## Contents

### Preface

15

- About the SKILL Language
- SKILL Development Helpful Hints
- SKILL API Documentation
- Document Conventions

### SKILL Language Functions

25

- abs
- acos
- add1
- addDefstructClass
- alias
- alphalessp
- alphaNumCmp
- and
- append
- append1
- apply
- argc
- argv
- arrayp
- arrayref
- asin
- assoc, assq, assv
- atan
- atof
- atoi
- atom
- band
- bcdp
- begin - SKILL mode
### SKILL Language Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>declare</td>
<td>98</td>
</tr>
<tr>
<td>declareLambda</td>
<td>100</td>
</tr>
<tr>
<td>declareNLambda</td>
<td>102</td>
</tr>
<tr>
<td>declareSQNLambda</td>
<td>103</td>
</tr>
<tr>
<td>define - SKILL++ mode</td>
<td>104</td>
</tr>
<tr>
<td>definemacro</td>
<td>106</td>
</tr>
<tr>
<td>defMathConstants</td>
<td>107</td>
</tr>
<tr>
<td>defprop</td>
<td>109</td>
</tr>
<tr>
<td>defstruct</td>
<td>110</td>
</tr>
<tr>
<td>defstructp</td>
<td>112</td>
</tr>
<tr>
<td>defun</td>
<td>113</td>
</tr>
<tr>
<td>defUserInitProc</td>
<td>115</td>
</tr>
<tr>
<td>defvar - SKILL mode only</td>
<td>116</td>
</tr>
<tr>
<td>deleteDir</td>
<td>117</td>
</tr>
<tr>
<td>deleteFile</td>
<td>118</td>
</tr>
<tr>
<td>difference</td>
<td>120</td>
</tr>
<tr>
<td>display</td>
<td>121</td>
</tr>
<tr>
<td>do - SKILL++ mode only</td>
<td>122</td>
</tr>
<tr>
<td>drain</td>
<td>125</td>
</tr>
<tr>
<td>dtpr</td>
<td>127</td>
</tr>
<tr>
<td>ed</td>
<td>128</td>
</tr>
<tr>
<td>edi</td>
<td>129</td>
</tr>
<tr>
<td>edit</td>
<td>130</td>
</tr>
<tr>
<td>edl</td>
<td>132</td>
</tr>
<tr>
<td>envobj</td>
<td>133</td>
</tr>
<tr>
<td>eq</td>
<td>134</td>
</tr>
<tr>
<td>equal</td>
<td>136</td>
</tr>
<tr>
<td>eqv</td>
<td>138</td>
</tr>
<tr>
<td>err</td>
<td>139</td>
</tr>
<tr>
<td>error</td>
<td>141</td>
</tr>
<tr>
<td>errset</td>
<td>143</td>
</tr>
<tr>
<td>errsetstring</td>
<td>145</td>
</tr>
<tr>
<td>eval</td>
<td>147</td>
</tr>
<tr>
<td>evalstring</td>
<td>149</td>
</tr>
<tr>
<td>evenp</td>
<td>151</td>
</tr>
<tr>
<td>exists</td>
<td>152</td>
</tr>
</tbody>
</table>
SKILL Language Reference

exit ................................................................. 154
exp ................................................................. 156
expandMacro ....................................................... 157
expt ............................................................... 158
fboundp ........................................................... 159
fileLength ......................................................... 160
fileSeek .......................................................... 161
fileTell ........................................................... 163
fileTimeModified ................................................ 164
fix ................................................................. 165
fixp ............................................................... 166
float ............................................................... 167
floatp ............................................................. 168
floor ............................................................... 169
for ................................................................. 170
forall ............................................................. 172
foreach .......................................................... 174
fprintf ........................................................... 177
tscanf, scanf, sscanf ............................................ 180
funcall ............................................................ 183
funobj ............................................................. 185
gc ................................................................. 186
gensym ............................................................ 188
ggeqp .............................................................. 189
get ................................................................. 190
get_filename ....................................................... 191
get_pname ........................................................ 192
get_string ........................................................ 193
getc ............................................................... 194
getchar ............................................................ 195
getCurrentTime .................................................. 196
getd ............................................................... 197
getDirFiles ....................................................... 198
getFnWriteProtect .............................................. 199
getFunType ........................................................ 200
getInstallPath .................................................... 201
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLogin</td>
<td>202</td>
</tr>
<tr>
<td>getPrompts</td>
<td>203</td>
</tr>
<tr>
<td>getq</td>
<td>204</td>
</tr>
<tr>
<td>getqq</td>
<td>206</td>
</tr>
<tr>
<td>getTempDir</td>
<td>208</td>
</tr>
<tr>
<td>gets</td>
<td>209</td>
</tr>
<tr>
<td>getShellEnvVar</td>
<td>210</td>
</tr>
<tr>
<td>getSkillPath</td>
<td>211</td>
</tr>
<tr>
<td>getSkillVersion</td>
<td>212</td>
</tr>
<tr>
<td>getVarWriteProtect - SKILL mode only</td>
<td>213</td>
</tr>
<tr>
<td>getVersion</td>
<td>214</td>
</tr>
<tr>
<td>getWarn</td>
<td>215</td>
</tr>
<tr>
<td>getWorkingDir</td>
<td>217</td>
</tr>
<tr>
<td>go</td>
<td>218</td>
</tr>
<tr>
<td>greaterp</td>
<td>219</td>
</tr>
<tr>
<td>help</td>
<td>220</td>
</tr>
<tr>
<td>if</td>
<td>222</td>
</tr>
<tr>
<td>importSkillVar - SKILL++ mode</td>
<td>224</td>
</tr>
<tr>
<td>index</td>
<td>226</td>
</tr>
<tr>
<td>infile</td>
<td>227</td>
</tr>
<tr>
<td>importp</td>
<td>229</td>
</tr>
<tr>
<td>inScheme</td>
<td>230</td>
</tr>
<tr>
<td>inSkill</td>
<td>231</td>
</tr>
<tr>
<td>instring</td>
<td>232</td>
</tr>
<tr>
<td>integerp</td>
<td>233</td>
</tr>
<tr>
<td>intToChar</td>
<td>234</td>
</tr>
<tr>
<td>isCallable</td>
<td>235</td>
</tr>
<tr>
<td>isDir</td>
<td>236</td>
</tr>
<tr>
<td>isExecutable</td>
<td>238</td>
</tr>
<tr>
<td>isFile</td>
<td>240</td>
</tr>
<tr>
<td>isFileEncrypted</td>
<td>241</td>
</tr>
<tr>
<td>isFileName</td>
<td>242</td>
</tr>
<tr>
<td>isInfinity</td>
<td>244</td>
</tr>
<tr>
<td>isLargeFile</td>
<td>245</td>
</tr>
<tr>
<td>isLink</td>
<td>246</td>
</tr>
<tr>
<td>isMacro</td>
<td>247</td>
</tr>
</tbody>
</table>
## SKILL Language Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNaN</td>
<td>248</td>
</tr>
<tr>
<td>isReadable</td>
<td>249</td>
</tr>
<tr>
<td>isWritable</td>
<td>250</td>
</tr>
<tr>
<td>lambda</td>
<td>251</td>
</tr>
<tr>
<td>last</td>
<td>252</td>
</tr>
<tr>
<td>lconc</td>
<td>253</td>
</tr>
<tr>
<td>leftshift</td>
<td>254</td>
</tr>
<tr>
<td>length</td>
<td>255</td>
</tr>
<tr>
<td>leqp</td>
<td>256</td>
</tr>
<tr>
<td>lessp</td>
<td>257</td>
</tr>
<tr>
<td>let - SKILL mode</td>
<td>258</td>
</tr>
<tr>
<td>let - SKILL++ mode</td>
<td>260</td>
</tr>
<tr>
<td>letrec - SKILL++ mode</td>
<td>263</td>
</tr>
<tr>
<td>letseq - SKILL++ mode</td>
<td>265</td>
</tr>
<tr>
<td>lineread</td>
<td>267</td>
</tr>
<tr>
<td>linereadstring</td>
<td>269</td>
</tr>
<tr>
<td>list</td>
<td>270</td>
</tr>
<tr>
<td>listp</td>
<td>271</td>
</tr>
<tr>
<td>listToVector</td>
<td>272</td>
</tr>
<tr>
<td>load</td>
<td>273</td>
</tr>
<tr>
<td>loadi</td>
<td>275</td>
</tr>
<tr>
<td>loadstring</td>
<td>277</td>
</tr>
<tr>
<td>log</td>
<td>279</td>
</tr>
<tr>
<td>log10</td>
<td>280</td>
</tr>
<tr>
<td>lowerCase</td>
<td>281</td>
</tr>
<tr>
<td>make &lt;name&gt;</td>
<td>282</td>
</tr>
<tr>
<td>makeTable</td>
<td>283</td>
</tr>
<tr>
<td>makeTempFileName</td>
<td>285</td>
</tr>
<tr>
<td>makeVector</td>
<td>286</td>
</tr>
<tr>
<td>map</td>
<td>287</td>
</tr>
<tr>
<td>mapc</td>
<td>289</td>
</tr>
<tr>
<td>mapcan</td>
<td>291</td>
</tr>
<tr>
<td>mapcar</td>
<td>293</td>
</tr>
<tr>
<td>maplist</td>
<td>295</td>
</tr>
<tr>
<td>max</td>
<td>297</td>
</tr>
<tr>
<td>measureTime</td>
<td>298</td>
</tr>
</tbody>
</table>
member, memq, memv ............................................................. 300
min .............................................................. 302
minus ............................................................. 303
minusp ........................................................... 304
mod ............................................................. 305
modf ............................................................ 306
modulo ......................................................... 307
mprocedure ...................................................... 309
nconc ........................................................... 311
ncons ........................................................... 313
needNCells ....................................................... 314
negativep ......................................................... 315
neq ............................................................. 316
nequal .......................................................... 317
newline .......................................................... 318
nindex ........................................................... 319
nlambda - SKILL mode only ........................................... 320
not ............................................................... 322
nprocedure - SKILL mode only ....................................... 323
nth ............................................................. 325
nthcdr .......................................................... 326
nthelem ........................................................ 327
null ............................................................. 328
numberp ........................................................ 329
numOpenFiles .................................................. 330
oddp ........................................................... 331
onep ........................................................... 332
openportp ....................................................... 333
or ............................................................... 334
otherp .......................................................... 335
outfile .......................................................... 336
outportp ........................................................ 338
pairp ........................................................... 339
parseString ....................................................... 340
plist ............................................................. 342
plus .............................................................. 343
SKILL Language Reference

plusp ................................................................. 344
portp ................................................................. 345
postdecrement .................................................... 346
postincrement ...................................................... 347
print ................................................................. 348
predecrement ....................................................... 349
preincrement ....................................................... 350
prependInstallPath ............................................. 351
print ................................................................. 352
printf ................................................................. 353
println ................................................................. 355
procedure ........................................................... 358
procedurep .......................................................... 362
prog ................................................................. 363
prog1 ................................................................. 365
prog2 ................................................................. 366
progn ................................................................. 367
putd ................................................................. 368
putprop ............................................................... 370
putpropq ............................................................. 371
putpropqq ............................................................ 373
quote ................................................................. 374
quotient ............................................................... 375
random ............................................................... 376
range ................................................................. 377
read ................................................................. 378
readstring .......................................................... 380
readTable ............................................................ 382
reap ................................................................. 383
regExitAfter ....................................................... 384
regExitBefore ...................................................... 385
remainder .......................................................... 386
remd ................................................................. 387
remdq ................................................................. 389
remExitProc ........................................................ 391
remove ................................................................. 392
remprop ............................................................... 394
remq ................................................................. 395
renameFile ......................................................... 396
return .............................................................. 397
reverse ........................................................... 399
rexCompile ......................................................... 400
rexExecute ......................................................... 403
rexMagic ........................................................... 404
rexMatchAssocList ............................................... 406
rexMatchList ..................................................... 408
rexMatchp ......................................................... 409
rexReplace ......................................................... 410
rexSubstitute ..................................................... 412
rightshift .......................................................... 413
rindex .............................................................. 414
round .............................................................. 415
rplaca ............................................................. 416
rplacd ............................................................. 417
schemeTopLevelEnv ............................................ 418
set ................................................................. 419
setarray ........................................................... 420
setcar ............................................................. 422
setcdr ............................................................. 423
setFnWriteProtect ............................................... 424
setof .............................................................. 425
setplist ........................................................... 427
setPrompts ....................................................... 428
setq .............................................................. 430
setqbitfield1 ..................................................... 432
setqbitfield ...................................................... 433
setShellEnvVar .................................................. 434
setSkillPath ...................................................... 435
setVarWriteProtect - SKILL mode only ..................... 437
sh, shell .......................................................... 439
simplifyFilename ............................................... 440
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>441</td>
</tr>
<tr>
<td>sort</td>
<td>442</td>
</tr>
<tr>
<td>sortcar</td>
<td>444</td>
</tr>
<tr>
<td>sprintf</td>
<td>446</td>
</tr>
<tr>
<td>sqrt</td>
<td>448</td>
</tr>
<tr>
<td>srandom</td>
<td>449</td>
</tr>
<tr>
<td>sstatus</td>
<td>450</td>
</tr>
<tr>
<td>status</td>
<td>453</td>
</tr>
<tr>
<td>strcat</td>
<td>454</td>
</tr>
<tr>
<td>strcmp</td>
<td>455</td>
</tr>
<tr>
<td>stringp</td>
<td>456</td>
</tr>
<tr>
<td>stringToFunction</td>
<td>457</td>
</tr>
<tr>
<td>stringToSymbol</td>
<td>458</td>
</tr>
<tr>
<td>stringToTime</td>
<td>459</td>
</tr>
<tr>
<td>strlen</td>
<td>460</td>
</tr>
<tr>
<td>strncat</td>
<td>461</td>
</tr>
<tr>
<td>strncmp</td>
<td>462</td>
</tr>
<tr>
<td>sub1</td>
<td>464</td>
</tr>
<tr>
<td>subst</td>
<td>465</td>
</tr>
<tr>
<td>substring</td>
<td>466</td>
</tr>
<tr>
<td>sxtd</td>
<td>468</td>
</tr>
<tr>
<td>symbolp</td>
<td>469</td>
</tr>
<tr>
<td>symbolToString</td>
<td>470</td>
</tr>
<tr>
<td>symeval</td>
<td>471</td>
</tr>
<tr>
<td>symstrp</td>
<td>472</td>
</tr>
<tr>
<td>system</td>
<td>473</td>
</tr>
<tr>
<td>tablep</td>
<td>474</td>
</tr>
<tr>
<td>tableToList</td>
<td>475</td>
</tr>
<tr>
<td>tailp</td>
<td>476</td>
</tr>
<tr>
<td>tan</td>
<td>477</td>
</tr>
<tr>
<td>tconc</td>
<td>478</td>
</tr>
<tr>
<td>theEnvironment</td>
<td>480</td>
</tr>
<tr>
<td>times</td>
<td>483</td>
</tr>
<tr>
<td>timeToString</td>
<td>484</td>
</tr>
<tr>
<td>timeToTm</td>
<td>485</td>
</tr>
<tr>
<td>tmToTime</td>
<td>487</td>
</tr>
</tbody>
</table>
truncate ................................................................. 489
type, typep .............................................................. 490
unalias ................................................................. 491
unless ................................................................. 492
upperCase ............................................................. 493
vector ................................................................. 494
vectorp ................................................................. 495
vectorToList .......................................................... 496
vi, vii, vil ............................................................... 497
warn ................................................................. 498
when ................................................................. 500
which ................................................................. 501
while ................................................................. 503
write ................................................................. 504
writeTable ........................................................... 505
xcons ................................................................. 506
xCoord ............................................................... 507
xdifference .......................................................... 508
xplus ................................................................. 509
xquotient ............................................................. 510
xtimes ............................................................... 511
yCoord ............................................................... 512
zerop ................................................................. 513
zxtd ................................................................. 514

A
Scheme/SKILL++ Equivalents Tables ........................................ 515
Lexical Structure ....................................................... 516
Expressions .......................................................... 517
Functions ............................................................. 518
Preface

This manual is for the following users

- Programmers beginning to program in SKILL
- CAD developers who have experience programming in SKILL, both Cadence internal users and Cadence customers.
- CAD integrators.

