
+
+        /* Get PHP program from Unicon */
+        ArgString(1)
+
         /* Compile the PHP test program defined above */
         rc = ph7_compile_v2(
-                pEngine,  /* PH7 engine */
-                PHP_PROG, /* PHP test program */
-                -1        /* Compute input length automatically*/,
-                &pVm,     /* OUT: Compiled PHP program */
-                0         /* IN: Compile flags */
+                pEngine,            /* PH7 engine */
+                StringVal(argv[1]), /* PHP test program */
+                -1                  /* Compute input length automatically*/,
+                &pVm,               /* OUT: Compiled PHP program */
+                0                   /* IN: Compile flags */
                 );
```

```
        if( rc != PH7_OK ){
                if( rc == PH7_COMPILE_ERR ){
```

A complete listing for clarity:

```
/*
 * Compile this file together with the ph7 engine source code to generate
 * the executable. For example:
 *  gcc -W -Wall -O6 -o ph7_test ph7_intro.c ph7.c
*/
/*
 * This simple program is a quick introduction on how to embed and start
 * experimenting with the PH7 engine without having to do a lot of tedious
 * reading and configuration.
 *
 * For an introduction to the PH7 C/C++ interface, please refer to this page
 *        http://ph7.symisc.net/api_intro.html
 * For the full C/C++ API reference guide, please refer to this page
 *        http://ph7.symisc.net/c_api.html
 */
/*
 * The following is the PHP program to execute.
 *   <?php
 *    echo 'Welcome guest'.PHP_EOL;
 *    echo 'Current system time is: '.date('Y-m-d H:i:s').PHP_EOL;
 *    echo 'and you are running '.php_uname();
 *   ?>
 * That is, this simple program when running should display a greeting
 * message, the current system time and the host operating system.
 * A typical output of this program would look like this:
 *
 *        Welcome guest
 *        Current system time is: 2012-09-14 02:08:44
 *        and you are running Microsoft Windows 7 localhost 6.1 build 7600 x86
 *
 */

/* PHP_PROG passed from Unicon */

/* Make sure you have the latest release of the PH7 engine
 * from:
 *  http://ph7.symisc.net/downloads.html
 */
#include <stdio.h>
#include <stdlib.h>
/* Make sure this header file is available.*/
#include "ph7.h"

/* Unicon loadfunc */
#include "icall.h"

/*
 * Display an error message and exit.
 */
static void Fatal(const char *zMsg)
{
        puts(zMsg);
        /* Shutdown the library */
```

```
        ph7_lib_shutdown();
        /* Exit immediately */
        exit(0);
}
/*
 * VM output consumer callback.
 * Each time the virtual machine generates some outputs, the following
 * function gets called by the underlying virtual machine  to consume
 * the generated output.
 * All this function does is redirecting the VM output to STDOUT.
 * This function is registered later via a call to ph7_vm_config()
 * with a configuration verb set to: PH7_VM_CONFIG_OUTPUT.
 */
static int Output_Consumer(const void *pOutput, unsigned int nOutputLen, void␣
↪*pUserData /* Unused */)
{
        /*
         * Note that it's preferable to use the write() system call to display the␣
↪output
         * rather than using the libc printf() which everybody now is extremely slow.
         */
        printf("%.*s",
                nOutputLen,
                (const char *)pOutput /* Not null terminated */
                );
        /* All done, VM output was redirected to STDOUT */
        return PH7_OK;
}
/*
 * Main program: Compile and execute the PHP program defined above.
 */
int uniph7(int argc, descriptor argv[])
{
        ph7 *pEngine; /* PH7 engine */
        ph7_vm *pVm;  /* Compiled PHP program */
        int rc;
        /* Allocate a new PH7 engine instance */
        rc = ph7_init(&pEngine);
        if( rc != PH7_OK ){
                /*
                 * If the supplied memory subsystem is so sick that we are unable
                 * to allocate a tiny chunk of memory, there is no much we can do␣
↪here.
                 */
                Fatal("Error while allocating a new PH7 engine instance");
        }

        /* Get PHP program from Unicon */
        ArgString(1)

        /* Compile the PHP test program defined above */
        rc = ph7_compile_v2(
                pEngine,            /* PH7 engine */
                StringVal(argv[1]), /* PHP test program */
                -1                  /* Compute input length automatically*/,
                &pVm,               /* OUT: Compiled PHP program */
                0                   /* IN: Compile flags */
                );
```

```
        if( rc != PH7_OK ){
                if( rc == PH7_COMPILE_ERR ){
                        const char *zErrLog;
                        int nLen;
                        /* Extract error log */
                        ph7_config(pEngine,
                                PH7_CONFIG_ERR_LOG,
                                &zErrLog,
                                &nLen
                                );
                        if( nLen > 0 ){
                                /* zErrLog is null terminated */
                                puts(zErrLog);
                        }
                }
                /* Exit */
                Fatal("Compile error");
        }
        /*
         * Now we have our script compiled, it's time to configure our VM.
         * We will install the VM output consumer callback defined above
         * so that we can consume the VM output and redirect it to STDOUT.
         */
        rc = ph7_vm_config(pVm,
                PH7_VM_CONFIG_OUTPUT,
                Output_Consumer,    /* Output Consumer callback */
                0                   /* Callback private data */
                );
        if( rc != PH7_OK ){
                Fatal("Error while installing the VM output consumer callback");
        }
        /*
         * And finally, execute our program. Note that your output (STDOUT in our␣
↪case)
         * should display the result.
         */
        ph7_vm_exec(pVm, 0);
        /* All done, cleanup the mess left behind.
        */
        ph7_vm_release(pVm);
        ph7_release(pEngine);
        return 0;
}
```

And a sample run:

```
prompt$ make -B --no-print-directory uniph7-v1
gcc -o uniph7-v1.so -shared -fpic uniph7-v1.c ph7.c \
  -Wno-unused -Wno-sign-compare
unicon -s uniph7-v1.icn -x

Welcome, btiffin
System time is: 2019-10-27 04:54:29
Running: Linux 4.4.0-166-generic #195-Ubuntu SMP Tue Oct 1 09:3...
```

There are some differences between the reference implementation of PHP and PH7, so large frameworks may not work, but small bits of PHP will, and the PH7 includes a foreign function interface to add features if required.

PH7 license obligation

```
/*
 * Copyright (C) 2011,2012 Symisc Systems. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Redistributions in any form must be accompanied by information on
 *    how to obtain complete source code for the PH7 engine and any
 *    accompanying software that uses the PH7 engine software.
 *    The source code must either be included in the distribution
 *    or be available for no more than the cost of distribution plus
 *    a nominal fee, and must be freely redistributable under reasonable
 *    conditions. For an executable file, complete source code means
 *    the source code for all modules it contains.It does not include
 *    source code for modules or files that typically accompany the major
 *    components of the operating system on which the executable file runs.
 *
 * THIS SOFTWARE IS PROVIDED BY SYMISC SYSTEMS ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT, ARE DISCLAIMED.  IN NO EVENT SHALL SYMISC SYSTEMS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

27.1.12 UnQLite

Another amalgam release from Symisc. This is a NoSQL database engine, with Jx9 scripting included (Jx9 is another Symisc software but included in the UnQLite distribution).

Similar build environment, include unqlite.c along with other sources to build a shared object file for use with Unicon *loadfunc*.

Aside: Being a COBOL programmer, and growing up on Vax/VMS, the term "NoSQL" is a sad state of word smithery. Key-value database would be better. Computers had ISAM and RMS and other indexed record management systems long before SQL became dominant, and the term NoSQL just shows a lack of educational history in the field of computer science. Unstructured Query Language is the new post-modern database paradigm, UnQL (pronounced Uncle) but records and keys is not "NoSQL". Rant over.

So here is a NoSQL data engine with UnQLite. A fairly unique blend of key-value store and document store.

First step is to see if it'll work.

```
#
# uniunql-v1.icn, Embed UnQLite in Unicon
#
# tectonics:
#    gcc -o uniunql-v1.so -shared -fpic uniunql-vi.c unqlite.c
#
procedure main()
    uniunql := loadfunc("./uniunql-v1.so", "uniunql")

    program := "/* Create the collection 'users'  */\n_
        if( !db_exists('users') ){\n_
           /* Try to create it */\n_
          $rc = db_create('users');\n_
          if ( !$rc ){\n_
            /*Handle error*/\n_
            print db_errlog();\n_
                  return;\n_
          }else{\n_
             print \"Collection 'users' successfuly created\n\";\n_
           }\n_
         }\n_
        /*The following is the records to be stored shortly in our collection*/ \n_
        $zRec = [\n_
        {\n_
          name : 'james',\n_
          age  : 27,\n_
          mail : 'dude@example.com'\n_
        },\n_
        {\n_
          name : 'robert',\n_
          age  : 35,\n_
          mail : 'rob@example.com'\n_
        },\n_
        {\n_
          name : 'monji',\n_
          age  : 47,\n_
          mail : 'monji@example.com'\n_
        },\n_
        {\n_
         name : 'barzini',\n_
         age  : 52,\n_
         mail : 'barz@mobster.com'\n_
        }\n_
        ];\n_
        /*Store our records*/\n_
        $rc = db_store('users',$zRec);\n_
        if( !$rc ){\n_
         /*Handle error*/\n_
         print db_errlog();\n_
         return;\n_
        }\n_
        /*Create our filter callback*/\n_
        $zCallback = function($rec){\n_
           /*Allow only users >= 30 years old.*/\n_
           if( $rec.age < 30 ){\n_
               /* Discard this record*/\n_
               return FALSE;\n_
           }\n_
```

```
            /* Record correspond to our criteria*/\n_
            return TRUE;\n_
         }; /* Dont forget the semi-colon here*/\n_
         /* Retrieve collection records and apply our filter callback*/\n_
         $data = db_fetch_all('users',$zCallback);\n_
         print \"Filtered records\n\";\n_
         /*Iterate over the extracted elements*/\n_
         foreach($data as $value){ /*JSON array holding the filtered records*/\n_
          print $value..JX9_EOL;\n_
         }"

    result := uniunql(":mem:", program)
    write("Unicon result: ", result)
end
```

The make rules

```
# UnQLite (embed a key value and document store engine in Unicon)
uniunql-v1.so: uniunql-v1.c
> gcc -o uniunql-v1.so -shared -fpic uniunql-v1.c unqlite.c \
  -Wno-unused

uniunql-v1: uniunql-v1.so uniunql-v1.icn
> unicon -s uniunql-v1.icn -x
```

A slightly modified `unqlite_doc_intro.c` for use with a *loadfunc* trial.

```
--- programs/unqlite_doc_intro.c
+++ programs/uniunql-v1.c
@@ -45,6 +45,7 @@
 #include <stdlib.h> /* exit() */
 /* Make sure this header file is available.*/
 #include "unqlite.h"
+#include "icall.h"
 /*
  * Banner.
  */
@@ -81,148 +82,30 @@
 }
 /* Forward declaration: VM output consumer callback */
 static int VmOutputConsumer(const void *pOutput,unsigned int nOutLen,void *pUserData␣
↪/* Unused */);
-/*
- * The following is the Jx9 Program to be executed later by the UnQLite VM:
- * This program store some JSON objects (a collections of dummy users) into
- * the collection 'users' stored in our database.
- * // Create the collection 'users'
- * if( !db_exists('users') ){
- *   // Try to create it
- *  $rc = db_create('users');
- *  if ( !$rc ){
- *    //Handle error
- *     print db_errlog();
- *          return;
- *  }
- * }
- * //The following is the records to be stored shortly in our collection
- * $zRec = [
```

```
- * {
- *  name : 'james',
- *  age  : 27,
- *  mail : 'dude@example.com'
- * },
- * {
- *  name : 'robert',
- *  age  : 35,
- *  mail : 'rob@example.com'
- * },
- *
- * {
- *  name : 'monji',
- *  age  : 47,
- *  mail : 'monji@example.com'
- * },
- * {
- * name : 'barzini',
- * age  : 52,
- * mail : 'barz@mobster.com'
- * }
- * ];
- *
- * //Store our records
- * $rc = db_store('users',$zRec);
- * if( !$rc ){
- *  //Handle error
- *  print db_errlog();
- *  return;
- * }
- * //Create our filter callback
- * $zCallback = function($rec){
- *   //Allow only users >= 30 years old.
- *   if( $rec.age < 30 ){
- *       // Discard this record
- *       return FALSE;
- *   }
- *   //Record correspond to our criteria
- *   return TRUE;
- * }; //Don't forget the semi-colon here
- *
- * //Retrieve collection records and apply our filter callback
- * $data = db_fetch_all('users',$zCallback);
- *
- * //Iterate over the extracted elements
- * foreach($data as $value){ //JSON array holding the filtered records
- * print $value..JX9_EOL;
- * }
- */
-#define JX9_PROG \
-"/* Create the collection 'users'  */"\
- "if( !db_exists('users') ){"\
- "   /* Try to create it */"\
- "  $rc = db_create('users');"\
- "  if ( !$rc ){"\
- "    /*Handle error*/"\
- "    print db_errlog();"\
- "          return;"\
```

```
- "  }else{"\
- "     print \"Collection 'users' successfuly created\\n\";"\
- "   }"\
- " }"\
- "/*The following is the records to be stored shortly in our collection*/ "\
- "$zRec = ["\
- "{"\
- "  name : 'james',"\
- "  age  : 27,"\
- "  mail : 'dude@example.com'"\
- "},"\
- "{"\
- "  name : 'robert',"\
- "  age  : 35,"\
- "  mail : 'rob@example.com'"\
- "},"\
- "{"\
- "  name : 'monji',"\
- "  age  : 47,"\
- "  mail : 'monji@example.com'"\
- "},"\
- "{"\
- " name : 'barzini',"\
- " age  : 52,"\
- " mail : 'barz@mobster.com'"\
- "}"\
- "];"\
- "/*Store our records*/"\
- "$rc = db_store('users',$zRec);"\
- "if( !$rc ){"\
- " /*Handle error*/"\
- " print db_errlog();"\
- " return;"\
- "}"\
- "/*Create our filter callback*/"\
- "$zCallback = function($rec){"\
- "   /*Allow only users >= 30 years old.*/"\
- "   if( $rec.age < 30 ){"\
- "       /* Discard this record*/"\
- "       return FALSE;"\
- "   }"\
- "   /* Record correspond to our criteria*/"\
- "   return TRUE;"\
- "}; /* Don't forget the semi-colon here*/"\
- "/* Retrieve collection records and apply our filter callback*/"\
- "$data = db_fetch_all('users',$zCallback);"\
- "print \"Filtered records\\n\";"\
- "/*Iterate over the extracted elements*/"\
- "foreach($data as $value){ /*JSON array holding the filtered records*/"\
- " print $value..JX9_EOL;"\
- "}"

-int main(int argc,char *argv[])
+int uniunql(int argc, descriptor argv[])
 {
        unqlite *pDb;       /* Database handle */
        unqlite_vm *pVm;    /* UnQLite VM resulting from successful compilation of␣
↪the target Jx9 script */
```

```
        int rc;

+        /* pass in the name of the data store, :mem: for in-memory */
+        ArgString(1)
+
+        /* Jx9 script as string */
+        ArgString(2)
+
         puts(zBanner);
+        fflush(stdout);

         /* Open our database */
-        rc = unqlite_open(&pDb,argc > 1 ? argv[1] /* On-disk DB */ : ":mem:" /* In-
↪mem DB */,UNQLITE_OPEN_CREATE);
+        rc = unqlite_open(&pDb,argc > 1 ? StringVal(argv[1]) /* On-disk DB */ :
↪":mem:" /* In-mem DB */,UNQLITE_OPEN_CREATE);
         if( rc != UNQLITE_OK ){
                 Fatal(0,"Out of memory");
         }

         /* Compile our Jx9 script defined above */
-        rc = unqlite_compile(pDb,JX9_PROG,sizeof(JX9_PROG)-1,&pVm);
+        rc = unqlite_compile(pDb, StringVal(argv[2]), strlen(StringVal(argv[2])),&
↪pVm);
         if( rc != UNQLITE_OK ){
                 /* Compile error, extract the compiler error log */
                 const char *zBuf;
@@ -232,19 +115,19 @@
                 if( iLen > 0 ){
                         puts(zBuf);
                 }
-                Fatal(0,"Jx9 compile error");
+                Fatal(0, "Jx9 compile error");
         }

         /* Install a VM output consumer callback */
         rc = unqlite_vm_config(pVm,UNQLITE_VM_CONFIG_OUTPUT,VmOutputConsumer,0);
         if( rc != UNQLITE_OK ){
-                Fatal(pDb,0);
+                Fatal(pDb, 0);
         }

         /* Execute our script */
         rc = unqlite_vm_exec(pVm);
         if( rc != UNQLITE_OK ){
-                Fatal(pDb,0);
+                Fatal(pDb, 0);
         }

         /* Release our VM */
@@ -252,7 +135,7 @@

         /* Auto-commit the transaction and close our database */
         unqlite_close(pDb);
-        return 0;
+        RetInteger(rc);
 }
```

```
 #ifdef __WINNT__
@@ -298,4 +181,4 @@

         /* All done, data was redirected to STDOUT */
         return UNQLITE_OK;
-}+}
```

A complete listing, for clarity

```
/*
 * Compile this file together with the UnQLite database engine source code
 * to generate the executable. For example:
 *  gcc -W -Wall -O6 unqlite_doc_intro.c unqlite.c -o unqlite_doc
*/
/*
 * This simple program is a quick introduction on how to embed and start
 * experimenting with UnQLite without having to do a lot of tedious
 * reading and configuration.
 *
 * Introduction to the UnQLite Document-Store Interfaces:
 *
 * The Document store to UnQLite which is used to store JSON docs (i.e. Objects,␣
↪Arrays, Strings, etc.)
 * in the database is powered by the Jx9 programming language.
 *
 * Jx9 is an embeddable scripting language also called extension language designed
 * to support general procedural programming with data description facilities.
 * Jx9 is a Turing-Complete, dynamically typed programming language based on JSON
 * and implemented as a library in the UnQLite core.
 *
 * Jx9 is built with a tons of features and has a clean and familiar syntax similar
 * to C and Javascript.
 * Being an extension language, Jx9 has no notion of a main program, it only works
 * embedded in a host application.
 * The host program (UnQLite in our case) can write and read Jx9 variables and can
 * register C/C++ functions to be called by Jx9 code.
 *
 * For an introduction to the UnQLite C/C++ interface, please refer to:
 *        http://unqlite.org/api_intro.html
 * For an introduction to Jx9, please refer to:
 *        http://unqlite.org/jx9.html
 * For the full C/C++ API reference guide, please refer to:
 *        http://unqlite.org/c_api.html
 * UnQLite in 5 Minutes or Less:
 *        http://unqlite.org/intro.html
 * The Architecture of the UnQLite Database Engine:
 *        http://unqlite.org/arch.html
 */
/* $SymiscID: unqlite_doc_intro.c v1.0 FreeBSD 2013-05-17 15:56 stable <chm@symisc.
↪net> $ */
/*
 * Make sure you have the latest release of UnQLite from:
 *  http://unqlite.org/downloads.html
 */
#include <stdio.h>  /* puts() */
#include <stdlib.h> /* exit() */
/* Make sure this header file is available.*/
#include "unqlite.h"
```

```
#include "icall.h"
/*
 * Banner.
 */
static const char zBanner[] = {
        "============================================================\n"
        "UnQLite Document-Store (Via Jx9) Intro                      \n"
        "                                          http://unqlite.org/\n"
        "============================================================\n"
};
/*
 * Extract the database error log and exit.
 */
static void Fatal(unqlite *pDb,const char *zMsg)
{
        if( pDb ){
                const char *zErr;
                int iLen = 0; /* Stupid cc warning */

                /* Extract the database error log */
                unqlite_config(pDb,UNQLITE_CONFIG_ERR_LOG,&zErr,&iLen);
                if( iLen > 0 ){
                        /* Output the DB error log */
                        puts(zErr); /* Always null termniated */
                }
        }else{
                if( zMsg ){
                        puts(zMsg);
                }
        }
        /* Manually shutdown the library */
        unqlite_lib_shutdown();
        /* Exit immediately */
        exit(0);
}
/* Forward declaration: VM output consumer callback */
static int VmOutputConsumer(const void *pOutput,unsigned int nOutLen,void *pUserData /
↪* Unused */);

int uniunql(int argc, descriptor argv[])
{
        unqlite *pDb;       /* Database handle */
        unqlite_vm *pVm;    /* UnQLite VM resulting from successful compilation of␣
↪the target Jx9 script */
        int rc;

        /* pass in the name of the data store, :mem: for in-memory */
        ArgString(1)

        /* Jx9 script as string */
        ArgString(2)

        puts(zBanner);
        fflush(stdout);

        /* Open our database */
        rc = unqlite_open(&pDb,argc > 1 ? StringVal(argv[1]) /* On-disk DB */ : ":mem:
↪" /* In-mem DB */,UNQLITE_OPEN_CREATE);
```

```
        if( rc != UNQLITE_OK ){
                Fatal(0,"Out of memory");
        }

        /* Compile our Jx9 script defined above */
        rc = unqlite_compile(pDb, StringVal(argv[2]), strlen(StringVal(argv[2])),&
↪pVm);
        if( rc != UNQLITE_OK ){
                /* Compile error, extract the compiler error log */
                const char *zBuf;
                int iLen;
                /* Extract error log */
                unqlite_config(pDb,UNQLITE_CONFIG_JX9_ERR_LOG,&zBuf,&iLen);
                if( iLen > 0 ){
                        puts(zBuf);
                }
                Fatal(0, "Jx9 compile error");
        }

        /* Install a VM output consumer callback */
        rc = unqlite_vm_config(pVm,UNQLITE_VM_CONFIG_OUTPUT,VmOutputConsumer,0);
        if( rc != UNQLITE_OK ){
                Fatal(pDb, 0);
        }

        /* Execute our script */
        rc = unqlite_vm_exec(pVm);
        if( rc != UNQLITE_OK ){
                Fatal(pDb, 0);
        }

        /* Release our VM */
        unqlite_vm_release(pVm);

        /* Auto-commit the transaction and close our database */
        unqlite_close(pDb);
        RetInteger(rc);
}

#ifdef __WINNT__
#include <Windows.h>
#else
/* Assume UNIX */
#include <unistd.h>
#endif
/*
 * The following define is used by the UNIX build process and have
 * no particular meaning on windows.
 */
#ifndef STDOUT_FILENO
#define STDOUT_FILENO        1
#endif
/*
 * VM output consumer callback.
 * Each time the UnQLite VM generates some outputs, the following
 * function gets called by the underlying virtual machine to consume
 * the generated output.
 *
```

```
 * All this function does is redirecting the VM output to STDOUT.
 * This function is registered via a call to [unqlite_vm_config()]
 * with a configuration verb set to: UNQLITE_VM_CONFIG_OUTPUT.
 */
static int VmOutputConsumer(const void *pOutput,unsigned int nOutLen,void *pUserData /
↪* Unused */)
{
#ifdef __WINNT__
        BOOL rc;
        rc = WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),pOutput,(DWORD)nOutLen,0,0);
        if( !rc ){
                /* Abort processing */
                return UNQLITE_ABORT;
        }
#else
        ssize_t nWr;
        nWr = write(STDOUT_FILENO,pOutput,nOutLen);
        if( nWr < 0 ){
                /* Abort processing */
                return UNQLITE_ABORT;
        }
#endif /* __WINT__ */

        /* All done, data was redirected to STDOUT */
        return UNQLITE_OK;
}
```

And a sample run:

```
prompt$ make -B --no-print-directory uniunql-v1
gcc -o uniunql-v1.so -shared -fpic uniunql-v1.c unqlite.c \
  -Wno-unused
unicon -s uniunql-v1.icn -x
============================================================
UnQLite Document-Store (Via Jx9) Intro
                                         http://unqlite.org/
============================================================

Collection 'users' successfuly created
Filtered records
{"name":"robert","age":35,"mail":"rob@example.com","__id":1}
{"name":"monji","age":47,"mail":"monji@example.com","__id":2}
{"name":"barzini","age":52,"mail":"barz@mobster.com","__id":3}
Unicon result: 0
```

A *JSON* document stored and then retrieved, filtered by `age > 30`, from `:mem:` in-memory storage.

A different (valid) filename passed from Unicon in `uniunql()` would create a disk persistent document store.

Next step will be a more capable Unicon binding.

Performance of UnQLite is impressive.

https://unqlite.org/

UnQLite license obligation

```
/*
 * Copyright (C) 2012, 2013 Symisc Systems, S.U.A.R.L [M.I.A.G Mrad Chems Eddine
↪<chm@symisc.net>].
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY SYMISC SYSTEMS ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT, ARE DISCLAIMED.  IN NO EVENT SHALL SYMISC SYSTEMS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

There is a visible conflict with this license and the Jx9 engine. Separated, Jx9 ships with more capabilities and a 3 clause license, comparable to the *vedis* and *PH7* licenses. Symisc has openly stated that the license intent with UnQLite is 2 clause, there is no obligation to produce source for all associated usage or work out a dual licensing contract when using the version of Jx9 that ships with UnQLite in closed source systems.

Not that Unicon is against strong copyleft freedoms, but the conflict is visible when inspecting the unqlite.c file when looking at the included jx9.h source file. Although it may be wise to treat UnQLite as a three clause system if you need to satisfy company attorneys, this forum post clarifies the author's intent

https://unqlite.org/forum/thread.php?file=can-i-use-the-jx9-with-unqlite-for-closed-source-project

```
...

So, even UnQLite uses a portion of the Jx9 core to implement it's document
storage engine, everything is covered by the UnQLite BSD license as far
you do not embed the entire Jx9 library (I mean here the independent
engine available here http://jx9.smisc.net) in your commercial software.
```

Or, just ship all the sources when using UnQLite and avoid any and all potential issues.

27.1.13 REXX

Restructured Extended Executor as Open Object Rexx, embedded in Unicon via *loadfunc*. Rexx was originally designed and implemented from 1979 to 1982 by Mike Cowlishaw. Rexx is a close relative to Icon, age wise. A version of Object Rexx was released by IBM as free software in 2004. That spawned Open Object Rexx, which is used here for the demonstration.

Regina Rexx would also work for classic Rexx, and may make more sense for `loadfunc`, being a solid C build environment, but ooRexx has some pretty nifty features and is keeping the C heritage available while the team builds out the new C++ API.

```
#
# unirexx.icn, Invoke ooRexx from Unicon, with C and C++ interfaces
#
# tectonics:
#     gcc -o unirexx.so -shared -fPIC unirexx.c -lrexx -lrexxapi
#     g++ -o oorexx.so -shared -fPIC oorexx.c -lrexx -lrexxapi
#
procedure main()
    unirexx := loadfunc("./unirexx.so", "unirexx")
    result := unirexx("hello.rexx")
    write("Unicon RexxStart from file = " || result)

    write()
    result := unirexx("[PARSE SOURCE data]",
                      "say \"Hello, from Rexx in Unicon\";" ||
                      " PARSE SOURCE a; return a")
    write("Unicon RexxStart from string = " || result)

    write("\n--------\nC++ API")
    oorexx := loadfunc("./oorexx.so", "oorexx")
    result := oorexx("hello.rexx")
    write("Unicon ooRexx C++ API rc = " || result)
end
```

The main ooRexx API is now a C++ implementation, but there is a classic interface based on C. Both are tested here.

The classic API.

```
/* Unicon integration with Open Object Rexx, classic C API */
/* tectonics: gcc -o unirexx.so -shared -fPIC unirexx.c */
#include <stdio.h>
#include <rexx.h>
#include "icall.h"
int
unirexx(int argc, descriptor argv[])
{
    int rc;

    /* RexxStart fields */
    size_t ArgCount = 0;
    PCONSTRXSTRING ArgList = NULL;
    const char *ProgramName;
    PRXSTRING PassStore;
    RXSTRING Instore[2];
    const char *EnvName = NULL;
    int CallType = RXCOMMAND;
    PRXSYSEXIT Exits = NULL;
    short ReturnCode;

    RXSTRING Result;
    char returnBuffer[256];

    /* Need a Rexx ProgramName and/or Instore evaluation string */
    fprintf(stderr, "argc: %d\n", argc);
    fflush(stderr);
```

```
    if (argc < 1) Error(105);  /* Need a filename or PARSE SOURCE */

    ArgString(1);

    /* Second string is optional, will be text to interpret */
    if (argc > 1) ArgString(2);

    /* Instore is two Rexx string descriptors */
    /*   one for the text and second for precompiled image */
    /*   no precompiled image is used here */
    RXNULLSTRING(Instore[0]);
    RXNULLSTRING(Instore[1]);
    if (argc > 1) {
        MAKERXSTRING(Instore[0], StringVal(argv[2]), StringLen(argv[2]));
        PassStore = &Instore[0];
    } else {
        /* If only the file name is passed, Instore is NULL */
        PassStore = NULL;
    }

    /* set up initial Result string space, Rexx may allocate its own */
    MAKERXSTRING(Result, returnBuffer, sizeof(returnBuffer));

    rc = RexxStart(ArgCount, ArgList, StringVal(argv[1]), PassStore,
                   EnvName, CallType, Exits, &ReturnCode, &Result);

    fprintf(stderr, "RexxStart rc: %d\nRexx ReturnCode: %d, Result: %s\n",
            rc, ReturnCode, RXSTRPTR(Result));
    fflush(stderr);

    /* A RetStringN, but Rexx may need to free the space */
    argv[0].dword = Result.strlength;
    argv[0].vword.sptr = alcstr(RXSTRPTR(Result), RXSTRLEN(Result));

    /* Rexx may have decided to allocate a return result space */
    if (RXSTRPTR(Result) != returnBuffer) {
        RexxFreeMemory(RXSTRPTR(Result));
    }

    Return;
}
```

The C++ API, with a slightly simpler Unicon interface. This is also testing whether Unicon *loadfunc* can manage C++ (which it does seem to, at least for the initial trials).