The companion for this manual is the SKILL Language User Guide, which

- Introduces the SKILL language to new users
- Leads users to understand advanced topics
- Encourages sound SKILL programming methods

See also:

- About the SKILL Language on page 16
- SKILL Development Helpful Hints on page 17
- SKILL API Documentation on page 18
- Document Conventions on page 19
About the SKILL Language

The SKILL programming language lets you customize and extend your design environment. SKILL provides a safe, high-level programming environment that automatically handles many traditional system programming operations, such as memory management. SKILL programs can be immediately executed in the Cadence environment.

SKILL is ideal for rapid prototyping. You can incrementally validate the steps of your algorithm before incorporating them in a larger program.

Storage management errors are persistently the most common reason cited for schedule delays in traditional software development. SKILL's automatic storage management relieves your program of the burden of explicit storage management. You gain control of your software development schedule.

SKILL also controls notoriously error-prone system programming tasks like list management and complex exception handling, allowing you to focus on the relevant details of your algorithm or user interface design. Your programs will be more maintainable because they will be more concise.

The Cadence environment allows SKILL program development such as user interface customization. The SKILL Development Environment contains powerful tracing, debugging, and profiling tools for more ambitious projects.

SKILL leverages your investment in Cadence technology because you can combine existing functionality and add new capabilities.

SKILL allows you to access and control all the components of your tool environment: the User Interface Management System, the Design Database, and the commands of any integrated design tool. You can even loosely couple proprietary design tools as separate processes with SKILL's interprocess communication facilities.
SKILL Development Helpful Hints

Here are some helpful hints:

- You can click Help in the SKILL Development Toolbox to access SKILL Development Help for information about utilities available in the toolbox. The Walkthrough can guide you through the tasks you perform when you develop SKILL programs using the SKILL Development Toolbox. You can also find information about SKILL lint messages, and message groups.

- You can use the SKILL Finder to access syntax and abstracts for SKILL language functions and application procedural interfaces (APIs).

- You can copy examples from windows and paste the code directly into the Command Interprete Window (CIW) or use the code in nongraphics SKILL mode. To select text:
  - Use Control+drag to select a text segment of any size.
  - Use Control+double-click to select a word.
  - Use Control+triple-click to select an entire section.

For more information about Cadence SKILL language and other related products, see:

- SKILL Development Help
- SKILL Development Functions Reference
- SKILL Language Functions Reference
- Interprocess Communication SKILL Functions Reference
- SKILL++ Object System Functions Reference

**Note:** The Cadence Installation Guide tells you how to install the product.

For more information about Cadence SKILL language and other related products, see:

- SKILL Development Help
- SKILL Development Functions Reference
- SKILL Language Functions Reference
- Interprocess Communication SKILL Functions Reference
- SKILL++ Object System Functions Reference

**Note:** The Cadence Installation Guide tells you how to install the product.
SKILL API Documentation

Cadence tools have their own application procedural interface functions. You can access the SKILL function references in the CDSDoc library by selecting Docs by Product and opening the SKILL folder. The set of books you will find there include the following:

- *Cadence Design Framework II SKILL Functions Reference* contains APIs for the graphics editor, database access, design management, technology file administration, online environment, design flow, user entry, display lists, component description format, and graph browser.

- *Cadence User Interface SKILL Functions Reference* contains APIs for management of windows and forms.
Document Conventions

The conventions used in this document are explained in the following sections. This includes the subsections used in the definition of each function and the font and style of the syntax conventions.

Section Names and Meaning

Each function can have up to seven sections. Not every section is required for every function description.

■ Syntax
  The syntax requirements for this function.

■ Prerequisites
  Steps required before calling this function.

■ Description
  A brief phrase identifying the purpose of the function.
  A text description of the operation performed by the function.

■ Arguments
  An explanation of the arguments input to the function.

■ Return Value
  An explanation of the value returned by the function.

■ Example
  Actual SKILL code using this function.

■ References
  Other functions that are relevant to the operation of this function: ones with partial or similar functionality or which could be called by or could call this function. Sections in this manual which explain how to use this function.
Syntax Conventions

This list describes the syntax conventions used in this document.

**literal (LITERAL)**

Nonitalic (UPPERCASE) words indicate keywords that you must enter literally. These keywords represent command (function, routine) or option names.

**argument (z_argument)**

Words in italics indicate user-defined arguments for which you must substitute a name or a value. (The characters before the underscore (_) in the word indicate the data types that this argument can take. Names are case sensitive. Do not type the underscore (z_) before your arguments.)

| Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character.

[ ] Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.

{ } Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.

... Three dots (…) indicate that you can repeat the previous argument. If you use them with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.

argument... ;specify at least one, ;but more are possible

[argument]... ;you can specify zero or more

,.. A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.

=> A right arrow points to the return values of the function. Variable values returned by the software are shown in italics. Returned literals, such as t and nil, are in plain text. The right arrow is also used in code examples in SKILL manuals.
A slash separates the possible values that can be returned by a SKILL function.

Note: The language requires any characters not included in the list above. You must enter required characters literally.

**SKILL Data Types**

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Internal Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>array</td>
<td>array</td>
</tr>
<tr>
<td>b</td>
<td>ddUserType</td>
<td>Boolean</td>
</tr>
<tr>
<td>c</td>
<td>opfcontext</td>
<td>OPF context</td>
</tr>
<tr>
<td>d</td>
<td>dbobject</td>
<td>Cadence database object (CDBA)</td>
</tr>
<tr>
<td>e</td>
<td>envobj</td>
<td>environment</td>
</tr>
<tr>
<td>f</td>
<td>flonum</td>
<td>floating-point number</td>
</tr>
<tr>
<td>F</td>
<td>opffile</td>
<td>OPF file ID</td>
</tr>
<tr>
<td>g</td>
<td>general</td>
<td>any data type</td>
</tr>
<tr>
<td>G</td>
<td>gdmSpecIlUserType</td>
<td>gdm spec</td>
</tr>
<tr>
<td>h</td>
<td>hdbobject</td>
<td>hierarchical database configuration object</td>
</tr>
<tr>
<td>l</td>
<td>list</td>
<td>linked list</td>
</tr>
<tr>
<td>m</td>
<td>nmpIlUserType</td>
<td>nmpIl user type</td>
</tr>
<tr>
<td>M</td>
<td>cdsEvalObject</td>
<td>—</td>
</tr>
<tr>
<td>n</td>
<td>number</td>
<td>integer or floating-point number</td>
</tr>
<tr>
<td>o</td>
<td>userType</td>
<td>user-defined type (other)</td>
</tr>
<tr>
<td>p</td>
<td>port</td>
<td>I/O port</td>
</tr>
<tr>
<td>q</td>
<td>gdmspecListIlUserType</td>
<td>gdm spec list</td>
</tr>
<tr>
<td>r</td>
<td>defstruct</td>
<td>defstruct</td>
</tr>
<tr>
<td>R</td>
<td>rodObj</td>
<td>relative object design (ROD) object</td>
</tr>
<tr>
<td>s</td>
<td>symbol</td>
<td>symbol</td>
</tr>
<tr>
<td>S</td>
<td>stringSymbol</td>
<td>symbol or character string</td>
</tr>
<tr>
<td>t</td>
<td>string</td>
<td>character string (text)</td>
</tr>
<tr>
<td>Prefix</td>
<td>Internal Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>$u$</td>
<td>function</td>
<td>function object, either the name of a function (symbol) or a lambda function body (list)</td>
</tr>
<tr>
<td>$U$</td>
<td>funobj</td>
<td>function object</td>
</tr>
<tr>
<td>$v$</td>
<td>hdbpath</td>
<td>—</td>
</tr>
<tr>
<td>$w$</td>
<td>wtype</td>
<td>window type</td>
</tr>
<tr>
<td>$x$</td>
<td>integer</td>
<td>integer number</td>
</tr>
<tr>
<td>$y$</td>
<td>binary</td>
<td>binary function</td>
</tr>
<tr>
<td>$s$</td>
<td>pointer</td>
<td>pointer type</td>
</tr>
</tbody>
</table>
SKILL Syntax Examples

The following examples show typical syntax characters used in SKILL.

Example 1

```
list( g_arg1 [ g_arg2 ] ... ) => l_result
```

This example illustrates the following syntax characters.

- `list` : Plain type indicates words that you must enter literally.
- `g_arg1` : Words in italics indicate arguments for which you must substitute a name or a value.
- `()` : Parentheses separate names of functions from their arguments.
- `_` : An underscore separates an argument type (left) from an argument name (right).
- `[ ]` : Brackets indicate that the enclosed argument is optional.
- `...` : Three dots indicate that the preceding item can appear any number of times.
- `=>` : A right arrow points to the description of the return value of the function. Also used in code examples in SKILL manuals.
- `l_result` : All SKILL functions compute a data value known as the return value of the function.

Example 2

```
needNCells( s_cellType | st_userType x_cellCount ) => t / nil
```

This example illustrates two additional syntax characters.

- `|` : Vertical bars separate a choice of required options.
- `/` : Slashes separate possible return values.
SKILL Language Functions

Cadence® SKILL is a high-level, interactive programming language based on the popular artificial intelligence language, Lisp.

This chapter describes the following:

- Functions that are common to all of the Cadence tools used in either a graphic or nongraphic environment. For information about using these functions, see the SKILL Language User Guide.

- SKILL++ core functions. SKILL++ is the second generation extension language for Cadence software. SKILL++ combines the ease-of-use of the SKILL environment with the power of the Scheme programming language. The major power brought in from Scheme is its use of lexical scoping and functions with lexically-closed environments called closures.

- Arithmetic and logical operators. All arithmetic operators are translated into calls to predefined SKILL functions. These operators are listed in Chapter 5, “Arithmetic and Logical Expressions” in the SKILL Language User Guide.
abs

abs(
    n_number
)
=> n_result

Description

Returns the absolute value of a floating-point number or integer.

Arguments

n_number Floating-point number or integer.

Value Returned

n_result Absolute value of n_number.

Example

abs( -209.625) => 209.625
abs( -23) => 23

Reference

min
acos

acos(
    n_number
)
=> f_result

Description

Returns the arc cosine of a floating-point number or integer.

Arguments

n_number  Floating-point number or integer.

Value Returned

f_result  Arc cosine of n_number.

Example

acos(0.3)
=> 1.266104

Reference

cos
**add1**

```
add1(
    n_number
)
=> n_result
```

**Description**

Adds one to a floating-point number or integer.

**Arguments**

- `n_number`  
  Floating-point number or integer to increase by one.

**Value Returned**

- `n_result`  
  `n_number` plus one.

**Example**

```
add1( 59 )
=> 60
```

**Reference**

- `sub1`
### addDefstructClass

```plaintext
addDefstructClass(
    s_name
)
=> u_classObject
```

#### Description

Creates a class for the `defstruct`.

By default, an instance of a `defstruct` does not have class. You cannot use `makeInstance` to instantiate this class. Use the instantiation function created by `defstruct`.

Using `addDefstructClass` to create a class for a `defstruct`, allows you to define methods for a `defstruct`.

#### Arguments

- `s_name` The name of the `defstruct`

#### Value Returned

- `u_classObject` The class object

#### Example

```plaintext
defstruct( card rank suit ) => t
x = make_card( ?rank 8 ?suit "spades" )
=> array[4]:3897312
type( x ) => card
findClass( 'card' ) => nil
classOf( x ) => nil
addDefstructClass( card ) => funobj:0x1c98f8
className( classOf( x ) ) => card
```

#### Reference

- `defstruct`, `makeInstance`
alias

```plaintext
alias(
  s_aliasName
  s_functionName
)
=> s_aliasName
```

**Description**

Defines a symbol as an alias for a function. This is an nlambda function.

Defines the `s_aliasName` symbol as an alias for the `s_functionName` function, which must already have been defined. The alias function does not evaluate its arguments.

⚠️ **Caution**

*Use alias only to speed up interactive command entry and never in programs.*

**Arguments**

- `s_aliasName` Symbol name of the alias.
- `s_functionName` Name of the function you are creating an alias for.

**Value Returned**

- `s_aliasName` Name of the alias.

**Example**

```plaintext
alias path getSkillPath => path
```

Aliases `path` to the `getSkillPath` function.

```plaintext
alias e edit => e
```

Aliases `e` to the `edit` function.
Reference

unalias
### alphalessp

alphalessp(
    S_arg1
    S_arg2
)
=> t | nil

#### Description

Compares two string or symbol names alphabetically.

This function returns `t` if the first argument is alphabetically less than the second argument. If `S_arg` is a symbol, then its name is its print name. If `S_arg` is a string, then its name is the string itself.

#### Arguments

- **S_arg1**: First name you want to compare.
- **S_arg2**: Name to compare against.

#### Value Returned

- **t**: If `S_arg1` is alphabetically less than the name of `S_arg2`.
- **nil**: In all other cases.

#### Example

- `alphalessp("name" "name1") => t`
- `alphalessp("third" "fourth") => nil`
- `alphalessp('a 'ab) => t`

#### Reference

- `strcmp`, `strncmp`
alphaNumCmp

alphaNumCmp(  
    S_arg1  
    S_arg2  
    [ g_arg3 ]  
)  
=> 1 | 0 | -1

Description

Compares two string or symbol names alphanumerically or numerically.

If the third optional argument is non-nil and the first two arguments are strings holding purely numeric values, then a numeric comparison is performed on the numeric representation of the strings.