```
/* Unicon integration with Open Object Rexx, C++ API sample */
/* tectonics: g++ -o oorexx.so -shared -fPIC oorexx.cpp -lrexx -lrexxapi */

#include <stdio.h>
#include <oorexxapi.h>

bool checkForCondition(RexxThreadContext *c, bool clear);

extern "C" {
#include "icall.h"

int
```

```
oorexx(int argc, descriptor argv[])
{
    RexxInstance *interpreter;
    RexxThreadContext *threadContext;
    RexxOption *options = NULL;
    int rc;

    /* Create a Rexx Interpreter */
    rc = RexxCreateInterpreter(&interpreter, &threadContext, options);
    fprintf(stderr, "rc = %d\n", rc);
    fflush(stderr);

    if (rc == 0) {
        fprintf(stderr, "Failed to create Rexx interpreter\n");
        exit(1);
    }

    /* Expect program name from Unicon */
    ArgString(1);

    /* Call a program */
    RexxArrayObject args = NULL;
    RexxObjectPtr result = threadContext->CallProgram(StringVal(argv[1]),
                                                     args);

    /* See if any conditions were raised */
    if (threadContext->CheckCondition()) {
        checkForCondition(threadContext, true);
    } else {
        if (result != NULLOBJECT) {
            fprintf(stderr, "\nProgram result = %s\n\n",
                   threadContext->ObjectToStringValue(result));
            fflush(stderr);
        }
    }

    /* this test just returns an integer code */
    interpreter->Terminate();
    RetInteger(rc);
}
} /* end extern C */

/* Support routines */
inline wholenumber_t conditionSubCode(RexxCondition *condition)
{
    return (condition->code - (condition->rc * 1000));
}

void standardConditionMsg(RexxThreadContext *c,
                          RexxDirectoryObject condObj,
                          RexxCondition *condition)
{
    RexxObjectPtr list = c->SendMessage0(condObj, "TRACEBACK");
    if ( list != NULLOBJECT )
    {
        RexxArrayObject a = (RexxArrayObject)c->SendMessage0(list,
                                                            "ALLITEMS");
        if ( a != NULLOBJECT )
```

```
        {
            size_t count = c->ArrayItems(a);
            for ( size_t i = 1; i <= count; i++ )
            {
                RexxObjectPtr o = c->ArrayAt(a, i);
                if ( o != NULLOBJECT )
                {
                    fprintf(stderr, "%s\n", c->ObjectToStringValue(o));
                    fflush(stderr);
                }
            }
        }
    }
    fprintf(stderr, "Error %d running %s line %ld: %s\n", (int)condition->rc,
           c->CString(condition->program), condition->position,
           c->CString(condition->errortext));

    fprintf(stderr, "Error %d.%03d:  %s\n", (int)condition->rc,
           (int)conditionSubCode(condition),
           c->CString(condition->message));
    fflush(stderr);
}

bool checkForCondition(RexxThreadContext *c, bool clear)
{
    if ( c->CheckCondition() )
    {
        RexxCondition condition;
        RexxDirectoryObject condObj = c->GetConditionInfo();

        if ( condObj != NULLOBJECT )
        {
            c->DecodeConditionInfo(condObj, &condition);
            standardConditionMsg(c, condObj, &condition);

            if ( clear )
            {
                c->ClearCondition();
            }
            return true;
        }
    }
    return false;
}
```

The build rules are not complicated.

```
# ooRexx integration (classic RexxStart interface)
unirexx.so: unirexx.c
> gcc -o unirexx.so -shared -fPIC unirexx.c -lrexx -lrexxapi

oorexx.so: oorexx.cpp
> g++ -o oorexx.so -shared -fPIC oorexx.cpp -lrexx -lrexxapi

unirexx: unirexx.icn unirexx.so oorexx.so
> unicon -s unirexx.icn -x
```

And the sample run. A single Unicon program tests both API implementation style.

```
prompt$ make -B unirexx
make[1]: Entering directory '/home/btiffin/wip/writing/unicon/programs'
gcc -o unirexx.so -shared -fPIC unirexx.c -lrexx -lrexxapi
g++ -o oorexx.so -shared -fPIC oorexx.cpp -lrexx -lrexxapi
unicon -s unirexx.icn -x
argc: 1
Hello, world
RexxStart rc: 0
Rexx ReturnCode: 0, Result: hello.rexx return value (string)
Unicon RexxStart from file = hello.rexx return value (string)

argc: 2
Hello, from Rexx in Unicon
RexxStart rc: 0
Rexx ReturnCode: 0, Result: LINUX COMMAND [PARSE SOURCE data]
Unicon RexxStart from string = LINUX COMMAND [PARSE SOURCE data]

--------
C++ API
rc = 1002
Hello, world

Program result = hello.rexx return value (string)

Unicon ooRexx C++ API rc = 1002
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/programs'
```

Open Object Rexx is available on SourceForge at

https://sourceforge.net/projects/oorexx/

This is 4.2, ooRexx has started in on 5.0 beta releases.

27.1.14 Internationalization and Localization

i18n/L10n

The GNU project provides very extensive localization tools. `gettext` being one of the main C functions provided to allow for runtime human language translations based on *Locale*.

```
/*
 unicon-i18n.c, call gettext for translations
 tectonics:
     gcc -o unicon-i18n.so -shared -fpic unicon-i18n.c
*/

#include <libintl.h>
#include <locale.h>
#include "icall.h"

/*
 translate the first string argument from Unicon
*/
int
translate(int argc, descriptor *argv)
{
```

```
    char *trans;

    /* Need a string argument */
    if (argc < 1) Error(500);
    ArgString(1);

    /* attempt translation */
    trans = gettext(StringVal(argv[1]));

    /* return translation, or original */
    if (trans) RetString(trans);
    RetString(StringVal(argv[1]));
}

/*
 set up for translations
*/
int
initlocale(int argc, descriptor *argv)
{
    /* Need a domain and a locale root directory */
    if (argc < 2) Error(500);
    ArgString(1);
    ArgString(2);

    /* set up according to environment variables */
    setlocale(LC_ALL, "");
    bindtextdomain(StringVal(argv[1]), StringVal(argv[2]));
    textdomain(StringVal(argv[1]));

    /* return nothing */
    RetNull();
}
```

```
#
# unicon-i18n.icn, demonstrate GNU gettext locale translation
#
# tectonics: gcc -o unicon-i18n.so -shared -fpic unicon-i18n.c
#
link printf
procedure main()
    write("Test error message about invalid width in environment")
    initlocale("coreutils", "/usr/share")
    write(printf(_("ignoring invalid width in environment _
                    variable COLUMNS: %s"), -4))

    write("\nTest message about write error")
    write(_("write error"))

    write("\nTest message about specifying fields")
    write(_("you must specify a list of bytes, characters, or fields"))

    write("\nTest messages about invalid pattern and regex")
    write(printf(_("%s: invalid pattern"), &progname || ":" || &line))
    write(printf(_("%s: invalid regular expression: %s"),
        &progname || ":" || &line, "["))
end
```

```
#
# i18n/L10n setup
#
procedure initlocale(domain, localedir)
    &error +:= 1
    initlocale := loadfunc("./unicon-i18n.so", "initlocale") | nolocale
    &error -:= 1
    return initlocale(domain, localedir)
end
procedure nolocale(domain, localedir)
    return &null
end

#
# i18n/L10n
#
procedure _(text:string)
    &error +:= 1
    _ := loadfunc("./unicon-i18n.so", "translate") | _none
    &error -:= 1
    return _(text)
end
procedure _none(text:string)
    return text
end
```

With a sample run (using messages from GNU coretils and Spanish translations)

```
prompt$ gcc -o unicon-i18n.so -shared -fpic unicon-i18n.c
```

Using local default locale, *English*

```
prompt$ unicon -s unicon-i18n.icn -x
Test error message about invalid width in environment
ignoring invalid width in environment variable COLUMNS: -4

Test message about write error
write error

Test message about specifying fields
you must specify a list of bytes, characters, or fields

Test messages about invalid pattern and regex
unicon-i18n:26: invalid pattern
unicon-i18n:28: invalid regular expression: [
```

Using a Spanish language locale setting

```
prompt$ LC_ALL="es_ES.UTF-8" LANG="spanish" LANGUAGE="spanish" ./unicon-i18n
Test error message about invalid width in environment
se descarta el ancho inválido de la variable de entorno COLUMNS: -4

Test message about write error
error de escritura

Test message about specifying fields
se debe indicar una lista de bytes, caracteres o campos
```

```
Test messages about invalid pattern and regex
./unicon-i18n:26: plantilla inválida
./unicon-i18n:28: la expresión regular no es válida: [
```

27.1.15 libsoldout markdown

A loadable function to process Markdown into HTML. libsoldout also ships with example renderers for LaTeX and man page outputs. Simple Markdown, and extended Discount and soldout features included.

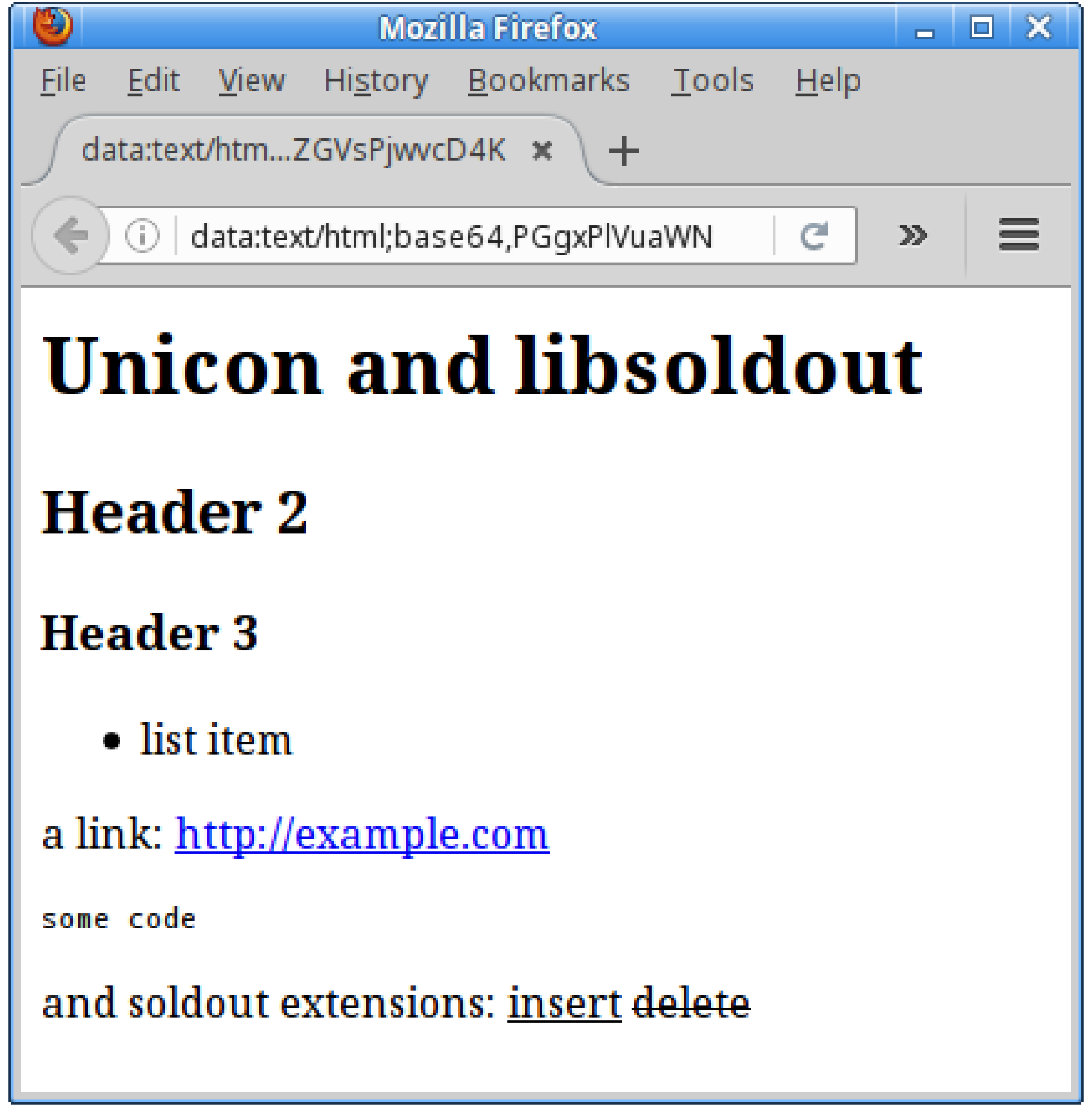
```
/*
 libsoldout markdown processor from Unicon
*/
#include <stdio.h>
#include <soldout/markdown.h>
#include <soldout/renderers.h>

#include "icall.h"

int
soldout(int argc, descriptor argv[])
{
    struct buf *ib, *ob;
    descriptor d;

    /* Need a string of markdown */
    if (argc < 1) Error(103);
    ArgString(1);

    /* set up input and output buffers */
    ib = bufnew(strlen(StringVal(argv[1])));
    ob = bufnew(1024);
    bufputs(ib, StringVal(argv[1]));

    markdown(ob, ib, &nat_html);
    Protect(StringAddr(d) = alcstr(ob->data, ob->size), Error(306));
    argv->dword = (int)ob->size;
    argv->vword.sptr = StringAddr(d);
    //RetStringN(ob->data, (int)ob->size);

    bufrelease(ib);
    bufrelease(ob);
    return 0;
}
```

```
#
# soldout.icn, a loadable Markdown to HTML demonstration
#
# tectonics:
#     gcc -o soldout.so -shared -fpic soldout.c -lsoldout
#
link base64

# any arguments will trigger firefox with the HTML output
procedure main(argv)
```

```
    markdown := "_
        Unicon and libsoldout\n_
        =====================\n_
        \n_
        ## Header 2\n_
        ### Header 3\n_
        - list item\n\n_
        a link: <http://example.com>\n\n_
        `some code`\n\n_
        and soldout extensions: ++insert++ --delete--"
    write(markdown)

    # load up the soldout library function
    soldout := loadfunc("./soldout.so", "soldout")

    # convert the Markdown and show the HTML
    markup := soldout(markdown)
    write("\n---HTML markup follows---\n")
    write(markup)

    # base64 encode the result and pass through a data url for browsing
    if \argv[1] then system("firefox \"data:text/html;base64, " ||
                            base64encode(markup) || "\" &")
end
```

And a sample run:

```
prompt$ unicon -s soldout.icn -x
Unicon and libsoldout
=====================

## Header 2
### Header 3
- list item

a link: <http://example.com>

`some code`

and soldout extensions: ++insert++ --delete--

---HTML markup follows---

<h1>Unicon and libsoldout</h1>

<h2>Header 2</h2>

<h3>Header 3</h3>

<ul>
<li>list item</li>
</ul>

<p>a link: <a href="http://example.com">http://example.com</a></p>

<p><code>some code</code></p>

<p>and soldout extensions: <ins>insert</ins> <del>delete</del></p>
```



27.1.16 ie modified for readline

`ie`, the Icon Evaluator, part of the *IPL* and built along with Unicon, is a handy utility for trying out Unicon expressions in an interactive shell. Nicer when the commands can be recalled. This example integrates GNU `readline` into `ie`.

Note: You could also use `rlwrap ie` to get the same effect.

```
/*
 unireadline.c, add readline powers to Unicon
 tectonics: gcc -o unireadline.so -shared -fpic unireadline.c

 Unicon usage: readline := loadfunc("./unireadline", "unirl")
```

```
 Dedicated to the public domain

 Date: September 2016
 Modified: 2016-09-07/04:53-0400
*/

#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>
#include "icall.h"

int
unirl(int argc, descriptor *argv)
{
    static char *line;

    /* need the string prompt */
    if (argc < 1) Error(500);
    ArgString(1);

    /* if line already allocated, free it */
    if (line) {
        free(line);
        line = (char *)NULL;
    }

    /* call readline with prompt */
    line = readline(StringVal(argv[1]));

    /* fail when no line read (EOF for instance) or save and return */
    if (!line) Fail;
    if (*line) add_history(line);
    RetString(line);
}
```

The changes to `ie` are minor. In the sources from `uni/prog/ie.icn`, change

```
writes(if *line = 0 then "][ " else "... ")
inline := (read()|stop())
```

to

```
inline := (readline(if *line = 0 then "uni> " else "... ")|stop())
```

and add

```
procedure reader(prompt)
    writes(prompt)
    return read()
end

procedure readline(prompt)
    &error +:= 1
    readline := loadfunc("./unireadline.so", "unirl") | reader
    &error -:= 1
    return readline(prompt)
```

```
end
```

Then recompile ie.

```
prompt$ gcc -o unireadline.so -shared -fpic unireadline -lreadline
prompt$ unicon ie.icn
prompt$ cp ie unireadline.so [INSTALL-DIR]/bin/
```

After that, when you run ie, you will have readline command recall available. *Assuming* readline *is installed*. If readline is not installed, you will get the old interface of *read*. To properly compile unireadline.c, you will need the GNU readline development headers installed on your system.

27.1.17 SNOBOL4

A short program to run SNOBOL4 programs with a pipe, and display any OUTPUT.

Some of the test programs that ship with SNOBOL4 are included, to highlight how complete the distribution is:

https://sourceforge.net/projects/snobol4/

Run *SNOBOL* files passed as arguments. This is a very lightweight program, results are simply written to *&output*. Much more could be done with the **snobol4** OUTPUT = data.

```
#
# snobol.icn, run snobol4 programs
#
# Requires snobol4
#
$define VERSION 0.1

link options

procedure main(argv)
    opts := options(argv, "-h! -v! -source!", optError)
    if \opts["h"] then return showHelp()
    if \opts["v"] then return showVersion()
    if \opts["source"] then return showSource()

    # run the rest of the arguments as snobol4 files
    every arg := !argv do snobol(arg)
end

#
# Run a snobol4 program, trim formfeeds
#
procedure snobol(filename)
    local sf
    sf := open("snobol4 " || filename, "p") | stop("no snobol4")
    while write(trim(read(sf), '\f', 0))
end

#
# show help, version and source info
#
procedure showVersion()
    write(&errout, &progname, " ", VERSION, " ", __DATE__)
```

```
end

procedure showHelp()
    showVersion()
    write(&errout, "Usage: snobol [opts] files...")
    write(&errout, "\t-h\tshow this help")
    write(&errout, "\t-v\tshow version")
    write(&errout, "\t-source\tlist source code")
    write(&errout)
    write(&errout, "all other arguments are run as SNOBOL4 sources")
end

procedure showSource()
    local f
    f := open(&file, "r") | stop("Source file ", &file, " unavailable")
    every write(!f)
    close(f)
end

#
# options error
#
procedure optError(s)
    write(&errout, s)
    stop("Try ", &progname, " -h for more information")
end
```

And a sample run, with `spitbol` based diagnostics and a hello:

```
prompt$ unicon -s snobol.icn -x hello.sno diag[12].sno
hello world
************************************************
**** s n o b o l      d i a g n o s t i c s ****
****          p h a s e    o n e            ****
************************************************
****  any trace output indicates an error   ****
************************************************
************************************************
**** n o     e r r o r s    d e t e c t e d ****
**** e n d    o f     d i a g n o s t i c s ****
************************************************
Dump of variables at termination

Natural variables

A = ARRAY('3')
AA = 'a'
AAA = ARRAY('10')
ABORT = PATTERN
AMA = ARRAY('2,2,2,2')
ARB = PATTERN
ATA = TABLE(2,10)
B = NODE
BAL = PATTERN
BB = 'b'
C = CLUNK
CC = 'c'
D = ARRAY('-1:1,2')
```

```
DIAGNOSTICS = 0
E = 'e'
EXPR = EXPRESSION
F = 'f'
FAIL = PATTERN
FENCE = PATTERN
FEXP = EXPRESSION
OUTPUT = '************************************************'
Q = 'qqq'
QQ = 'x'
REM = PATTERN
SEXP = EXPRESSION
STARS = '  error detected          ***'
SUCCEED = PATTERN
T = TABLE(10,10)
TA = ARRAY('2,2')

Unprotected keywords

&ABEND = 0
&ANCHOR = 0
&CASE = 1
&CODE = 0
&DUMP = 2
&ERRLIMIT = 999
&FILL = ' '
&FTRACE = 0
&FULLSCAN = 0
&GTRACE = 0
&INPUT = 1
&MAXLNGTH = 4294967295
&OUTPUT = 1
&STLIMIT = -1
&TRACE = 1000000
&TRIM = 0
**********************************************
**** snobol  diagnostics -- phase two     ****
**********************************************
****           &fullscan = 0              ****
****** error detected at 67 ********
***** resuming execution *******
**********************************************
****           &fullscan = 1              ****
****           no errors detected         ****
**********************************************
****           end of diagnostics         ****
**********************************************
Dump of variables at termination

Natural variables

ABORT = PATTERN
ARB = PATTERN
BAL = PATTERN
ERRCOUNT = 0
FAIL = PATTERN
FENCE = PATTERN
OUTPUT = '**********************************************'
```

```
REM = PATTERN
SUCCEED = PATTERN
TEST = 'abcdefghijklmnopqrstuvwxyz'
VAR = 'abc'
VARA = 'i'
VARD = 'abc'
VARL = 'abc'
VART = 'abc'

Unprotected keywords

&ABEND = 0
&ANCHOR = 0
&CASE = 1
&CODE = 0
&DUMP = 2
&ERRLIMIT = 99
&FILL = ' '
&FTRACE = 0
&FULLSCAN = 1
&GTRACE = 0
&INPUT = 1
&MAXLNGTH = 4294967295
&OUTPUT = 1
&STLIMIT = -1
&TRACE = 1000
&TRIM = 0
```

The SNOBOL sources are from the SNOBOL4 distribution `test/` directory downloaded from SourceForge. *Tabs replaced with spaces at tab stop 8.*

```
        OUTPUT = 'hello world'
END
```

```
*-title snobol test program #1 -- diagnostics phase one
*
*         this is a standard test program for spitbol which tests
*         out functions, operators and datatype manipulations
*
         &dump = 2
         trace(.test)
         &trace = 1000000
         stars =                     '  error detected          ***'
         &errlimit = 1000
         setexit(.errors)
         output = '************************************************'
         output = '**** s n o b o l      d i a g n o s t i c s ****'
         output = '****          p h a s e    o n e            ****'
         output = '************************************************'
         output = '****  any trace output indicates an error   ****'
         output = '************************************************'
-eject
*
*         test replace function
*
         test = differ(replace('axxbyyy','xy','01'),'a00b111') stars
         a = replace(&alphabet,'xy','ab')
```

```
         test = differ(replace('axy',&alphabet,a),'aab') stars
*
*        test convert function
*
         test = differ(convert('12','integer') , 12) stars
         test = differ(convert(2.5,'integer'),2)       stars
         test = differ(convert(2,'real'),2.0) stars
         test = differ(convert('.2','real'),0.2) stars
*
*        test datatype function
*
         test = differ(datatype('jkl'),'STRING') stars
         test = differ(datatype(12),'INTEGER') stars
         test = differ(datatype(1.33),'REAL') stars
         test = differ(datatype(null),'STRING') stars
-eject
*
*        test arithmetic operators
*
         test = differ(3 + 2,5) stars
         test = differ(3 - 2,1) stars
         test = differ(3 * 2,6) stars
         test = differ(5 / 2,2) stars
         test = differ(2 ** 3,8) stars
         test = differ(3 + 1,4) stars
         test = differ(3 - 1,2) stars
         test = differ('3' + 2,5) stars
         test = differ(3 + '-2',1) stars
         test = differ('1' + '0',1) stars
         test = differ(5 + null,5) stars
         test = differ(-5,0 - 5) stars
         test = differ(+'4',4) stars
         test = differ(2.0 + 3.0,5.0) stars
         test = differ(3.0 - 1.0,2.0) stars
         test = differ(3.0 * 2.0,6.0) stars
         test = differ(3.0 / 2.0,1.5) stars
         test = differ(3.0 ** 3,27.0) stars
         test = differ(-1.0,0.0 - 1.0) stars
*
*        test mixed mode
*
         test = differ(1 + 2.0,3.0) stars
         test = differ(3.0 / 2,1.5) stars
-eject
*
*        test functions
*
*        first, a simple test of a factorial function
*
         define('fact(n)')                  :(factend)
fact     fact = eq(n,1) 1          :s(return)
         fact = n * fact(n - 1)             :(return)
factend  test = ne(fact(5),120) stars
         test = differ(opsyn(.facto,'fact')) stars
         test = differ(facto(4),24) stars
*
*        see if alternate entry point works ok
*
```

```
         define('fact2(n)',.fact2ent)       :(fact2endf)
fact2ent fact2 = eq(n,1) 1         :s(return)
         fact2 = n * fact2(n - 1) :(return)
fact2endf output = ne(fact(6),720) stars
*
*        test function redefinition and case of argument = func name
*
         test = differ(define('fact(fact)','fact3')) stars
.                                           :(fact2end)
fact3    fact = ne(fact,1) fact * fact(fact - 1)
.                                           :(return)
fact2end
         test = ne(fact(4),24) stars
*
*        test out locals
*
         define('lfunc(a,b,c)d,e,f')        :(lfuncend)
lfunc    test = ~(ident(a,'a') ident(b,'b') ident(c,'c')) stars
         test = ~(ident(d) ident(e) ident(f)) stars
         a = 'aa' ; b = 'bb' ; c = 'cc' ; d = 'dd' ; e = 'ee' ; f = 'ff'
.                                 :(return)
lfuncend aa = 'a' ; bb = 'b' ; cc = 'c'
         d = 'd' ; e = 'e' ; f = 'f'
         a = 'x' ; b = 'y' ; c = 'z'
         test = differ(lfunc(aa,bb,cc)) stars
         test = ~(ident(a,'x') ident(b,'y') ident(c,'z')) stars
         test = ~(ident(aa,'a') ident(bb,'b') ident(cc,'c')) stars
         test = ~(ident(d,'d') ident(e,'e') ident(f,'f')) stars
*
*        test nreturn
*
         define('ntest()')                  :(endntest)
ntest    ntest = .a                         :(nreturn)
endntest a = 27
         test = differ(ntest(),27) stars    :f(st59) ;st59
         ntest() = 26                       :f(st60) ;st60
         test = differ(a,26) stars
-eject
*
*        continue test of functions
*
*
*        test failure return
*
         define('failure()')                :(failend)
failure                           :(freturn)
failend  test = failure() stars
-eject
*
*        test opsyn for operators
*
         opsyn('@',.dupl,2)
         opsyn('|',.size,1)
         test = differ('a' @ 4,'aaaa') stars
         test = differ(|'string',6) stars
*
*        test out array facility
*
```

```
         a = array(3)
         test = differ(a<1>) stars
         a<2> = 4.5
         test = differ(a<2>,4.5) stars
         test = ?a<4> stars
         test = ?a<0> stars
         test = differ(prototype(a),'3') stars
         b = array(3,10)
         test = differ(b<2>,10) stars
         b = array('3')
         b<2> = 'a'
         test = differ(b<2>,'a') stars
         c = array('2,2')
         c<1,2> = '*'
         test = differ(c<1,2>,'*') stars
         test = differ(prototype(c),'2,2') stars
         d = array('-1:1,2')
         d<-1,1> = 0
         test = differ(d<-1,1>,0) stars
         test = ?d<-2,1> stars
         test = ?d<2,1> stars
-eject
*
*        test program defined datatype functions
*
         data('node(val,lson,rson)')
         a = node('x','y','z')
         test = differ(datatype(a),'NODE') stars
         test = differ(val(a),'x') stars
         b = node()
         test = differ(rson(b)) stars
         lson(b) = a
         test = differ(rson(lson(b)),'z') stars
         test = differ(value('b'),b) stars
*
*        test multiple use of field function name
*
         data('clunk(value,lson)')
         test = differ(rson(lson(b)),'z') stars
         test = differ(value('b'),b) stars
         c = clunk('a','b')
         test = differ(lson(c),'b') stars
-eject
*
*        test numerical predicates
*
         test = lt(5,4) stars
         test = lt(4,4) stars
         test = ~lt(4,5) stars
         test = le(5,2) stars
         test = ~le(4,4) stars
         test = ~le(4,10) stars
         test = eq(4,5) stars
         test = eq(5,4) stars
         test = ~eq(5,5) stars
         test = ne(4,4) stars
         test = ~ne(4,6) stars
         test = ~ne(6,4) stars
```

```
        test = gt(4,6) stars
        test = gt(4,4) stars
        test = ~gt(5,2) stars
        test = ge(5,7) stars
        test = ~ge(4,4) stars
        test = ~ge(7,5) stars
        test = ne(4,5 - 1) stars
        test = gt(4,3 + 1) stars
        test = le(20,5 + 6) stars
        test = eq(1.0,2.0) stars
        test = gt(-2.0,-1.0) stars
        test = gt(-3.0,4.0) stars
        test = ne('12',12) stars
        test = ne('12',12.0) stars
        test = ~convert(bal,'pattern') stars
-eject
*
*       test integer
*
        test = integer('abc') stars
        test = ~integer(12) stars
        test = ~integer('12') stars
*
*       test size
*
        test = ne(size('abc'),3) stars
        test = ne(size(12),2) stars
        test = ne(size(null),0) stars
*
*       test lgt
*
        test = lgt('abc','xyz') stars
        test = lgt('abc','abc') stars
        test = ~lgt('xyz','abc') stars
        test = lgt(null,'abc') stars
        test = ~lgt('abc',null) stars
*
*       test indirect addressing
*
        test = differ($'bal',bal) stars
        test = differ($.bal,bal) stars
        $'qq' = 'x'
        test = differ(qq,'x') stars
        test = differ($'garbage') stars
        a = array(3)
        a<2> = 'x'
        test = differ($.a<2>,'x') stars
*
*       test concatenation
*
        test = differ('a' 'b','ab')         stars
        test = differ('a' 'b' 'c','abc') stars
        test = differ(1 2,'12') stars
        test = differ(2 2 2,'222') stars
        test = differ(1 3.4,'13.4') stars
        test = differ(bal null,bal)         stars
        test = differ(null bal,bal) stars
-eject
```

```
*
*        test remdr
*
         test = differ(remdr(10,3),1) stars
         test = differ(remdr(11,10),1) stars
*
*        test dupl
*
         test = differ(dupl('abc',2),'abcabc') stars
         test = differ(dupl(null,10),null) stars
         test = differ(dupl('abcdefg',0),null) stars
         test = differ(dupl(1,10),'1111111111')  stars
*
*        test table facility
*
         t = table(10)
         test = differ(t<'cat'>) stars
         t<'cat'> = 'dog'
         test = differ(t<'cat'>,'dog')   stars
         t<7> = 45
         test = differ(t<7>,45)   stars
         test = differ(t<'cat'>,'dog')  stars
         ta = convert(t,'array')
         test = differ(prototype(ta),'2,2') stars
         ata = convert(ta,'table')
         test = differ(ata<7>,45) stars
         test = differ(ata<'cat'>,'dog') stars
*
*        test item function
*
         aaa = array(10)
         item(aaa,1) = 5
         test = differ(item(aaa,1),5) stars
         test = differ(aaa<1>,5) stars
         aaa<2> = 22
         test = differ(item(aaa,2),22) stars
         ama = array('2,2,2,2')
         item(ama,1,2,1,2) = 1212
         test = differ(item(ama,1,2,1,2),1212) stars
         test = differ(ama<1,2,1,2>,1212) stars
         ama<2,1,2,1> = 2121
         test = differ(item(ama,2,1,2,1),2121) stars
-eject
*
*        test eval
*
         expr = *('abc' 'def')
         test = differ(eval(expr),'abcdef') stars
         q = 'qqq'
         sexp = *q
         test = differ(eval(sexp),'qqq') stars
         fexp = *ident(1,2)
         test = eval(fexp) stars
*
*        test arg
*
jlab     define('jlab(a,b,c)d,e,f')
         test = differ(arg(.jlab,1),'A') stars
```

```
         test = differ(arg(.jlab,3),'C') stars
         test = arg(.jlab,0) stars
         test = arg(.jlab,4) stars
*
*        test local
*
         test = differ(local(.jlab,1),'D') stars
         test = differ(local(.jlab,3),'F') stars
         test = local(.jlab,0) stars
         test = local(.jlab,4) stars
*
*        test apply
*
         test = apply(.eq,1,2) stars
         test = ~apply(.eq,1,1) stars
         test = ~ident(apply(.trim,'abc '),'abc') stars
-eject
*
*        final processing
*
         output = '************************************************'
         diagnostics = 1000000 - &trace
         eq(diagnostics,0)        :s(terminate)
         &dump = 2
         output = '****    number of errors detected  '
.                         diagnostics '    ****'
         output = '**** e n d    o f     d i a g n o s t i c s ****'
         output = '************************************************'
.                                            :(end)
terminate output = '**** n o     e r r o r s    d e t e c t e d ****'
         output = '**** e n d    o f     d i a g n o s t i c s ****'
         output = '************************************************'
 :(end)
*
*        error handling routine
*
errors   eq(&errtype,0)                      :(continue)
         output = '****  error at '
.        lpad(&lastno,4)    '      &errtype = ' lpad(&errtype,7,' ')
.                                            ' ****'
         &trace = &trace - 1
         setexit(.errors)                    :(continue)
end
```

```
* title snobol test program #2 -- diagnostics phase two
*
*
*        this is the standard test program for spitbol which
*        tests pattern matching using both fullscan and quickscan
*
         &dump = 2
         define('error()')
         &trace = 1000
         &errlimit = 00
         trace(.errtype,'keyword')
         &fullscan = 0
         output = '**********************************************'
         output = '**** snobol  diagnostics -- phase two     ****'
```

```
        output = '**********************************************'
floop   errcount = 0
        output = '****          &fullscan = ' &fullscan
.       '               ****'
        test = 'abcdefghijklmnopqrstuvwxyz'
*
*       test pattern matching against simple string
*
        test  'abc' :s(s01) ; error()
s01     test 'bcd' :s(s02) ; error()
s02     test 'xyz' :s(s03) ; error()
s03     test 'abd' :f(s04) ; error()
s04     &anchor = 1
        test 'abc' :s(s05) ; error()
s05     test 'bcd' :f(s06) ; error()
s06     test test :s(s06a) ; error()
*
*       test simple cases of $
*
s06a    test 'abc' $ var :s(s07) ; error()
s07     ident(var,'abc') :s(s08) ; error()
s08     test 'abc' . vard :s(s09) ; error()
s09     ident(vard,'abc') :s(s10) ; error()
*
*       test len
*
s10     &anchor = 0
        test len(3) $ varl :s(s11) ; error()
s11     ident(varl,'abc') :s(s12) ; error()
s12     test len(26) $ varl :s(s13) ; error()
s13     ident(varl,test) :s(s14) ; error()
s14     test len(27) :f(s15) ; error()
*
*       test tab
*
s15     test tab(3) $ vart :s(s16) ; error()
s16     ident(vart,'abc') :s(s17) ; error()
s17     test tab(26) $ vart :s(s18) ; error()
s18     ident(test,vart) :s(s19) ; error()
s19     test tab(0) $ vart :s(s20) ; error()
s20     ident(vart) :s(s21) ; error()
-eject
*
*       test arb
*
s21     test arb $ vara 'c' :s(s22) ; error()
s22     ident(vara,'ab') :s(s23) ; error()
s23     &anchor = 1
        test arb $ vara pos(60) :f(s24) ; error()
s24     ident(vara,test) :s(s25) ; error()
*
*       test pos
*
s25     test arb $ vara pos(2) $ varp :s(s26) ; error()
s26     (ident(vara,'ab') ident(varp)) :s(s27) ; error()
s27     &anchor = 0
        test arb $ vara pos(26) $ varp :s(s28) ; error()
s28     (ident(vara,test) ident(varp)) : s(s29) ; error()
```

```
s29      test arb $ vara pos(0) $ varp :s(s30) ; error()
s30      ident(vara varp) :s(s31) ; error()
s31      test pos(0) arb $ vara pos(26) :s(s32) ; error()
s32      ident(test,vara) :s(s33) ; error()
s33      test pos(2) arb $ vara pos(3) :s(s34) ; error()
s34      ident(vara,'c') :s(s35) ; error()
s35      test pos(27) :f(s36) ; error()
*
*        test rpos
*
s36      test arb $ vara rpos(25) :s(s37) ; error()
s37      ident(vara,'a') :s(s38) ; error()
s38      test arb $ vara rpos(0) :s(s39) ; error()
s39      ident(test,vara) :s(s39a) ; error()
s39a     test arb $ vara rpos(26) :s(s40) ; error()
s40      ident(vara) :s(s41) ; error()
s41      test rpos(27) :f(s42) ; error()
*
*        test rtab
*
s42      test rtab(26) $ vara :s(s43) ; error()
s43      ident(vara) :s(s44) ; error()
s44      test rtab(27) :f(s45) ; error()
s45      test rtab(0) $ vara :s(s46) ; error()
s46      ident(vara,test) :s(s47) ; error()
s47      test rtab(25) $ vara :s(s48) ; error()
s48      ident(vara,'a') :s(s49) ; error()
*
*        test @
*
s49      test len(6) @vara :s(s50) ; error()
s50      ident(vara,6) :s(s51) ; error()
s51      test @vara :s(s52) ; error()
s52      ident(vara,0) :s(s53) ; error()
s53      test len(26) @vara :s(s54) ; error()
s54      ident(vara,26) :s(s55) ; error()
-eject
*
*        test break
*
s55      test break('c') $ vara :s(s56) ; error()
s56      ident(vara,'ab') :s(s57) ; error()
s57      test break('z()') $ vara :s(s58)      ; error()
s58      ident(vara,'abcdefghijklmnopqrstuvwxy') :s(s59) ; error()
s59      test break(',') :f(s60) ; error()
s60
*
*        test span
*
s63      test span(test) $ vara :s(s64) ; error()
s64      ident(test,vara) :s(s65) ;error()
s65      test span('cdq') $ vara :s(s66) ; error()
s66      ident(vara,'cd') :s(s67) ; error()
s67      test span(',') :f(s68) ; error()
s68
*
*
*        test any
```

```
*
s73      test any('mxz') $ vara :s(s74) ; error()
s74      ident(vara,'m') :s(s75) ; error()
s75      test any(',.') :f(s76) ; error()
-eject
*
*        test notany
*
s76      test notany('abcdefghjklmpqrstuwxyz') $ vara :s(s77) ; error()
s77      ident(vara,'i') :s(s78) ; error()
s78      test notany(test) :f(s79) ; error()
*
*        test rem
*
s79      test rem $ vara :s(s80) ; error()
s80      ident(vara,test) :s(s81) ; error()
s81      test len(26) rem $ vara :s(s82) ; error()
s82      ident(vara) :s(s83) ; error()
*
*        test alternation
*
s83      test ('abd' | 'ab') $ vara :s(d84) ; error()
d84      ident(vara,'ab') :s(d85) ; error()
d85      test (test 'a' | test) $ varl :s(d86) ; error()
d86      ident(varl,test) :s(d00) ; error()
*
*        test deferred strings
*
d00      test *'abc' :s(d01) ; error()
d01      test *'abd' :f(d06) ; error()
*
*        test $ . with deferred name arguments
*
d06      test 'abc' $ *var :s(d07) ; error()
d07      ident(var,'abc') :s(d08) ; error()
d08      test 'abc' . *$'vard' :s(d09) ; error()
d09      ident(vard,'abc') :s(d10) ; error()
*
*        test len with deferred argument
*
d10      &anchor = 0
         test len(*3) $ varl :s(d11) ; error()
d11      ident(varl,'abc') :s(d15) ; error()
*
*        test tab with deferred argument
*
d15      test tab(*3) $ vart :s(d16) ; error()
d16      ident(vart,'abc') :s(d21) ; error()
-eject
*
*        test pos with deferred argument
*
d21      &anchor = 1
         test arb $ vara pos(*2) $ varp :s(d26) ; error()
d26      (ident(vara,'ab') ident(varp)) :s(d27) ; error()
d27      &anchor = 0
         test arb $ vara pos(*0) $ varp :s(d35) ; error()
d35      ident(vara varp) :s(d36) ; error()
```

```
*
*        test rpos with deferred argument
*
d36      test arb $ vara rpos(*25) :s(d37) ; error()
d37      ident(vara,'a') :s(d38) ; error()
*
*        test rtab with deferred argument
*
d38      test rtab(*26) $ vara :s(d43) ; error()
d43      ident(vara) :s(d49) ; error()
*
*        test @ with deferred argument
*
d49      test len(6) @*vara :s(d50) ; error()
d50      ident(vara,6) :s(d51) ; error()
d51      test @*$'vara' :s(d52) ; error()
d52      ident(vara,0) :s(d55) ; error()
*
*        test break with deferred argument
*
d55      test break(*'c') $ vara :s(d56) ; error()
d56      ident(vara,'ab') :s(d57) ; error()
*
*        test span with deferred argument
*
d57      test span(*test) $ vara :s(d64) ; error()
d64      ident(test,vara) :s(d70) ; error()
*
*        test breakx with deferred argument
*
d70
*       (test test) pos(*0) breakx(*'e') $ vara '.' :f(d71) ; error()
*d71       ident(vara,test 'abcd') :s(d73) ; error()
-eject
*
*        test any with deferred argument
*
d73      test any(*'mxz') $ vara :s(d74) ; error()
d74      ident(vara,'m') :s(d75) ; error()
*
*        test notany with deferred argument
*
d75      test notany(*'abcdefghjklmpqrstuwxyz') $ vara :s(d77) ; error()
d77      ident(vara,'i') :s(d79) ; error()
d79      :(alldone)
         eject
*
*        error handling routine
*
error    output = '****** error detected at ' &lastno ' ********'
         errcount = errcount + 1
         output = '***** resuming execution *******'       :(return)
*
*        termination routine
*
alldone
         errcount = errcount + &errlimit - 100
         &errlimit = 100
```

```
        output = eq(errcount,0)
.                '****           no errors detected         ****'
        output = '*********************************************'
        &fullscan = eq(&fullscan,0) 1            :s(floop)
        output = '****           end of diagnostics         ****'
        output = '*********************************************'
end
```

Many thanks to the SNOBOL4 in C team, Philip L. Budne, and the other contributors.

Ralph Griswold sure did some amazing design work.

Be sure to check out the SourceForge link given above, and grab a copy of the distribution kit.

CSNOBOL4 License obligation


```
Copyright © 1993-2015, Philip L. Budne
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

27.1.18 fizzbuzz

FizzBuzz without modulo. Chase the fizz.

FizzBuzz is a task on Rosetta Code, *Unicon on rosettacode.org*. Most entries use some form of modulo test, this one (idea from a COBOL entry by Steve Williams) simply adds to the fizz and the buzz during the loop.

```
#
# fizzbuzz, chase the fizz
#
link wrap
procedure main()
```

```
    fizz := 3; buzz := 5;
    every i := 1 to 100 do {
        write(wrap(left(
            ((i = fizz = buzz & fizz +:= 3 & buzz +:= 5 & "fizzbuzz, ") |
            (i = fizz & fizz +:= 3 & "fizz, ") |
            (i = buzz & buzz +:= 5 & "buzz, ") |
            i || ", ") \ 1
        , 10), 60))
    }
    write(trim(wrap(), ', '))
end
```

And a sample run:

```
prompt$ unicon -s fizzbuzz.icn -x
1,        2,        fizz,     4,        buzz,     fizz,
7,        8,        fizz,     buzz,     11,       fizz,
13,       14,       fizzbuzz, 16,       17,       fizz,
19,       buzz,     fizz,     22,       23,       fizz,
buzz,     26,       fizz,     28,       29,       fizzbuzz,
31,       32,       fizz,     34,       buzz,     fizz,
37,       38,       fizz,     buzz,     41,       fizz,
43,       44,       fizzbuzz, 46,       47,       fizz,
49,       buzz,     fizz,     52,       53,       fizz,
buzz,     56,       fizz,     58,       59,       fizzbuzz,
61,       62,       fizz,     64,       buzz,     fizz,
67,       68,       fizz,     buzz,     71,       fizz,
73,       74,       fizzbuzz, 76,       77,       fizz,
79,       buzz,     fizz,     82,       83,       fizz,
buzz,     86,       fizz,     88,       89,       fizzbuzz,
91,       92,       fizz,     94,       buzz,     fizz,
97,       98,       fizz,     buzz
```

27.1.19 eval

A poor person's *expensive* `eval` procedure.

Putting the multi-tasker to work with on the fly compilation, *load* of new code and co-expression reflective properties.

```
#
# uval.icn, an eval function
#
$define base "/tmp/child-xyzzy"

link ximage

#
# try an evaluation
#
global cache
procedure main()
    cache := table()
    program := "# temporary file for eval, purge at will\n_
        global var\n_
        procedure main()\n_
```

```
            var := 5\n_
            suspend ![1,2,3] do var +:= 5\n_
        end"

    while e := eval(program) do {
        v := variable("var", cache[program])
        write("child var: ", v, " e: ", ximage(e))
    }

    # BUG HERE, can't refresh the task space: ^cache[program]

    # test cache
    v := &null
    e := eval(program)
    v := variable("var", cache[program])
    write("child var: ", v)
    write("e: ", ximage(e))

    # eval and return a list
    program := "# temporary file for eval, purge at will\n_
        procedure main()\n_
            return [1,2,3]\n_
        end"
    e := eval(program)
    write("e: ", ximage(e))

end

#
# eval, given string (either code or filename with isfile)
#
procedure eval(s, isfile)
    local f, code, status, child, result

    if \isfile then {
        f := open(s, "r") | fail
        code ||:= every(!read(f))
    } else code := s

    # if cached, just refresh the co-expression
    # otherwise, compile and load the code
    if member(cache, code) then write("^cache[code]")
    else {
        codefile := open(base || ".icn", "w") | fail
        write(codefile, code)
        close(codefile)

        status := system("unicon -s -o " || base  || " " ||
                         base || ".icn 2>/dev/null")
        if \status then
            cache[code] := load(base)
    }

    # if there is code, activate the co-expression
    if \cache[code] then result := @cache[code]

    remove(base || ".icn")
    remove(base)
```

```
    return \result | fail
end
```

And a sample run:

```
prompt$ unicon -s uval.icn -x
child var: 5 e: 1
^cache[code]
child var: 10 e: 2
^cache[code]
child var: 15 e: 3
^cache[code]
^cache[code]
child var: 20
e: 3
e: L1 := list(3)
   L1[1] := 1
   L1[2] := 2
   L1[3] := 3
```

27.1.20 unilist

Creating list results with *loadfunc*.

This is a pass at coming to grips with building heterogeneous lists from C functions.

First the C side:

```
/* unilist.c, trials with C functions and Unicon lists */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "icall.h"

/* access src/runtime/rstruct.r low level put */
void c_put(descriptor *, descriptor *);

/* some characters for random strings */
#define ALPHABET "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
#define ALPHASIZE (sizeof(ALPHABET) - 1)

/* replace all character positions with a random element from ALPHABET */
char *
randomize(char *str)
{
    int i;
    unsigned int rnd;
    static unsigned int seed;
    char *p = str;

    if (seed == 0) {
        seed = ((unsigned) time(NULL)<<8) ^ (unsigned)clock();
    }
```

```
    srand(seed++);
    while (*p) {
        rnd = rand() % ALPHASIZE;
        *p++ = (ALPHABET)[rnd];
    }
    return str;
}


/*
  Build out a heterogeneous list
*/
int
unilist(int argc, descriptor argv[])
{
    char *str;
    int len;
    int size;
    int limit;

    double dbl;

    descriptor listReturn;
    descriptor stringReturn;
    descriptor fileReturn;
    descriptor realReturn;

    /*
      Cheating, need list to work with
      todo: create your own list ya lazy git
    */
    ArgList(1);

    /* set a limit on randomized string lengths */
    ArgInteger(2);
    limit = IntegerVal(argv[2]);

    /* make a randomized string, 0 to limit, don't care about bias */
    size = 0;
    while (size < 1) {
        size = ((float)rand()) / RAND_MAX * limit+1;
    }

    str = malloc(size);
    if (!str) exit(1);
    memset(str, ' ', size);
    str[size-1] = '\0';
    len = size;
    randomize(str);

    /* add string to end of list */
    Protect(StringAddr(stringReturn) = alcstr(str, len), Error(306));
    StringLen(stringReturn) = len;
    c_put(&argv[1], &stringReturn);
    free(str);

    /* random real from [0, size], ignoring bias */
    dbl = ((double)rand() / (double)(RAND_MAX)) * (size-1);
```

```
    realReturn.dword = D_Real;
#if defined(DescriptorDouble)
    realReturn.vword.realval = dbl;
#else
    Protect(realReturn = alcreal(dbl), Error(307));
    realReturn.vword.bptr = (union block *)realReturn;
#endif
    /* add real to end of list */
    c_put(&argv[1], &realReturn);

    /* return string size */
    RetInteger(size-1);
}
```

Then a Unicon test pass:

```
#
# unilist.icn, demonstrate heterogeneous lists from loadfunc
#
# tectonics:
#   gcc -o unilist.so -shared -fPIC unilist.c
#
procedure main()
    allocated()
    unilist := loadfunc("./unilist.so", "unilist")

    # pass in an empty list, and limit on random string length
    L := []
    limit := 32

    # do a reasonably fat pass
    every i := 1 to 1000 do {
        rc := unilist(L, limit)
        L := put(L, rc)
    }
    write(i, " ", image(L))

    # dump out the first 10 triplets
    every i := 1 to 30 by 3 do write(left(L[i+2], 3),
                                     left(":" || L[i] || ":", limit+2),
                                     L[i + 1])
    allocated()
end

# Display current memory region allocations
procedure allocated()
    local allocs

    allocs := [] ; every put(allocs, &allocated)

    write()
    write("&allocated")
    write("----------")
    write("Heap   : ", allocs[1])
    write("Static : ", allocs[2])
    write("String : ", allocs[3])
    write("Block  : ", allocs[4])
    write()
```

```
end
```

Build the loadable:

```
prompt$ gcc -o unilist.so -shared -fPIC unilist.c
```

Run the test under `valgrind` to watch for leaks, should be 0.

```
prompt$ valgrind unicon -s unilist.icn -x
==18783== Memcheck, a memory error detector
==18783== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==18783== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==18783== Command: /home/btiffin/unicon-git/bin/unicon -s unilist.icn -x
==18783==
==18784== Warning: invalid file descriptor -1 in syscall close()
==18785==
==18785== HEAP SUMMARY:
==18785==     in use at exit: 10,182 bytes in 59 blocks
==18785==   total heap usage: 66 allocs, 7 frees, 10,894 bytes allocated
==18785==
==18785== LEAK SUMMARY:
==18785==    definitely lost: 0 bytes in 0 blocks
==18785==    indirectly lost: 0 bytes in 0 blocks
==18785==      possibly lost: 0 bytes in 0 blocks
==18785==    still reachable: 10,182 bytes in 59 blocks
==18785==         suppressed: 0 bytes in 0 blocks
==18785== Rerun with --leak-check=full to see details of leaked memory
==18785==
==18785== For counts of detected and suppressed errors, rerun with: -v
==18785== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==18784==
==18784== HEAP SUMMARY:
==18784==     in use at exit: 2,017 bytes in 59 blocks
==18784==   total heap usage: 64 allocs, 5 frees, 2,201 bytes allocated
==18784==
==18784== LEAK SUMMARY:
==18784==    definitely lost: 0 bytes in 0 blocks
==18784==    indirectly lost: 0 bytes in 0 blocks
==18784==      possibly lost: 0 bytes in 0 blocks
==18784==    still reachable: 2,017 bytes in 59 blocks
==18784==         suppressed: 0 bytes in 0 blocks
==18784== Rerun with --leak-check=full to see details of leaked memory
==18784==
==18784== For counts of detected and suppressed errors, rerun with: -v
==18784== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

&allocated
----------
Heap   : 34528
Static : 0
String : 0
Block  : 34528

1000 list_2(3000)
26 :AtqbcHWFCXIXnTYNdkXCBYwLky\:     11.2612881303119
0  :\:                                0.0
8  :oodyAwhj\:                        0.9550582696474429
25 :puGfsWEdhszHWxwtRaBIpxwMH\:      6.107686567636061
```

```
1  :n\:                            0.9917704826182548
27 :gMTLUzctVPJxutZkYGXXCDVjGLX\:    0.2817263562612824
21 :ogzezbqqhMRsNgmeXLgwo\:          4.551997850906103
24 :QDXFehIbexREdszAgbfHVjBM\:       21.05572956290828
5  :iGbIk\:                          1.28611881578626
23 :oOGHaIFpwTVkFkbQWUjzgQJ\:        5.691450906773774

&allocated
----------
Heap   : 102186
Static : 0
String : 17178
Block  : 85008
```

27.1.21 tcc

Embedding and integrating Tiny C with *loadfunc*.

This is a pass at building *loadfunc* dynamic shared object files with Tiny C.

As a first trial, the `unilist.c` and `unilist.icn` listed above is used:

Build the loadable:

```
prompt$ tcc -o unilist.so -shared unilist.c
```

Run the test:

```
prompt$ unicon -s unilist.icn -x

&allocated
----------
Heap   : 34528
Static : 0
String : 0
Block  : 34528

1000 list_2(3000)
26 :MKKOlROariRXwikvjQJsXUqOsX\:     8.975710052519901
19 :hpnaWQGUmaaDxWRmaSr\:            12.53669600958782
4  :VYJB\:                           3.728886930145736
5  :FVBhO\:                          4.509417311991293
6  :atKuuX\:                         0.2543022931806288
31 :nPkmHHiTLmyFtFwiDuQOLvXvgOOLjHD\:11.7287685860548
24 :UEPxLSerZRIRqWUcyvPIsKSB\:       12.6808377377134
8  :rnUYBCiz\:                       5.914912569296972
19 :tsqARymactHiIwLRnoe\:            15.05751783450019
4  :GKrV\:                           3.885928295499612

&allocated
----------
Heap   : 101712
Static : 0
String : 16704
Block  : 85008
```

So, *tcc* can be used to build Unicon loadable shared libraries. *gcc* produced a 12K shared object file, *tcc* produced an 8K file.

Embedded tcc

And then, to embed a C compiler in a Unicon function:

```
/* program.c description */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "libtcc.h"
#include "icall.h"

int
unitcc(int argc, descriptor argv[])
{
    /* Crank up tcc */
    TCCState *s;
    int (*func)(int);

    int result;

    /* First arg is the C code */
    ArgString(1);
    /* Second arg is the integer parameter */
    ArgInteger(2);

    s = tcc_new();
    if (!s) {
        fprintf(stderr, "Could not create tcc state\n");
        exit(1);
    }

    /* if tcclib.h and libtcc1.a are not installed, where can we find them */
    /*
    if (argc == 2 && !memcmp(argv[1], "lib_path=",9))
        tcc_set_lib_path(s, argv[1]+9);
    */

    /* MUST BE CALLED before any compilation */
    tcc_set_output_type(s, TCC_OUTPUT_MEMORY);

    if (tcc_compile_string(s, StringVal(argv[1])) == -1)
        return 1;

    /* as a test, we add a symbol that the compiled program can use.
       You may also open a dll with tcc_add_dll() and use symbols from that */
    /*
    tcc_add_symbol(s, "add", add);
    */

    /* relocate the code */
    if (tcc_relocate(s, TCC_RELOCATE_AUTO) < 0)
        return 1;
```

```
    /* get entry symbol, Unicon passes code that compiles trytcc */
    func = tcc_get_symbol(s, "trytcc");
    if (!func)
        return 1;

    /* run the code */
    result = func(IntegerVal(argv[2]));

    /* delete the state */
    tcc_delete(s);

    RetInteger(result);
}
```

Unicon test file, expects to find a post compile symbol of `trytcc` that takes an integer and returns that parameter multiplied by seven.

```
#
# unitcc.icn, demonstrate an embedded Tiny C compiler
#
# tectonics:
#   tcc -o unitcc.so unitcc.c -ltcc -L\usr\local\lib
#
procedure main()
    tcc := loadfunc("./unitcc.so", "unitcc")

    # compile some code with unitcc and pass an integer argument
    result := tcc(

        "int trytcc(int i) {\n_
            printf(\"Hello, world\n\");\n_
            return i*7;\n_
         }", 6)

    # the inner trytcc function, compiled by tcc is invoked with arg
    write("result from tcc: ", result)
end
```

Build the loadable using `libtcc.so` from `/usr/local/lib`.

```
prompt$ tcc -o unitcc.so -shared unitcc.c -ltcc -L/usr/local/lib
```

This test loads the external function, which is an embedded C compiler, and then compiles and links a C function, `trytcc` directly into memory. The external function trial expects the `trytcc` entry point and invokes the function with an integer that is passed by Unicon along with the code. We expect to see “Hello, world” and a result that returns 6 * 7. Six is passed from Unicon and the C code multiples the input parameter by seven.

Note that the *loadfunc DSO* was created by *tcc*.

```
prompt$ unicon -s unitcc.icn -x
Hello, world
result from tcc: 42
```

Tiny C is very neat. And not really a toy. It’s a full fledged ANSI C compiler, that even includes an inline assembler. Originally by Fabrice Bellard, now famous for designing and developing QEMU.

http://bellard.org/tcc/

C code, compiled on the fly from Unicon, then invoked with arguments, and results returned using the features of *loadfunc*.

CHAPTER

TWENTYEIGHT

MULTILANGUAGE

28.1 Multilanguage programming

By necessity, this chapter is not solely Unicon focused. Development with multiple programming languages, across multiple paradigms can be highly rewarding when applied to non-trivial application development. This can be as basic as adding a scripting engine to a program, or far more sophisticated, with multiple modules integrated into a cohesive whole.

For example, Emacs, a text editor now over 30 years old, merged a C programming core with Lisp scripting. Not the first multilanguage environment, but a relatively famous, and highly successful one.

Unicon is well placed for `general purpose programming`, the feature set allowing for many problems to be solved with only Unicon source code. This is especially true when programming in the small, utility programs and light applications. One thing missing from base Unicon is end user scripting. Extending a core Unicon program means writing new Unicon source and recompiling the application. Emacs has shown that allowing the end user to extend a program with scripts is a powerful attribute, providing a path to add useful features that were not originally designed into an application.

Along with scripting, there are vast quantities of libraries written in other languages that may benefit a Unicon program, alleviating the need to write new code to solve an already solved problem.

Luckily, Unicon allows for extending programs with foreign language libraries and exposing some powerful scripting engines (with some small effort).

A small downside is that Unicon requires a specific interface protocol,[1] so there is usually a few lines of C language source involved. This extra effort is fairly boilerplate, and working samples already exist, making this a rather convenient inconvenience.

Unicon ships with multiple forms of C integration (*loadfunc* and *callout* to name two), and from there, all kinds of potential is exposed.

Multilanguage programming with Unicon will usually be a two or three language mix, `Unicon`, `C`, and any other language in use for the modules at hand. This is possible because almost every programming language in existence

[1] The Unicon foreign function interface protocol is in place to manage the high level Unicon *datatypes* expectations. Unicon data types are held in encoded descriptors, and these need conversion to and from the native ABI.

provides access to an *ABI*, application binary interface, compatible with platform specific C compilers. C becomes the crossroads of a *many to one to many* integration hub. The required C layer is usually small in terms of binary size and source code line count, and as mentioned earlier, there are existing examples to ease the burden on a Unicon developer.

See *S-Lang*, *COBOL integration*, *Javascripting* and *Mini Ruby* for some example integrations.

28.1.1 loadfunc

loadfunc is a handy piece of kit. It allows Unicon to load libraries and then define Unicon functions that will invoke special C functions. The[1] requires a small piece of C that accepts `int argc, descpriptor argv[]`, or a count of arguments and pointer to an array of Unicon descriptors.

Argument 0 is reserved for the return result, so `argc` is always at least 1.

The `descriptor` data structures are define in `ipl/cfuncs/icall.h` and that header file must be included in the source that defines the callable functions.

The `descriptor` is what gives Unicon its variant data type powers. A data item can be a string, a number, a list, a real, etcetera and the structure can assigned to a variable (or passed to C functions). Variables do not have a type, and can reference any data, determined by the descriptor. The *Programs* section has many examples of using *loadfunc*.