Arguments

S_arg1  First string or symbol to compare.

S_arg2  String or symbol to compare against S_arg1.

g_arg3  If non-nil, can cause a numeric comparison of S_arg1 and S_arg2 depending whether those arguments are strings holding purely numeric values.

Value Returned

1  If S_arg1 is alphanumerically greater than S_arg2

0  If S_arg1 is alphanumerically identical to S_arg2.

-1  If S_arg2 is alphanumerically greater than S_arg1.

Example

alphaNumCmp( "a" "b" ) => -1
alphaNumCmp( "b" "a" ) => 1
alphaNumCmp( "name12" "name12" ) => 0
alphaNumCmp( "name23" "name12" ) => 1
alphaNumCmp( "00.09" "9.0E-2" t) => 0
Reference

str cmp, strncmp, equal, eq
and

```
and(g_arg1
    g_arg2
    [ g_arg3... ]
)
=> nil | g_val
```

**Description**

Evaluates from left to right its arguments to see if the result is `nil`. As soon as an argument evaluates to `nil`, and returns `nil` without evaluating the rest of the arguments. Otherwise, and evaluates the next argument. If all arguments except for the last evaluate to non-nil, and returns the value of the last argument as the result of the function call. Prefix form of the `&&` binary operator.

**Arguments**

- `g_arg1`: Any SKILL object.
- `g_arg2`: Any SKILL object.
- `g_arg3`: Any SKILL object.

**Value Returned**

- `nil`: If an argument evaluates to `nil`.
- `g_val`: Value of the last argument if all the preceding arguments evaluate to non-nil.

**Example**

```
and(nil t) => nil
and(t nil) => nil
and(18 12) => 12
```

**Reference**

`band`, `bland`, `bnor`, `bnot`, `bor`, `bxnor`, `bxor`, `not`
append

append(
  l_list1
  l_list2
)
=> l_result

append(
  o_table
  g_assoc
)
=> o_table

Description

Creates a list containing the elements of \texttt{l_list1} followed by the elements of \texttt{l_list2} or returns the original association table including new entries.

The top-level list cells of \texttt{l_list1} are duplicated and the \texttt{cdr} of the last duplicated list cell is set to point to \texttt{l_list2}; therefore, this is a time-consuming operation if \texttt{l_list1} is a long list.

\textbf{Note:} This is a slow operation and the functions \texttt{tconc}, \texttt{lconc}, and \texttt{nconc} can be used instead for adding an element or a list to the end of a list. The command \texttt{cons} is even better if the new list elements can be added to the beginning of the list.

The \texttt{append} function can also be used with association tables as shown in the second syntax statement. Key/value pairs are added to the original association table (not to a copy of the table). This function should be used mainly in converting existing association lists or disembodied property lists to an association table. See “Association Table” in the \textit{SKILL Language User Guide} for more details.

Arguments

\begin{itemize}
  \item \texttt{l_list1} List of elements to be added to a list.
  \item \texttt{l_list2} List of elements to be added.
  \item \texttt{o_table} Association table to be updated.
  \item \texttt{g_assoc} Key/value pairs to be added to the association table.
\end{itemize}
Value Returned

$l_result$ Returns a list containing elements of $l_list1$ followed by elements of $l_list2$.

$o_table$ Returns the original association table including the new entries.

Example

/* List Example */
append( '(1 2) '(3 4) )
=> (1 2 3 4)

/* Association Table Example */
myTable = makeTable("myAssocTable")
=> table:myAssocTable
myTable[ 'a' ] = 1
=> 1
append(myTable '((b 2) (c 3)))
=> table:myAssocTable

/* Check the contents of the assoc table */
tableToList(myTable)
=> ((a 1) (b 2) (c 3))

Reference

tconc, lconc, nconc, append1, cons
append1

append1(
  l_list
  g_arg
)
=> l_result

Description

Adds new arguments to the end of a list.

Returns a list just like \textit{l\_list} with \textit{g\_arg} added as the last element of the list.

\textbf{Note:} This is a slow operation and the functions \texttt{tconc}, \texttt{lconc}, and \texttt{nconc} can be used instead for adding an element or a list to the end of a list. The command \texttt{cons} is even better if the new list elements can be added to the beginning of the list.

Arguments

\begin{itemize}
  \item \textit{l\_list} \quad List to which \textit{g\_arg} is added.
  \item \textit{g\_arg} \quad Argument to be added to the end of \textit{l\_list}.
\end{itemize}

Value Returned

\begin{itemize}
  \item \textit{l\_result} \quad Returns a copy of \textit{l\_list} with \textit{g\_arg} attached to the end.
\end{itemize}

Example

append1('(1 2 3) 4) => (1 2 3 4)

Like append, append1 duplicates the top-level list cells of \textit{l\_list}.

Reference

append
apply

apply(
    slu_func
    l_args
)
=> g_result

Description

Applies the given function to the given argument list.

The first argument to apply must be either the name of a function or a list containing a lambda/nlambda/macro expression or a function object. The second argument is a list of arguments to be passed to the function.

The argument list l_args is bound to the formal arguments of slu_func according to the type of function. For lambda functions the length of l_args should match the number of formal arguments, unless keywords or optional arguments exist. For nlambda and macro functions, l_args is bound directly to the single formal parameter of the function.

Note: If slu_func is a macro, apply evaluates it only once, that is, it expands it and returns the expanded form, but does not evaluate the expanded form again (as eval does).

Arguments

slu_func Name of the function.

l_args Argument list to apply to the function.

Value Returned

g_result Returns the result of applying the function to the given arguments.

Example

apply('plus (list 1 2) ); Apply plus to its arguments.
=> 3

procedure( sumTail(l) apply( 'plus cdr(l)))
=> sumTail ;Define a procedure
sumTail( '(1 2 3))
=> 5
Reference

`eval`, `funcall`
**argc**

```
argc(
  )
  => n | 0 | -1 | -2
```

**Description**

Returns the number of arguments passed to a SKILL script. Used to enhance the SKILL script environment.

**Value Returned**

- $n$: $n$ arguments were passed ($n$ is an integer).
- 0: No arguments were passed, but `argv(0)` has a value.
- -1: Argument list is `nil` (no arguments passed, and `argv(0)` is `nil`). This can occur when using SKILL interactively.
- -2: Error caused by a problem with the argument list property.

**Example**

Assume that arguments passed to a SKILL script file are `("my.il" "1st" "2nd" "3rd")`:

```
argc() => 3
```

**Reference**

- `argv`
**argv**

```lisp
argv(
    [ x_int ]
) => g_result
```

**Description**

Returns the arguments passed to a SKILL script. Used to enhance the SKILL script environment.

**Arguments**

`x_int` Optional argument; it must be a positive integer.

**Value Returned**

`g_result` The return value depends on the arguments passed.

- **Argument**
  - `argv()`: List of all arguments (list of strings or nil).
  - `argv(0)`: Name of the calling script.
  - `argv(n)`: nth argument as a string or nil if there is no nth argument.

**Example**

Assume that arguments passed to a SKILL script file are (“my.il” “1st” “2nd” “3rd”):

- `argv() => (“1st” “2nd” “3rd”)`
- `argv(0) => “my.il”`
- `argv(1) => “1st”`
- `argv(4) => nil`

**Reference**

`argc`
arrayp

arrayp(
  g_value
)
=> t | nil

Description

Checks if an object is an array.

The suffix p is usually added to the name of a function to indicate that it is a predicate function.

Arguments

\( g_value \)

Any data object.

Value Returned

\( t \)

If \( g_value \) is an array object.

\( \text{nil} \)

Otherwise.

Example

\[
\begin{align*}
declare(x[10])
\text{arrayp}(x) & \Rightarrow t \\
\text{arrayp}('x) & \Rightarrow \text{nil}
\end{align*}
\]

Reference

\text{declare}
arrayref

arrayref(  
g_collection  
g_index  
)  
=> g_element

Description

Returns the element in a collection that is in an array or a table of the given index.

This function is usually called implicitly using the [ ] syntax.

Arguments

g_collection       An array or a table.
g_index            An integer for indexing an array. An arbitrary object for indexing a table.

Value Returned

g_element         The element selected by the given index in the given collection.

Example

a[3]  
=> 100       ;if the fourth element of the array is 100
(arrayref a 3 )  
=> 100       ;same as a[3]

Reference

The syntax a[i] = b, referred to as the setarray function.
**asin**

```plaintext
asin(
    n_number
) => f_result
```

**Description**

Returns the arc sine of a floating-point number or integer.

**Arguments**

- `n_number`: Floating-point number or integer.

**Value Returned**

- `f_result`: Arc sine of the value passed in.

**Example**

```
asin(0.3) => 0.3046927
```

**Reference**

- `sin`
assoc, assq, assv

assv(
  g_key
  l_alist
)
=> l_association | nil

Description

The assoc, assq, and assv functions find the first list in l_alist whose car field is g_key and return that list. assq uses \textsf{eq} to compare \textit{g_key} with the car fields of the lists in \textit{alist}. assoc uses \textsf{equal}. assv uses \textsf{eqv}.

The association list, \textit{l_alist}, must be a list of lists. An association list is a standard data structure that has the form ((key1 value1) (key2 value2) (key3 value3) ...). These functions find the first list in \textit{l_alist} whose car field is \textit{g_key} and return that list. assq uses \textsf{eq} to compare \textit{g_key} with the car fields of the lists in \textit{l_alist}. assv uses \textsf{eqv}. assoc uses \textsf{equal}.

Arguments

\textit{g_key} \hspace{1cm} An arbitrary object as the search key.

\textit{l_alist} \hspace{1cm} Association list. Must be a list of lists.

Value Returned

\textit{l_association} \hspace{1cm} The returned list is always an element of \textit{l_alist}.

\textit{nil} \hspace{1cm} If no list in \textit{l_alist} has \textit{g_key}, as its car.

Example

\begin{verbatim}
e = '((a 1) (b 2) (c 3))
(assq 'a e) => (a 1)
(assq 'b e) => (b 2)
(assq 'd e) => nil
(assq (list 'a) '(((a)) ((b)) ((c)))) => nil
(assoc (list 'a) '(((a)) ((b)) ((c)))) => ((a))
(assv 5 '(((2 3) (5 7) (11 13))) => (5 7)
\end{verbatim}
Reference

eq, equal, eqv
atan

atan(
    \text{n\_number}
) => \text{f\_result}

Description

Returns the arc tangent of a floating-point number or integer.

Arguments

\text{n\_number}               Floating-point number or integer.

Value Returned

\text{f\_result}               Arc tangent of \text{n\_number}.

Example

atan(0.3) => 0.2914568

Reference

tan
**atof**

```plaintext
atof(
    t_string
)
=> f_result | nil
```

**Description**

Converts a string into a floating-point number. Returns `nil` if the given string does not denote a number.

The `atof` function calls the C library function `strtod` to convert a string into a floating-point number. It returns `nil` if `t_string` does not represent a number.

**Arguments**

`t_string`

A string.

**Value Returned**

`f_result`

The floating-point value represented by `t_string`.

`nil`

If `t_string` does not denote a floating-point number.

**Example**

```plaintext
atof("123")   => 123.0
atof("abc")   => nil
atof("123.456") => 123.456
atof("123abc") => 123.0
```

**Reference**

`atoi`
SKILL Language Functions

atoi

atoi(
  t_string
)
  => x_result | nil

Description

Converts a string into an integer. Returns nil if the given string does not denote an integer.

The atoi function calls the C library function strtol to convert a string into an integer. It returns nil if t_string does not represent an integer.

Arguments

t_string
  A string.

Value Returned

x_result
  The integer value represented by t_string.
nil
  If t_string does not denote an integer.

Example

atoi("123")  => 123
atoi("abc")  => nil
atoi("123.456")  => 123
atoi("123abc")  => 123

Reference

atoi
atom

atom(
    g_arg
)
=> t | nil

Description

Checks if an object is an atom.

Atoms are all SKILL objects except non-empty lists. The special symbol \texttt{nil} is both an atom and a list.

Arguments

\textit{g\_arg}

Any SKILL object.

Value Returned

\texttt{t}

If \texttt{g\_arg} is an atom.

\texttt{nil}

If \texttt{g\_arg} is not an atom.

Example

\begin{verbatim}
atom( 'hello' ) => t
x = '(a b c)
atom( x ) => nil
atom( nil ) => t
\end{verbatim}

Reference

\texttt{dtpr, listp}
band

band(
   x_op1
   x_op2
   [ x_op3 ... ]
)
=> x_result

Description

Returns the integer result of the Boolean AND operation on each parallel pair of bits in each operand. Prefix form of the & bitwise operator.

Arguments

x_op1        Operand to be evaluated.
x_op2        Operand to be evaluated.
x_op3        Optional additional operands to be evaluated.

Value Returned

x_result     Result of the operation.

Example

band(12 13)  => 12
band(1 2 3 4 5) => 0

Reference

and, bnand, bnor, bnot, bor, bxnor, bxor, not
**bcdp**

```plaintext
bcdp(
    g_value
)
=> t | nil
```

**Description**

Checks if an object is a binary primitive function.

The suffix \( p \) is usually added to the name of a function to indicate that it is a predicate function.

**Arguments**

- **g_value**
  
  Object to check.

**Value Returned**

- **t**
  
  If \( g_value \) is a binary function.

- **nil**
  
  Otherwise.

**Example**

```plaintext
bcdp(getd('plus')) => t
bcdp('plus') => nil
```

**Reference**

`getd`
begin - SKILL mode

```
begin(
    g_exp1
    [ g_exp2 ...
    g_expN ]
)
=> g_result
```

**Description**

In SKILL mode `begin` is a syntax form used to group a sequence of expressions. Evaluates expressions from left to right and returns the value of the last expression. Equivalent to `progn`.

This expression type is used to sequence side effects such as input and output.

**Arguments**

`g_exp1, g_exp2, g_expN`

Arbitrary expressions.

**Value Returned**

`g_result` Value of the last expression, `g_expN`.

**Example**

```
begin( x = 1 y = 2 z = 3 )
=> 3
```

**Reference**

`progn`
begin - SKILL++ mode

begin(
  def1
  [ def2 ... 
  defN ]
)
=> g_result

begin(
  exp1
  [ exp2 ... 
  expN ]
)
=> g_result

Description

In SKILL++ mode begin is a syntax form used to group either a sequence of expressions or a sequence of definitions.

begin( exp1 [exp2 ... expN] )

The expressions are evaluated sequentially from left to right, and the value of the last expression is returned. This expression type is used to sequence side effects such as input and output.

begin( [def1 def2 ... defN] )

This form is treated as though the set of definitions is given directly in the enclosing context. It is most commonly found in macro definitions.

Value Returned

$g_{result}$ Value of the last expression or definition.