The small wrappers, written in C, are then free to call any other C function as a native call. Results from these functions can then be passed back to Unicon by setting the `argv[0]` element. There are macros to make this very easy. `Fail`, `Return`, `RetString`, `RetInteger`, `RetReal`, and so on, all documented in `ipl/cfuncs/icall.h`.

There are also a helper macros for testing the Unicon type passed into the wrapper, and for converting the descriptor values to native C data types.

`IconType` is used for testing. `ArgString` will ensure (and convert if necessary) a particular descriptor is a C `char` pointer. `StringVal` will return the `char *`. Same for `ArgInteger`, `ArgReal`, `ArgList` (and others) along with the corresponding `IntegerVal`, `RealVal`, `ListVal`, etcetera. Read `icall.h` for all current details.

28.1.2 C Native

Note: Superseded, see *libffi* below. But read through this section as it details a lot of important background. The libffi version uses a very similar Unicon programmer interface, but offers a lot more platforms from a much more mature and well tested codebase.

Here is an experimental enhancement to Unicon that allows directly calling almost any C function, without need for a wrapper. This builds on *loadfunc* and defines 2 (or 3) new loadable functions. `addLibrary` will add a dynamic shared object library to the loader search path. `native` allows for calling C functions directly, by string name and an enumerated constant to determine the return type. Other arguments are tested by Unicon type and passed to C having built a call frame, using inline assembler.

The listing below provides support for x86_64, System V ABI calling conventions. Other platform specific assembler will be required to add support for other systems. This is relatively straight forward inline assembler, 17 instructions for the initial trials. It allows up to 6 arguments, mixed integer, real or pointer/handle argument types and handles void returns along with integral (numbers/pointers), and `double` floating point data for use as Unicon *Real numbers*, along with *String* types.

The code needs to be loaded into Unicon space, using *loadfunc*. `addLibrary` allows Unicon to add libraries to the dynamic search path. `native` is then used to lookup the call vectors (by string name, similar to *loadfunc* but then marshals the other arguments and sets up a valid Unicon return descriptor. No other wrappers are required.

First the new functions:

```
/* A new Unicon C native FFI layer, with a little assembler thrown in */
#include <stdio.h>
#include <dlfcn.h>

#include "icall.h"
#include "natives.h"

static void *dlHandle;

int
addLibrary(int argc, descriptor argv[])
{
    //union block dlBlock;
    //descriptor dlBlock;

    /* Add a new library to the dynamic search path */
    ArgString(1)
    if (!dlHandle) {
        dlHandle = dlopen(StringVal(argv[1]), RTLD_LAZY | RTLD_GLOBAL);
        if (!dlHandle) {
            fprintf(stderr, "dlHandle error\n");
            fflush(stderr);
            Error(500);
        }
    }

    //dlBlock = mkExternal(dlHandle, sizeof(dlHandle));
    //RetExternal(dlBlock);
    RetInteger((long)dlHandle);
}

//static void *(*func)();
static void (*func)();
union blob {
    long lvalue;
    double rvalue;
    float fvalue;
    char *svalue;
};

static union blob fromc;
static union blob fromf;

/* Let the good times roll */
int
native(int argc, descriptor argv[])
{
    /* a dlsym function pointer */
    char *dlMsg;

    /* Integers and pointers go in RDI, RSI, RDX, RCX/R10, R8, R9 */
    /* Floating point in XMM0 - XMM6 */
    /* They are two seperate sets */
    union blob ipregs[7];
    union blob fregs[7];

    long retType;
    char inType;
```

```
int ips = 1; /* count of integer/pointer args, reserve 0 */
int rs = 1;  /* count of floating args, reserved 0, might not use */

/* first the function name */
ArgString(1);

/* second is return type, encoded in natives.inc matched in natives.h */
ArgInteger(2);
retType = IntegerVal(argv[2]);

/* Load the function pointer */
dlerror();
*(void **)(&func) = dlsym(dlHandle, StringVal(argv[1]));
dlMsg = dlerror();
if (dlMsg) {
    fprintf(stderr, "dlsym fail: %s\n", dlMsg);
    fflush(stderr);
    Error(500);
}
if (!func) Fail;

/* marshalling by assembly to set up the variant call frames */
for (int argi = 3; argi <= argc; argi++) {
    inType = IconType(argv[argi]);

    /* Integers and pointers go in RDI, RSI, RDX, RCX/R10, R8, R9 */
    /* Floating point in XMM0 - XMM6 */
    /* They are two seperate sets */
    switch(inType) {
    case 'i':
        ArgInteger(argi);
        ipregs[ips++].lvalue = IntegerVal(argv[argi]);
        break;
    case 'r':
        ArgReal(argi);
        fregs[rs++].rvalue = RealVal(argv[argi]);
        break;
    case 's':
        ArgString(argi);
        ipregs[ips++].svalue = StringVal(argv[argi]);
        break;
    }

}

/* Function vector in r11 */
asm("movq func(%%rip), %%r11;"
    : /* no output operand */
    : /* no input operand */
    :"%r11"
);
/* Take values from ipregs ints and fregs floats */
asm("movq %0, %%r9;"
    : /* no output operand */
    :"r"(ipregs[6])
    :"%r9" /* clobber */
);
```

```
    asm("movq %0, %%r8;"
        :
        :"r"(ipregs[5])
        :"%r8"
    );
    asm("movq %0, %%rcx;"
        :
        :"r"(ipregs[4])
        :"%rcx"
    );
    asm("movq %0, %%r10;"   /* For Linux kernel calls instead of RCX */
        :
        :"r"(ipregs[4])
        :"%r10"
    );
    asm("movq %0, %%rdx;"
        :
        :"r"(ipregs[3])
        :"%rdx"
    );
    asm("movq %0, %%rsi;"
        :
        :"r"(ipregs[2])
        :"rsi"
    );
    asm("movq %0, %%rdi;"
        :
        :"r"(ipregs[1])
        :"%rdi"
    );

    asm("movsd %0, %%xmm5;"
        :
        :"m"(fregs[6])
        :"xmm5"
    );
    asm("movsd %0, %%xmm4;"
        :
        :"m"(fregs[5])
        :"xmm4"
    );
    asm("movsd %0, %%xmm3;"
        :
        :"m"(fregs[4])
        :"xmm3"
    );
    asm("movsd %0, %%xmm2;"
        :
        :"m"(fregs[3])
        :"xmm2"
    );
    asm("movsd %0, %%xmm1;"
        :
        :"m"(fregs[2])
        :"xmm1"
    );
    asm("movsd %0, %%xmm0;"
        :
```

```
        :"m"(fregs[1])
        :"xmm0"
    );

    asm("call *%r11");
    asm("movsd %xmm0, fromf(%rip)");
    asm("movq %rax, fromc(%rip)");

    /* Return type, as specfied in argument 2 enum */
    switch(retType) {
    case TYPEVOID:
        Return;
        break;
    case TYPESTAR:
        RetInteger(fromc.lvalue);
        break;
    case TYPEINT:
        RetInteger((long)fromc.lvalue);
        break;
    case TYPEFLOAT:
        RetReal((float)fromf.rvalue);
        break;
    case TYPEDOUBLE:
        RetReal((double)fromf.rvalue);
        break;
    case TYPESTRING:
        RetString(fromc.svalue);
        break;
    default:
        RetInteger((long)fromc.lvalue);
        break;
    }
}

/* Delta between managing float versus double */
/* Requires some refactoring to merge with above */
int
nativeFloat(int argc, descriptor argv[])
{
    /* a dlsym function pointer */
    char *dlMsg;

    /* Integers and pointers go in RDI, RSI, RDX, RCX/R10, R8, R9 */
    /* Floating point in XMM0 - XMM6 */
    /* They are two seperate sets here, duplicate work, more coverage */
    union blob ipregs[7];
    union blob fregs[7];

    long retType;
    char inType;

    int ips = 1; /* count of integer/pointer args, reserve 0 */
    int rs = 1;  /* count of floating args, reserved 0, might not use */

    /* first the function name */
    ArgString(1);

    /* second is return type, encoded in natives.inc */
```

```
    ArgInteger(2);
    retType = IntegerVal(argv[2]);

    /* load the function pointer */
    dlerror();
    *(void **)(&func) = dlsym(dlHandle, StringVal(argv[1]));
    dlMsg = dlerror();
    if (dlMsg) {
        fprintf(stderr, "dlsym fail: %s\n", dlMsg);
        fflush(stderr);
        Error(500);
    }
    if (!func) Fail;

    /* marshalling by assembly to set up the variant call frames */
    for (int argi = 3; argi <= argc; argi++) {
        inType = IconType(argv[argi]);
        /* Integers and pointers go in RDI, RSI, RDX, RCX/R10, R8, R9 */
        /* Floating point in XMM0 - XMM6 */
        /* They are two seperate sets */
        switch(inType) {
        case 'i':
            ArgInteger(argi);
            ipregs[ips++].lvalue = IntegerVal(argv[argi]);
            break;
        case 'r':
            ArgReal(argi);
            fregs[rs++].fvalue = (float)RealVal(argv[argi]);
            break;
        case 's':
            ArgString(argi);
            ipregs[ips++].svalue = StringVal(argv[argi]);
            break;
        }

    }

    /* Function vector in r11 */
    asm("movq func(%%rip), %%r11;"
        : /* no output operand */
        : /* no input operand */
        :"%r11"
    );
    /* Take values from ipregs ints and fregs floats */
    asm("movq %0, %%r9;"
        : /* no output operand */
        :"r"(ipregs[6])
        :"%r9" /* clobber */
    );
    asm("movq %0, %%r8;"
        :
        :"r"(ipregs[5])
        :"%r8"
    );
    asm("movq %0, %%rcx;"
        :
        :"r"(ipregs[4])
        :"%rcx"
```

```
    );
    asm("movq %0, %%r10;"   /* For Linux kernel calls instead of RCX */
        :
        :"r"(ipregs[4])
        :"%r10"
    );
    asm("movq %0, %%rdx;"
        :
        :"r"(ipregs[3])
        :"%rdx"
    );
    asm("movq %0, %%rsi;"
        :
        :"r"(ipregs[2])
        :"rsi"
    );
    asm("movq %0, %%rdi;"
        :
        :"r"(ipregs[1])
        :"%rdi"
    );

    asm("movss %0, %%xmm5;"
        :
        :"m"(fregs[6])
        :"xmm5"
    );
    asm("movss %0, %%xmm4;"
        :
        :"m"(fregs[5])
        :"xmm4"
    );
    asm("movss %0, %%xmm3;"
        :
        :"m"(fregs[4])
        :"xmm3"
    );
    asm("movss %0, %%xmm2;"
        :
        :"m"(fregs[3])
        :"xmm2"
    );
    asm("movss %0, %%xmm1;"
        :
        :"m"(fregs[2])
        :"xmm1"
    );
    asm("movss %0, %%xmm0;"
        :
        :"m"(fregs[1])
        :"xmm0"
    );

    asm("call *%r11");
    asm("movss %xmm0, fromf(%rip)");
    asm("movq %rax, fromc(%rip)");

    /* Return type, as specfied in argument 2 enum */
```

```
    switch(retType) {
    case TYPEVOID:
        Return;
        break;
    case TYPESTAR:
        RetInteger(fromc.lvalue);
        break;
    case TYPEINT:
        RetInteger((long)fromc.lvalue);
        break;
    case TYPEFLOAT:
        RetReal((float)fromf.fvalue);
        break;
    case TYPEDOUBLE:
        RetReal((double)fromf.fvalue);
        break;
    case TYPESTRING:
        RetString(fromc.svalue);
        break;
    default:
        RetInteger((long)fromc.lvalue);
        break;
    }
}
```

Before the sample run, here are the two small support files for enumerating the encoded return types, and other related paperwork. These two files must be kept in synch.

A Unicon `$include` file:

```
#
# native.inc, Native type constants
#
$define TYPEVOID 0
$define TYPESTAR 1
$define TYPECHAR 2
$define TYPESHORT 3
$define TYPEINT 4
$define TYPEFLOAT 5
$define TYPEDOUBLE 6
$define TYPESTRING 7
```

(Note the filename, `.inc`), it is a Unicon preprocessor include file.

A small C `#include` header.

```
/* Native return types, must match natives.icn from Unicon */
#define TYPEVOID 0
#define TYPESTAR 1
#define TYPECHAR 2
#define TYPESHORT 3
#define TYPEINT 4
#define TYPEFLOAT 5
#define TYPEDOUBLE 6
#define TYPESTRING 7
```

Now a test head of small C functions to try out various call frames.

```
/* testnative.c, testing an experimental Unicon native call layer */
#include <stdio.h>

int
testnative(int one, int two)
{
    int ivalue;
    fprintf(stderr, "In testnative %d %d\n", one, two);
    fflush(stderr);
    ivalue = one + two;
    return ivalue;
}

double
testdouble(double one, double two)
{
    fprintf(stderr, "In testdouble %f %f\n", one, two);
    fflush(stderr);
    return one + two;
}

double
testfive(double one, double two, int three, double four, double five)
{
    double sum;
    fprintf(stderr, "In testfive %f %f %d %f %f\n", one, two, three,
                                                    four, five);
    fflush(stderr);
    sum = one + two + three + four + five;
    return sum;
}

void *
teststar()
{
    fprintf(stderr, "In teststar\n");
    fflush(stderr);
    return &teststar;
}

char *
teststring(char *echo)
{
    fprintf(stderr, "In teststring with #%s#\n", echo);
    fflush(stderr);
    return echo;
}

char *
testmulti(char *echo, int i, double d)
{
    fprintf(stderr, "In testmulti with #%s# %d %f\n", echo, i, d);
    fflush(stderr);
    return echo;
}

double
testmultid(char *echo, int i, double d)
```

```
{
    fprintf(stderr, "In testmultid with #%s# %d %f\n", echo, i, d);
    fflush(stderr);
    return d;
}

/* Requires using nativeFloat from Unicon */
float
testfloat(float one, float two)
{
    fprintf(stderr, "In testfloat %f %f\n", one, two);
    fflush(stderr);
    return one + two;
}

float
testmultif(char *echo, int i, float f)
{
    fprintf(stderr, "In testmultif with #%s# %d %f\n", echo, i, f);
    fflush(stderr);
    return f;
}
```

Unicon code to put in all in motion:

```
#
# testffi.icn, demonstrate an experimental C FFI
#
$include "natives.inc"

procedure main()
    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uninative.so", "addLibrary")

    # allow arbitrary C functions, marshalled by a piece of assembler
    native := loadfunc("./uninative.so", "native")

    # add the testing functions to the dlsym search path,
    #  the handle is somewhat irrelevant, but won't be soonish
    dlHandle := addLibrary("./libtestnative.so")
    write("Unicon dlHandle: ", dlHandle)
    write()

    # pass two integers, get the sum as int
    ans := native("testnative", TYPEINT, 40, 2)
    write("Unicon: called testnative and got ", ans)
    if ans ~= 42 then write("ERROR with testnative")
    write()

    # pass two reals, get the sum as real
    ans := native("testdouble", TYPEDOUBLE, 9.0, 8.0)
    write("Unicon: called testdouble and got ", ans)
    if ans ~= 17.0 then write("ERROR with testdouble")
    write()

    # third arg is an integer, returns real
    ans := native("testfive", TYPEDOUBLE, 1.0, 2.0, 3, 4.0, 5.0)
```

```
    write("Unicon: called testfive and got ", ans)
    if ans ~= 15.0 then write("ERROR with testfive")
    write()

    # get a pointer/handle
    ans := native("teststar", TYPESTAR)
    write("Unicon: called teststar and got ", ans)
    if ans = 0 then write("ERROR with teststar")
    write()

    # a string
    ans := native("teststring", TYPESTRING, "this is a string")
    write("Unicon: called teststring and got ", ans)
    if ans ~== "this is a string" then write("ERROR with teststring")
    write()

    # a string, and int and a real, returning string
    ans := native("testmulti", TYPESTRING,
                  "this is a string for multi", 42, &pi)
    write("Unicon: called testmulti and got ", ans)
    if ans ~== "this is a string for multi" then
        write("ERROR with testmulti")
    write()

    # a string, and int and a real, returning real
    ans := native("testmultid", TYPEDOUBLE,
                  "this is a string for multid", 42, &pi)
    write("Unicon: called testmultid and got ", ans)
    if ans ~= &pi then write("ERROR with testmultid")
    write()

    #
    # Variant for float versus double
    #
    # pass two reals, get the sum as real
    write("float variant")
    nativeFloat := loadfunc("./uninative.so", "nativeFloat")

    ans := nativeFloat("testfloat", TYPEFLOAT, 9.0, 8.0)
    write("Unicon: called testfloat and got ", ans)
    if ans ~= 17.0 then write("ERROR with testfloat")
    write()

    # a string, and int and a real, returning real
    ans := nativeFloat("testmultif", TYPEFLOAT,
                       "this is a short string for multif", 21, &pi/2)
    write("Unicon: called testmultif and got ", ans)
    if ans - &pi/2 > 0.0000001 then write("ERROR with testmultif")
    write()
end
```

Two commands to prep the new support function and build the test heads

```
prompt$ gcc -o uninative.so -shared -fPIC native.c
```

```
prompt$ gcc -o libtestnative.so -shared -fPIC testnative.c
```

And finally, the sample run:

```
prompt$ unicon -s testffi.icn -x
Unicon dlHandle: 28311408

In testnative 40 2
Unicon: called testnative and got 42

In testdouble 9.000000 8.000000
Unicon: called testdouble and got 17.0

In testfive 1.000000 2.000000 3 4.000000 5.000000
Unicon: called testfive and got 15.0

In teststar
Unicon: called teststar and got 139819910732339

In teststring with #this is a string#
Unicon: called teststring and got this is a string

In testmulti with #this is a string for multi# 42 3.141593
Unicon: called testmulti and got this is a string for multi

In testmultid with #this is a string for multid# 42 3.141593
Unicon: called testmultid and got 3.141592653589793

float variant
In testfloat 9.000000 8.000000
Unicon: called testfloat and got 17.0

In testmultif with #this is a short string for multif# 21 1.570796
Unicon: called testmultif and got 1.570796370506287
```

Woohoo, swing you partner round and round. Calling C without any wrapper functions.

---

libharu

This makes things a little easier when it comes to some of the more feature rich libraries available, first trial is `libharu` a C library for creating PDF documents.

```
#
# haru.icn, demonstrate a new C FFI (first fail, double versus float)
#
$include "natives.inc"

$define HPDF_COMP_ALL 15
$define HPDF_PAGE_MODE_USE_OUTLINE 1
$define HPDF_PAGE_SIZE_LETTER 0
$define HPDF_PAGE_PORTRAIT 0

procedure main()
    local dlHandle, pdf, page1, rc, savefile := "harutest.pdf"

    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uninative.so", "addLibrary")

    # allow arbitrary C functions, marshalled by a piece of assembler
```

```
    # assume float instead of double, changes the inline assembler
    # movsd versus movdd
    native := loadfunc("./uninative.so", "nativeFloat")

    # add libhpdf to the dlsym search path, the handle is irrelevant
    dlHandle := addLibrary("libhpdf.so")

    pdf := native("HPDF_New", TYPESTAR, 0, 0)

    rc := native("HPDF_SetCompressionMode", TYPEINT, pdf, HPDF_COMP_ALL)
    rc := native("HPDF_SetPageMode", TYPEINT, pdf,
                 HPDF_PAGE_MODE_USE_OUTLINE)

$ifdef PROTECTED
    rc := native("HPDF_SetPassword", TYPEINT, pdf, "owner", "user")
    savefile := "harutest-pass.pdf"
$endif

    page1 := native("HPDF_AddPage", TYPESTAR, pdf)

    rc := native("HPDF_Page_SetHeight", TYPEINT, page1, 220.0)
    rc := native("HPDF_Page_SetWidth", TYPEINT, page1, 200.0);

    #/* A part of libharu pie chart sample, Red*/
    rc := native("HPDF_Page_SetRGBFill", TYPEINT, page1, 1.0, 0.0, 0.0);
    rc := native("HPDF_Page_MoveTo", TYPEINT, page1, 100.0, 100.0);
    rc := native("HPDF_Page_LineTo", TYPEINT, page1, 100.0, 180.0);
    rc := native("HPDF_Page_Arc", TYPEINT, page1, 100.0, 100.0, 80.0,
                                                   0.0, 360 * 0.45);

    #pos := native("HPDF_Page_GetCurrentPos (page);

    rc := native("HPDF_Page_LineTo", TYPEINT, page1, 100.0, 100.0);
    rc := native("HPDF_Page_Fill", TYPEINT, page1);

    rc := native("HPDF_SaveToFile", TYPEINT, pdf, savefile)
    native("HPDF_Free", TYPEVOID, pdf)
end
```

And then generating a password protected sample PDF.

```
prompt$ unicon -s -DPROTECTED haru-v1.icn -x
```

In a less short example, the password PROTECTED option would be tested at runtime, not at compile time, along with other control options for things like supported page sizes, compression and default character encodings.

```
prompt$ ls -l harutest-pass.pdf
-rw-rw-r-- 1 btiffin btiffin 1113 Oct 27 04:53 harutest-pass.pdf
```

GnuCOBOL

Calling COBOL modules is now a breeze.

```
      *>
      *> Demonstrate Unicon native call of COBOL modules
      *>
       identification division.
       program-id. cobolnative.

       data division.
       working-storage section.
       linkage section.
       01 one usage binary-long.
       01 two usage binary-long.

       procedure division using by value one two.
       display "GnuCOBOL got " one ", " two
       compute return-code = one + two
       goback.
       end program cobolnative.
```

A caller:

```
#
# testcob.icn, test calling COBOL without wrapper
#
$include "natives.inc"

procedure main()
    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uninative.so", "addLibrary")

    # allow arbitrary C functions, marshalled by a small piece of assembler
    native := loadfunc("./uninative.so", "native")

    # add the testing functions to the dlsym search path,
    #  the handle is somewhat irrelevant, but won't be soonish
    dlHandle := addLibrary("./cobolnative.so")

    # initialize GnuCOBOL
    native("cob_init", TYPEVOID)

    # pass two integers, get back a sum
    ans := native("cobolnative", TYPEINT, 40, 2)
    write("Unicon: called sample and got ", ans)

    # rundown the libcob runtime
    native("cob_tidy", TYPEVOID)
end
```

A build, and a run:

```
prompt$ cobc -m cobolnative.cob -Wno-unfinished
```

```
prompt$ unicon -s testcob.icn -x
GnuCOBOL got +0000000040, +0000000002
Unicon: called sample and got 42
```

And there is Unicon calling COBOL, with no additional wrappers, besides the `native` functions. Data passed, and results returned to Unicon.

Supported platforms

Currently, the assembler required to make this work is x86_64 GNU/Linux only. That will change if/when people show interest.

Note: libffi fixes that support problem, lots of platforms supported. libffi supersedes, but does not displace, the above information.

28.1.3 libffi

After playing with the C Native interface, and actually looking into how many platforms would require the assembly layer, bumped into `libffi`. `libffi` is a library that does the call frame setup, very much like the native interface described above. Except it is mature, well tested, and already supports many tens of platforms and various compiler systems. So, while the above interface works, and will be available, the `libffi` system is already ahead of the game, and will be put to use instead of hand rolled assembler.

https://sourceware.org/libffi/

The user interface from a Unicon point of view, is almost identical to what is shown above, but extended with a few more features, like support for `struct`, something that would have been a little trickier with hand rolled assembly developed from scratch.

A new `native(...)` function is defined in the same way with *loadfunc*, but instead of including inline assembler, it uses `ffi_prep_cif` and `ffi_call`.

FFI Foreign Function Interface

CIF Call InterFace

The only visible change is `uninative` in now `uniffi`, and `-lffi` is included for access to the `libffi` features.

Using `libffi` suffers the same problem of `float` versus `double` that was in the previous native sequence. A Unicon Real is a `double` and sometimes *C* requires a `float`. A new interface was developed to help get around this problem; a datatype override can be included by using a *List (arrays)* structure as part of any arguments passed on to *C*.

As an explanation:

The current marshalling layer pulls data from Unicon and tests the type of data passed. Integer values are mapped to `long`, Real values are mapped to `double`.

For example, the following code, just works:

```
ans := native("j0", TYPEDOUBLE, &pi)
```

That code will call the `libm` Bessel function of the first kind, order 0, which assumes a `double` floating point input and returns a `double`. The Unicon type does not need any further conversion. A real is effectively a double.

The problem comes when you want to call a lower precision `float` version.

The first crack at a solution was to have `native(...)` and `nativeFloat(...)` routines. The `nativeFloa(...)` function assumed that all real values were to be demoted to `float` values. It worked, but it meant that you could never mix `float` and `double` arguments. The new sequence drops the `nativeFloat` function and provides an override option to the Unicon programmer.

The new experiment uses *List (arrays)* data for those times when an override is required.

```
# pass pi as a real (as double), get back a real (as double)
ans := native("j0", TYPEDOUBLE, &pi)
write("j0(pi) = ", ans)

# pass pi as a real (as float), get back a real (as float)
ans := naitive("j0f", TYPEFLOAT, [&pi, TYPEFLOAT])
write("j0f(pi) = ", ans)

# pass pi as a normal double, pass e as a float, get back a double
ans := native("mymath", TYPEDOUBLE, &pi, [&e, TYPEFLOAT])
```

The override allows freely mixed `float` and `double` arguments pulled from the Unicon Real type.

An alternative was an overly burdensome type specifier required for every argument, which make the source code less friendly to write and a little bit harder to read with all the extra type specifiers getting in the way.

As a bonus, it also allows other hard to qualify options in Unicon, such as

```
# pass integer i (by address) get back an integer
ans := native("indirect", TYPEINT, [i, TYPESTAR])
write("indirect( ", i, ") = ", ans)
```

If the *C* function is defined as

```
int
indirect(int *i)
{
    return *i * 2;
}
```

The Unicon code listing above will call the function by passing the address *of a copy of* the integer value. This is not quite the same as pass by reference, but is pass by content. The called routine gets a pointer (common in *C* library functions) but will not be able to change the source data.

This is a limitation, but it protects the immutable property of Unicon base data types. This limitation will stand. The type override opens the possibility of calling a *C* functions with pointers, without falling back to writing a wrapper (unless the routine actually needs to change the referenced data for proper functioning). At that point, to protect Unicon immutable data, an extra wrapper routine would need to be written, burden on the Unicon programmer to follow the *loadfunc* model of C integration. *It's not a huge burden really, but does require a little bit of C code.*

There may be a future addition to the `uniffi.c` feature set to allow changing referenced data, by passing a list from Unicon. The caller would then be able to reassign the output values using normal Unicon assignment operators.

Note: There will *never* be a feature added that allows immutable Unicon data to be changed in place with `uniffi`. That goes too far against the grain and the spirit of Unicon programming.