Example

begin( x = 1 y = 2 z = 3 ) => 3
begin( define( x 1 ) define( y 2 ) define( z 3 ) ) => z

Reference

define - SKILL++ mode
bitfield1

```
bitfield1(
    x_val
    x_bitPosition
)
=> x_result
```

**Description**

Returns the value of a specified bit of a specified integer. Prefix form of the <> operator.

**Arguments**

- **x_val**
  Integer for which you want to extract the value of a specified bit.
- **x_bitPosition**
  Position of the bit whose value you want to extract.

**Value Returned**

- **x_result**
  Value of a single bit.

**Example**

```
x = 0b1001
bitfield1(x 0) => 1
bitfield1(x 3) => 1
```

**Reference**

- [bitfield](#), [setqbitfield1](#), [setqbitfield](#)
bitfield

bitfield(
    x_val
    x_msb
    x_lsb
)
=> x_result

Description

Returns the value of a specified set of bits of a specified integer. Prefix form of the <::> operator.

Arguments

x_val
    Integer for which you want to extract the value of a specified set of bits.

x_msb
    Leftmost bit of the set of bits to be extracted.

x_lsb
    Rightmost bit of the set of bits to be extracted.

Value Returned

x_result
    Value of the set of bits.

Example

x = 0b1011
bitfield(x 2 0) => 3
bitfield(x 3 0) => 11

Reference

bitfield1, setqbitfield1, setqbitfield
**blankstrp**

```lang
blankstrp(
    t_string
) => t | nil
```

**Description**

Checks if the given string is empty or has blank space characters only and returns `true`. If there are non-space characters `blankstrp` returns `nil`.

**Arguments**

- `t_string` A string.

**Value Returned**

- `t` If `t_string` is blanks or is an empty string.
- `nil` If there are non-space characters.

**Example**

```lang
    blankstrp( "")
    t
    blankstrp( " ")
    t
    blankstrp( "a string")
    nil
```
bnand

bnand(
x_op1
x_op2
[ x_op3 ... ]
)
=> x_result

Description

Returns the integer result of the Boolean NAND operation on each parallel pair of bits in each operand. Prefix form of the ~& bitwise operator.

Arguments

x_op1 Operand to be evaluated.

x_op2 Operand to be evaluated.

x_op3 Optional additional operands to be evaluated.

Value Returned

x_result Result of the operation.

Example

bnand(12 13) => -13
bnand(1 2 3 4 5) => -1

Reference

and, band, bnor, bnot, bor, bxnor, bxor, not
bnor

bnor(
  x_op1
  x_op2
  [ x_op3 ... ]
) => x_result

Description

Returns the integer result of the Boolean NOR operation on each parallel pair of bits in each operand. Prefix form of the \(~|~\) bitwise operator.

Arguments

- **x_op1**: Operand to be evaluated.
- **x_op2**: Operand to be evaluated.
- **x_op3**: Optional additional operands to be evaluated.

Value Returned

- **x_result**: Result of the operation.

Example

bnor(12 13) => -14
bnor(1 2 3 4 5) => -8

Reference

- and, band, bnand, bnot, bor, bxnor, bxor, not
bnot

bnot (  
    x_op  
)  
=> x_result

Description

Returns the integer result of the Boolean NOT operation on each parallel pair of bits in each operand. Prefix form of the \(~\) (one’s complement) unary operator.

Arguments

x_op  
Operand to be evaluated.

Value Returned

x_result  
Result of the operation.

Example

bnot(12) => -13  
bnot(-12) => 11

Reference

and, band, bnor, bor, bxnor, bxor, not
booleanp

booleanp(  
  g_obj  
)  
=> t | nil

Description

Checks if an object is a boolean. Returns t if the object is t or nil. Returns nil otherwise.

Arguments

g_obj

Any SKILL object.

Value Returned

   t    If g_obj is either t or nil.
  nil  Otherwise.

Example

(booleanp 0 ) => nil
(booleanp nil) => t
(booleanp t) => t
bor

bor(
  x_op1
  x_op2
  [ x_op3 ... ]
)
=> x_result

Description

Returns the integer result of the Boolean OR operation on each parallel pair of bits in each operand. Prefix form of the | bitwise operator.

Arguments

x_op1               Operand to be evaluated.

x_op2               Operand to be evaluated.

x_op3               Optional additional operands to be evaluated.

Value Returned

x_result            Result of the operation.

Example

bor(12 13) => 13
bor(1 2 3 4 5) => 7

Reference

and, band, bnand, bnor, bnot, bxnor, bxor, not
**boundp**

boundp(
    s_arg
    [ e_environment ]
)
=> t | nil

**Description**

Checks if the variable named by a symbol is bound, that is, has been assigned a value. The single argument form of `boundp` only works in SKILL mode.

Remember that a variable can be set to the special symbol `unbound`.

**Note:** The single argument form of `boundp` only works in SKILL mode.

**Arguments**

- **s_arg**
  Symbol to be tested to see if it is bound.

- **e_environment**
  If this argument is given, SKILL++ semantics are used. The symbol will be searched for within the given (lexical) environment.

**Value Returned**

- **t**
  If the symbol `s_arg` has been assigned a value.

- **nil**
  If the symbol `s_arg` has not been assigned a value.

**Example**

```lisp
x = 5 ; Binds x to the value 5.
y = 'unbound ; Unbind y

boundp( 'x )
=> t

boundp( 'y )
=> nil
```
y = 'x
boundp( y )
=> t

; Bind y to the constant x.
; Returns t because y evaluates to x,
; which is bound.
buildString

buildString(
   l_strings
   [ S_glueCharacters ]
) => t_string

Description

Concatenates a list of strings with specified separation characters.

Arguments

l_strings        List of strings. A null string is permitted.

S_glueCharacters Separation characters you use within the strings. A null string is permitted. If this argument is omitted, the default single space is used.

Value Returned

t_string        Strings concatenated with S_glueCharacters. Signals an error if l_strings is not a list of strings.

Example

buildString( ["test" "il"] ",") => "test.il"
buildString( ["usr" "mnt"] "/") => "usr/mnt"
buildString( ["a" "b" "c"] ) => "a b c"
buildString( ["a" "b" "c"] ) => "abc"
buildString( ["A" "B"] "and") => "AandB"

Reference

parseString
bxnor

bxnor(
    x_op1
    x_op2
    [ x_op3 ... ]
) => x_result

Description

Returns the integer result of the Boolean XNOR operation on each parallel pair of bits in each operand. Prefix form of the ~^ bitwise operator.

Arguments

x_op1
Operand to be evaluated.

x_op2
Operand to be evaluated.

x_op3
Optional additional operands to be evaluated.

Value Returned

x_result
Result of the operation.

Example

bxnor(12 13) => -2
bxnor(1 2 3 4 5) => -2

Reference

and, band, bnot, bnor, bor, bxor, not
**bxor**

```plaintext
bxor(
    x_op1
    x_op2
    [ x_op3 ... ]
) => x_result
```

**Description**

Returns the integer result of the Boolean XOR operation on each parallel pair of bits in each operand. Prefix form of the `^` bitwise operator.

**Arguments**

- `x_op1`  
  Operand to be evaluated.
- `x_op2`  
  Operand to be evaluated.
- `x_op3`  
  Optional additional operands to be evaluated.

**Value Returned**

- `x_result`  
  Result of the operation.

**Example**

```plaintext
bxor(12 13) => 1
bxor(1 2 3 4 5) => 1
```

**Reference**

- `and`, `band`, `bnand`, `bnor`, `bnot`, `bor`, `bxnor`, `not`
caar, caaar, caadr, cadr, caddr, cdar, cddr, ...

ca|d[ a|d ][ a|d ][ a|d ]r(
  l_list
)
=> g_result

Description

Performs operations on a list using repeated applications of car and cdr. For example, caaar is equivalent to car( car( car( l_list))). The possible combinations are caaaaar, caaadr, caaar, caadr, caddr, cadr, caaddr, cdaaar, cdaadr, cdaar, cdadar, cdadder, cddadr, cddar, cddr, cdadder, cddadr, cddar, cdddar, cdddr, cddr.

The cadr(l_list) expression, for example, applies cdr to get the tail of the list and then applies car to get the first element of the tail, in effect extracting the second element from the list. SKILL implements all c...r functions with any combination of a and d up to four characters.

Arguments

l_list
List of elements.

Value Returned

g_result
Returns the value of the specified operation.

Example

caaaar('(((1 2 3)(4 5 6))(7 8 9))) => 1

caaaar is equivalent to car( car( car( l_list))).
caaadr('(((1 2 3)(4 5 6))(7 8 9))) => 7

Equivalent to car( car( cdr( l_list))).
caar('(((1 2 3)(4 5 6))(7 8 9))) => (1 2 3)

Equivalent to car( car( l_list)).
z = '(1 2 3)  =>  (1 2 3)
cadr(z) =>  2

Equivalent to \texttt{car( cdr( l_list))}.

\textbf{Reference}

\texttt{car}, \texttt{cdr}
car

```
car(
    l_list
  )
=> g_result
```

**Description**

Returns the first element of a list. **car** is nondestructive, meaning that it returns the first element of a list but does not actually modify the list that was its argument.

The functions **car** and **cdr** are typically used to take a list of objects apart, whereas the **cons** function is usually used to build up a list of objects. **car** was a machine language instruction on the first machine to run Lisp. **car** stands for *contents of the address register*.

**Arguments**

- **l_list**: A list of elements.

**Value Returned**

- **g_result**: Returns the first element in a list. Note that **car(nil)** returns **nil**.

**Example**

```
car( '(a b c) ) => a
z = '(1 2 3) => (1 2 3)
y = car(z) => 1
y => 1
z => (1 2 3)
car(nil) => nil
```

**Reference**

**cdr**, **cons**
case, caseq

case(
    g_selectionExpr
    l_clause1
    [ l_clause2 ... ]
)
=> g_result | nil

Description

Evaluates the selection expression, matches the resulting selector values sequentially against comparators defined in clauses, and executes the expressions in the matching clause. This is a syntax function.

Each `l_clause` is a list of the form `(g_comparator g_expr1 [g_expr2 ...])`, where a comparator is either an atom (that is, a scalar) of any data type or a list of atoms. Comparators are always treated as constants and are never evaluated. The `g_selectionExpr` is evaluated and the resulting selector value is matched sequentially against comparators defined in `l_clause1, l_clause2, ...` and so on. A match occurs when either the selector is equal to the comparator or the selector is equal to one of the elements in the list given as the comparator. If a match is found, the expressions in that clause and that clause only (that is, the first match) are executed. The value of `case` is then the value of the last expression evaluated (that is, the last expression in the clause selected). If there is no match, `case` returns `nil`.

The symbol `t` has special meaning as a comparator in that it matches anything. It is typically used in the last clause to serve as a default case when no match is found with other clauses.

Comparing case with caseq

caseq is a considerably faster version of `case`. caseq uses the function `eq` rather than `equal` for comparison. The comparators for `caseq` are therefore restricted to being either symbols or small integer constants (`-256 <= i <= 255`), or lists containing symbols and small integer constants.

Arguments

g_selectionExpr  An expression whose value is evaluated and tested for equality against the comparators in each clause. When a match is found the rest of the clause is evaluated.
**l_clause1**
An expression whose first element is an atom or list of atoms to be compared against the value of `g_selectionExpr`. The remainder of the `l_clause` is evaluated if a match is found.

**l_clause2**
Zero or more clauses of the same form as `l_clause1`.

**Value Returned**

**g_result**
Returns the value of the last expression evaluated in the matched clause, or `nil` if there is no match.

**nil**
If there is no match.

**Example**

```
nameofmonth = "February"
month = case( nameofmonth
    ("January" 1)
    ("February" 2)
    (t 'Other))
=> 2

procedure( testCase( selector )
    caseq(selector
        {0 println("selector is 0")
         1 println("selector is 1")
         (2 3) println("selector is either 2 or 3")
         (a b) println("selector is either the symbol a or b")
         (t println("selector is none of the above"))
    )}

testCase( 1 )
=> testCase
"selector is 1" ; Printed by caseq statement.
=> nil ; Value returned by println.

testCase( 'b )
"selector is either the symbol a or b" ; Printed by caseq.
=> nil ; Value returned by println.
```

**Reference**

`eq`, `equal`
**cdr**

```
cdr(
    l_list
)
=> l_result
```

**Description**

Returns the tail of the list, that is, the list without its first element.

The expression `cdr(nil)` returns `nil`. `cdr` was a machine language instruction on the first machine to run Lisp. `cdr` stands for *contents of the decrement register*.

**Arguments**

- `l_list`: List of elements.

**Value Returned**

- `l_result`: Returns the end of a list, or the list minus the first element.

**Example**

```
cdr( '(a b c) ) => (b c)
z = '(1 2 3)
cdr(z) => (2 3)
```

**Note:** `cdr` always returns a list, so `cdr(''(2 3))` returns the list `(3)` rather than the integer `3`.

**Reference**

- `caar`, `caaar`, `caadr`, `cadr`, `caddr`, `cdar`, `cdadr`, `...`
**cdsGetInstPath**

```plaintext
cdsGetInstPath(
    [ t_name ]
) => t_string
```

**Description**

Returns the absolute path of the Cadence installation directory as a string. `cdsGetInstPath` is for the cds root hierarchy and is meant to be used by all DFII and non-DFII applications.

**Arguments**

- `t_name`  
  The optional argument `t_name` is appended to the end of the cds root path with a directory separator if necessary.

**Value Returned**

- `t_string`  
  Returns the installation path as a string.

**Example**

```plaintext
cdsGetInstPath() => "/cds/99.02/latest.il"
cdsGetInstPath("tools") =>"/cds/99.02/latest.il/tools"
```

**Reference**

- `getInstallPath`, `getSkillPath`, `getWorkingDir`, `prependInstallPath`
ceiling

ceiling(  
  n_number  
)  
=> x_integer

Description

Returns the smallest integer not smaller than the given argument.

Arguments

n_number    Any number.

Value Returned

x_integer    Smallest integer not smaller than n_number.

Example

(ceiling -4.3)  => -4  
(ceiling 3.5)   => 4

Reference

floor, round, truncate
changeWorkingDir

```lua
changeWorkingDir(
    [ S_name ]
) => t
```

**Description**

Changes the working directory to `S_name`.

Different error messages are printed if the operation fails because the directory does not exist or you do not have search (execute) permission.

**Caution**

*Use this function with care: if “.” is either part of the SKILL path or the libraryPath, changing the working directory can affect the visibility of SKILL files or design data.*

**Arguments**

`S_name`  
Name of the working directory you want to use. Can be specified with either a relative or absolute path. If you supply a relative path, the shell environment is used to search for the directory, not the SKILL path.

**Value Returned**

`t`  
Returns `t` if the function executes successfully. Prints an error message if the directory you tried to change to does not exist. Prints a permission denied message if you do not have search permission.

**Example**

Assume there is a directory `/usr5/design/cpu` with proper permission and there is no test directory under `/usr5/design/cpu`.

```shell
changeWorkingDir( "/usr5/design/cpu") => t
changeWorkingDir( "test")
```
Signals an error about a non-existent directory.

**Reference**

`getWorkingDir`
charToInt

```skill
charToInt(
    s_char
)
=> x_ascii
```

**Description**

Returns the ASCII code of the first character of the given symbol. In SKILL, a single character symbol can be used as a `character` value.

**Arguments**

- `s_char` A symbol.

**Value Returned**

- `x_ascii` The ASCII code of the (first) character of the given symbol.

**Example**

```skill
charToInt('B')
=> 66
charToInt('Before')
=> 66
```

**Reference**

`intToChar`
clearExitProcs

clearExitProcs(
    )
  => t

Description

Removes all registered exit functions (takes no arguments).

Arguments

None.

Value Returned

t

Always returns t.

Example

clearExitProcs( )=> t

Reference

exit, reqExitBefore, regExitAfter, remExitProc
close

close(
    p_port
)
=> t

Description

Drains, closes, and frees a port.

When a file is closed, it frees the FILE* associated with p_port. Do not use this function on piport, poport, stdin, stdout, and stderr.

Arguments

p_port Name of port to close.

Value Returned

t Returns t if the port is closed successfully.

Example

p = outfile("~/test/myFile") => port:"~/test/myFile"
close(p) => t

Reference

outfile, infile, drain
**compareTime**

```skill
compareTime(
    t_time1
    t_time2
)
=> x_difference
```

**Description**

Compares two string arguments, representing a clock-calendar time.

**Arguments**

- `t_time1`: First string in the month day hour:minute:second year format.
- `t_time2`: Second string in the month day hour:minute:second year format.

**Value Returned**

- `x_difference`: An integer representing a time that is later than (positive), equal to (zero), or earlier than (negative) the second argument. The units are seconds.

**Example**

```skill
=> -687777.
```

687,777 seconds have occurred between the two dates given. For a positive number of seconds, the most recent date needs to be given as the first argument.

```skill
=> 600
```

600 seconds (10 minutes) have occurred between the two dates.

**Reference**

- `getCurrentTime`
**compress**

compress(
    t_sourceFile
    t_destFile
)
=> t | error message

**Description**

Reduces the size of a SKILL file, which must be SKILL source code, and places the output into another file.

Compression renders the data less readable because indentation and comments are lost. It is not the same as encrypting the file because the representation of \textit{t\_destFile} is still in ASCII format. This process does not remove the source file.

**Arguments**

\textit{t\_sourceFile} Name of the SKILL source file.

\textit{t\_destFile} Name of the destination file.

**Value Returned**

\textit{t} Returns \textit{t} when function executes successfully.

\textit{error message} Signals an error if problems are encountered compressing the file.

**Example**

compress("triad.il" "triad_cmp.il") => t

**Reference**

encrypt
concat

concat(
  Sx_arg1
  [ Sx_arg2 ... ]
)
=> s_result

Description

Concatenates strings, symbols, or integers into a single symbol.

This function is useful for converting strings to symbols. To concatenate several strings and have a single string returned, use the strcat function. Symbol names are limited to 255 characters.

Symbol functions such as eq, memq, and caseq are much faster than their siblings equal, member, and case because they compare pointers rather than data. You can use concat to convert a string to a symbol before performing memq on large lists for increased speed.

Arguments

Sx_arg1 String, symbol, or integer to be concatenated.
Sx_arg2 Zero or more strings, symbols, or integers to be concatenated.

Value Returned

s_result Returns a symbol whose print name is the result of concatenating the printed representation of the argument or arguments.

Example

concat("string") => string
concat("ab" 123 'xy) => ab123xy
memq( concat( "c" ) '(a b c d e)) => (c d e)

This demonstrates using concat to take advantage of the faster functions such as memq.
Reference

strcat, eq, member, memq, memv, case, caseq
cond

cond(
    l_clause1 ...
)
=> g_result

Description

Examines conditional clauses from left to right until either a clause is satisfied or there are no more clauses remaining. This is a syntax function.

Each clause has the form (g_condition g_expr1 ...). cond examines a clause by evaluating the condition associated with the clause. The clause is said to be “satisfied” if g_condition evaluates to non-nil, in which case expressions in the rest of the clause are evaluated from left to right, and the value returned by the last expression in the clause is returned as the value of the cond form. If g_condition evaluates to nil, however, cond skips the rest of the clause and moves on to the next clause.

Arguments

l_clause1  Each clause should be of the form (g_condition g_expr1 ...) where if g_condition evaluates to non-nil then all the succeeding expressions are evaluated.

Value Returned

g_result  Value of the last expression of the satisfied clause, or nil if no clause is satisfied.

Example

procedure( test(x)
    cond((null x) (println "Arg is null"))
    (numberp x)(println "Arg is a number")
    ((stringp x)(println "Arg is a string"))
    (t (println "Arg is an unknown type")))

test( nil )  => nil; Prints "Arg is null".
test( 5 )    => nil; Prints "Arg is a number".
test( 'sym ) => nil; Prints "Arg is an unknown type".
cons

cons(
    g_element
    l_list
)
=> l_result

Description

Adds an element to the beginning of a list.

Thus the car of l_result is g_element and the cdr of l_result is l_list. l_list can be nil, in which case a new list containing the single element is created.

Arguments

g_element Element to be added to the beginning of l_list.

l_list List that can be nil.

Value Returned

l_result List whose first element is g_element and whose cdr is l_list.

Example

cons(1 nil) => (1)
cons('a '(b c)) => (a b c)

The following example shows how to efficiently build a list from 1 to 100. You can reverse the list if necessary.

x = nil
for( i 1 100 x = cons( i x )) => t
x => (100 99 98 .. 2 1)
x = reverse( x ) => (1 2 3 .. 100)

Reference

car, cdr, append, appendi
constar

constar(
    [ g_arg1 ... ]
    l_list
)
=> l_result

Description

Adds elements to the beginning of a list.

This function is equivalent to cons\(*(), and should be used instead.

The last argument, l_list, must be a list. l_list can be nil, in which case a new list containing the elements is created. The car of l_result is the first argument passed to constar() and the cdr of l_result is rest of the elements of the newly created list (including l_list).

Arguments

[ g_arg1 ... ] Elements to be added to the beginning of l_list.

l_list The last argument that must be a list (which can be nil).

Value Returned

l_result List whose first element is the first argument and whose cdr is rest of the elements of the newly created list (including l_list).

Example

The first element of the newly created list is the first argument while cdr is rest of the elements (including l_list):

```
newList = constar( '(a b) "hello" 1 2.3 (x y) )
=> ((a b) ("hello") 1 2.3 x y z)
car( newList ) => (a b)
cdr( newList ) => (("hello") 1 2.3 x y z)
```
The last argument can be nil:

\texttt{constar( 1 2 3 nil )} => \texttt{(1 2 3)}

The last argument must be a list:

\texttt{constar( 'x 1 2 )}

*Error* \texttt{constar: the last arg must be a list} - 2

\texttt{constar( )} is cleaner and more efficient in adding multiple elements to the beginning of a list than \texttt{cons( )}:

\texttt{cons(1 cons(2 cons(3 '('a b c))))} => \texttt{(1 2 3 a b c)}
\texttt{constar( 1 2 3 '('a b c))} => \texttt{(1 2 3 a b c)}
copy

```lisp
(copy (l_arg)
  => l_result)
```

**Description**

Returns a copy of a list, that is, a list with all the top-level cells duplicated.

Because list structures in SKILL are typically shared, it is usually only necessary to pass around pointers to lists. If, however, any function that modifies a list destructively is used, `copy` is often used to create new copies of a list so that the original is not inadvertently modified by those functions. This call is costly so its use should be limited. This function only duplicates the top-level list cells, all lower level objects are still shared.

**Arguments**

`l_arg` List of elements.

**Value Returned**

`l_result` Returns a copy of `l_arg`.

**Example**

```lisp
z = '('1 2 3 4)' => (1 2 3 4)
x = copy(z) => (1 2 3 4)
equal(z x) => t
```

`z` and `x` have the same value.

```lisp
eq(z x) => nil
```

`z` and `x` are not the same list.
**copy_<name>**

```lisp
(copy_<name>(
  r_defstruct
)
  => r_defstruct
)
```

**Description**

Creates and returns a copy of a structure. This function is created by the `defstruct` function where `<name>` is the name of the defstruct.

Structures can contain instances of other structures; therefore you need to be careful about structure sharing. If sharing is not desired, use the `copyDefstructDeep` function to generate a copy of the structure and its sub-elements.

**Arguments**

- `r_defstruct` An instance of a defstruct.

**Value Returned**

- `r_defstruct` Copy of the given instance

**Example**

```lisp
defstruct(myStruct a b c) => t
m1 = make_myStruct(?a 3 ?b 2 ?c 1) => array[x]:xxxx
m2 = copy_myStruct(m1) => array[x]:xxxx
```

**Reference**

- `copyDefstructDeep`, `defstruct`, `defstructp`, `make_<name>`, `printstruct`
**copyDefstructDeep**

```scl
copyDefstructDeep(
    r_object
) => r_defstruct
```

**Description**

Performs a deep or recursive copy on defstructs with other defstructs as sub-elements, making copies of all the defstructs encountered.

The various `copy_<name>` functions are called to create copies for the various defstructs encountered in the deep copy.

**Note:** Only defstruct sub-elements are recursively copied. Other data types, like lists, are still shared.

**Arguments**

- `r_object` An instance of a defstruct.

**Value Returned**

- `r_defstruct` A deep copy of the given instance.

**Example**

```scl
defstruct(myStruct a b c) => t ; creates a function make_myStruct

m1 = make_myStruct(?a 3 ?b 2 ?c 1)
=> array[5]:3873024

m2 = make_myStruct(?a m1 ?b '(a b c) ?c 5)
=> array[5]:3873208 ; m1 is m2's sub-element

m3 = copyDefstructDeep(m2)
=> array[5]:3873056 ; uses deep copy

m3->a
=> array[5]:3873344 ; a new object

eq(m3->a m2->a) => nil ; eq checks object identity
```
m2->b
=> (a b c)

eq(m3->b m2->b)
=> t  ; still sharing the same object because
       ; the sub-element b is not a defstruct

m4 = copy_myStruct(m2)
=> array[5]:3873376  ; uses shallow copy

m4->a => array[5]:3873024
eq(m4->a m2->a) => t  ; share identical substructure
eq(m4->b m2->b) => t  ; the same object

Reference

copy <name>, defstruct, printstruct, defstructp
**COS**

\[
\cos(n_{\text{number}}) \Rightarrow f_{\text{result}}
\]

**Description**

Returns the cosine of a floating-point number or integer.

**Arguments**

- \(n_{\text{number}}\)  
  Floating-point number or integer.

**Value Returned**

- \(f_{\text{result}}\)  
  Cosine of \(n_{\text{number}}\).

**Example**

- \(\cos(0.3) \Rightarrow 0.9553365\)
- \(\cos(3.14/2) \Rightarrow 0.0007963\)

**Reference**

acos
cputime

cputime() => x_result

Description

Returns the total amount of CPU time (user plus system) used in units of 60ths of a second.

Value Returned

x_result CPU time in 60ths of a second.

Example

cputime() => 8
integerp( cputime() ) => t
floatp( cputime() ) => nil
createDir

createDir(
  S_name
)
  => t | nil

Description

Creates a directory.

The directory name can be specified with either an absolute or relative path; the SKILL path is used in the latter case. Note that a path which is anchored to current directory, for example, ./,../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name  Name of the directory you are creating.

Value Returned

t  If the directory is created.

nil  If the directory is not created because it already exists.

If the directory cannot be created because you do not have permission to update the parent directory, or a parent directory does not exist, an error is signaled.

Example

createDir("/usr/tmp/test") => t
createDir("/usr/tmp/test") => nil  ;Directory already exists.

Reference

deleteDir, isDir, isFile
csh

csh(
    [ t_command ]
) => t | nil

Description

Starts the UNIX C-shell as a child process to execute a command string.
Identical to the sh function, but invokes the C-shell (csh) rather than the Bourne-shell (sh).

Arguments

t_command Command string to execute.

Value Returned

t If the exit status of executing the given shell command is 0.

nil Otherwise.

Example

csh( "mkdir ~/tmp" ) => t

Creates a sub-directory called tmp in your home directory.

Reference

sh, shell
**declare**

```
declare(
    s_arrayName
    [ x_sizeOfArray ]
)
=> a_newArray
```

**Description**

Creates an array with a specified number of elements. This is a syntax form. All elements of the array are initialized to `unbound`.

**Arguments**

- `s_arrayName` Name of the array. There must be no white space between the name of an array and the opening bracket containing the size.
- `x_sizeOfArray` Size of the array as an integer.

**Value Returned**

- `a_newArray` Returns the new array.