The current `libffi` interface tests just as well as the hand rolled assembler. *To be honest it actually works better, many edge and corner cases have been debugged in libffi, and the cross platform support is a complete boon.*

```
/*
  A new Unicon C native FFI layer, with libffi
  Tectonics: gcc -o uniffi.so -shared -fPIC uniffi.c -lffi
  Phase 2 trial
     todo refactor the argument scanner
          work out ORing in the TYPESTAR arg override
*/
#include <stdio.h>
```

```
#include <ffi.h>

#include "icall.h"
#include "natives.h"

/* a dlopen handle */
static void *dlHandle;
/* a dlsym function pointer */
static void (*func)();

/* storage blob for arguments */
union blob {
    long lvalue;
    double rvalue;
    float fvalue;
    char *svalue;
};


#ifndef RTLD_LAZY                        /* normally from <dlfcn.h> */
#define RTLD_LAZY 1
#endif                                   /* RTLD_LAZY */

#if NT
void *
dlopen(char *name, int flag)
{ /* LoadLibrary */
    return (void *)LoadLibrary(name);
}
void *
dlsym(void *handle, char *sym)
{
    return (void *)GetProcAddress((HMODULE)handle, sym);
}
int
dlclose(void *handle)
{ /* FreeLibrary */
    return FreeLibrary((HMODULE)handle);
}

char *
dlerror(void)
{
    return "undiagnosed dynamic load error";
}
#else                                    /* NT */
#include <dlfcn.h>
#endif                                   /* NT */

#ifdef FreeBSD
/*
  If DL_GETERRNO exists, this is an FreeBSD 1.1.5 or 2.0
  which lacks dlerror(); supply a substitute.
*/
#ifdef DL_GETERRNO
char *
dlerror(void)
{
```

```
    int no;

    if (0 == dlctl(NULL, DL_GETERRNO, &no))
        return(strerror(no));
    else
        return(NULL);
}
#endif
#endif                                  /* __FreeBSD__ */

int
ffi(int argc, descriptor argv[])
{

    /* ffi_retval will point to an appropriate return slot */
    void *ffi_retval;
    /* ffi_rettype will point to the address of an ffi_type indicator */
    void *ffi_rettype;

    /* The cif setup requires the return type and value slot */
    ffi_cif cif;

    /* allow for 127 arguments, the C standard minimum limit */
    ffi_type *args[127];
    void *values[127];

    /* a variant result block */
    ffi_arg rc;

    /* ffi library call result codes */
    int ffi_stat;

    /* function lookup name, and possible alternate */
    char *funcname;
    char *funcname2;

    /* dlsym lookup error messages */
    char *dlMsg;

    /* local copy of args, probably dumped in phase 2 */
    union blob ipregs[127];
    union blob fregs[127];

    /* the return type enum from Unicon, in natives.h */
    long retType;
    /* the base Unicon argument data type, may be overridden */
    char inType;

    /* pointed to by ffi_retval, for ffi_prep_cif */
    long intSlot;
    double doubleSlot;
    void *pointerSlot;
    float floatSlot;

    /* current count stashed arguments */
    int ips = 0;

    /* first the function name */
```

```
    ArgString(1);

    /* second is return type, from natives.inc matched in natives.h */
    ArgInteger(2);
    retType = IntegerVal(argv[2]);

    /* look up the function entry point */
    funcname = StringVal(argv[1]);
    dlerror();
    *(void **)(&func) = dlsym(dlHandle, funcname);
    dlMsg = dlerror();
    if (dlMsg) {
        fprintf(stderr, "dlsym fail: %s\n", dlMsg);
        fflush(stderr);
        Fail; //Error(500);
    }

    /* try alternative name with initial underscore */
    if (!func) {
        funcname2 = malloc(strlen(funcname + 2));
        if (funcname2) {
            *funcname2 = '_';
            strcpy(funcname2 + 1, funcname);
            *(void **)(&func) = dlsym(dlHandle, funcname2);
            free(funcname2);
        }
    }
    if (!func) Fail;

    /* Return type, pass an indirect pointer to ffi_call */
    switch(retType) {
    case TYPEVOID:
        ffi_rettype = &ffi_type_void;
        ffi_retval = NULL;
        break;
    case TYPESTAR:
        ffi_rettype = &ffi_type_pointer;
        ffi_retval = &pointerSlot;
        break;
    case TYPEINT:
        ffi_rettype = &ffi_type_slong;
        ffi_retval = &intSlot;
        break;
    case TYPEFLOAT:
        ffi_rettype = &ffi_type_float;
        ffi_retval = &floatSlot;
        break;
    case TYPEDOUBLE:
        ffi_rettype = &ffi_type_double;
        ffi_retval = &doubleSlot;
        break;
    case TYPESTRING:
        ffi_rettype = &ffi_type_pointer;
        ffi_retval = &pointerSlot;
        break;
    default:
        ffi_rettype = &ffi_type_slong;
        ffi_retval = &intSlot;
```

```
        break;
    }

    /* pull out types and values */
    for (int argi = 3; argi <= argc; argi++) {

        inType = IconType(argv[argi]);
        /*
          fprintf(stderr, "%d arg: %d, IconType: %d '%c'\n",
                argc, argi, inType, inType);
          fflush(stderr);
        */
        int llen;
        struct descrip slot[2];

        int forceType;

        switch(inType) {
        /* Special case for lists, second value is forced datatype */
        /* currently a cheater stub only handling double/float cases */
        case 'L':
            llen = ListLen(argv[argi]);
            if (llen != 2) {
                ipregs[ips].rvalue = 0.0;
                ipregs[ips].fvalue = 0.0;
                break;
            }
            /* from looking at IListVal and RListVal */
            cpslots(&argv[argi], &slot[0], 1, 3);
            forceType = IntegerVal(slot[1]);
            switch(forceType) {
            case TYPEFLOAT:
                ipregs[ips].fvalue = (float)RealVal(slot[0]);
                args[ips] = &ffi_type_float;
                values[ips] = &ipregs[ips].fvalue;
                ips++;
                break;
            case TYPEDOUBLE:
                ipregs[ips].rvalue = RealVal(slot[0]);
                fregs[ips].rvalue = RealVal(slot[0]);
                args[ips] = &ffi_type_double;
                values[ips] = &(fregs[ips].rvalue);
                ips++;
                break;
            default:
                fprintf(stderr, "forceType %d not yet supported\n",
                        forceType);
                fflush(stderr);
                break;
            }
            break;
        case 'i':
            ArgInteger(argi);
            ipregs[ips].lvalue = IntegerVal(argv[argi]);

            args[ips] = &ffi_type_slong;
            values[ips] = &(ipregs[ips].lvalue);
            ips++;
```

```
            break;
        case 'r':
            ArgReal(argi);
            fregs[ips].rvalue = RealVal(argv[argi]);
            args[ips] = &ffi_type_double;
            values[ips] = &(fregs[ips].rvalue);
            ips++;
            break;
        case 's':
            ArgString(argi);
            ipregs[ips].svalue = StringVal(argv[argi]);

            args[ips] = &ffi_type_pointer;
            values[ips] = &(ipregs[ips].svalue);
            ips++;
            break;
        }
    }

    /* Initialize the cif */
    ffi_stat = ffi_prep_cif(&cif, FFI_DEFAULT_ABI, argc - 2,
                            ffi_rettype, args);

    if (ffi_stat == FFI_OK) {
        /* the magic call */
        ffi_call(&cif, *func, ffi_retval, values);
    } else {
        fprintf(stderr, "ffi_prep_cif failed: %d\n", ffi_stat);
        fflush(stderr);
        Fail;
    }

    /* Return type, as specfied in argument 2 enum */
    switch(retType) {
    case TYPEVOID:
        Return;
        break;
    case TYPESTAR:
        RetInteger((long)pointerSlot);
        break;
    case TYPEINT:
        RetInteger(intSlot);
        break;
    case TYPEFLOAT:
        RetReal(floatSlot);
        break;
    case TYPEDOUBLE:
        RetReal(doubleSlot);
        break;
    case TYPESTRING:
        RetString(pointerSlot);
        break;
    default:
        RetInteger(intSlot);
        break;
    }
    Error(500);
}
```

```
/* add a library to the dlsym search path */
/* need to pull code from src/runtime/fload.r */
int
addLibrary(int argc, descriptor argv[])
{
    //union block dlBlock;
    //descriptor dlBlock;
    /* Add a new library to the dynamic search path */
    ArgString(1)
    dlHandle = dlopen(StringVal(argv[1]), RTLD_LAZY | RTLD_GLOBAL);
    if (!dlHandle) {
        /*
          fprintf(stderr, "dlHandle error\n");
          fflush(stderr);
          Error(500);
        */
        Fail;
    }

    /* The return pointer, needs to get stashed properly */
    //dlBlock = mkExternal(dlHandle, sizeof(dlHandle));
    //RetExternal(dlBlock);
    RetInteger((long)dlHandle);
}
```

And an updated test head:

```
#
# uniffi.icn, demonstrate an experimental C FFI with libffi
#
$include "natives.inc"

link io
procedure main()
    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := pathload("uniffi.so", "addLibrary")

    # allow arbitrary C functions, marshalled by libffi
    native := pathload("uniffi.so", "ffi")

    # add the testing functions to the dlsym search path,
    #  the handle is somewhat irrelevant, but won't be soonish
    dlHandle := addLibrary("./testnative.so")
    write("Unicon dlHandle: ", dlHandle)
    write()

    # pass two integers, get the sum as int
    ans := native("testnative", TYPEINT, 40, 2)
    write("Unicon: called testnative and got ", ans)
    if ans ~= 42 then write("ERROR with testnative")
    write()

    # pass two reals, get the sum as real
    ans := native("testdouble", TYPEDOUBLE, 9.0, 8.0)
    write("Unicon: called testdouble and got ", ans)
    if ans ~= 17.0 then write("ERROR with testdouble")
    write()
```

```
# third arg is an integer, returns real
ans := native("testfive", TYPEDOUBLE, 1.0, 2.0, 3, 4.0, 5.0)
write("Unicon: called testfive and got ", ans)
if ans ~= 15.0 then write("ERROR with testfive")
write()

# get a pointer/handle
ans := native("teststar", TYPESTAR)
write("Unicon: called teststar and got ", ans)
if ans = 0 then write("ERROR with teststar")
write()

# a string
ans := native("teststring", TYPESTRING, "this is a string")
write("Unicon: called teststring and got ", ans)
if ans ~== "this is a string" then write("ERROR with teststring")
write()

# a string, and int and a real, returning string
ans := native("testmulti", TYPESTRING,
              "for multi", 42, &pi)
write("Unicon: called testmulti and got ", ans)
if ans ~== "for multi" then
    write("ERROR with testmulti")
write()

# a string, and int and a real, returning real
ans := native("testmultid", TYPEDOUBLE,
              "for multid", 42, &pi)
write("Unicon: called testmultid and got ", ans)
if ans ~= &pi then write("ERROR with testmultid")
write()

#
# Variant for float versus double
#
write("float variant")

# pass two reals (as float), get the sum as real (float)
ans := native("testfloat", TYPEFLOAT,
              [9.0, TYPEFLOAT], [8.0, TYPEFLOAT])
write("Unicon: called testfloat and got ", ans)
if ans ~= 17.0 then write("ERROR with testfloat")
write()

# a string, and int and a real (as float), returning real (as float)
ans := native("testmultif", TYPEFLOAT,
              "for multif",
              21, [&pi/2, TYPEFLOAT])
write("Unicon: called testmultif and got ", ans)
if ans - &pi/2 > 0.0000001 then write("ERROR with testmultif")
write()

#
# standard math lib
#
# todo: loader needs a path, but pathload stops, need a fail version
```

```
    dlHandle := addLibrary("libm.so")
    write("Unicon libm dlHandle: ", dlHandle)

    # call a Bessel function double form
    ans := native("j0", TYPEDOUBLE, &pi)
    write("j0(pi) = ", ans)
    # call Bessel function float form
    ans := native("j0f", TYPEFLOAT, [&pi, TYPEFLOAT])
    write("j0f(pi) = ", ans)
    write()

    # test switching back to the previous load library
    ans := native("testfloat", TYPEFLOAT,
                  [9.0, TYPEFLOAT], [8.0, TYPEFLOAT])
    write("Unicon: called testfloat and got: ", ans)
    if ans ~= 17.0 then write("ERROR with testfloat")
    write()
end
```

Nearly the same commands to prep the support function and build the test heads, as listed above under *C Native*. The only addition is `-lffi` to the *gcc* compile line, and a change of filename from `native.c` to `uniffi.c`. (I've taken to pronouncing that as "unify").

```
prompt$ gcc -o uniffi.so -shared -fPIC uniffi.c -lffi
```

The `testnative.c` *C* code remains almost the same, but added `libm.so` to test the `j0` and `j0f` functions (and to test that loaded libraries stick around and function lookups are cumulative).

```
prompt$ gcc -o testnative.so -shared -fPIC testnative.c
```

Here is a test run with a new top level Unicon filename `uniffi.icn` with the new list `[arg, TYPE]` specifiers included in some of the tests:

```
prompt$ unicon -s uniffi.icn -x
Unicon dlHandle: 14938640

In testnative 40 2
Unicon: called testnative and got 42

In testdouble 9.000000 8.000000
Unicon: called testdouble and got 17.0

In testfive 1.000000 2.000000 3 4.000000 5.000000
Unicon: called testfive and got 15.0

In teststar
Unicon: called teststar and got 140173140630067

In teststring with #this is a string#
Unicon: called teststring and got this is a string

In testmulti with #for multi# 42 3.141593
Unicon: called testmulti and got for multi

In testmultid with #for multid# 42 3.141593
Unicon: called testmultid and got 3.141592653589793

float variant
```

```
In testfloat 9.000000 8.000000
Unicon: called testfloat and got 17.0

In testmultif with #for multif# 21 1.570796
Unicon: called testmultif and got 1.570796370506287

Unicon libm dlHandle: 14938640
j0(pi) = -0.3042421776440938
j0f(pi) = -0.3042422235012054

In testfloat 9.000000 8.000000
Unicon: called testfloat and got: 17.0
```

Unified Foreign Function Interface. (Uniconified FFI). Running your Unicon on a Mac, Windows, 32bit, 64bit, FreeBSD or a z/Linux mainframe? `uniffi` has your back. Call as many library functions as you like, all from Unicon sources, no wrappers required.

The GnuCOBOL sample listed in *C Native* is identical.

The `libharu` integration now uses the type override system, as the PDF generator prefers `float` data arguments. It makes the listing a listing a little less easy to read, with all the type overrides, but not too bad, given the level of flexibility provided.

```
#
# haru.icn, demonstrate a newer C FFI
#
$include "natives.inc"

$define HPDF_COMP_ALL 15
$define HPDF_PAGE_MODE_USE_OUTLINE 1
$define HPDF_PAGE_SIZE_LETTER 0
$define HPDF_PAGE_PORTRAIT 0

procedure main()
    local dlHandle, pdf, page1, rc, savefile := "harutest.pdf"

    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uniffi.so", "addLibrary")

    # allow arbitrary C functions, marshalled by a piece of assembler
    # assume float instead of double, changes the inline assembler
    # movsd versus movdd
    native := loadfunc("./uniffi.so", "ffi")

    # add libhpdf to the dlsym search path, the handle is irrelevant
    dlHandle := addLibrary("libhpdf.so")

    pdf := native("HPDF_New", TYPESTAR, 0, 0)

    rc := native("HPDF_SetCompressionMode", TYPEINT, pdf, HPDF_COMP_ALL)
    rc := native("HPDF_SetPageMode", TYPEINT, pdf,
                 HPDF_PAGE_MODE_USE_OUTLINE)

$ifdef PROTECTED
    rc := native("HPDF_SetPassword", TYPEINT, pdf, "owner", "user")
    savefile := "harutest-pass.pdf"
$endif
```

```
    page1 := native("HPDF_AddPage", TYPESTAR, pdf)

    rc := native("HPDF_Page_SetHeight", TYPEINT, page1,
                 [220.0, TYPEFLOAT]);
    rc := native("HPDF_Page_SetWidth", TYPEINT, page1,
                 [200.0, TYPEFLOAT]);

    #/* A part of libharu pie chart sample, Red*/
    rc := native("HPDF_Page_SetRGBFill", TYPEINT, page1,
                 [1.0, TYPEFLOAT], [0.0, TYPEFLOAT], [0.0, TYPEFLOAT]);
    rc := native("HPDF_Page_MoveTo", TYPEINT, page1,
                 [100.0, TYPEFLOAT], [100.0, TYPEFLOAT]);
    rc := native("HPDF_Page_LineTo", TYPEINT, page1,
                 [100.0,  TYPEFLOAT],[180.0, TYPEFLOAT]);
    rc := native("HPDF_Page_Arc", TYPEINT, page1,
                 [100.0, TYPEFLOAT], [100.0, TYPEFLOAT],
                 [80.0, TYPEFLOAT], [0.0, TYPEFLOAT],
                 [360 * 0.45, TYPEFLOAT]);

    #pos := native("HPDF_Page_GetCurrentPos (page);

    rc := native("HPDF_Page_LineTo", TYPEINT, page1,
                 [100.0, TYPEFLOAT], [100.0, TYPEFLOAT]);
    rc := native("HPDF_Page_Fill", TYPEINT, page1);

    rc := native("HPDF_SaveToFile", TYPEINT, pdf, savefile);
    native("HPDF_Free", TYPEVOID, pdf);
end
```

What comes next? That is up to the creativity, imagination and needs of august Unicon developers. No need to write any *C* to get at *C*, or any language that uses the *C* application binary interface. As a (not completely) informed guess, I'd peg that at well over 75% of currently available computing resources, world wide. Even an Android phone plays nice with the *C* ABI internally.

Other features of `libffi` can be exposed as needs are determined. The `ffi_call` layer supports `sysv`, `unix64`, `win64`, `stdcall`, `fastcall`, `thiscall` and `cdecl` call conventions. Possibly others, depending on platform specific builds. By default, `uniffi` uses the calling convention most appropriate for the system used during builds by using `FFI_DEFAULT_ABI` when preparing the Call InterFace block.

28.1.4 baconffi

Putting *libffi* to use with *BaCon*, the BASIC Converter.

Like the GnuCOBOL example, this routine just sums two numbers passed in as arguments and returns the resulting integer.

```
REM basicnative.bac, uniffi call to BaCon BASIC
FUNCTION basic(NUMBER one, NUMBER two)
    RETURN one + two
END FUNCTION
```

This source is then converted to C, and a shared library is created.

```
prompt$ bacon -q -f basicnative.bac
```

```
Converting 'basicnative.bac'...
Converting 'basicnative.bac'... done, 15 lines were processed in 0.003 seconds.
Compiling 'basicnative.bac'... make[1]: Entering directory '/home/btiffin/wip/writing/
↪unicon/programs/uniffi'
cc  -fPIC -c basicnative.bac.c
cc -o basicnative.so basicnative.bac.o -L. -lbacon -lm -shared -rdynamic
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/programs/uniffi'
Done, program 'basicnative.so' ready.
```

A slightly more sophisticated BaCon program, that embeds some inline assembler. *Sample derived from a thread on the BaCon forums by vovchik and Axelfish*, http://basic-converter.proboards.com/thread/752/assembler-bacon

```
REM mixedbacon.bac, uniffi call to BaCon BASIC
FUNCTION embed(int operand1,int operand2) TYPE int
    USEC
    #define ASM asm(
    #define GAS );

    int sum, accumulator;

    ASM
        "movl %1, %0\n\t"
        "addl %2, %0"
        : "=r" (sum)                        /* output operands */
        : "r" (operand1), "r" (operand2)    /* input operands */
        : "0"                               /* clobbered operands */
    GAS
    accumulator = sum;

    ASM
        "addl %1, %0\n\t"
        "addl %2, %0"
        : "=r" (accumulator)
        : "0" (accumulator), "g" (operand1), "r" (operand2)
        : "0"
    GAS
    END USEC

    RETURN accumulator
END FUNCTION

REM
REM asmmix, to be called from Unicon uniffi
REM
FUNCTION asmmix(int one, int two)
    PRINT "BaCon received: ", one, ", ", two
    my_accumulator=embed(one, two)
    RETURN my_accumulator
END FUNCTION
```

Again, convert the source and compile to a shared library.

```
prompt$ bacon -q -f asmmix.bac

Converting 'asmmix.bac'...
Converting 'asmmix.bac'... done, 48 lines were processed in 0.005 seconds.
Compiling 'asmmix.bac'... make[1]: Entering directory '/home/btiffin/wip/writing/
↪unicon/programs/uniffi'
```

```
cc  -fPIC -c asmmix.bac.c
cc -o asmmix.so asmmix.bac.o -L. -lbacon -lm -shared -rdynamic
make[1]: Leaving directory '/home/btiffin/wip/writing/unicon/programs/uniffi'
Done, program 'asmmix.so' ready.
```

A Unicon test head to load in the `basic` function defined in `basicnative.bac` and then invoke the function through the *libffi* interface. Same sequence for the `asmmix` call.

```
#
# baconffi.icn, test calling BaCon without wrapper with libffi
#
$include "natives.inc"

procedure main()
    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uniffi.so", "addLibrary")

    # allow arbitrary C functions, marshalled by libffi
    native := loadfunc("./uniffi.so", "ffi")

    # add the testing functions to the dlsym search path,
    #  the handle is somewhat irrelevant, but won't be soonish
    dlHandle := addLibrary("./basicnative.so")

    # pass two integers, get back a sum
    write("Unicon calling BaCon function \"basic\" with 40 and 2")
    ans := native("basic", TYPEINT, 40, 2)
    write("Unicon got ", ans)

    # pass two integers that create an accumulator in assembly
    dlHandle := addLibrary("./asmmix.so")
    write("Unicon calling \"asmmix\" with 2 and 4")
    ans := native("asmmix", TYPEINT, 2, 4)
    write("Unicon accumulator of 2+4 + 2+4 + 2: ", ans)
end
```

Sample run:

```
prompt$ unicon -s baconffi.icn -x
Unicon calling BaCon function "basic" with 40 and 2
Unicon got 42
Unicon calling "asmmix" with 2 and 4
BaCon received: 2, 4
Unicon accumulator of 2+4 + 2+4 + 2: 14
```

And mixing Unicon with BASIC becomes another easy thing to do.

BaCon is an extraordinarily powerful BASIC translator. Lots of features.

http://www.basic-converter.org/

CHAPTER

TWENTYNINE

THEORY

29.1 Computer programming theory

Todo

only random thoughts so far, goal is codification

29.1.1 Unicon and Computer Science

Unicon is *very* well placed to be put to use advancing computational theory, computer science, software engineering, and other critical areas of digital construction.

Everything from simple Hello, world examples to complex graphing, execution monitoring and visualization, and untold as yet uncodified digital computing topics.

Below software development, in the theories surrounding result oriented computer programming.

29.1.2 Proving Unicon

Not easy, in the large, as there are building blocks and notations to create, and within these smaller pieces might be facts that can be proven.

```
seq()\1
```

How hard will it be to prove that that code will generate a one, and then be forced to fail by a limit expression?

Part of the "how hard will it be", is figuring out the level of detail involved in that expression. There are many. The C source code that writes the compiler, that compiles the Unicon, that runs a virtual machine that evaluates expressions. Running on a machine with effectively infinite combinations of electronic circuit state. It won't be easy.

Perhaps it is simply the lack of a viable notation. The Roman's, having recently conquered the Normans, could not tally the spoils of war. Despite having a great interest in doing so, and putting some of the best and brightest of the time on the task. The problems with the Roman numeral system did not allow for reckoning the sums. This type of math is easily solved by school children now that the notations used in arithmetic have changed.

Maybe a Unicon programmer will see the light that brings a quantum leap in computer science notation and level of mass understanding. Of all the programming languages, and design influences, maybe *Ralph Griswold* was on to something, and the situation just needs a little bit of extra insight to open up a new level of possibilities.

At the moment, correct software development is hard. Maybe we are in a phase of Roman numeral notation when it comes to source code and the human machine interface.

Or maybe we should step back to $\mathcal{P}''$ by Corrado Böhm, and start smaller? https://en.wikipedia.org/wiki/Corrado_B%C3%B6hm. Corrado describes a Turing complete theoretical system with 4 symbols, 4 rules of syntax, and 5 semantic clauses. https://en.wikipedia.org/wiki/P%E2%80%B2%E2%80%B2. Maybe more people should study the simple things, in hopes of making the step that replaces current models with a more capable system.

Either way, forward or backwards, I'll opine that Unicon can play a role in the evolution of provable digital construction and the as yet undiscovered notations that may get us there.

29.1.3 f = ma

Force equals mass times acceleration (velocity squared). $f = mv^2$. Energy equals mass times the fastest thing squared. $e = mc^2$.

Where is the

Result equals mass (lines of code, concrete instructions) times electrical impulse (squared?) theory/law? Result = instructions times pulses squared, $r = ip^2$?? I'd like to believe that Unicon can help narrow that down (if such a principle or relation actually exists).

Physics comes with massive parallelism. All points of gravity are self powered in the n-body equations, and each exerts an influence on the system independently. Unicon concurrent co-expressions kinda fit that bill. Discrete "points of gravity" able to influence the state of the system independently. Maybe.

CHAPTER THIRTY

ROSETTACODE

30.1 Unicon on rosettacode.org

Along with the *IPL*, rosettacode.org contains another treasure trove of Unicon programming examples, tips, and learning materials.

http://rosettacode.org

http://rosettacode.org/wiki/Category:Unicon

Everything from simple Hello, world examples to complex graphing and other meaty topics.

30.1.1 Some samples

Combinations and Permutations

A nice example of concise Unicon, and the benefits of large integer support.

http://rosettacode.org/wiki/Combinations_and_permutations#Icon_and_Unicon

```
procedure C(n,k)
    every (d:=1) *:= 2 to k
    return P(n,k)/d
end

procedure P(n,k)
    every (p:=1) *:= (n-k+1) to n
    return p
end
```

Where:

```
write("P(1000,10) = ",P(1000,10))
write("P(1000,15) = ",P(1000,15))
```

Gives:

```
P(1000,10) = 955860613004397508326213120000
P(1000,15) = 899864387800469514043972248293019354931200000
```

Two Sum

Not yet a Unicon guru, here is a possibly less than stellar RosettaCode entry by this author.

http://rosettacode.org/wiki/Two_Sum

```
#
# twosum.icn, find two array elements that add up to a given sum
# Dedicated to the public domain
#
link fullimag
procedure main(arglist)
    sum := pop(arglist) | 21
    L := []
    if *arglist > 0 then every put(L, integer(!arglist)) & L := sort(L)
    else L := [0, 2, 11, 19, 90]

    write(sum)
    write(fullimage(L))
    write(fullimage(twosum(sum, L)))
end

# assume sorted list, only interested in zero or one solution
procedure twosum(sum, L)
    i := 1
    j := *L
    while i < j do {
        try := L[i] + L[j]
        if try = sum then return [i,j]
        else
            if try < sum then
                i +:= 1
            else
                j -:= 1
    }
    return []
end
```

Sample run:

```
prompt$ unicon -s onesum.icn -x
21
[0,2,11,19,90]
[2,4]
```

The above sample does not quite highlight the expressive power of Unicon, the task requires zero or one solution. Unicon can do much better than that.

```
#
# twosum.icn, find two array elements that add up to a given sum
# a modification of http://rosettacode.org/wiki/Two_Sum
#
link fullimag
procedure main(arglist)
    # predictable results for the UP docset
    &random := 1

    # target sum
    sum := pop(arglist) | ?100

    # create a list of the rest of the arguments, or make it up
    L := []
    if *arglist > 0 then every put(L, integer(!arglist))
    else every 1 to 20 do put(L, ?100)
    writes(sum, ", ")
    write(fullimage(L))

    # keep track of result count
    results := 0
    every write(fullimage(pair := twosum(sum, L)), " ",
        L[pair[1]], " ", L[pair[2]]) do results +:= 1
    if results = 0 then write("[], no solution")
end

# O(n^2), nested iterations
procedure twosum(sum, L)
    every i := 1 to *L-1 do
        every j := i+1 to *L do
            if L[i] + L[j] = sum then suspend [i,j]
end
```

Sample runs:

```
prompt$ unicon -s twosum.icn -x
73, [94,32,38,27,97,30,37,18,59,85,91,64,70,92,46,52,9,16,56,56]
[4,15] 27 46
[12,17] 64 9
```

```
prompt$ unicon -s twosum.icn -x 10 0 1 2 3 4 5 6 7 8 9 10
10, [0,1,2,3,4,5,6,7,8,9,10]
[1,11] 0 10
[2,10] 1 9
[3,9] 2 8
[4,8] 3 7
[5,7] 4 6
```

Pathological floating point problems

http://rosettacode.org/wiki/Pathological_floating_point_problems

Found this one interesting, and was initially disappointed (but not for long), as the Unicon *real* data type falls into the trap of not having enough precision to produce correct results for this task.

The initial quick trial for the sequence convergence shows the problem:

```
#
# patho.icn, Pathological problems with floating point
#   From RosettaCode
#
link printf
procedure main()
    iterations := [3,4,5,6,7,8,20,30,50,100,200]
    every n := !iterations do {
        ans := patho(n, 2.0, -4.0)
        printf("%3d %19.15r\n", n, ans)
    }
end

# this sequence should converge on 6.0
procedure patho(n, a, v)
    if n < 3 then return v
    return patho(n -:= 1, v, 111-1130/v+3000/(v*a))
end
```

The sample run starts miscalculating the proper convergence pretty much after the first iteration, and then shortly fails catastrophically converging on 100, instead of 6.

```
prompt$ unicon -s patho-real.icn -x
  3  18.500000000000000
  4   9.378378378378379
  5   7.801152737752169
  6   7.154414480975333
  7   6.806784736924811
  8   6.592632768721792
 20  98.349503122165360
 30  99.999999999998930
 50 100.000000000000000
100 100.000000000000000
200 100.000000000000000
```

I mentioned it on the SourceForge forum, and how a REXX solution with inherent decimal arithmetic (and explicit control on how many digits are used in calculations) made for a very easy to read and correct solution. Bruce Rennie came to the rescue almost immediately.

Unicon supports infinite length integers, which can be scaled out to allow very precise control over the number of digits used in computations. A little bit of fixed point decimal arithmetic, and large integers make this problem a cake walk. Runs fast and accurate, and the source code is a very clean read.

Bruce posted a small example, which was modified a bit to suit a RosettaCode entry (and to fill out other parts of the task, to further demonstrate the ease of using highly precise fixed point decimal math in Unicon programs).

```
#
# Pathological floating point problems
#
procedure main()
    sequence()
    chaotic()
end

#
# First task, sequence convergence
#
```

```
link printf
procedure sequence()
     local l := [2, -4]
     local iters := [3, 4, 5, 6, 7, 8, 20, 30, 50, 100, 200]
     local i, j, k
     local n := 1

     write("Sequence convergence")
     # Demonstrate the convergence problem with various precision values
     every k := (100 | 300) do {
         n := 10^k
         write("\n", k, " digits of intermediate precision")

         # numbers are scaled up using large integer powers of 10
         every i := !iters do {
             l := [2 * n, -4 * n]
             printf("i: %3d", i)

             every j := 3 to i do {
                 # build out a list of intermediate passes
                 # order of scaling operations matters
                 put(l, 111 * n - (1130 * n * n / l[j - 1]) +
                        (3000 * n * n * n / (l[j - 1] * l[j - 2])))
             }
             # down scale the result to a real
             # some precision may be lost in the final display
             printf(" %20.16r\n", l[i] * 1.0 / n)
         }
     }
end

#
# Task 2, chaotic bank of Euler
#
procedure chaotic()
    local euler, e, scale, show, y, d

    write("\nChaotic Banking Society of Euler")
    # format the number for listing, string form, way overboard on digits
    euler :=
"27182818284590452353602874713526624977572470936999595749669676277240766303535_
  5475945713821785251664274274663919320030599218174135966290435729003342952605_
  5956307381323286279434907632338298807531952510190115738341879307021540891499_
  9348841675092447614606680822648001684774118537423454424371075390777449920695_
  5517027618386062613313845830007520449338265602976067371132007093287091274437_
  4704723069697720931014169283681902551510865746377211125238978442505695369677_
  7078544996996794686445490598793163688923009879312773617821542499922957635148_
  2208269895193668033182528869398496465105820939239829488793320362509443117301_
  2381970684161403970198376793206832823764648042953118023287825098194558153017_
  5671736133206981125099618188159304169035159888851934580727386673858942287922_
  8499892086805825749279610484198444363463244968487560233624827041978623209002_
  1609902353043699418491463140934317381436405462531520961836908887070167683964_
  2437814059271456354906130310720851038375051011574770417189861068739696552126_
  7154688957035035"
```

```
    # 1000 digits after the decimal for scaling intermediates and service fee
    scale := 10^1000

    # initial deposit, e - $1
    d := e - scale

    # show balance with 16 digits
    show := 10^16
    write("Starting balance:       $", d * show / scale * 1.0 / show, "...")

    # wait 25 years, with only a trivial $1 annual service fee
    every y := 1 to 25 do {
        d := d * y - scale
    }

    # show final balance with 4 digits after the decimal (truncation)
    show := 10^4
    write("Balance after ", y, " years: $", d * show / scale * 1.0 / show)
end
```

A much more satisfying run:

```
prompt$ unicon -s patho.icn -x
Sequence convergence

100 digits of intermediate precision
i:   3  18.5000000000000000
i:   4   9.3783783783783790
i:   5   7.8011527377521620
i:   6   7.1544144809752490
i:   7   6.8067847369236330
i:   8   6.5926327687044380
i:  20   6.0435521101892680
i:  30   6.0067860930312060
i:  50   6.0001758466271870
i: 100  99.9999999999998400
i: 200 100.0000000000000000

300 digits of intermediate precision
i:   3  18.5000000000000000
i:   4   9.3783783783783790
i:   5   7.8011527377521610
i:   6   7.1544144809752490
i:   7   6.8067847369236320
i:   8   6.5926327687044380
i:  20   6.0435521101892680
i:  30   6.0067860930312060
i:  50   6.0001758466271870
i: 100   6.0000000193194780
i: 200   6.0000000000000000

Chaotic Banking Society of Euler
Starting balance:       $1.718281828459045...
Balance after 25 years: $0.0399
```

With only a small bit of fixed point management, Unicon is more than capable of very precise fractional mathematics when *real* native floating point data may not be up to the task. Some care needs to be taken with the order of scaling within the large integer operations, but other than that, this style of programming is fairly straight forward. It shines

another bright light on Unicon for use in general purpose programming.