**Example**

When the name of an array appears on the right side of an assignment statement, only a pointer to the array is used in the assignment; the values stored in the array are not copied. It is therefore possible for an array to be accessible by different names. Indices are used to specify elements of an array and always start with 0; that is, the first element of an array is element 0. SKILL checks for an out of bounds array index with each array access.

```
declare(a[10])
a[0] = 1
a[1] = 2.0
a[2] = a[0] + a[1]
```

Creates an array of 10 elements. `a` is the name of the array, with indices ranging from 0 to 9. Assigns the integer 1 to element 0, the float 2.0 to element 1, and the float 3.0 to element 2.

```
b = a
```

`b` now also refers to the same array as `a`. 

```
declare(c[10])
declares another array of 10 elements.
declare(d[2])
declares d as array of 2 elements.
d[0] = b
d[0] now refers to the array pointed to by b and a.
d[1] = c
d[1] is the array referred to by c.
d[0][2]
Accesses element 2 of the array referred to by d[0].
This is the same element as a[2].

Brackets ([ ]) are used in this instance to represent array references and are part of the statement syntax.

Reference
makeVector
**declareLambda**

```snippet
declareLambda(
   s_name1 ...
   s_nameN
)
=> s_nameN
```

**Description**

Tells the evaluator that certain (forward referenced) functions are of lambda type (as opposed to nlambda or macro).

Declares `s_name1 ... s_nameN` as procedures (lambdas) to be defined later. This is much like C’s “extern” declarations. Because the calling sequence for nlambdas is quite different from that of lambdas, the evaluator needs to know the function type in order to generate more efficient code. Without the declarations, the evaluator can still handle things properly, but with some performance penalty. The result of evaluating this form is the last name given (in addition to the side-effects to the evaluator).

This (and declareNLambda) form has effect only on undefined function names, otherwise it is ignored. Also, when the definition is provided later, if it is of a different function type (for example, declared as lambda but defined as nlambda) a warning will be given and the definition is used regardless of the declaration. In this case (definition is inconsistent with declaration), if there is any code already loaded that made forward references to these names, that part of code should be reloaded in order to use the correct calling sequence.

**Arguments**

- `s_name1` One or more function names.

**Value Returned**

- `s_nameN` The last name in the arguments.

**Example**

```
declareLambda(fun1 fun2 fun3) => fun3
```
Reference

`declareNLambda`
declareNLambda

```
declareNLambda(  
      s_name1 ...  
      s_nameN  
  )  
=> s_nameN
```

**Description**

Tells the evaluator that certain (forward referenced) functions are of nlambda type (as opposed to lambdas or macros).

Declares $s_{\text{name1}} \ldots s_{\text{nameN}}$ as nprocedures (nlambdas) to be defined later. This is much like C’s “extern” declarations. Because the calling sequence for nlambdas is quite different from that of lambdas, the evaluator needs to know the function type in order to generate more efficient code. Without the declarations, the evaluator can still handle things properly, but with some performance penalty. The result of evaluating this form is the last name given (in addition to the side-effects to the evaluator).

**Arguments**

$s_{\text{name1}}$ One or more function names.

**Value Returned**

$s_{\text{nameN}}$ The last name in the arguments.

**Example**

```
declareNLambda(nfun1 nfun2 nfun3) => nfun3
```

**Reference**

declareLambda
**declareSQNLambda**

```lisp
declareSQNLambda(
    s_functionName ...
)
=> nil
```

**Description**

Declares the given nlambda functions to be *solely-quoting nlambdas*.

This is an nlambda function. The named functions are defined as nlambdas only to save typing the explicit quotes to the arguments.

The compiler has been instructed to allow the calling of these kinds of nlambdas from SKILL++ code without giving a warning message.

All the debugging commands have been declared as SQNLambdas already.

**Arguments**

- `s_functionName` Function to be declared as a *solely-quoting nlambda*.

**Value Returned**

- `nil` Always. This function is for side-effects only.

**Example**

```lisp
declareSQNLambda( step next stepout ) => nil
```
**define - SKILL++ mode**

```plaintext
define(
   s_var
   g_expression
)
=> s_var

define(
   (s_var
   [ s_formalVar1 ... ]
   )
g_body ...
)
=> s_var
```

**Description**

`define`, *used in SKILL++ mode only*, is a syntax form used to provide a definition for a global or local variable. The `define` syntax form has two variations.

Definitions are allowed only at the top-level of a program and at the beginning of a body within the following syntax forms: `lambda`, `let`, `letrec`, and `letseq`. If occurring within a body, the `define`’s variable is local to the body.

- **Top Level Definitions**
  A definition occurring at the top level is equivalent to an assignment statement to a global variable.

- **Internal Definitions**
  A definition that occurs within the body of a syntax form establishes a local variable whose scope is the body.

- **define(`s_var g_expression`)**
  This is the primary variation. The other variation can be rewritten in this form. The expression is evaluated in enclosing lexical environment and the result is assigned or bound to the variable.

- **define(`( s_var [s_formalVar1 ...] ) g_body`)**
  In this variation, body is a sequence of one or more expressions optionally preceded by one or more nested definitions. This form is equivalent to the following define

```plaintext
define(s_var
       lambda(( [sformalVar1 ...] ) g_body ...)
```
Example

- **First variation**

  ```
  define( x 3 ) => x
  define( addTwoNumbers lambda( ( x y ) x+y ) ) => addTwoNumbers
  ```

- **Second variation**

  ```
  define( ( addTwoNumbers x y ) x+y ) => addTwoNumbers
  ```

- **Local definition using second variation**

  ```
  let( (( x 3 ))
      define( ( add y ) x+y ) ; define
      add( 5 )
    ) ; let
  => 8
  ```

  Defines a local function `add`, then invokes it.

  ```
  let( ()
      define( ( f n )
        if( n > 0 then n*f(n-1) else 1 ) ; if
        ) ; define
      f( 5 )
    ) ; let
  => 120
  ```

  Declares a single recursive local function `f` that computes the factorial of its argument. The `let` expression returns the factorial of 5.

Reference

- `lambda`, `let` - SKILL mode,
- `letrec` - SKILL++ mode,
- `letseq` - SKILL++ mode,
- `begin` - SKILL++ mode
defmacro

defmacro(
    s_macroName
    ( l_formalArglist )
    g_exprl ...
)
=> s_macroName

Description

Defines a macro which can take a list of formal arguments including @optional, @key, and @rest (instead of the more restrictive format as required by using mprocedure).

The actual arguments will be matched against the formals before evaluating the body.

Arguments

s_macroName       Name of the macro you are defining.
l_formalArglist   Formal argument list.
g_exprl           Expression or expressions to be evaluated.

Value Returned

s_macroName       Returns the name of the macro being defined.

Example

defmacro( whenNot (cond @rest body)
    '(if ! ,cond then ,@body) )
=> whenNot
expandMacro( '(whenNot x > y z = f(y) x*z) )
=> if(!x > y then (z = (f y))(x * z))
whenNot(1 > 2 "hello" 1+2)
=> 3
\[ \sqrt{2} \]

Reference

expandMacro, isMacro, mprocedure
defMathConstants

defMathConstants(
    s_id
)
=> s_id

Description

Associates a set of predefined math constants as properties of the given symbol.

Arguments

s_id Must be a symbol. The properties to be associated with the symbol are listed as name/value pairs. The names are explained in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>The base of natural logarithms. (e)</td>
</tr>
<tr>
<td>LOG2E</td>
<td>The base-2 logarithm of e</td>
</tr>
<tr>
<td>LOG10E</td>
<td>The base-10 logarithm of e</td>
</tr>
<tr>
<td>LN2</td>
<td>The natural logarithm of 2.</td>
</tr>
<tr>
<td>LN10</td>
<td>The natural logarithm of 10.</td>
</tr>
<tr>
<td>PI</td>
<td>The ratio of the circumference of a circle to its diameter. (( \pi ))</td>
</tr>
<tr>
<td>PI_OVER_2</td>
<td>( \pi /2 )</td>
</tr>
<tr>
<td>PI_OVER_4</td>
<td>( \pi /4 )</td>
</tr>
<tr>
<td>ONE_OVER_PI</td>
<td>1/( \pi )</td>
</tr>
<tr>
<td>TWO_OVER_PI</td>
<td>2/( \pi )</td>
</tr>
<tr>
<td>TWO_OVER_SQRTPI</td>
<td>2/( \sqrt{\pi} )</td>
</tr>
<tr>
<td>SQRT_TWO</td>
<td>( \sqrt{2} ) (The positive square root of 2.)</td>
</tr>
<tr>
<td>SQRT_POINT_FIVE</td>
<td>(The positive square root of 1/2.)</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>The maximum value of a SKILL integer.</td>
</tr>
<tr>
<td>INT_MIN</td>
<td>The minimum value of a SKILL integer.</td>
</tr>
</tbody>
</table>
Value Returned

\( s_id \) Returns the symbol ID.

Example

```python
def MathConstants('m) => m
m.?? => (SQRT_POINT_FIVE 0.7071068
SQRT_TWO 1.414214
TWO_OVER_SQRTPI 1.128379
TWO_OVER_PI 0.6366198
ONE_OVER_PI 0.3183099
PI_OVER_4 0.7853982
PI_OVER_2 1.570796
PI 3.141593
LN10 2.302585
LN2 0.6931472
LOG10E 0.4342945
LOG2E 1.442695
E 2.718282
DBL_MIN 2.225074e-308
DBL_MAX 1.797693e+308
INT_MIN -2147483648
INT_MAX 2147483647
SHRT_MIN -32768
SHRT_MAX 32767)
m.SQRT_POINT_FIVE => 0.7071068
m.INT_MIN => -2147483648
m.PI => 3.141593
printf("%0.17f\n" m.PI) => 3.14159265358979312
```

Reference

`printf, getqq, plist, setplist`
defprop

defprop(
    s_id
    g_value
    s_name
)
=> g_value

Description

Adds properties to symbols but none of its arguments are evaluated. This is a syntax form.

The same as putprop except that none of its arguments are evaluated.

Arguments

s_id  
Symbol to add property to.

g_value  
Value of the named property.

s_name  
Named property.

Value Returned

\(g\_value\)  
Value of the named property.

Example

defprop(s 3 x)  => 3

Sets property \(x\) on symbol \(s\) to 3.

defprop(s 1+2 x)  =>  (1+2)

Sets property \(x\) on symbol \(s\) to the unevaluated expression 1+2.

Reference

get, putprop
defstruct

defstruct(
    s_name
    s_slot1
[    [ s_slot2.. ]
)
=> t

Description

Creates a defstruct, a named structure that is a collection of one or more variables.

Defstructs can have slots of different types that are grouped together under a single name for handling purposes. They are the equivalent of structs in C. The defstruct form also creates an instantiation function, named make_<name> where <name> is the structure name supplied to defstruct. This constructor function takes keyword arguments: one for each slot in the structure. Once created, structures behave just like disembodied property lists.

Note: Just like disembodied property lists, structures can have new slots added at any time. However these dynamic slots are less efficient than the statically declared slots, both in access time and space utilization.

Structures can contain instances of other structures; therefore one needs to be careful about structure sharing. If sharing is not desired, a special copy function can be used to generate a copy of the structure being inserted. The defstruct form also creates a function for the given defstruct called copy_<name>. This function takes one argument, an instance of the defstruct. It creates and returns a copy of the given instance. An example appears after the description of the other defstruct functions.

Arguments

s_name A structure name.

s_slot1 Name of the first slot in structure s_name.

s_slot2 Name of the second slot in structure s_name.

Value Returned

t Always.
Example

defstruct(myStruct slot1 slot2 slot3) => t
struct = make_myStruct(?slot1 "one" ?slot2 "two"
?slot3 "three")
struct->slot1 => "one"

Returns the value associated with a slot of an instance.

struct->slot1 = "new" => "new"

Modifies the value associated with a slot of an instance.

struct->? => (slot3 slot2 slot1)

Returns a list of the slot names associated with an instance.

struct->?? => (slot3 "three" slot2 "two" slot1 "new")

Returns a property list (not a disembodied property list) containing the slot names and values associated with an instance.

Reference

defstructp, printstruct
defstructp

defstructp(
  g_object
  [ S_name ]
) => t | nil

Description

Checks if an object is an instance of a particular defstruct.

If the optional second argument is given, it is used as the defstruct name to check against. The suffix \texttt{p} is usually added to the name of a function to indicate that it is a predicate function.

Arguments

\texttt{g_object} \hspace{1cm} A data object.

\texttt{S_name} \hspace{1cm} Name of the structure to be tested for.

Value Returned

\texttt{t} \hspace{1cm} If \texttt{g_object} is an instance of defstruct \texttt{S_name}.

\texttt{nil} \hspace{1cm} Otherwise.

Example

defstruct(myStruct slot1 slot2 slot3)
  => t
struct = make_myStruct(\?slot1 "one" \?slot2 "two" \?slot3 "three")
  => array[5]:3555552
defstructp( "myDefstruct")
  => nil
defstructp(struct 'myStruct)
  => t

Reference

defstruct, printstruct
defun

```lisp
defun(
    s_funcName
    ( l_formalArglist )
    g_expr1 ...
)
=> s_funcName
```

Description

Defines a function with the name and formal argument list you specify. This is a syntax form.

The body of the procedure is a list of expressions to be evaluated one after another when `s_funcName` is called. There must be no white space between `defun` and the open parenthesis that follows.

However, for `defun` there must be white space between `s_funcName` and the open parenthesis. This is the only difference between the `defun` and `procedure` forms. `defun` has been provided principally so that you can make your code appear more like other LISP dialects.

Expressions within a function can reference any variable on the formal argument list or any global variable defined outside the function. If necessary, local variables can be declared using the `let` function.

Arguments

- `s_funcName` Name of the function you are defining.
- `l_formalArglist` Formal argument list.
- `g_expr1` Expression or expressions to be evaluated when `s_funcName` is called.

Value Returned

- `s_funcName` The name of the function being defined.
ARGUMENT LIST PARAMETERS

Several parameters provide flexibility in procedure argument lists. These parameters are referred to as @ (“at” sign) options. The parameters are @rest, @optional, and @key. See procedure for a detailed description of these argument list parameters.

Example

```
procedure( cube(x) x**3 ) ; Defines a function to compute the
debug cube  ; cube of a number using procedure.
cube( 3 ) => 27

defun( cube (x) x**3 ) ; Defines a function to compute the
=> cube  ; cube of a number using defun.
```

The following function computes the factorial of its positive integer argument by recursively calling itself.

```
procedure( factorial(x)
    if( (x == 0) then 1
    else x * factorial(x - 1))) => factorial

defun( factorial (x)
    if( (x == 0) then 1
      else x * factorial( x - 1))) => factorial
```

```
factorial( 6 ) => 720
```

Reference

procedure, let - SKILL mode, prog, nprocedure - SKILL mode only, nlambda - SKILL mode only
defUserInitProc

defUserInitProc(
    t_contextName
    s_procName
)
=> ( t_contextName s_procName )

Description

Registers a user-defined function that the system calls immediately after autoloading a context.

Lets you customize existing Cadence contexts. In the general case, most Cadence-supplied contexts have internally defined an initialization function through the defInitProc function. This function defines a second initialization function, called after the internal initialization function, thereby allowing you to customize on top of Cadence supplied contexts. This is best done in the .cdsinit file.

Arguments

\[ t\_contextName \]
Name of context file to load.

\[ s\_procName \]
Function to be called when context file is loaded.

Value Returned

\[(t\_contextName s\_procName)\]
Always returns an association list when set up. Note that the function is not actually called at this point, but is called when the \[t\_contextName\] context is loaded.

Example

defUserInitProc( "myContext" 'initMyContext)
=> ("myContext" initMyContext)

Reference

defInitProc, callInitProc
**defvar - SKILL mode only**

```scheme
defvar(
    s_varName
    [ g_value ]
)
=> g_value | nil
```

**Description**

Defines a global variable and assigns it a value. Use in SKILL mode only. Use the `define` syntax form to define global variables in SKILL++ mode.

**Arguments**

- `s_varName` Name of the variable to be defined.
- `g_value` Value to assign to the variable. If `g_value` is not given, `nil` is assigned to the variable.

**Value Returned**

- `g_value` If given.
- `nil` Otherwise.

**Example**

```scheme
defvar(x 3) => 3
```

Assigns `x` a value of 3.

**Reference**

- `defprop`, `set`, `setq`
deleteDir

defleteDir(
    S_name
)
    => t | nil

Description

Deletes a directory.

The directory name can be specified with either an absolute or relative path; the SKILL path is used in the latter case. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name Name of directory to delete.

Value Returned

t If the directory has been successfully deleted.

nil If the directory does not exist.

Signals an error if you do not have permission to delete a directory or the directory you want to delete is not empty.

Example

createDir("/usr/tmp/test") => t
deleteDir("/usr/tmp/test") => t
deleteDir("/usr/bin")

Signals an error about permission violation.

deleteDir("~")

Assuming there are some files in ~, signals an error that the directory is not empty.

Reference

createDir, deleteFile, isDir, isFile
deleteFile

deleeteFile(
    S_name
)
=> t | nil

Description

Deletes a file.

The file name can be specified with either an absolute or relative path; the SKILL path is used
in the latter case. If a symbolic link is passed in as the argument, it is the link itself, not the file
or directory referenced by the link, that gets removed. Note that a path which is anchored to
current directory, for example, ./././, or ././././, etc., is not considered as a relative
path.

Arguments

S_name  Name of file you want to delete.

Value Returned

t  File is successfully deleted.

nil  File does not exist.

Signals an error if you do not have permission to delete a file.

Example

deleteFile("~/test/out.1") => t

If the named file exists and is deleted.
deleteFile("~/test/out.2") => nil

If the named file does not exist.
deleteFile("/bin/ls")

If you do not have write permission for /bin, signals an error about permission violation.
Reference

delteDir, isFile, isDir
difference

difference(
  n_op1
  n_op2
  [ n_op3 ... ]
)
=> n_result

Description

Returns the result of subtracting one or more operands from the first operand. Prefix form of the – arithmetic operator.

Arguments

n_op1 Number from which the others are to be subtracted.
n_op2 Number to subtract.
n_op3 Optional additional numbers to subtract.

Value Returned

n_result Result of the operation.

Example

difference(5 4 3 2 1) => -5
difference(-12 13) => -25
difference(12.2 -13) => 25.2

Reference

xdifference
**display**

```lisp
display(
  g_obj
  [ p_port ]
)
=> t | nil
```

**Description**

Writes a representation of an object to the given port.

Strings that appear in the written representation are not enclosed in double quotes, and no characters are escaped within those strings.

**Arguments**

- `g_obj` Any SKILL object.
- `p_port` Optional output port. `p_port` is the default.

**Value Returned**

- `t` Usually ignored. Function is for side effects only.
- `nil` Usually ignored. Function is for side effects only.

**Example**

```lisp
(display "Hello!"
=> t
```

The side effect is to display `Hello!` to `p_port`.

**Reference**

`drain`, `print`, `write`
do - SKILL++ mode only

do(
  (   
    s_var1
    g_initExp1 
    [ g_stepExp1 ]   
  )  
  (   
    s_var2
    g_initExp2 
    [ g_stepExp2 ]   
  ) ...  
)  
( 
  g_terminationExp  
  g_terminationExp1 ... 
)   
  g_loopExp1  
  g_loopExp2 ...  
)  
=> g_value

Description

Iteratively executes one or more expressions. Used in SKILL++ mode only.

Use do to iteratively execute one or more expressions. The do expression provides a do-while facility allowing multiple loop variables with arbitrary variable initializations and step expressions. You can declare

- One or more loop variables, specifying for each variable both its initial value and how it gets updated each time around the loop.
- A termination condition which is evaluated before the body expressions are executed.
- One or more termination expressions to be evaluated upon termination to determine a return value.

A do Expression Evaluates in Two Phases

- Initialization phase
  
  The initialization expressions g_initExp1, g_initExp2, ... are evaluated in an unspecified order and the results bound to the local variables var1, var2, ...
Iteration phase

This phase is a sequence of steps, informally described as going around the loop zero or more times with the exit determined by the termination condition.

More formally stated:

1. Each iteration begins by evaluating the termination condition.

   If the termination condition evaluates to a non-nil value, the do expression exits with a return value computed as follows:

2. The termination expressions \( \text{terminationExp1, terminationExp2, ...} \) are evaluated in order. The value of the last termination condition is returned as the value of the do expression.

   Otherwise, the do expression continues with the next iteration as follows.

3. The loop body expressions \( g\_loopExp1, g\_loopExp2, ... \) are evaluated in order.

4. The step expressions \( g\_stepExp1, g\_stepExp2, ... \), if given, are evaluated in an unspecified order.

5. The local variables \( \text{var1, var2, ...} \) are bound to the above results. Reiterate from step one.

Example

By definition, the sum of the integers \( 1, ..., N \) is the \( N \)th triangular number. The following example finds the first triangular number greater than a given limit.

```.skill
procedure( trTriangularNumber( limit )
  do(
    ;;; start loop variables
    ( i 0 i+1 )
    ( sum 0 )
    ;;; no step expression
    ;;; same as ( sum 0 sum )
    ;;; end loop variables
    ( sum > limit ;; test
      sum
      ;;; return result
      )
    sum = sum+i ;;; body
    )
  )
)
```

trTriangularNumber( 4 ) => 6
trTriangularNumber( 5 ) => 6
trTriangularNumber( 6 ) => 10
Reference

for, while
**drain**

```
    drain(  
        [ p_outputPort ]  
    )  
    => t | nil
```

**Description**

Writes out all characters that are in the output buffer of a port.

Analogous to `fflush` in C (plus `fsync` if the port is a file). Not all systems guarantee that the disk is updated on each write. As a result, it is possible for a set of seemingly successful writes to actually fail when the port is closed.

To protect your data, call `drain` after a logical set of writes to a file port. It is not recommended that you call `drain` after every write however, because this could impact your program's performance.

**Arguments**

- `p_outputPort` Port to flush output from. If no argument is given this function does nothing.

**Value Returned**

- `t` If all buffered data was successfully written out.
- `nil` There was a problem writing out the data, and some or all of it was not successfully written out.

- Signals an error if the port to be drained is an input port or has been closed.

**Example**

```
drain() => t
drain(poport) => t
myPort = outfile("/tmp/myfile")
=> port:="/tmp/myfile"
for(i 0 15 fprintf(myPort "Test output%d\n" i))
=> t
```
system( "ls -l /tmp/myfile")
--rw-r--r-- 1 root 0 Aug12 14:44 /tmp/myFile
fileLength( "/tmp/myfile")
=> 0


drain(myPort)
=> t

fileLength( "/tmp/myfile" )
=> 230

close(myPort)
=> t

drain(myPort)
=> *Error* drain: cannot send output to a closed port - port:
    "/tmp/myfile"
drain(piport)
=> *Error* drain: cannot send output to an input port -
    port:"*stdin*"
drain(poport)
=> t

defun(handleWriteError (x)
    printf("WARNING - %L write unsuccessful\n" x) nil)
=> handleWriteError

myPort=outfile("/tmp/myfile")
=> port:"/tmp/myfile"

for(i 0 15 fprintf(myPort "%d\n" (2**i)))
=> t

if(!drain(myPort) handleWriteError(myPort) t)
=> t

Reference

outfile, close
dtpr

dtpr(
    g_value
)
=> t | nil

Description

Checks if an object is a non-empty list.

dtpr is a predicate function that is equivalent to pairp.

Arguments

$g\_value$  
An object.

Value Returned

t  
Object is a non-empty list.

nil  
Otherwise. Note that dtpr(nil) returns nil.

Example

dtpr( 1 ) => nil
dtpr( list(1)) => t

Reference

listp, null, pairp
ed

ed(
    [ t_fileName ]
) => t | nil

Description
Edits the named file.

Arguments

| t_fileName   | File to edit. If no argument is given, defaults to the previously edited file, or temp.il, if there is no previous file. |

Value Returned

| t | The operation was successfully completed. |
| nil | The file does not exist or there is an error condition. |

Reference
edi, edl, edit
edi

edi(
    [ t_fileName ]
)
=> t | nil

Description
Edits the named file, then includes the file into SKILL.

Arguments

\textit{t\_fileName} \hspace{1cm} \text{File to edit. If no argument is given, defaults to the previously edited file, or temp.il, if there is no previous file.}

Value Returned

t \hspace{1cm} \text{The operation was successfully completed.}
nil \hspace{1cm} \text{The file does not exist or there is an error condition.}

Example

edi( "~/myFile.il" )

Reference
\textit{ed, edit, edl}
**edit**

```lisp
edit(
    S_object
    [ g_loadFlag ]
)
=> x_childId
```

**Description**

Edits a file, function, or variable. This function only works if you are in graphical mode. This is an nlambda function.

*edit* brings up an editor window in a separate process and thus doesn’t lock up the CIW. If the object being edited is a function that was loaded after debug mode was turned on, then *edit* opens up the file that contains the function. If the editor is vi or emacs it jumps to the start of the function. If *g_loadFlag* is *t* the file is loaded into SKILL when the editor is exited. Be sure the *editor* variable is set up properly if you are using an editor other than vi or emacs.

**Arguments**

- **S_object**
  - If you are editing a file, the object you are editing must be a string. If you are editing a function or variable, it must be an unquoted symbol.

- **g_loadFlag**
  - Determines whether to load the file after the editor window is exited.
  - **Valid values:** *t* or *nil*
  - **Default:** *nil*.

**Value Returned**

- **x_childId**
  - Integer identifying the process spawned for the editor.

**Example**

```
edit( "~/cdsinit" )
```

Edits the *cdsinit* file in your home directory.

```
edit( myFun)
```
Edits the myFun function.

```
edit( myVar )
```

Edits the myVar variable and loads in the new value when the editor window is closed.

**Reference**

`ed`, `edl`, `edi`, `isFile`
edl

edl(  
    [ t_fileName ]  
)  
=> t | nil

Description
Edits the named file, then loads the file into SKILL.

Arguments

\textit{t\_fileName}  
File to edit. If no argument is given, defaults to the previously edited file, or \texttt{temp.il}, if there is no previous file.

Value Returned

t  
The operation was successfully completed.

\texttt{nil}  
The file does not exist or there is an error condition.

Example

edl( "/tmp/demo.il" )

Reference

\texttt{ed, edi, edit}


**envobj**

envobj(
    \(x_{id}\)
) => e_environment

**Description**

Returns the environment object whose print representation has the ID \(x_{id}\). You can consider \(x_{id}\) to be the address of the environment object.

**Arguments**

\(x_{id}\) The environment object’s ID.

**Value Returned**

e_environment Environment object specified by the given object ID. An error is signaled if the given object ID does not designate an environment object.

**Example**

\[E = \text{theEnvironment}() \Rightarrow \text{envobj:0xlad018}\]
\[; \text{only meaningful in SKILL++ mode}\]
\[\text{eq( envobj( 0xlad018 ) E ) } \Rightarrow \text{t}\]

This example retrieves the enclosing lexical environment and assigns it to a variable. Next extract the ID by inspection from the print representation, and pass it to the envobj function. Using the eq function demonstrates that return value is \(E\).

**Reference**

funobj, theEnvironment - SKILL++ mode only
eq

eq(g_arg1, g_arg2)
 => t | nil

Description

Checks addresses when testing for equality.

Returns \( t \) if \( g\_arg1 \) and \( g\_arg2 \) are exactly the same (that is, are at the same address in memory). The \( \text{eq} \) function runs considerably faster than \( \text{equal} \) but should only be used for testing equality of symbols or shared lists. Using \( \text{eq} \) on types other than symbols and lists will give unpredictable results and should be avoided.

For testing equality of numbers, strings, and lists in general, the \( \text{equal} \) function and not the \( \text{eq} \) function should be used. You can test for equality between symbols using \( \text{eq} \) more efficiently than using the \( \text{==} \) operator, which is the same as the \( \text{equal} \) function.

Arguments

\( g\_arg1 \)
Any SKILL object. \( g\_arg1 \) is compared with \( g\_arg2 \) to see if they point to the same object.

\( g\_arg2 \)
Any SKILL object.

Value Returned

\( t \)
Both arguments are the same object.

\( \text{nil} \)
The two objects are not identical.

Example

\( x = \text{'dog} \)
\( \text{eq}( x \text{'dog} ) \) \( \Rightarrow \text{t} \)
\( \text{eq}( x \text{'cat} ) \) \( \Rightarrow \text{nil} \)

\( y = \text{'dog} \)
\( \text{eq}( x \ y ) \) \( \Rightarrow \text{t} \)
Reference

equal
equal

equal(
  g_arg1
  g_arg2
)
  => t | nil

Description

Checks contents of strings and lists when testing for equality.

Checks if two arguments are equal or if they are logically equivalent, for example, $g_{arg1}$ and $g_{arg2}$ are equal if they are both lists/strings and their contents are the same. Note that this test is slower than using eq but works for comparing objects other than symbols.

- If the arguments are the same object in virtual memory (that is, they are eq), equal returns t.
- If the arguments are the same type and their contents are equal (for example, strings with identical character sequence), equal returns t.
- If the arguments are a mixture of fixnums and flonums, equal returns t if the numbers are identical (for example, 1.0 and 1).

Arguments

$g_{arg1}$ Any SKILL object. $g_{arg1}$ and $g_{arg2}$ are tested to see if they are logically equivalent.

$g_{arg2}$ Any SKILL object.

Value Returned

t If $g_{arg1}$ and $g_{arg2}$ are equal.

nil Otherwise.

Example

```plaintext
x = 'cat
equal(x 'cat ) => t
```
x == 'dog       => nil     ; == is the same as equal.

x = "world"
equal(x "world") => t

x = '(a b c)
equal(x '(a b c)) => t
equal(2 2.0)    => t

Reference

eq
eqv

eqv(
    g_obj1
    g_obj2
  ) => t | nil

Description
Tests for object identity or equality between two numbers of the same type (for example, both numbers are integers). Except for numbers, eqv is like eq.

Arguments

  g_obj1    Any SKILL object.
  g_obj2    Any SKILL object.

Value Returned

t  g_obj1 and g_obj2 are the same object or the same number.

nil  Otherwise.

Example

(eqv 1.5 1.5) => t
(equal 1.5 1.5) => t
(eq 1.5 1.5) => nil
(eqv (list 1 2) (list 1 2)) => nil

Reference

  eq, equal
err

err([ g_value ])
=> none

Description

Causes an error.

If this error is caught by an errset, nil is returned by that errset. However, if the optional g_value argument is given then g_value is returned from the errset and can be used to identify which err signaled the error. The err function never returns a value.

Arguments

g_value SKILL object that becomes the return value for errset.

Value Returned

Never returns a value.

Example

errset(err('ErrorType)) => (ErrorType)
errset.errset => nil

procedure(test(x))
  if(equal errset(foo(x)) 'throw')
    println( "Throw caught" )
  else if(errset.errset println( "Error: divide by zero")))=> test

procedure(foo(x))
  if(equal (4 / x) 1)
    then err('throw')
  else println(x)) => foo

  test(4) => nil ; Prints Throw caught
  test(2) => nil ; Prints 2
  test(0) => nil ; Prints Error: divide by zero
Reference

errset, error
error

error(
  [ S_message1
  [ S_message2 ] ... ]
)
=> none

Description

Prints error messages and calls err.

Prints the $S_message1$ and $S_message2$ error messages if they are given and then calls err, causing an error. The first argument can be a format string, which causes the rest of the arguments to be printed in that format.

Arguments

$S_message1$  
Message string or symbol.

$S_message2$  
More message strings or symbols. Note that more than two arguments should be given only if the first argument is a format string.

Value Returned

Prints the $S_message1$ and $S_message2$ error messages if they are given and then calls err, causing an error. error never returns.

Example

error( "myFunc" "Bad List"")

Prints *Error* myFunc: Bad List

error( "bad args - %s %d %L" "name" 100 '(1 2 3) )

Prints *Error* bad args - name 100 (1 2 3)

errset( error( "test" ) t) => nil

Prints out *Error* test and returns nil.
SKILL Language Reference
SKILL Language Functions

Reference

err, errset
**errset**