Note: The COBOL 2014 specification calls for 1,000 digits of internal precision for financially sound computations. Unicon can easily meet and exceed that requirement when using large integers in fixed point decimal calculations. At speed.

CHAPTER

THIRTYONE

NOTES

31.1 ABI

Application Binary Interface. A low level operating environment (chip and OS) specification detailing how call frames are expected to be handled when invoking subroutines; in terms of order of arguments and burden of cleanup being on caller or callee, to name a few of the issues involved.

C compiler tool chains eventually resolve to a platform specific binary form appropriate for the operating system and hardware. This ABI is what allows machine code to integrate assembler and other high level languages with an operating system. Unicon relies on a local C ABI when interfacing with the outside and inside world and for ensuring the virtual machine is correct for the local environment. Those details are all handled by the Unicon compiler and virtual machine implementation. Very few developers will ever have to worry about these details.

31.2 API

Application Program Interface. A specification detailing the routines, protocols and tools for building software applications. Developers can access a wide range of pre-existing application building blocks when using Unicon. Each API is different and (hopefully well) documented along with the library or framework that is was designed for. The API is the blueprint for building software from these components. The *POSIX* standard is one of the major API specifications available to a Unicon programmer on some operating systems, and the Microsoft Windows API is another.

Unicon itself, with the built in functions, form an API; to properly use Unicon functions the arguments must be of the right types, and be in the correct order. Sometimes a specific sequence of events is required to set up proper use and handle any values produced. All of these details make up an application program interface.

There is rarely a name given to any particular API, but the details are still required for proper use. Fortunately, Unicon is well documented and each large and small API is detailed for use by programmers, even though these details may not be explicitly labelled as an `API`.

31.3 ASCII

American Standard Code for Information Interchange. A character set encoding.

EBCDIC is another character encoding, mostly in use on mainframes.

Unicode is the newest and most encompassing character encoding.

Unicon is as of release 13, an ASCII programming language, Unicode support is on the *Help Wanted* list.

31.4 BaCon

The BASIC Converter. An amazing piece of software. Starts life as a shell script that converts the main BaCon source, written in BaCon, converting to C, and producing a BaCon native binary. A nicely complete, well documented BASIC system (and more than just BASIC being a BASIC/C hybrid), featuring an integrated Highlevel Universal GUI, full access to C libraries and copious examples. Generates *DSO* or native executable. The dynamic shared object modules can be loaded from Unicon, both as *loadfunc* and *libffi* functions. Direct C access with the `USEC` statement even allows for use of inline assembler from BaCon sources.

http://www.basic-converter.org/

Due to the C translation intermediates and the ease of *DSO* creation, almost all (if not all) BaCon functions and subroutines are accessible to the Unicon programmer with a simple *libffi* call. As with Unicon, there is an ever growing number of highly useful BaCon procedures available online. Perhaps not quite as organized as the *IPL*, the principal author of BaCon, Peter van Eerten keeps a handy index page on the BaCon website, http://www.basic-converter.org/#examples

31.5 BASIC

Beginners All purpose Symbolic Instruction Code. A programming language developed in 1964 by John Kemeny and Thomas Kurtz. BASIC evolved along with the use of personal computers to now having many hundreds of dialects and implementations. Originally a line numbered system with not much structure other than IF and GOTO, most dialects are now full fledged structured programming languages, that retain the ease of use paradigm.

For Unicon on Unix-like systems, the *BaCon* translator makes for easy BASIC integration with multilanguage solutions.

31.6 C

The C programming language was developed by the late Dennis Ritchie starting in 1969 while at Bell Labs, and became the re-implementation language of the Unix operating system.

C is also the base language for the reference implementation of Unicon, along with a custom variant, rtt, the runtime translator and rtl, the runtime language used when building Icon and Unicon.

31.7 comprehension

List comprehension is a term used to describe the syntactic construct of creating a list based on a generator expression, not a simple list of values.

Based on set comprehension and set-builder notation from mathematical set theory, distinct from other list operations such as map and filter.

For example:

$S = \{ 2 \cdot x \mid x \in \mathbb{N},\ 0 > x < 6 \}$

Is the set of all positive natural integer values less than 6, multiplied by 2.

In Unicon, the syntax would be something like

```
L := [: 2 * (1 to 5) :]
```

31.8 Creative Commons

Share alike licensing. https://creativecommons.org/licenses/by-nc/2.5/

The Creative Commons initiative has created a full suite of licensing terms for creative works. These licenses can be applied to images, software, text, or just about anything that can be covered by copyright. Authors are allowed wide leeway when deciding how a work can be shared.

Note: *Being not a lawyer, the paragraph below is personal opinion and carries no authority, nor can it be treated as legal advice:*

The `by-nc` version requires attribution of the original author and stipulates that redistribution rights are only given if there is no commercial aspect to the sharing, but derivative works are allowed.

See the link given above for the precise legal wordings and definitions. Even then, consult a lawyer if you require authoritative understanding of the issues at hand.

31.9 COBOL

The COmmon Business Oriented Language. An imperative programming language dating back to the very early 1960's. Still in heavy use in financial, government and other enterprise scale business environments. COBOL programming comes part and parcel with most mainframe settings. *GnuCOBOL* is a freely available compiler for this industrial strength programming language.

As a programming language, COBOL is designed to solve practical business problems more than the *computer science* that influences almost all other environments. COBOL syntax is verbose, words are commonly used instead of symbols. This can be viewed as pro or a con, depending on who you ask, but it makes for highly readable source code.

Until fairly recently, there was no free compiler option, and due to the nature of big business, COBOL can be a very expensive field to work in. Or used to be. *GnuCOBOL* provides a free alternative, so it may help keep people on the technology. There are billions upon billions of lines of production COBOL source code in the field, and lowering year to year cost burdens associated with most COBOL runtime environments may breathe new life into an aging, yet critical, part of the computing landscape.

The latest Standard, COBOL 2014, is just starting to make the rounds among COBOL compiler vendors.

31.10 DSO

Dynamic Shared Object. This is a technical acronym that differentiates shared libraries from dynamic shared link libraries. Linkage to DSO files happens at runtime.

31.11 Expect

A pseudo terminal interface allowing hosted control of interactive console applications, by Don Libes. `Expect` is in the public domain, being developed by a Department of the U.S. government and therefore not subject to copyright.

https://en.wikipedia.org/wiki/Expect

`Expect` is an extension of the *Tcl/Tk* programming language, mentioned here for comparison with the *Pseudo terminals* API added to Unicon by Qutaiba Mahmoud, which is built on the `pty` interface introduced in BSD UNIX back in 1983. Pseudo terminal support is now part of many operating environments, including GNU/Linux and Windows. Unicon boasts a very easy to use, cross platform, pseudo terminal interface.

31.12 Farberisms

Sayings, with style.

```
From here on up, it's down hill all the way.

We can dig ourselves out of this hole.

We took our eyes off the wrong ball.
```

To name but a few of the quaint expressions made famous by *David Farber*. See `farb.icn` and `farb2.icn` in the *IPL* for a whole bunch of these quotables.

31.13 Forth

A stack and extensible threaded word programming language, by Chuck Moore.

31.14 GCC

The *GNU* Compiler Collection.

Use it to build Unicon.

31.15 GNU

GNU is Not Unix.

Use GNU, Linux, and free software.

It's good for you, and your/my/our freedoms.

31.16 GnuCOBOL

A free software implementation of *COBOL*.

https://sourceforge.net/projects/open-cobol/

Due to some of the implementation details of `GnuCOBOL`, *it is a transpiler, emitting intermediate C code*, it can be integrated with `Unicon` executables, each gaining from the innate strengths of the other.

See *unicob.cob* for a working example.

31.17 Graphics Programming in Icon

The original Icon graphics programming book by *Ralph Griswold*, *Clinton Jeffery*, and Gregg M. Townsend, published by Peer-To-Peer Communications in 1998. It went out of print, and the rights reverted to the authors, who then placed it in the public domain. Many thanks to the authors and Peer-To-Peer.

Get a copy from https://www2.cs.arizona.edu/icon/gb/index.htm

Still highly relevant for modern Unicon programming.

31.18 Help Wanted

The Unicon project maintains a Help Wanted section. Some items on the wish list require very high levels of technical skill and programming literacy, some just require a willingness to help out.

http://unicon.sourceforge.net/helpwanted.html

Check it out, and if you want to help, drop a note to the contact listed on the Unicon homepage.

http://unicon.sourceforge.net/

31.19 Icon

The Icon programming language, heavily leveraged by Unicon, awesome.

https://www2.cs.arizona.edu/icon/

31.20 Icon version 9

Version 9 is the latest, and last Icon release version. Before his passing, Ralph Griswold let it be known that he wanted the feature set of Icon to be frozen; a completed work. *And so it shall be. A testament to a great teacher, and a true Computer Scientist.*

The University of Arizona Computer Science Department will make periodic updates, to fix any critical bugs (not many of those left to find, as Icon is a very mature, and meticulous code base), or to keep up with changes in C compilers, and operating systems; so the sources will always properly build.

Version 9.5.1 (at time of writing) can be found at

https://www2.cs.arizona.edu/icon/current/

31.21 JSON

JavaScript Object Notation.

31.22 Locale

A system used in Internationalization and Localization. i18n/L10n.

Localization is a detail rich subset of modern programming.

See https://www.gnu.org/software/gettext/manual/html_node/index.html, the GNU `gettext` manual, for a full explanation of the issues involved with robust locale aware programming.

Note: Unicon, being an *ASCII* based system by design, cannot elegantly deal with Unicode or other multi-byte character encodings at this time. Special care and attention must be paid when dealing with non ASCII character sets in Unicon programs. Treat the data as binary buffers, and do not rely on any one to one correspondence between code points and bytes.

31.23 mutable colour

A colour created with *NewColor*. Supports changing existing pixel colours on the display hardware. Not supported by all builds of Unicon. Mutable colours are encoded in Unicon as a negative number entry in the colour map.

31.24 PHP

Currently a recursive backronym, *PHP: Hypertext Preprocessor*. Originally it was *Personal Home Page*, a utility layer developed by Ramus Lerdorf in 1994 to assist in the creation of websites. Much has changed since Ramus started the project, and PHP is now one of the most widely used programming languages in existence.

Unicon can compete with, and integrates nicely with PHP when developing web applications. A Technical Report written by *Clinton Jeffery* includes information on using Unicon for CGI and interoperating with PHP.

http://unicon.org/utr/utr4.html

See https://secure.php.net/ for more information.

31.25 POSIX

Portable Operating System Interface. An IEEE standard for maintaining compatibility between operating systems. An acronym suggested by *Richard Stallman* of *GNU*.

https://en.wikipedia.org/wiki/POSIX

31.26 REXX

Restructured Extended Executor, an interpreted programming language developed in the late 1970s by Mike Cowlishaw while at IBM.

Unicon can integrate REXX programming using either the ooRexx or Regina implementations. See *REXX* for a working example that allows use of both the C and C++ *API*.

```
SAY "Your name please? "
PARSE PULL name
SAY "Well met, ", name

math = "say 6 * 7"
INTERPRET math
```

Standard documented in *ANSI X3.274–1996 "Information Technology – Programming Language REXX*

31.27 Richard Stallman

rms originated the Free Software movement. He founded the Free Software Foundation, launched the *GNU* project, and developed *GCC* and GNU Emacs, (along with many other software systems).

https://en.wikipedia.org/wiki/Richard_Stallman

31.28 SEXI

String EXtraction Interpreter, a name originally used for *SNOBOL*.

31.29 serif

In typography, serifs are the line extenders attached to the end of a line in a letter or symbol.

31.30 SNOBOL

An early string and pattern manipulation programming language, circa 1963, designed and originally implemented by *Ralph Griswold* and others. Still in use.

StriNg Oriented symBOlic Language.

See *SNOBOL4* for a few examples, and a small Unicon program that runs and captures output from SNOBOL4 programs, as implemented by the SNOBOL in C project homed at https://sourceforge.net/projects/snobol4/

The links to SNOBOL run deep in Unicon. See *Patterns* for one major example of the influence.

SNOBOL, a precursor to Icon, hence Unicon is a label and branch heavy programming language.

```
Label   Statement body                    :Goto
```

That's SNOBOL in a nutshell, three fields. A label, a statement and a jump.

Labels can be pretty much any name, except `END`, the end of program text marker. When there is no label, statements can't start in column 1, and are usually indented to column 8 or 9.

Statements actually have three parts, `Subject Pattern = Replacement`. There can be `Subject` only (which can be function calls, primitives and expressions), `Subject = Replacement` assignments, `Subject Pattern` pattern matching, or all three components for replacement of matched patterns within the subject.

This seed of an idea grew to be Unicon string scanning, within a much more structured syntax.

The goto can be unconditional `:(label)`, jump on success `:S(label)`, jump on failure `:F(label)` or separate branches for both `:S(label) F(label)`. Target labels could also be computed, and referenced in variables `:($VAR)`.

More fully, source lines can be

```
LABEL   SUBJECT PATTERN = REPLACEMENT   :S(success-label) F(fail-label)
```

There are quite a few reserved words, in tune with computer programming of the era. PUNCH was a reserved word that meant write data to a punch card. INPUT was for reading from a paper based console keyboard (or card reader in the early early days), and OUTPUT wrote to the console printer. At the time, machines were mostly upper case. These device names could be redirected to and from files as time went on and 50 pound display screens became a thing and disk drives were invented (costing a mere $10,000 per megabyte (or more) in an era when top tier programmers made less than that in a year).

```
* Hello, SNOBOL style, circa 1964 (before Hello, world was a thing)
        OUTPUT = "HELLO, WORLD"
END
```

```
* Prompt, on paper, read from keyboard, echoed on paper, loop
PROMPT  OUTPUT = "NAME PLEASE? "
        NAME = INPUT                               :F(END)
        OUTPUT = "HELLO " NAME                     :(PROMPT)
END
```

Within the winds of fate, SNOBOL eventually evolved into structured Unicon. *Hooray, fate*.

31.31 Tcl/Tk

A scripting language invented by John Ousterhout, designed for inclusion inside other applications to provide a Tool Command Language. The goal was to alleviate the need for building custom scripting engines in these applications. Initial releases of `Tcl` were quite successful, and use expanded beyond an embedded scripting engine to become a full fledged development platform. Tcl also includes a graphical user interface layer, `Tk`, ToolKit.

https://en.wikipedia.org/wiki/Tcl

A Tcl version of the *Summing integers* integer summation comparison is included in the *Performance* chapter of this docset.

31.32 Tectonics

The expression **tectonics** used through this document is based on the following (fairly archaic) definition.

```
"Tectonics" gcide
"The Collaborative International Dictionary of English v.0.48"
Tectonics \Tec*ton"ics\, n.
1. The science, or the art, by which implements, vessels,
dwellings, or other edifices, are constructed, both
agreeably to the end for which they are designed, and in
conformity with artistic sentiments and ideas.
[1913 Webster]
```

Found from a lookup using the **dict://** protocol and the bank of open servers.

The term is used here as nerd slang, for describing the code building process.

The inference is that building code with Unicon is a very agreeable mix of art and science. With overtones of the geological term, and being rock solid.

31.33 uniclass

`uniclass.dir` and `uniclass.pag` are two external database files which make up `uniclass`, used with the class and package system of Unicon. These files form a DBM database file, entirely managed by the Unicon compile and link system, much like *ucode*. Being a *DBM* dataset, the entries are simply treated as a persistent table. Use *key* after `open("uniclass", "dr")` to see the keys which can then be used as table indexes to see the data.

```
# use "dr" for read only, don't muck with this database
uc := open("uniclass", "dr")
every k := key(uc) do write(k, ": ", uc[k])
close(uc)
```

31.34 Unix epoch

When Unix(tm) was first in development, the engineers needed a time reference. They picked January 1st, 1970. Many computer clocks have been counting up seconds based on that epoch ever since.

31.35 VAX

A midrange computer, by Digital Electronics Corporation.

31.36 VimL

The de facto naming convention of the Vim script language. Bram never really published a name for the internal language used for Vim scripting. Someone eventually coined `VimL` as a name for the language, and it has spread to *unofficial* common use; with the side benefit of helping find relevant internet search results.

31.37 VMS

Virtual Memory System, by Digital Electronics Corporation.

Version 9 of Icon (with graphics) is known to work with OpenVMS.

31.38 VOIP

Voice Over IP, or more fully, Voice Over Internet Protocol. A telephony feature optionally supported by Unicon. Accessed with *open* mode `v`.

See http://www2.cs.uidaho.edu/~jeffery/unicon/reports/zsharif-voip.pdf for details.

31.39 y2k

The Year 2000 problem, an issue caused by using 2 digit years in computer systems developed in the 20th century, ill prepared to rollover from the 1900s to the 2000s. The *Year 2038 problem* may end up being worse.

31.40 Year 2038 problem

Is a pending, possible problem, when epoch based signed 32 bit counters of seconds roll into the sign bit and turn hugely negative. The `epoch` was arbitrarily set to January 1st, 1970 during the early days of UNIX design and development. Internal computer clocks have been counting seconds since then. The counter field is (or was) a signed 32 bit field, and the 31 bits of count will rollover into the sign field at 03:14:07 UTC on 19 January 2038. The computer will then think it is 8:45 pm, December 13th, 1901. How most low level systems will react to that will require case by case study.

New systems (as of about 2012), use an internal 64 bit counter, enough seconds to exceed the theoretical life span of the Universe, but embedded and older systems with a 32 bit clock, will encounter unpredictable issues. *Well, these are computer programs. Rarely "unpredictable". But without explicit study of each and every occurrence of clock field use, the overall system outcomes may as well be viewed as unpredictable. The number of possible faults, and the effect of clock counter rollover will need analysis to determine the overall system impact.*

An event similar to epoch rollover may have already occurred. The Deep Impact mission satellite stopped responding, 2^32 1/10ths of a second past its internal epoch of January 1st, 2000, on August 11th, 2013 at 00:38:49. Engineers at NASA believe that the system went into a perpetual reboot state, when the internal clock counter (counting 10 times per second) rolled. That meant commands could no longer be received by the satellite, or acted on. Corrections could not be uploaded due to the perpetual reboot. Without thruster control the spacecraft is in an unknown orientation, antennas possibly pointed in the wrong direction, and solar panels not in a position to recharge batteries. Attempts to correct the problem and contact the spacecraft were officially abandoned on September 20th, 2013.

31.40.1 Unicon epoch rollover risk is nearly zero

Very old copies of Icon might be at risk of the `epoch bug`, which is more dependant on the operating system than the actual Icon runtime.

`Unicon`, being in active development, will almost certainly be based on 64 bit clocks by the time 2038 rolls around. Unicon developers will not generally need to worry about the `epoch rollover bug`.

Or to put it another way. If, *long if*, Unicon is running on a system that is susceptible to epoch counter sign rollover, it won't be Unicon that takes the system down or causes problems. There will be far too many other possible component failures to see that forest from those trees.

CHAPTER THIRTYTWO

CHANGELOG

19-Oct-2019, Oct-20, Oct-25, Oct-27 Small corrections and typo fixes. More consistent use of label syntax for internal :ref: tags (work in progress). https fix, link string scanning to the ? operator. Update IPL entry with core.icn hint, update www.cs. links to www2.cs..

14-Sep-2019 Correction to the (1 to 3) & 4 curiousity in expression.rst.

12-Apr-2019 Name typo, update some of the build options in unicon.rst.

02-Feb-2019, 23-Feb Add scenarios.rst for use case scenarios for Unicon. Index entries for tighloop samples.

22-Nov-2018 Add blurb about complex systems in Icon.

22-Oct-2018, 23-Oct Corrections suggested by Clinton, added ucl.rst, the Unicon Class Library with a first entry of the new JSON support class. Typo fixes and try to add space to the HTML bottom previous/next links.

12-Sep-2017 Add more function entries.

12-Jul-2017, 16-Jul Corrected some of the wording in the Year 2038 notes entry. Moving back to the GPL for licensing.

13-Jun-2017 Add Groovy and Nickle to performance (unfinished in terms of charting). Fix typo in charting.icn that had loadfunc faster than C timing.

10-May-2017 More operators, add Smalltalk performance.

03-Apr-2017, 08-Apr, 09-Apr, 11-Apr, 16-Apr, 17-Apr, 27-Apr Duplicate the precedence chart in operators.rst, add PHP tightloop. Update unicon.rst overview and add some notes, add Ada tightloop. Add Elixir performance, reorganize summary chart. Add more on run-benchmark. More operators. Even more operators. Add Rust performance.

04-Mar-2017, 15-Mar Tweak CSS with a nudge over and light gray filler. More functions.

09-Feb-2017, 28-Feb Operators as functions sample. Bump to v0.6 with builder in Python3, drop i18n-l10n gettext warning about invalid utf-8, drop second preamble.

13-Jan-2017, 14-Jan, 18-Jan, 28-Jan, 31-Jan More notes. Small tweaks to Unicon overview. Background colour for PDF verbatim boxes. More tweaks, and experiments with CSS. Left align placement of document pages.

01-Jan-2017, 02-Jan, 03-Jan, 04-Jan, 05-Jan, 07-Jan, 08-Jan, 09-Jan, 12-Jan Tweaks to libffi entry. More tweaks with uniffi haru.icn added, new blog entry. Updated some notes. Attempt css tweaks (widen in particular). More tweaks, simplify reserved word formatting to bold. Move uniffi and unitest samples to a separate directory. Add unilist list sample, add Tiny C embedding sample. Add numbers.icn to IPL blurb, more functions. Add BaCon BASIC uniffi sample to multilanguage.rst.

30-Dec-2016, 31-Dec Add the experimental native C calling feature via assembler to multilanguage. Add libffi info along with native assembly, uniffi.

16-Dec-2016, 17-Dec, 18-Dec, 20-Dec, 21-Dec, 23-Dec, 24-Dec, 25-Dec, 27-Dec Re-org performance headings, add ooRexx loadfunc sample. Add more on ucode. Add some icode blurbs, add assembler tightloop. Add a performance timing barchart and reorganize the section. Small tweaks to chart. A few more function entries. Added some RosettaCode entries, touched on theory meanderings. Bump to version 5, move down to LGPL license, status codes carried through with unicon -x (custom change at the moment). Touch on tools.rst.

01-Dec-2016, 02-Dec, 03-Dec, 06-Dec, 08-Dec, 09-Dec, 11-Dec, 13-Dec, 14-Dec Added an assembler loadfunc sample. Add PH7 sample, add vedis sample. Add UnQLite sample. Changed in unicon.rst, new note, more on operators. Tweak preamble. Add libcox sample, typo fixes and some corrections as pointed out by Clinton. Tweaks to unicon.rst, start in on using function directive. Add Lua to performance. Add ALGOL, shell, Neko, Nim, Vala, Vala/Genie and ECMAScript performance samples.

22-Nov-2016, 23-Nov, 24-Nov, 25-Nov, 26-Nov, 27-Nov, 28-Nov, 30-Nov A little bit more on structures, change body_text_align conf to justify, bump to 0.3. Touch ups in expression.rst, reserved and tools, work on image sizing between PDF and HTML builds. Tweak overview section. More overview details. More functions, expand on tools and vim syntax file, fight with new favicon.ico, added features.rst along with some new notes. Add EvSend (bogus), libz sample, reorgs and fix ups, added SNOBOL, REXX D, and Perl to the performance samples, more functions. More functions, try different rst roles. Fixed parent() description, add Java to performance, fix parent function description. Add a ucode highlighter to Pygments icon.py.

08-Nov-2016, 10-Nov, 11-Nov, 13-Nov, 14-Nov, 15-Nov, 16-Nov, 19-Nov, 21-Nov Small tweaks to performance code listing formatting. More functions, update tools.rst. Add libsoldout program sample. Update Unicon overview, move to sourceforge, add a favicon. New blog entry about the forge build, add Lua loadable program sample, mention make in tools.rst. Add Fortran alpha loadfunc trials. Add Fortran, Guile and BaCon to the performance timings. Added AJAX to networking.rst. Touch on threading.rst, move SVN checkout location to shorten command listings.

14-Oct-2016, 15-Oct, 17-Oct, 23-Oct, 25-Oct, 26-Oct, 27-Oct, 28-Oct, 30-Oct Done most of the core Unicon functions. More graphic functions, add theory.rst. Update repl entry to show off new list replication, add a rosettacode sample, add Tcl and COBOL to the tightloop time trials. Lots of tweaks, more functions. Add ficl loadfunc program sample, added ficl and S-Lang performance entries. Dropped the alpha status warning and bump to release 0.2, blog entry on the admin invite, updated tools.rst, more keywords. More keywords, Event sample now using Enqueue. Finished pending items in keywords.rst. Update intro, tweaks, more functions, fill in graphic attribute list.

02-Oct-2016, 03-Oct, 04-Oct, 06-Oct, 08-Oct, 09-Oct, 10-Oct, 12-Oct, 13-Oct Add multilanguage.rst. Start using start-after for code listing includes and update all current sample headers. Start in on testing.rst. Touched on documentation.rst, update one of the build machines to Fedora 24 and associated update of Sphinx, more functions, keywords. More functions. More functions. Licensed under the GPL 3.0+. More functions. More functions.

29-Sep-2016 Updated precedence chart.

16-Sep-2016, 17-Sep, 18-Sep, 19-Sep, 20-Sep, 21-Sep, 26-Sep, 27-Sep, 28-Sep Added lists.icn to ipl.rst, and wrap sample. More function samples (into the C's). More functions, new blog entry mentioning unittest. More functions (starting in on the D's), touched on threading. More reserved word entries. Initial pass of reserved.rst almost complete. Tweaks, more functions. Added link to Graphics Programming in Icon. More functions, a little tweaking of icon.py highlighter, removed race condition from the thread reserved word sample (thanks to Jafar), more on string scanning, try actdiag.

01-Sep-2016, 02-Sep, 06-Sep, 07-Sep, 08-Sep, 09-Sep, 11-Sep, 13-Sep, 14-Sep Second step with Duktape, added valgrind report to the sample. Add objects.rst, add uniruby loadfunc for mruby. Rename statements to reserved.rst, categorize the reserved words. Added readline and gettext sample programs, with notes about the Spanish capture being wrong and the need to update the Sphinx build tool to Python 3. Added initial expression precedence list, add threading.rst, change all -quiets to -s. Added fizzbuzz, added colour name options provided by Jafar, added to ipl.rst, added list comprehension example. Added uval program trials for eval(), expanded on patterns and regex. A few more function examples.

27-Aug-2016, 28-Aug, 30-Aug, 31-Aug Added patterns.rst, first SNOBOL conversion sample, Kudos blog post, added the full todo list to the bottom of the main Blog.rst file, added programs.rst with an initial S-Lang embedded interpreter example. Update statements, add COBOL loadfunc sample program. Update slang.c, add slangFile and flatten S-Lang arrays, update to unicon -s for quiet mode. icall.h fixed by Jafar, mkRlist now properly an array of double, added the first step of a Duktape ECMAScript integration program, more operator stubs.

18-Aug-2016, 19-Aug, 20-Aug, 21-Aug, 22-Aug, 23-Aug, 24-Aug, 25-Aug, 26-Aug Corrections as pointed out by Jafar, started in on structures. Execution Monitoring, add sphinxcontrib-blockdiag, add networking.rst, start adding get the code links. Added statement stubs, reserved words and starting in on functions.rst, added debugging.rst and testing.rst. Put pending on most programs to speed up the doc build, added documentation.rst and preprocessor.rst. Miscellaneous updates and filling in entries. More functions, some edits suggested by Jafar regarding ifdef feature testing. Finished keywords from data in src/runtime/keywords.r, consistent footers, simple loadfunc example. Experiment with ABlog. Update networking.rst, add a Blog entry with another Jafar fix, and Ruby to performance, tweaked explanations in expressions.rst, added some pictures to unicon.rst, statements was missing a stub for abstract.

01-Aug-2016, 02-Aug, 03-Aug, 11-Aug, 13-Aug, 14-Aug, 15-Aug, 16-Aug, 17-Aug More work on the Icon and Unicon Pygments source highlighter. Start using program-output Sphinx contribution for working samples. Adding keywords, added show_related to Alabaster config, add note about Icon vs Unicon and the origin of some features. Separate ChangeLog file from notes, adding keyword stubs and some people. More keywords, add monitoring.rst. More examples, add graphics.rst. Add expressions.rst, more keyword samples. (Birthday) Add rosettacode.rst and the PlotPairs sample. Added the gui.icn button sample. Add database.rst, performance.rst. Add operators.rst.

29-Jul-2016, 30-Jul, 31-Jul Started the Unicon Programming document set. Icon history added. Adding datatypes.rst and structures.rst.

CHAPTER THIRTYTHREE

LICENSE

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
```

```
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.
```

```
  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
```

```
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
```

```
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
```

```
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
```

```
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
```

```
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
```

```
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
```

```
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
```

```
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs
```

```
  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

CHAPTER

THIRTYFOUR

BLOG ENTRIES

34.1 Execution Monitoring

Jafar Al-Gharaibeh and *Clinton Jeffery* fixed an execution monitoring bug that was caused by interactions with Concurrency and fast Native Co-Expression switching.

https://sourceforge.net/p/unicon/code/4484/

34.2 https with no newline in response

There was a problem producing a single line response from a web request that lacked a terminating newline. Or, the last line of any response that lacked a newline.

Discovered while testing SourceForge JSON API calls.

Jafar Al-Gharaibeh fixed this in https://sourceforge.net/p/unicon/code/4489/

34.3 https with redirection

Jafar Al-Gharaibeh fixed an https redirection problem, first noticed with SourceForge.

https://sourceforge.net/p/unicon/code/4486/

34.4 Unicon bug fixing

While starting up the `Unicon Programming` docset, I have been more than pleasantly surprised at the turn around times with `Unicon` development. The few bugs that have been reported have been fixed within a matter of hours.

This is a good testament to the core Unicon team, and bodes well for developers that decide to use `Unicon` when coding solutions to their problems.

See https://sourceforge.net/p/unicon/bugs/199/ for an example. The new pattern operators tripped up automatic semicolon insertion, and *Clinton Jeffery* had the fix posted, ready for testing, in no time.

All the other reports have had similar treatment. Execution monitoring, HTTPS support issues, to name a few. All handled and fixed, within very short time frames. What more can you ask?

34.5 SNOBOL pattern example

Added a document file for Unicon `patterns` and *pattern* data.

Patterns

34.6 Unicon loadfunc

I've been playing around with the Unicon *loadfunc* built in function.

After the first couple of sample *Programs*, it is becoming abundantly apparent that Unicon has vast potentials.

With very little effort, Unicon is now sporting prototype integrations with

- S-Lang
- COBOL
- Duktape ECMAScript
- mruby

These layers will only get stronger and more robust as build outs continue, and already imply that great things lie in store.

All it takes is an idea, and some few lines of code, to expose entirely new ways of leveraging the powers baked into Unicon. High level Unicon can enlist various assistants and allow all kinds of pre-existing solutions to be part of a larger Unicon development effort. If only as a temporary measure until full Unicon source solutions can be put in place.

Along with the already tight integration with C, it seems the sky is nowhere near the limit when it comes to applying Unicon to modern programming problems.

Can't recommend strongly enough how all developers should spend some time getting to know the Unicon programming language. The future awaits the intrepid programmer. A future, just waiting to be conquered.

34.6.1 In other news

Clinton Jeffery has been busy polishing the *Pattern* features in Unicon 13 alpha. Recent commits include extensions to *Unicon monitoring* and instrumentation surrounding patterns. These additions are sure to make some SNOBOL programmers quite jealous, and increase the incentives to port some SNOBOL code over to a Unicon implementation. See Rev 4504 in the Unicon source tree for a few of the details.

Have good, make well.

34.7 Impressed

Continue to be impressed by the responsiveness of the Unicon development team.

Unicon is building up toward release 13. The beta working copies in SVN are used to build up this document set, and rarely fail to deliver.

I've put in a few bug reports for edge case failures, and all the critical ones have been fixed in well under an 8 hour window. Much kudos to the team.

Jafar Al-Gharaibeh and *Clinton Jeffery* are two highly productive and engaged developers. Nice to be following.

Some examples: take a look at https://sourceforge.net/p/unicon/bugs/202/ and https://sourceforge.net/p/unicon/bugs/200/. The first, a segfault issue, and the other a beta work in progress build failure. Both fixed within 3 hours of reporting. Nice. *And this is on top of all the other code and documentation commits that are happening. Nicer.*

34.7.1 The UP docs

This Unicon Programming doc set is progressing along nicely. Happy to have taken on the task. There are so many features in Unicon that it is a complete pleasure to add each new entry. Every sit down is a joy.

Unicon is both practical, and deep. A single expression chain can be a wonder to behold, and sometimes a challenge to come up with just the right idiomatic Unicon. Not hard, per se, but immensely satisfying. Do you add an `if` or use a conjunction? Separate out an assignment or chain it inside an indexing operation? When things need to get done, they get done, but there are times when pondering a single line, polishing and crafting, is more than half the fun. A sensation that I'd guess the great authors experience when the sentence just seems perfect. *(Or perhaps the pleasure and pain, and the angst, when the right combination of words seems just out of reach, only to suddenly appear on the page, with a sigh of relief and a smile.)*

34.7.2 Unit testing

Along with some experiments with inline expression evaluation, using Tasks and multi-tasking loadable co-expressions, I've started in on a small Unicon unit testing framework. So far, some few 140 lines of framework code is allowing for simple unit test passes. For example:

```
##-
# Author: Brian Tiffin
# Dedicated to the public domain
#
# Date: September 2016
# Modified: 2016-10-23/05:11-0400
##+
#
# literate.icn, Unit testing, in source
#

$ifndef UNITTEST
##
# unit testing experiment
procedure main()
    write("compile with unicon -DUNITTEST for the real show")
end

$else
link unittest
##
# unit test trial
procedure main()
    ##-
    #
    #
    #
    ##+
    speaktest := 1
    looplimit := -1
    test("1 + 2")
    test("return 1 + 2", 3)
```

```
    test("return 1 + 2", 0)
    test("return write(1 + 2)", 3, "3\n")
    tests("suspend 1 to 3", [1,2,3])
    tests("syntaxerror 1 to 3", [1,2,3])
    looplimit := 4
    tests("suspend seq()\\5", [1,2,3,4,5])
    tests("suspend seq()\\4", [1,2,3,4])
end
$endif

$ifdef DOC
===================
Unicon unit testing
===================

- test(code, result, output, errorout) - singleton
- tests(code, result, output, errorout) - generators

Set ``speaktest`` to non-null for verbose reporting
Set ``looplimit`` to a reasonable value for infinite loop break.
$endif
```

Sample run:

```
prompt$ unicon -s -DUNITTEST literate.icn -x
################## Test:    1 ##################
1 + 2
Trials: 1 Errors: 0  Pass: 1 Fail: 0
################################################

################## Test:    2 ##################
return 1 + 2
Expecting: 3
Received: integer, 3
Trials: 2 Errors: 0  Pass: 2 Fail: 0
################################################

################## Test:    3 ##################
return 1 + 2
Expecting: 0
Received: integer, 3
Trials: 3 Errors: 0  Pass: 2 Fail: 1
################################################

################## Test:    4 ##################
return write(1 + 2)
Expecting: 3
3
Received: integer, 3
Trials: 4 Errors: 0  Pass: 3 Fail: 1
################################################

######## Generator test:    1 ##################
suspend 1 to 3
Expecting: [1,2,3]
Received: [1,2,3]
Trials: 1  Errors: 0  Breaks: 0  Pass: 1 Fail: 0
################################################
```

```
######## Generator test:    2 ##################
syntaxerror 1 to 3
Expecting: [1,2,3]
Received: []
Trials: 2  Errors: 1  Breaks: 0  Pass: 1 Fail: 0
################################################

######## Generator test:    3 ##################
suspend seq()\5
Expecting: [1,2,3,4,5]
fail lequiv 4
Received: [1,2,3,4]
Trials: 3  Errors: 1  Breaks: 1  Pass: 1 Fail: 1
################################################

######## Generator test:    4 ##################
suspend seq()\4
Expecting: [1,2,3,4]
Received: [1,2,3,4]
Trials: 4  Errors: 1  Breaks: 1  Pass: 2 Fail: 1
################################################
```

This will be worked on, on the side, to see if it can't be made xUnit and/or TAP compatible, with possible XML reporting features to allow for future Unicon programs to take part in Jenkins style auto build setups.

Have good, test well.

34.8 Invited

Just started in on conversations regarding taking on a co-admin role with the Unicon project. Honoured, and looking forward. If it works out, *and there is no reason to think it won't*, the Unicon Programming docset will be moved to SourceForge and become part of the Unicon project proper.

I have plans to leverage some of the offerings provided by the great people at SourceForge. And to continue to advocate the use of Unicon for generating solutions to modern computing needs.

34.8.1 Recent news

The Unicon Programming doc set is still progressing along nicely. The alpha status warning has been dropped and the version bumped to 0.2. The todo list is shrinking, *and growing*. Shrinking in terms of getting all the reference material in place, growing in terms of all the nifty application potentials for Unicon programming that await discovery and implementation.

List support in repl

Clinton Jeffery has recently updated the *repl* function to support *List (arrays)* data.

```
link lists
procedure main()
    L := [1,2,3]
    write(limage(L))
    write(limage(repl(L, 3)))
end
```

```
```

That code will replicate the list `L` three times, giving a result of

```
[1,2,3,1,2,3,1,2,3]
```

A handy feature.

More loadfunc

Created a sample that embeds `ficl`, the Forth Inspired Command Language, a shared library that exposes a very nice Forth engine. `unificl` embeds the interpreter for use from Unicon.

See *ficl* for all the details.

Linux Containers

Jafar Al-Gharaibeh has created a layer that allows Unicon to interface with LXC, the Linux Container system. This allows sandbox operations and container management in a few lines of Unicon source.

https://sourceforge.net/p/unicon/discussion/contributions/thread/ec34b78c/

Recommended read.

Vim editor syntax file

A Unicon specific highlighter has been created for use inside the Vim editor. Builds on the existing Icon syntax file that ships with Vim, adding in Unicon specific (and Icon graphic) reserved words, keywords and function lists.

https://sourceforge.net/p/unicon/discussion/contributions/thread/27a00aa9/

```
" Vim syntax file
" Language: Unicon
" Maintainer:   Brian Tiffin (btiffin@gnu.org)
" URL: https://sourceforge.net/projects/unicon
" Last Change:  2016 Oct 22

" quit when a syntax file was already loaded
if exists("b:current_syntax")
  finish
endif

" Read the Icon syntax to start with
runtime! syntax/icon.vim
unlet b:current_syntax

" Unicon function extensions
syn keyword uniconFunction Abort Any Arb Arbno array
syn keyword uniconFunction Break Breakx chmod chown
syn keyword uniconFunction chroot classname cofail Color
syn keyword uniconFunction condvar constructor
syn keyword uniconFunction crypt ctime dbcolumns dbdriver
syn keyword uniconFunction dbkeys dblimits dbproduct dbtables display
syn keyword uniconFunction eventmask EvGet EvSend
syn keyword uniconFunction exec Fail fdup Fence fetch fieldnames
syn keyword uniconFunction filepair
syn keyword uniconFunction flock fork
```

```
syn keyword uniconFunction getegid geteuid getgid getgr
syn keyword uniconFunction gethost getpgrp getpid getppid getpw
syn keyword uniconFunction getrusage getserv gettimeofday
syn keyword uniconFunction getuid globalnames gtime
syn keyword uniconFunction ioctl istate
syn keyword uniconFunction keyword kill Len link load localnames lock
syn keyword uniconFunction max membernames methodnames
syn keyword uniconFunction methods min mkdir mutex name
syn keyword uniconFunction NotAny Nspan opencl oprec
syn keyword uniconFunction paranames parent pipe
syn keyword uniconFunction Pos proc
syn keyword uniconFunction readlink ready
syn keyword uniconFunction receive Rem rmdir Rpos Rtab
syn keyword uniconFunction select send setenv setgid setgrent
syn keyword uniconFunction sethostent setpgrp setpwent setservent setuid
syn keyword uniconFunction signal Span spawn sql stat staticnames
syn keyword uniconFunction structure Succeed symlink
syn keyword uniconFunction sys_errstr syswrite Tab
syn keyword uniconFunction trap truncate trylock
syn keyword uniconFunction umask unlock utime wait

" Unicon graphics, audio and VOIP
syn keyword uniconGraphics Active Alert
syn keyword uniconGraphics Attrib Bg
syn keyword uniconGraphics Clip Clone Color
syn keyword uniconGraphics ColorValue CopyArea
syn keyword uniconGraphics Couple
syn keyword uniconGraphics DrawArc DrawCircle DrawCube DrawCurve
syn keyword uniconGraphics DrawCylinder DrawDisk DrawImage DrawLine
syn keyword uniconGraphics DrawPoint DrawPolygon DrawRectangle
syn keyword uniconGraphics DrawSegment DrawSphere DrawString DrawTorus
syn keyword uniconGraphics EraseArea Event
syn keyword uniconGraphics Eye Fg
syn keyword uniconGraphics FillArc FillCircle FillPolygon
syn keyword uniconGraphics FillRectangle Font FreeColor
syn keyword uniconGraphics GotoRC GotoXY
syn keyword uniconGraphics IdentityMatrix
syn keyword uniconGraphics Lower MatrixMode
syn keyword uniconGraphics MultMatrix
syn keyword uniconGraphics NewColor Normals
syn keyword uniconGraphics PaletteChars PaletteColor PaletteKey
syn keyword uniconGraphics Pattern Pending
syn keyword uniconGraphics Pixel PlayAudio PopMatrix
syn keyword uniconGraphics PushMatrix PushRotate PushScale PushTranslate
syn keyword uniconGraphics QueryPointer Raise ReadImage
syn keyword uniconGraphics Refresh Rotate
syn keyword uniconGraphics Scale
syn keyword uniconGraphics StopAudio
syn keyword uniconGraphics Texcoord Texture
syn keyword uniconGraphics TextWidth Translate
syn keyword uniconGraphics Uncouple
syn keyword uniconGraphics VAttrib
syn keyword uniconGraphics WAttrib WDefault WFlush
syn keyword uniconGraphics WindowContents
syn keyword uniconGraphics WriteImage WSection WSync

" Unicon system specific
syn keyword uniconSpecific FreeSpace GetSpace InPort Int86
```

```
syn keyword uniconSpecific OutPort Peek Poke Swi
syn keyword uniconSpecific WinAssociate WinButton WinColorDialog
syn keyword uniconSpecific WinEditRegion WinFontDialog WinMenuBar
syn keyword uniconSpecific WinOpenDialog WinPlayMedia WinSaveDialog
syn keyword uniconSpecific WinScrollBar WinSelectDialog

" Unicon and Icon Graphic Keywords
syn match uniconKeyword "&col"
syn match uniconKeyword "&column"
syn match uniconKeyword "&control"
syn match uniconKeyword "&errno"
syn match uniconKeyword "&eventcode"
syn match uniconKeyword "&eventsource"
syn match uniconKeyword "&eventvalue"
syn match uniconKeyword "&interval"
syn match uniconKeyword "&ldrag"
syn match uniconKeyword "&lpress"
syn match uniconKeyword "&lrelease"
syn match uniconKeyword "&mdrag"
syn match uniconKeyword "&meta"
syn match uniconKeyword "&mpress"
syn match uniconKeyword "&mrelease"
syn match uniconKeyword "&pick"
syn match uniconKeyword "&now"
syn match uniconKeyword "&rdrag"
syn match uniconKeyword "&resize"
syn match uniconKeyword "&row"
syn match uniconKeyword "&rpress"
syn match uniconKeyword "&rrelease"
syn match uniconKeyword "&shift"
syn match uniconKeyword "&window"
syn match uniconKeyword "&x"
syn match uniconKeyword "&y"

" New reserved words
syn keyword uniconReserved critical import initially invocable method
syn keyword uniconReserved package thread

" Storage class reserved words
syn keyword uniconStorageClass  abstract class

" Define the highlighting colour groups
hi def link uniconStorageClass  StorageClass
hi def link uniconFunction      Statement
hi def link uniconGraphics      Special
hi def link uniconSpecific      SpecialComment
hi def link uniconReserved      Label
hi def link uniconKeyword       Operator

let b:current_syntax = "unicon"
```

Graphics fixes

Reported some issues with 3D graphic support on GNU/Linux systems, and within a few days, Clinton and Jafar had fixes posted. More attributes work properly with the *open* function, and *WriteImage* can now be used to capture the result of a 3D graphic canvas, saved in any of the many supported image formats.

The Unicon project continues to improve. Nice.

Have good, make well.

34.9 SourceForge

Well met,

Just spent a few minutes, and it was just a few minutes, building Unicon on SourceForge using the freely offered developer web services. Revision 4616, pulled fresh from svn, configured and built inside a SourceForge developer shell.

Worked out great. First sample of Unicon CGI is up at

http://btiffin.users.sourceforge.net/form.html

```
<HTML><HEAD><title> An HTML Form Example </title></HEAD>
<!--
 From Programming with Unicon
 Copyright (C) 1999-2015 Clinton Jeffery, Shamim Mohamed,
     Jafar Al Gharaibeh, Ray Pereda, and Robert Parlett
-->
<BODY>
<h1> A <tt>cgi.icn</tt> Demonstration</h1>
<form method="GET" action="/cgi-bin/simple.cgi">
    1. Name: <input type="text" name="name" size=25> <p>
    2. Age: <input type="text" name="age" size=3>  Years <p>
    3. Quest:
        <input type="checkbox" name="fame">Fame</input>
        <input type="checkbox" name="fortune">Fortune</input>
        <input type="checkbox" name="grail">Grail</input><p>
    4. Favorite Color:
        <select name="color">
            <option>Red
            <option>Green
            <option>Blue
            <option selected>Don't Know (Aaagh!)
        </select><p>
    Comments:<br>
        <textarea rows=5 cols=60 name="comments"></textarea><p>
    <input type="submit" value="Submit Data">
    <input type="reset" value="Reset Form">
</form>
</BODY>
</HTML>
```

That form has a Submit action that invokes a small server side Unicon CGI program. The code was taken from the Programming with Unicon book and modified slightly to make it safer for hosting on a public facing web site.

```
#
# simple-cgi.icn
# tectonics:
#    unicon -B simple-cgi.icn
#    mv simple ../cgi-bin/simple.cgi
#
link cgi
procedure cgimain()
    # set defaults for both CGI and AJAX usage
```

```
    if /cgi["name"] | cgi["name"] === "" then cgi["name"] := "Guest"
    if /cgi["age"] | cgi["age"] === "" then cgi["age"] := "no"
    if /cgi["comments"] then cgi["comments"] := ""
    if /cgi["word"] then cgi["word"] := ""

    # remove any potentially dangerous characters
    cgi["name"] := map(cgi["name"], "<>&%", "....")
    cgi["age"] := map(cgi["age"], "<>&%", "....")
    cgi["comments"] := map(cgi["comments"], "<>&%", "....")
    cgi["word"] := map(cgi["word"], "<>&%", "....")

    # output for the web
    cgiEcho("Hello, ", cgi["name"], "!")
    cgiEcho("Are you really ", cgi["age"], " years old?")
    cgiEcho("You seek: ", cgi["fame"]==="on" & "fame")
    cgiEcho("You seek: ", cgi["fortune"]==="on" & "fortune")
    cgiEcho("You seek: ", cgi["grail"]==="on" & "grail")
    cgiEcho("Your favorite color is: ", cgi["color"])
    cgiEcho("Your comments: ", cgi["comments"])
    cgiEcho("")
    cgiEcho("Your AJAX word: ", cgi["word"])
    cgiEcho("")
    cgiEcho("<a href=\"/demos/\">Home</a> / " ||
        "<a href=\"/demos/simple-form.html\">Back to HTML form</a> / " ||
        "<a href=\"/demos/simple-ajax.html\">Back to AJAX form</a>")
end
```

That trial code simply echos the form data, after ruthlessly sanitizing any data to avoid any potential cross site scripting efforts. The code was compiled, on SourceForge with `unicon -B simple.icn` and then the resulting executable was moved into the `cgi-bin` directory as `simple.cgi`

All tests so far have come up golden.

As a very satisfied Unicon customer, the good folk that provide and maintain SourceForge services deserve a round of applause.

34.9.1 Recent news

Timers

Jafar Al-Gharaibeh posted up a little `loadfunc` sample for accessing interval timers with `setitimer`.

https://sourceforge.net/p/unicon/discussion/contributions/thread/db34541b/

Resizing windows

A bug with window resizing is being discussed on the mailing list. If things progress as they normally do, this will be fixed shortly. *The code under discussion works fine here, an X11 build on Xubuntu, and this seems to be an issue only with certain configurations of Unicon.*

Procedural or Object oriented

Unicon, being a multi-paradigm programming environment, offers a lot of flexibility when it comes to making design and implementation choices. I asked for some opinions on the Discussion forum on whether Procedural/Imperative or Object oriented development is preferred by the language designers for small Unicon programs.

Both *Clinton Jeffery* and *Jafar Al-Gharaibeh* opined that *it depends on the developer and the problem being faced, there is no preferred style*. This is good news in terms of letting programmers attack problems from the most comfortable position, and is a sign that no single paradigm is given more weight during language development. Unicon programmers are free to make choices without worrying about decisions being *wrong* in any way. Both paradigms are a good choice.

By the nature of Unicon, the styles can easily be mixed, so that becomes yet another valid choice available for developers.

https://sourceforge.net/p/unicon/discussion/general/thread/b68a2d34/

Markdown

The UP docs now include a seed work example of calling `libsoldout` by way of *loadfunc*. The soldout engine ships with example (production ready) output renderers. The `soldout.icn` sample parses extended Markdown from a Unicon string, and produces HTML, returned as a string from the loaded wrapper function. See *libsoldout markdown* for code listings and some explanations.

More information about `libsoldout`, by Natacha Porté, can be found at

http://fossil.instinctive.eu/libsoldout/home

Have good, make well.

34.10 Unicon FFI

Well met,

Let 2017 be the year of the `uniffi`. Unicon Foreign Function Interface.

Ok, Unicon already has a Foreign Function Interface, *loadfunc* and similar C function interfacing has been in Unicon since its inception, dating back to at least Icon version 8.10, March of 1993. There were two C interfaces documented for that release, outbound, *callout* and inbound `icon_call`.

Sadly, the inbound code in Unicon for `icon_call` is no longer available, *but read on for a possible future, perhaps better alternative*.

The outbound interface *callout* is still in Unicon version 13, but requires a special build of the entire compiler/runtime system to replace an internal stub function called `extcall`, in `src/runtime/extcall.r` which by default just returns and error code 216. Anyone is free to dig into this interface, actually fairly well documented by *Ralph Griswold* in IPD217, http://www2.cs.arizona.edu/icon/ftp/doc/ipd217.pdf

It's old, and usable, but all the recent activity has been focused on *loadfunc*. A small layer of code was added in version 9, (the base Icon used for Unicon core, much has changed in Unicon since then) to load C function entry points at runtime, from dynamic shared object libraries. And *loadfunc* was born. Foreign functions could/can be loaded into Unicon at runtime without need of special builds that `extcall` and *callout* require.

There are a lot of *loadfunc* examples peppered throughout the Unicon Programming document set. It opens up doors to C libraries, which are numerous and ubiquitous.

One issue with *loadfunc* is that the functions called have to comply with a Unicon calling convention. Routines are passed an `argc argv` style Unicon frame, using a count of passed in descriptors. These descriptors need to be manually converted to C native data, passed on to other C routines, and then converted back to Unicon data types for returning results. There are copious examples of managing this protocol, and support macros in `ipl/cfuncs/icall.h` that make this all pretty easy. But, it is still an extra layer of burden placed on a Unicon programmer aiming to use an existing C library solution to a problem, or for a speed boost.

And now a step up.

34.10.1 libffi

`libffi` is a foreign function interface library, that manages the call frame setup for all kinds of different calling conventions. 32bit, 64bit and many different operating systems are all supported. This layer was put to use to alleviate the need to use *loadfunc* for many/most/all C functions that a Unicon programmer may want to call. Once loaded (the experimental `native(...)` function is not built into Unicon, so it uses *loadfunc* to bootstrap), all a Unicon programmer needs to do is call `native`:

```
dlHandle := addLibrary("libraryName")

result := native("function", returnType, arguments,...)
more := native("otherFunction", returnType, argumens,...)
...
```

And that's it. Under the covers the `native` function finds an entry point (usually after a supporting call to `addLibrary` which is the name of a Dynamic Shared Object module archive (a DLL)), marshals the *Unicon* arguments by for use by *C*, and dispatches a call/return sequence. Results from *C* are converted to the specified Unicon `returnType` and passed back to *Unicon*. Almost of this become invisible to the Unicon programmer. All you need to do is call `native` with a function link name and arguments. Almost all *C* native data types are supported.

And that is a wrinkle. *C* call frames need to know the exact type of each argument, and what type to return (including nothing, termed `void`). For many types, `native` can just convert to reasonable *C* types. Integer to `int`, Real to `double`, String to `char *` etc, using the handy macros built into `icall.h`. Sometimes this is wrong. *C* (currently) has two types of floating point values, 32 bit `float`, and 64 bit `double`. There are also distinctions for 8bit, 16bit, 32bit, 64bit integers, in both signed and unsigned forms. *Unicon* just has Integer and Real.

`native` allows for type overrides in the function call, using two element lists.

```
result := native("function", TYPEFLOAT, [x, TYPEFLOAT], [y, TYPEFLOAT])
```

The real values from *Unicon* are demoted to *C* `float` data, and the returning type is promoted from `float` to an acceptable *Unicon Real numbers* form.

These type specifications can be freely mixed

```
result := native("mixed", TYPEINT, [x, TYPEFLOAT], [y, TYPEDOUBLE])
```

That assumes that mixed has a *C* prototype of `int mixed(float x, double y)` and makes the proper arrangements for the function call, returning an Integer result back to Unicon.

Note: Please note that this experiment is at a very early stage, and some of the type constant names, and argument lists may change before this ever gets accepted into Unicon proper; if it ever gets accepted.

---

libharu

This entire exercise started with a desire to integrate PDF generation in Unicon by leveraging `libharu`, the PDF writer library. There are many tens of functions in `libharu` and each one would have required a small `loadfunc` call convention wrapper, written in *C* to accommodate. That led to an initial version of `native()` that took on the task of preparing a *C* call frame using inline assembler, which works, but is limited to x86_64 System V call conventions. See *C Native* for that blurb.

After finishing a trial of *C Native*, `libffi` was discovered. It does the same job and far more than *C Native*; there is a single interface, no burden to write umpteen dozen small pieces of assembler to support the various platforms that Unicon is currently built to run on, and is well supported by a team of experts in the area of foreign function calls.

Here is what the `libharu` integration example looks like:

```
#
# haru.icn, demonstrate a newer C FFI
#
$include "natives.inc"

$define HPDF_COMP_ALL 15
$define HPDF_PAGE_MODE_USE_OUTLINE 1
$define HPDF_PAGE_SIZE_LETTER 0
$define HPDF_PAGE_PORTRAIT 0

procedure main()
    local dlHandle, pdf, page1, rc, savefile := "harutest.pdf"

    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uniffi.so", "addLibrary")

    # allow arbitrary C functions, marshalled by a piece of assembler
    # assume float instead of double, changes the inline assembler
    # movsd versus movdd
    native := loadfunc("./uniffi.so", "ffi")

    # add libhpdf to the dlsym search path, the handle is irrelevant
    dlHandle := addLibrary("libhpdf.so")

    pdf := native("HPDF_New", TYPESTAR, 0, 0)

    rc := native("HPDF_SetCompressionMode", TYPEINT, pdf, HPDF_COMP_ALL)
    rc := native("HPDF_SetPageMode", TYPEINT, pdf,
                 HPDF_PAGE_MODE_USE_OUTLINE)

$ifdef PROTECTED
    rc := native("HPDF_SetPassword", TYPEINT, pdf, "owner", "user")
    savefile := "harutest-pass.pdf"
$endif

    page1 := native("HPDF_AddPage", TYPESTAR, pdf)

    rc := native("HPDF_Page_SetHeight", TYPEINT, page1,
                 [220.0, TYPEFLOAT]);
    rc := native("HPDF_Page_SetWidth", TYPEINT, page1,
                 [200.0, TYPEFLOAT]);

    #/* A part of libharu pie chart sample, Red*/
    rc := native("HPDF_Page_SetRGBFill", TYPEINT, page1,
                 [1.0, TYPEFLOAT], [0.0, TYPEFLOAT], [0.0, TYPEFLOAT]);
    rc := native("HPDF_Page_MoveTo", TYPEINT, page1,
                 [100.0, TYPEFLOAT], [100.0, TYPEFLOAT]);
    rc := native("HPDF_Page_LineTo", TYPEINT, page1,
                 [100.0,  TYPEFLOAT],[180.0, TYPEFLOAT]);
    rc := native("HPDF_Page_Arc", TYPEINT, page1,
                 [100.0, TYPEFLOAT], [100.0, TYPEFLOAT],
                 [80.0, TYPEFLOAT], [0.0, TYPEFLOAT],
                 [360 * 0.45, TYPEFLOAT]);

    #pos := native("HPDF_Page_GetCurrentPos (page);

    rc := native("HPDF_Page_LineTo", TYPEINT, page1,
```

```
                  [100.0, TYPEFLOAT], [100.0, TYPEFLOAT]);
    rc := native("HPDF_Page_Fill", TYPEINT, page1);

    rc := native("HPDF_SaveToFile", TYPEINT, pdf, savefile);
    native("HPDF_Free", TYPEVOID, pdf);
end
```

Fairly short, and sweet.

This sample barely scratches the surface of `libharu` features (simply drawing a partial arc, filled in red). What it highlights is that the calls occurred with no extra *C* source required.

This is where the excitement might start to build. *Unicon* programmers can focus on *Unicon*, leaving *C* to the *C* folk.

Here is a small GnuCOBOL program that was used during testing

```
      *>
      *> Demonstrate Unicon native call of COBOL modules
      *>
       identification division.
       program-id. cobolnative.

       data division.
       working-storage section.
       linkage section.
       01 one usage binary-long.
       01 two usage binary-long.

       procedure division using by value one two.
       display "GnuCOBOL got " one ", " two
       compute return-code = one + two
       goback.
       end program cobolnative.
```

The Unicon caller:

```
#
# cobffi.icn, test calling COBOL without wrapper with libffi
#
$include "natives.inc"

procedure main()
    # will be RTLD_LAZY | RTLD_GLOBAL (so add to the search path)
    addLibrary := loadfunc("./uniffi.so", "addLibrary")

    # allow arbitrary C functions, marshalled by libffi
    native := loadfunc("./uniffi.so", "ffi")

    # add the testing functions to the dlsym search path,
    #  the handle is somewhat irrelevant, but won't be soonish
    dlHandle := addLibrary("./cobolnative.so")

    # initialize GnuCOBOL
    native("cob_init", TYPEVOID)

    # pass two integers, get back a sum
    ans := native("cobolnative", TYPEINT, 40, 2)
    write("Unicon: called sample and got ", ans)

```

```
    # rundown the libcob runtime
    native("cob_tidy", TYPEVOID)
end
```

And a sample run:

```
prompt$ cobc -m -Wno-unfinished cobolnative.cob
```

```
prompt$ unicon -s cobffi.icn -x
GnuCOBOL got +0000000040, +0000000002
Unicon: called sample and got 42
```

`libffi` makes calling GnuCOBOL modules from Unicon, a complete breeze.

Next steps

I plan on pestering Clinton and Jafar, and who ever else will listen to help polish this up, and hopefully get it added to the Unicon build system proper. It currently lacks some features; not all datatypes are properly supported and there needs to be some deep discussion about how indirect data references (*C* pointers) should be handled (they cannot be allowed to change immutable Unicon data, so an interstitial layer will need to be worked out).

I'd be honoured to continue this with a formal Unicon Technical Report, and will do so if that's what it takes to advance this flag.

On the other side of the coin...

34.10.2 C calling Unicon

The `unicon -C` native compile sequence is pretty handy. It creates a native executable by generating C source code and compiling that intermediate into a native binary. The one point lacking is that it assumes a `main` is generated from the Unicon side, and does all the linking steps assuming that point of view. I'd like to extend `unicon -C` with a new compile time option (something like `--no-main` or `--object` or `-c` meaning compile/don't link (but the `-c` idea was deemed to conflict with the current meaning of *generate ucode*), to produce object code, ready for linking to other programs.

Initial trials for this have been proven (in a hack sort of way) by changing the generated *C* code output by `unicon -C` to change the name of *main* to *somecode* and then removing the link phase from invocation of `gcc` that is used, to simply generate an object file with `gcc -c`. That code was then linked to a GnuCOBOL test program, and *Unicon* was called, data passed in, results returned.

The hack even went as far as returning a pointer to the *Unicon* *global* variable structure that is part of native executables, but that part would not be part of any production level release. First a shared memory space sequence would be worked out, instead of pointers into *Unicon* space (which can be garbage collected and moved at any time, outside normal control of a developer).

Unicon object files (meaning `.o` files, not class objects) will alleviate some of the need to resurrect `call_icon` to allow *C* programs to call *Unicon* programs. *Unicon* will then be able to take part in all forms of mixed language programming. Shareable libraries could be created that will allow foreign languages to enjoy direct benefit from *Unicon* language features without knowing anything about *Unicon* source code. *Though one of the goals will be to demonstrate how easy that code is to read and write.*

The first round of experiments relied on statically linking to the *Unicon* runtime system, but another phase may provide for a `libunicon.so` that could be dynamically linked into these callable *Unicon* modules. This would make for very small, easy to manage *Unicon* application level link libraries (or singleton object files).

Continuing this experiment has been given the nod by *Clinton Jeffery*, but there are many details to work out, and it won't be part of *Unicon* until the entire sequence is ready at a level of quality expected by *Unicon* developers. There will be copious amounts of documentation available during the design, development and implementation stages.

There are lots of things to discuss, and many possibilities await.

You can follow along in the SourceForge Discussion pages at

https://sourceforge.net/p/unicon/discussion/contributions/

Have good, make well, happiest of 2017s

Unicon Programming documentation set.

- *SNOBOL pattern example*

34.11 Unicon Programming docset

Started the Unicon Programming document. Releases will be made early, and often.

Todo

add more details to package scoping

(The original entry is located in /home/btiffin/wip/writing/unicon/expressions.rst, line 186.)

Todo

co-expression entries

(The original entry is located in /home/btiffin/wip/writing/unicon/expressions.rst, line 403.)

Todo

control structure examples

(The original entry is located in /home/btiffin/wip/writing/unicon/expressions.rst, line 411.)

Todo

add pty sample

(The original entry is located in /home/btiffin/wip/writing/unicon/features.rst, line 86.)

Todo

fix plot range and domain handling

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 147.)

Todo

plot image of atanh

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 594.)

Todo

entry for function callout

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 805.)

Todo

Not working

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 2083.)

Todo

3D points

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 2376.)

Todo

3D polygons

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 2443.)

Todo

entry for function Fence

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 3202.)

Todo

ioctl demo

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 4953.)

Todo

what do the attribute fields actually mean

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 5043.)

Todo

a constant todo item is keeping this list up to date

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 6434.)

Todo

entry for function PushScale

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 7348.)

Todo

entry for function PushTranslate

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 7379.)

Todo

entry for function Rotate

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8011.)

Todo

entry for function Scale

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8204.)

Todo

entry for function sethostent

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8527.)

Todo

entry for function setpgrp

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8558.)

Todo

entry for function setpwent

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8589.)

Todo

entry for function setservent

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8620.)

Todo

entry for function setuid

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8651.)

Todo

entry for function Span

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 8837.)

Todo

entry for function Texcoord

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 9526.)

Todo

entry for function Texture

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 9557.)

Todo

entry for function TextWidth

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 9588.)

Todo

entry for function Translate

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 9619.)

Todo

entry for function Uncouple

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 9840.)

Todo

entry for function VAttrib

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 9995.)

Todo

this graphics section is woefully incomplete

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 10063.)

Todo

entry for function WDefault

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 10157.)

Todo

entry for function WinColorDialog

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 10338.)

Todo

entry for function WinFontDialog

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 10473.)

Todo

entry for function WinMenuBar

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 10504.)

Todo

entry for function WinScrollBar

(The original entry is located in /home/btiffin/wip/writing/unicon/functions.rst, line 10647.)

Todo

samples of Robert Parlett's GUI classes

(The original entry is located in /home/btiffin/wip/writing/unicon/graphics.rst, line 357.)

Todo

fill in more wisdoms

(The original entry is located in /home/btiffin/wip/writing/unicon/ipl.rst, line 477.)

Todo

add some plots

(The original entry is located in /home/btiffin/wip/writing/unicon/monitoring.rst, line 114.)

Todo

add a mail sending sample

(The original entry is located in /home/btiffin/wip/writing/unicon/networking.rst, line 198.)

Todo

add a mail reader sample

(The original entry is located in /home/btiffin/wip/writing/unicon/networking.rst, line 207.)

Todo

add Unicon, PHP integration sample

(The original entry is located in /home/btiffin/wip/writing/unicon/networking.rst, line 321.)

Todo

map out Unicon examples of each SOLID principle

(The original entry is located in /home/btiffin/wip/writing/unicon/objects.rst, line 71.)

Todo

Much to do regarding the Objects chapter.

(The original entry is located in /home/btiffin/wip/writing/unicon/objects.rst, line 87.)

Todo

fill out all the random element types.

(The original entry is located in /home/btiffin/wip/writing/unicon/operators.rst, line 332.)

Todo

more examples and clarification required.

(The original entry is located in /home/btiffin/wip/writing/unicon/operators.rst, line 592.)

Todo

extend this further to handle more datatypes

(The original entry is located in /home/btiffin/wip/writing/unicon/programs.rst, line 274.)

Todo

add string scanning samples

(The original entry is located in /home/btiffin/wip/writing/unicon/strings.rst, line 85.)

Todo

more on tables

(The original entry is located in /home/btiffin/wip/writing/unicon/structures.rst, line 162.)

Todo

monitoring test mode not yet ready for prime time. Nor is xUnit compatibility actually finished for that matter.

(The original entry is located in /home/btiffin/wip/writing/unicon/testing.rst, line 103.)

Todo

only random thoughts so far, goal is codification

(The original entry is located in /home/btiffin/wip/writing/unicon/theory.rst, line 27.)

Todo

complete the list of core tools

(The original entry is located in /home/btiffin/wip/writing/unicon/tools.rst, line 35.)

Todo

ADD SAMPLE

(The original entry is located in /home/btiffin/wip/writing/unicon/tools.rst, line 165.)

INDEX

Symbols

*, 52
+, 53
-, 54
., 55
/, 54
=, 55
?, 56, 58
?:=, 58
??, 58
#line, 346
$define, 339
$else, 340
$endif, 340
$error, 340
$ifdef, 341
$ifndef, 342
$include, 343
$line, 343
$undef, 345
&, 57
&:=, 57
&allocated, 293
&ascii, 294
&clock, 295
&col, 295
&collections, 296
&column, 297
&control, 297
&cset, 298
¤t, 299
&date, 299
&dateline, 300
&digits, 300
&dump, 301
&e, 302
&errno, 302
&error, 303
&errornumber, 304
&errortext, 304
&errorvalue, 305
&errout, 306
&eventcode, 306
&eventsource, 307
&eventvalue, 308
&fail, 309
&features, 309
&file, 311
&host, 311
&input, 312
&lcase, 313
&ldrag, 313
&letters, 314
&level, 314
&line, 315
&lpress, 315
&lrelease, 316
&main, 317
&mdrag, 317
&meta, 318
&mpress, 319
&mrelease, 320
&now, 321
&null, 321
&output, 322
&phi, 322
&pi, 323
&pick, 323
&pos, 324
&progname, 325
&random, 325
&rdrag, 326
®ions, 327
&resize, 328
&row, 328
&rpress, 329
&rrelease, 330
&shift, 331
&source, 331
&storage, 332
&subject, 333
&time, 333
&trace, 334
&ucase, 335

&version, 335
&window, 336
&x, 337
&y, 337
~, 56
\, 54
|, 56, 57
||, 57
||:=, 58
|||, 58
|||:=, 58

A

ABI, 633
Abort, 93
Abort() (built-in function), 93
abs, 94
abs() (built-in function), 94
abstract, **80**
acos, 94
acos() (built-in function), 94
actions, 62
Active, 96
Ada, 445
Al-Gharaibeh
    Jafar, 17
Alert, 97
Alert() (built-in function), 97
ALGOL, 446
all, **89**
allocated.icn, 293
Any, 98
any, 98
API, 633
Arb, 98
Arbno, 99
ArgError (C function), 199
ArgInteger (C function), 199
args, 99
array, 100
ASCII, 633
ascii.icn, 294
asin, 101
Assembler, 446, 533
atan, 103
atanh, 103
Attrib, 104

B

BaCon, 447, 619, 634
Bal, 105
bal, 106
Balbi
    Federico, 16
base, 19
BASIC, 447, 634
benchmark, 443, 465
Bg, 106
Blog, 660
blog
    2016/08/22, 661
    2016/08/25, 661
    2016/08/25b, 661
    2016/08/27, 661
    2016/09/02, 662
    2016/09/18, 662
    2016/10/26, 665
    2016/11/13, 669
    2017/01/02, 671
bound, 43
bounded, 43
Break, 107
break, **62**
Breakx, 108
build from source, 12
by, **90**

C

C, 445, 634
callout, 108
case, **63**
cc-by-nc-2.5, 635
center, 108
cfunction (C function), 198
CGI, 407
CGI spec, 409
ChangeLog, 643
char, 109
chdir, 109
chmod, 110
chown, 111
chroot, 111
class, **81**
classname, 112
Clint, 16
Clip, 113
Clone, 113
close, 114
co-expression, 45
COBOL, 448, 505, 635
coercion
    floating point, 23
    string to numeric, 24
cofail, 115
collect, 115
collections.icn, 296
Color, 116
colors, 387

ColorValue, 117
colour
    mutable, 638
colours, 387
column.icn, 297
combinations, 625
comprehension, 29, 634
compression, 418
condvar, 118
constructor, 119
conventions, 355
copy, 120
CopyArea, 121
core, 468
cos, 121
Couple, 123
Cowlishaw
    Mike, 556
create, **64**
critical, **64**
crypt, 124
Cset, **25**
cset, 124
cset.icn, 298
ctime, 125
current.icn, 299

D

D, 448
database, 395
    dbm, 396
    ODBC, 396
    tables, 395
datatype
    cset, 25
    integer, 19
    real, 22
    string, 26
dbcolumns, 126
dbdriver, 128
dbkeys, 129
dblimits, 130
dbm, 396
dbproduct, 131
dbtables, 131
debugger, 359
debugging, 435
declaratives, 79
default, **90**
delay, 132
delete, 133
descriptor (C type), 198
detab, 133
digits.icn, 301
display, 134
do, **90**
documentation, 421
downsides, 464
DrawArc, 135
DrawCircle, 135
DrawCube, 136
DrawCurve, 137
DrawCylinder, 138
DrawDisk, 139
DrawImage, 140
drawing, 389
DrawLine, 141
DrawPoint, 142
DrawPolygon, 144
DrawRectangle, 145
DrawSegment, 145
DrawSphere, 146
DrawString, 147
DrawTorus, 148
DSO, 635
dtor, 149
Duktape, 449
duktape, 511
dump.icn, 301

E

e.icn, 302
ECMAScript, 449
editors, 371
Elixir, 449
else, **90**
EM, 437
emacs, 376
    vim emulation, 376
emphatics, 62
end, **90**
entab, 149
environment variable
    TRACE, 435
epoch, 643
EraseArea, 150
error.icn, 303
errorclear, 150
errornumber.icn, 304
eval, 583
Event, 151
eventmask, 152
every, **65**
EvGet, 152
EvSend, 153
exec, 154
Execution Monitoring, 437
exit, 155

exp, 155
Expect, 636
expression
    bound, 43
expressions, 37
Eye, 156

F

Fail, 157
fail, **69**
failure, 37
Farber
    David, 17
Farberisms, 636
fcntl, 157
fdup, 159
features, 417
features.icn, 310
Fence, 159
fetch, 159
Fg, 160
Ficl, 450
ficl, 519
fieldnames, 161
File, 27
filepair, 162
FillArc, 163
FillCircle, 163
FillPolygon, 164
FillRectangle, 165
find, 165
fixed-point, 627
fizzbuzz, 582
floating-point, 627
flock, 166
flush, 167
Font, 167
fonts
    default, 167
fork, 168
Forth, 450, 519, 636
Fortran, 450
fortran, 530
FreeColor, 169
FreeSpace, 170
function, 170
    Abort, 93
    abs, 94
    acos, 94
    Active, 96
    Alert, 97
    Any, 98
    any, 98
    Arb, 98
    Arbno, 99
    args, 99
    array, 100
    asin, 101
    atan, 103
    atanh, 103
    Attrib, 104
    Bal, 105
    bal, 106
    Bg, 106
    Break, 107
    Breakx, 108
    callout, 108
    center, 108
    char, 109
    chdir, 109
    chmod, 110
    chown, 111
    chroot, 111
    classname, 112
    Clip, 113
    Clone, 113
    close, 114
    cofail, 115
    collect, 115
    Color, 116
    ColorValue, 117
    condvar, 118
    constructor, 119
    copy, 120
    CopyArea, 121
    cos, 121
    Couple, 123
    crypt, 124
    cset, 124
    ctime, 125
    dbcolumns, 126
    dbdriver, 128
    dbkeys, 129
    dblimits, 130
    dbproduct, 131
    dbtables, 131
    delay, 132
    delete, 133
    detab, 133
    display, 134
    DrawArc, 135
    DrawCircle, 135
    DrawCube, 136
    DrawCurve, 137
    DrawCylinder, 138
    DrawDisk, 139
    DrawImage, 140
    DrawLine, 141

DrawPoint, 142
DrawPolygon, 144
DrawRectangle, 145
DrawSegment, 145
DrawSphere, 146
DrawString, 147
DrawTorus, 148
dtor, 149
entab, 149
EraseArea, 150
errorclear, 150
Event, 151
eventmask, 152
EvGet, 152
EvSend, 153
exec, 154
exit, 155
exp, 155
Eye, 156
Fail, 157
fcntl, 157
fdup, 159
Fence, 159
fetch, 159
Fg, 160
fieldnames, 161
filepair, 162
FillArc, 163
FillCircle, 163
FillPolygon, 164
FillRectangle, 165
find, 165
flock, 166
flush, 167
Font, 167
fork, 168
FreeColor, 169
FreeSpace, 170
function, 170
get, 171
getch, 172
getche, 172
getegid, 173
getenv, 173
geteuid, 174
getgid, 174
getgr, 175
gethost, 175
getpgrp, 176
getpid, 176
getppid, 177
getpw, 177
getrusage, 178
getserv, 179
GetSpace, 180
gettimeofday, 181
getuid, 181
globalnames, 182
GotoRC, 182
GotoXY, 183
gtime, 184
hardlink, 184
iand, 185
icom, 185
IdentityMatrix, 186
image, 187
InPort, 187
insert, 188
Int86, 188
integer, 189
ioctl, 189
ior, 190
ishift, 190
istate, 191
ixor, 192
kbhit, 192
key, 193
keyword, 194
kill, 194
left, 195
Len, 196
list, 196
load, 197
loadfunc, 198
localnames, 200
lock, 200
log, 201
Lower, 201
lstat, 202
many, 203
map, 203
match, 204
MatrixMode, 205
max, 206
member, 207
membernames, 207
methodnames, 208
methods, 208
min, 209
mkdir, 210
move, 211
MultMatrix, 211
mutex, 212
name, 213
NewColor, 214
Normals, 214
NotAny, 215
Nspan, 215

numeric, 216
open, 217
opencl, 218
oprec, 219
ord, 220
OutPort, 220
PaletteChars, 220
PaletteColor, 221
PaletteKey, 222
paramnames, 222
parent, 222
Pattern, 223
Peek, 224
Pending, 224
pipe, 225
Pixel, 226
PlayAudio, 227
Poke, 227
pop, 228
PopMatrix, 228
Pos, 229
pos, 230
proc, 230
pull, 231
push, 231
PushMatrix, 232
PushRotate, 233
PushScale, 233
PushTranslate, 234
put, 234
QueryPointer, 234
Raise, 235
read, 236
ReadImage, 236
readlink, 238
reads, 238
ready, 239
real, 239
receive, 240
Refresh, 240
Rem, 241
remove, 242
rename, 242
repl, 243
reverse, 243
right, 244
rmdir, 244
Rotate, 245
Rpos, 245
Rtab, 245
rtod, 246
runerr, 247
save, 247
Scale, 248
seek, 248
select, 249
send, 249
seq, 250
serial, 250
set, 250
setenv, 251
setgid, 251
setgrent, 252
sethostent, 253
setpgrp, 253
setpwent, 253
setservent, 253
setuid, 254
signal, 254
sin, 254
sort, 256
sortf, 257
Span, 257
spawn, 257
sql, 259
sqrt, 260
stat, 261
staticnames, 262
stop, 262
StopAudio, 263
string, 263
structure, 264
Succeed, 264
Swi, 265
symlink, 265
sys_errstr, 266
system, 266
syswrite, 266
Tab, 267
tab, 267
table, 268
tan, 268
Texcoord, 270
Texture, 270
TextWidth, 270
Translate, 271
trap, 271
trim, 272
truncate, 272
trylock, 272
type, 273
umask, 274
Uncouple, 275
unlock, 275
upto, 275
utime, 276
variable, 276
VAttrib, 277

wait, 277
WAttrib, 278
WDefault, 280
WFlush, 281
where, 281
WinAssociate, 282
WinButton, 282
WinColorDialog, 283
WindowContents, 283
WinEditRegion, 285
WinFontDialog, 285
WinMenuBar, 286
WinOpenDialog, 286
WinPlayMedia, 286
WinSaveDialog, 287
WinScrollBar, 287
WinSelectDialog, 288
write, 288
WriteImage, 289
writes, 289
WSection, 290
WSync, 291
functions, 9

G

GCC, 636
Genie, 459
get, 171
getch, 172
getche, 172
getegid, 173
getenv, 173
geteuid, 174
getgid, 174
getgr, 175
gethost, 175
getpgrp, 176
getpid, 176
getppid, 177
getpw, 177
getrusage, 178
getserv, 179
GetSpace, 180
gettext, 561
gettimeofday, 181
getuid, 181
gforth, 450
global, **82**
globalnames, 182
GNU, 636
GnuCOBOL, 606, 637
GotoRC, 182
GotoXY, 183
graphics, 387
Graphics Programming in Icon, 637
Griswold
Dr. Ralph, 17
Groovy, 451
gst, 458
gtime, 184
gui, 387, 392
Guile, 456

H

hardlink, 184
Help Wanted, 637
history, 14
hpdf, 605
http, 401
https, 403

I

i18n, 561
iand, 185
icom, 185
Icon, 637
version 9, 637
icont, **352**
iconx, **353**
IdentityMatrix, 186
IdentityMatrix() (built-in function), 186
idioms, 58
IDOL, 386
ie, 566
if, **70**
image, 187
import, **82**
indexing, 26
initial, **71**
initially, **71**
InPort, 187
insert, 188
install, 12
Int86, 188
Integer, **19**
integer, 189
large, 22
Internationalization, 561
invocable, **83**
ioctl, 189
ior, 190
IPL, 467
core, 468
lists, 472
numbers, 474
options, 469
strings, 471
wrap, 470

ximage, 470
ipp, 339
ishift, 190
istate, 191
IVIB, 359
ixor, 192

J

Java, 451
Javascript, 449
Jeffery
Dr. Clinton, 3, 16
jimsh, 458
JSON, 493, 638

K

kbhit, 192
key, 193
keyboard, 417
keyword, 194
&ascii, 294
&clock, 295
&col, 295
&collections, 296
&column, 297
&control, 297
&cset, 298
¤t, 299
&date, 299
&dateline, 300
&digits, 300
&dump, 301
&e, 302
&errno, 302
&error, 303
&errornumber, 304
&errortext, 304
&errorvalue, 305
&errout, 306
&eventcode, 306
&eventsource, 307
&eventvalue, 308
&fail, 309
&features, 309
&file, 311
&host, 311
&input, 312
&lcase, 313
&ldrag, 313
&letters, 314
&level, 314
&line, 315
&lpress, 315
&lrelease, 316
&main, 317
&mdrag, 317
&meta, 318
&mpress, 319
&mrelease, 320
&null, 321
&output, 322
&phi, 322
&pi, 323
&pick, 323
&pos, 324
&progname, 325
&random, 325
&rdrag, 326
®ions, 327
&resize, 328
&row, 328
&rpress, 329
&rrelease, 330
&shift, 331
&source, 331
&storage, 332
&subject, 333
&trace, 334
&ucase, 335
&version, 335
&window, 336
&x, 337
&y, 337
kill, 194

L

L10n, 561
left, 195
Len, 196
libcox, 538
libffi, 608
libharu, 605
libsoldout, 564
libz, 418
license, 649
line.icn, 315
link, **84**
linker, 468
List, 29
list, 196
lists, 472
load, 197
loadfunc, 198, 460, 594
loadfunc() (built-in function), 198
local, **85**
locale, 638
Localization, 561
localnames, 200

lock, 200
log, 201
Lower, 201
lstat, 202
Lua, 452, 525

M

Mahmoud, 17
main, 87
make, 357
many, 203
map, 203
markdown, 564
match, 204
MatrixMode, 205
MatrixMode() (built-in function), 205
max, 206
member, 207
membernames, 207
method, **85**
methodnames, 208
methods, 208
min, 209
mkdir, 210
mode
    m, 401
    n, 401
    na, 401
    nl, 401
    nu, 401
Mohamed
    Shamim, 16
monitoring, 437
move, 211
mruby, 517
multi-tasking, 415
multilanguage, 593
MultMatrix, 211
MultMatrix() (built-in function), 211
mutable colour, 638
mutex, 212

N

name, 213
native, 594
Neko, 452
networking, 401
NewColor, 214
next, **72**
Nickle, 453
Nim, 453
nodejs, 449
Normals, 214
Normals() (built-in function), 215
not, **73**
NotAny, 215
Nspan, 215
null, 38
numbers, 474
numeric, 216

O

objects, 385
ODBC, 396
of, **90**
ooRexx, 455, 556
open, 217
opencl, 218
operator functions, 59
operators, 45
oprec, 219
options, 469
ord, 220
OutPort, 220
overview, 6

P

package, **86**
PaletteChars, 220
PaletteChars() (built-in function), 221
PaletteColor, 221
PaletteColor() (built-in function), 221
PaletteKey, 222
PaletteKey() (built-in function), 222
paramnames, 222
parent, 222
parent() (built-in function), 223
Parlett
    Robert, 17
Pattern, 223
pattern
    scanning, 381
patterns, 381
    internals, 382
    operators, 383
    regex, 383
    SNOBOL, 381
    syntax, 383
PDF, 605
Peek, 224
Pending, 224
Pending() (built-in function), 224
people, 16
performance, 443
Perl, 454
permutations, 625
PH7, 541
PHP, 407, 411, 454, 541, 638

pipe, 225
pipe() (built-in function), 225
Pixel, 226
PlayAudio, 227
Poke, 227
Polonsky
    Ivan, 17
pop, 228
PopMatrix, 228
PopMatrix() (built-in function), 228
Pos, 229
pos, 230
POSIX, 639
precedence, 39
predefined symbols, 346
preprocessor, 339
preprocessor substitutions, 347
proc, 230
procedure, **87**
progname.icn, 325
programming language
    unicon, 3
programs, 499
pseudo terminals, 417
pty, 417
pull, 231
push, 231
PushMatrix, 232
PushMatrix() (built-in function), 232
PushRotate, 233
PushScale, 233
PushTranslate, 234
put, 234
Python, 444

Q

QueryPointer, 234
Qutaiba, 17

R

radix, 19
Raise, 235
Ralph, 17
read, 236
ReadImage, 236
readline, 566
readlink, 238
readlink() (built-in function), 238
reads, 238
reads() (built-in function), 238
ready, 239
Real, **22**
real, 239
REBOL, 455
receive, 240
Record, 34
record, **88**
Refresh, 240
Regina, 455
regular expressions, 383
Rem, 241
remove, 242
rename, 242
repeat, **74**
repl, 243
reserved word list, 61
reserved words
    action, 62
    declarative, 79
    syntax, 89
RetInteger (C function), 199
return, **74**
reverse, 243
REXX, 455, 556, 639
RFC3875, 409
right, 244
rmdir, 244
rosettacode, 625
Rotate, 245
Rpos, 245
Rtab, 245
rtod, 246
Ruby, 456
runerr, 247
Rust, 456

S

S-Lang, 457, 499
save, 247
Scale, 248
scaling suffix, 21
scanning
    pattern, 381
    string, 379
Scenarios, 495
Scheme, 456
scope, 9, 41
seek, 248
select, 249
semicolon
    automatic insertion, 42
send, 249
seq, 250
serial, 250
serif, 640
set, 250
setenv, 251
setgid, 251

setgrent, 252
sethostent, 253
setpgrp, 253
setpwent, 253
setservent, 253
setuid, 254
SEXI, 639
signal, 254
sin, 254
skeleton
    ipl, 376
skeleton.icn, 421
Smalltalk, 458
SNOBOL, 458, 640
    patterns, 381
snobol, 568
soldout, 564
sort, 256
sortf, 257
Span, 257
spawn, 257
sql, 259
sqrt, 260
Stallman
    Richard, 639
stat, 261
static, **88**
staticnames, 262
stop, 262
StopAudio, 263
String, **26**
string, 263
    scanning, 379
string scanning, 379
strings, 471
structure, 264
style, 355
subscripts, 26
Succeed, 264
success, 37
suspend, **75**
Swi, 265
Symisc, 535, 538, 541, 546
symlink, 265
sys_errstr, 266
system, 266
syswrite, 266

T

Tab, 267
tab, 267
Table, 32
table, 268
tan, 268
tasks, 415
tcc, 589
Tcl, 458, 641
tectonics, 641
testing, 425
Texcoord, 270
Texture, 270
TextWidth, 270
then, **90**
theory, 623
thread, **76**
threads, 413
tighloop
    Python, 444
tightloop, 443
    Ada, 445
    ALGOL, 446
    Assembler, 446
    BASIC, 447
    C, 445
    COBOL, 448
    D, 448
    ECMAScript, 449
    Elixir, 449
    Forth, 450
    Fortran, 450
    Groovy, 451
    Java, 451
    loadfunc, 460
    Lua, 452
    Neko, 452
    Nickle, 453
    Nim, 453
    Perl, 454
    PHP, 454
    REBOL, 455
    REXX, 455
    Ruby, 456
    Rust, 456
    S-Lang, 457
    Scheme, 456
    shell, 457
    Smalltalk, 458
    SNOBOL, 458
    sources, 444
    Tcl, 458
    Unicon, 444
    Vala, 459
Tiny C, 589
Tk, 641
to, **78**
tools, 349, 357
TP, 437
TRACE, 435

trace, 435
tracing, 435
transforms, 204
Translate, 271
transliterations, 348
trap, 271
trig
    sin, 254
trim, 272
truncate, 272
trylock, 272
twosum, 626
type, 273

U

UCL, 493
UDB, 435
udb, 359
ui, 359
umask, 274
Uncouple, 275
uniclass, 641
unicob.cob, 508
unicob.icn, 510
Unicon, 3
unicon, **350**
unicon-i18n.c, 561
unicon-i18n.icn, 562
unidoc, 423
uniduk-v1, 511
uniduk.c, 512
uniffi, 608
unificl-v1.c, 519
unificl.c, 521
unilist, 585
unilua-v1.c, 526
unilua.c, 527
unireadline.c, 566
uniruby-v1.c, 517
unit testing, 425
unitest, 426
Unix epoch, 642
unlock, 275
unqlite, 546
until, **78**
upto, 275
Use cases, 495
utime, 276
utr, 423
uval.icn, 583

V

Vala, 459
variable, 276
VAttrib, 277
VAX, 642
vedis, 535
vib, 359
vidget, 392
vidgets, 391
Vim, 371
vim, 422
VimL, 642
VM, 359
VMS, 642
VOIP, 642

W

wait, 277
WAttrib, 278
WAttrib() (built-in function), 278
WDefault, 280
WFlush, 281
WFlush() (built-in function), 281
where, 281
while, **79**
widget, 392
WinAssociate, 282
WinButton, 282
WinColorDialog, 283
Window, 27
WindowContents, 283
WinEditRegion, 285
WinFontDialog, 285
WinMenuBar, 286
WinOpenDialog, 286
WinPlayMedia, 286
WinSaveDialog, 287
WinScrollBar, 287
WinSelectDialog, 288
wrap, 470
write, 288
WriteImage, 289
writes, 289
WSection, 290
WSync, 291

X

ximage, 470

Y

y2k, 642