```
errset(
    g_expr
    [ g_errprint ]
) => l_result | nil
```

**Description**

Encapsulates the execution of an expression in an environment safe from the error mechanism. This is a syntax form.

If an error occurs in the evaluation of the given expression, control always returns to the command following the `errset` instead of returning to the nearest toplevel. If `g_errprint` is non-nil, error messages are issued; otherwise they are suppressed. In either case, information about the error is placed in the `errset` property of the `errset` symbol. Programs can therefore access this information with the `errset.errset` construct after determining that `errset` returned nil.

**Arguments**

- **g_expr**
  - Expression to be evaluated; while evaluating it, any errors cause immediate return from the `errset`.

- **g_errprint**
  - Flag to control the printout of error messages. If t then prints the error message encountered in `errset`, defaults to nil.

**Value Returned**

- **l_result**
  - List with value from successful evaluation of `g_expr`.

- **nil**
  - An error occurred.

**Example**

```
errset(1+2) => (3)
errset.errset => nil
errset(sqrt('x)) => nil
```

Because `sqrt` requires a numerical argument.
errset.errset => ("sqrt" 0 t nil ("*Error* sqrt: can’t handle sqrt(x)...))

Reference

err, error
errsetstring

errsetstring(
    t_string
    [ g_errprint ]
    [ s_langMode ]
)
=> l_value | nil

Description

Reads and evaluates an expression stored in a string. Same as evalstring except that it calls errset to catch any errors that might occur during the parsing and evaluation.

If an error has occurred, nil is returned, otherwise a list containing the value of the evaluation is returned. Should an error occur, it is stored in errset.errset. If errprint is non-nil, error messages are printed out; otherwise they are suppressed.

Arguments

- **t_string**: String to be evaluated.
- **g_errprint**: Flag for controlling the printout of error messages. If t, then prints the error message encountered in errset. Defaults to nil.
- **s_langMode**: Must be a symbol. Valid values:
  - 'ils: Evaluates the given string in SKILL++ mode.
  - 'il: Evaluates the given string in SKILL mode. This is the default.

Value Returned

- **l_value**: List with the value from successful evaluation of t_string.
- **nil**: An error occurs.
**Example**

```
errsetstring("1+2")  => (3)
errsetstring("1+'a")  => nil
```

Returns *nil* because an error occurred.
```
errsetstring("1+'a" t)  => nil
```

Prints out error message:

*Error* plus: can't handle (1+a)...

**Reference**

`err, error, errset, evalstring`
eval

```skill
eval(g_expression [ e_environment ])
=> g_result
```

Description

Evaluates an argument and returns its value. If an environment argument is given, `g_expression` is treated as SKILL++ code, and the expression is evaluated in the given (lexical) environment. Otherwise `g_expression` is treated as SKILL code.

This function gives you control over evaluation. If the optional second argument is not supplied, it takes `g_expression` as SKILL code. If an environment argument is given, it treats `g_expression` as SKILL++ code, and evaluates it in the given (lexical) environment.

For SKILL++'s `eval`, if the given environment is not the top-level one, the effect is like evaluating `g_expression` within a `let` construct for the bindings in the given environment, with the following exception:

If `g_expression` is a definitional form (such as `(define ... )`), it is treated as a global definition instead of local one. Therefore any variables defined will still exist after executing the `eval` form.

Arguments

- `g_expression` Any SKILL expression.
- `e_environment` If this argument is given, SKILL++ semantics is assumed. The forms entered will be evaluated within the given (lexical) environment.

Value Returned

- `g_result` Result of evaluating `g_expression`.

Example

```skill
eval( 'plus( 2 3 )') => 5
```
Evaluates the expression `plus(2 3)`.

\[
\begin{align*}
x &= 5 & \Rightarrow & & 5 \\
\text{eval( 'x') } & \Rightarrow & & 5
\end{align*}
\]

Evaluates the symbol `x` and returns the value of symbol `x`.

\[
\begin{align*}
\text{eval( list( 'max 2 1' ) ) } & \Rightarrow & & 2 \\
\text{Evaluates the expression max(2 1).}
\end{align*}
\]

**Reference**

`evalstring`, `funcall`
evalstring

evalstring(
  t_string
  [ s_langMode ]
)
=> g_value | nil

Description

Reads and evaluates an expression stored in a string.

The resulting value is returned. Notice that evalstring does not allow the outermost set of parentheses to be omitted from the evaluated expression, as in load or in the top level.

Arguments

<table>
<thead>
<tr>
<th>t_string</th>
<th>String containing the SKILL expression to be evaluated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_langMode</td>
<td>Must be a symbol.</td>
</tr>
</tbody>
</table>

Valid values:

| 'ils                  | Evaluates the given string in SKILL++ mode.            |
| 'il                   | Evaluates the given string in SKILL mode. This is the default. |

Value Returned

| g_value               | The value of the argument expression after evaluation. |
| nil                   | No form is read.                                       |

Example

evalstring("1+2") => 3

The 1+2 infix notation is the same as (plus 1 2).

evalstring("cons('a '(b c))") => (a b c)
car '(1 2 3) => 1
evalstring("car '(1 2 3)")

Signals that car is an unbound variable.
Reference

eval
evenp

```lisp
evenp(
    x_num
)
=> t  |  nil
```

**Description**

Checks if a number is an even integer.

**Arguments**

- **x_num**
  - Number to check.

**Value Returned**

- **t**
  - If `x_num` is an even integer.
- **nil**
  - Otherwise.

**Example**

```lisp
evenp( 59 )
=> nil
```

```lisp
evenp( 60 )
=> t
```

```lisp
evenp( 2.0 )
=> nil
      ; Number is even, but not an integer.
```

**Reference**

- minusp
- oddp
- onep
- plusp
- zerop
exists

exists(
  s_formalVar
  l_valueList
  g_predicateExpr
)
=> g_result

exists(
  s_key
  o_table
  g PredicateExpr
)
=> t | nil

Description

Returns the first tail of l_valueList whose car satisfies a predicate expression. Also verifies whether an entry in an association table satisfies a predicate expression. This is a syntax form.

This process continues to apply the cdr function successively through l_valueList until it finds a list element that causes g_predicateExpr to evaluate to non-nil. It then returns the tail that contains that list element as its first element.

This function can also be used to verify whether an entry in an association table satisfies g_predicateExpr.

Arguments

s_formalVar Local variable that is usually referenced in g_predicateExpr.

l_valueList List of elements that are bound to s_formalVar, one at a time.

g_predicateExpr SKILL expression that usually uses the value of s_formalVar.

s_key Key portion of an association table entry.

o_table Association table containing the entries to be processed.
Value Returned

\( g_{\text{result}} \)

First tail of \( l_{\text{valueList}} \) whose \text{car} satisfies \( g_{\text{predicateExpr}} \).

nil

If none of the elements in \( l_{\text{valueList}} \) can satisfy it.

t

Entry in an association table satisfies \( g_{\text{predicateExpr}} \).

Example

\[
\text{exists( x '}(1 \ 2 \ 3 \ 4) \ (x > 1) ) \ \Rightarrow (2 \ 3 \ 4) \\
\text{exists( x '}(1 \ 2 \ 3 \ 4) \ (x > 4) ) \ \Rightarrow \text{nil}
\]

\[
\text{exists( key myTable (and (stringp key) (stringp myTable[key]))))} \\
\Rightarrow \text{t}
\]

Tests an association table and verifies the existence of an entry where both the key and its corresponding value are of type \text{string}.

Reference

\text{car, cdr, forall}
exit

exit(
    [ x_status ]
) => nil

**Description**

Causes SKILL to exit with a given process status (defaults to 0), whether in interactive or batch mode.

Use `exit` functions to customize the behavior of an exit call. Sometimes you might like to do certain cleanup actions before exiting SKILL. You can do this by registering `exit-before` and/or `exit-after` functions.

An `exit-before` function is called before `exit` does anything, and an `exit-after` function is called after `exit` has performed its bookkeeping tasks and just before it returns control to the operating system. The user-defined exit functions do not take any arguments.

To give you even more control, an `exit-before` function can return the atom `ignoreExit` to abort the exit call totally. When `exit` is called, first all the registered `exit-before` functions are called in the reverse order of their registration. If any of them returns the special atom `ignoreExit`, the exit request is aborted and it returns `nil` to the caller.

After the `exit-before` functions are called:

1. Some bookkeeping tasks are called.
2. All the registered `exit-after` functions are called in the reverse order of their registration.
3. Finally the process exits to the operating system.

For compatibility with earlier versions of SKILL, you can still define the functions named `exitbefore` and `exitafter` as one of the exit functions. They are treated as the first registered exit functions (the last to be called). To avoid confusing the system setup, do not use these names for other purposes.

**Arguments**

- `x_status` Process exit status; defaults to 0.
Value Returned

nil

The exit request is aborted. Otherwise there is no return value because the process exits.

Example

(defun myExitBefore ()
  (if (closeMyDataBase)
      t ; if OK in closeMyDataBae then exit
    'ignoreExit)) ; otherwise we want to abort exit
regExitBefore('myExitBefore)
=> t ; exit function is registered
exit()

Depending on the result from calling closeMyDataBase, the system either exits the application (after asking for confirmation if running in graphic mode) or aborts the exit and returns nil.

Reference

regExitBefore, regExitAfter
**exp**

```
exp(
    n_number
)
=> f_result
```

**Description**

Raises e to a given power.

**Arguments**

```
n_number  Power to raise e to.
```

**Value Returned**

```
f_result  Value of e raised to the n_numberth power.
```

**Example**

```
exp( 1 ) => 2.718282
exp( 3.0) => 20.08554
```

**Reference**

acos, asin, atan, cos, log, sin, tan
**expandMacro**

```lisp
expandMacro(
    g_form
) => g_expandedForm
```

**Description**

Expands one level of macro call for a form.

Checks if the given form `g_form` is a macro call and returns the expanded form if it is. Otherwise it returns the original argument. The macro expansion is done only once (one level). That is, if the expanded form is another macro call, it is not further expanded (unless another `expandMacro` is called with the expanded form as its argument).

**Arguments**

`g_form` Form that can be a macro call.

**Value Returned**

`g_expandedForm` Expanded form or the original form if the given argument is not a macro call.

**Example**

```lisp
mprocedure( testMsg(args)
    '(printf "test %s -- %L\n",(cadr args)progn,@(cddr args)))
) => testMsg
expandMacro( '(testMsg "alpha1" y = f(x) g(y 100)) )
=> printf("test %s -- %L\n" "alpha1"
    progn((y = (f x)) (g y 100)))
```

**Reference**

`mprocedure`, `defmacro`
**expt**

```lisp
expt(
    n_base
    n_power
)
=> n_result
```

**Description**

Returns the result of raising a base number to a power. Prefix form of the ** exponentiation operator.

**Arguments**

- `n_base` Number to be raised to a power.
- `n_power` Power to which the number is raised.

**Value Returned**

- `n_result` Result of the operation.

**Example**

```
expt(2 3) => 8
expt(-2 3) => -8
expt(3.3 2) => 10.89
```
fboundp

```lisp
fboundp(
    s_functionName
) => t | nil
```

Description

Returns true (that is, some non-nil value) if the given name has a function binding.

This function returns a non-nil (that is, true) value if the given name has a function binding and returns nil otherwise. Note that `fboundp` examines the current function binding only and does not check for any potential definitions from autoloading. `fboundp` can be considered as an alias to `getd`.

Arguments

- `s_functionName`: Name to check for function binding.

Value Returned

- `t`: If there is a function binding for the given name.
- `nil`: If no function binding exists currently for the name.

Example

```lisp
fboundp( 'xyz ) => nil ;assuming there is no function named xyz
fboundp( 'defstruct) => funobj:0x261108 ;a non-nil result
```

Reference

- `getd`
fileLength

fileLength(
    $S_name$
)
=> $x_size$ | 0

Description

Determines the number of bytes in a file.

A directory is viewed just as a file in this case. Uses the current SKILL path if a relative path is given. Note that a path which is anchored to current directory, for example,. /, . /, or . /.

Arguments

$S_name$ Name of the file you want the size of.

Value Returned

$x_size$ Number of bytes in the $S_name$ file.

0 The file exists but is empty. Signals an error if the named file does not exist.

Example

fileLength("/tmp") => 1024

Return value is system-dependent.
fileLength("~/test/out.1") => 32157

Assuming the named file exists and is 32157 bytes long.

Reference

isDir,.isFile, isFileName
**fileSeek**

```plaintext
fileSeek(  
    p_port  
    x_offset  
    x_whence  
  )  
  => t | nil
```

**Description**

Sets the position for the next operation to be performed on the file opened on a port. The position is specified in bytes.

**Arguments**

- **p_port**: Port associated with the file.
- **x_offset**: Number of bytes to move forward (or backward with negative argument).
- **x_whence**: Valid Values:
  - 0  Offset from the beginning of the file.
  - 1  Offset from current position of file pointer.
  - 2  Offset from the end of the file.

**Value Returned**

- **t**: The operation was successfully completed.
- **nil**: The file does not exist or the position given is out of range for an input file.

**Example**

Let the file `test.data` contain the single line of text:

```
0123456789 test xyz
```

```plaintext
p = infile("test.data")  => port:"test.data"
fileTell(p)  => 0
for(i 1 10 getc(p))  => t  ; Skip first 10 characters
```
fileTell(p) => 10
fscanf(p "%s" s) => 1 ; s = "test" now
fileTell(p) => 15

fileSeek(p 0 0) => t
fscanf(p "%d" x) => 1 ; x = 123456789 now
fileSeek(p 6 1) => t
fscanf(p "%s" s) => 1 ; s = "xyz" now

Reference

fileTell, isDir, isFile, isFileName
fileTell

fileTell(
    p_port
)
=> x_offset

Description

Returns the current offset in bytes for the file opened on a port.

Arguments

p_port Port associated with the file.

Value Returned

x_offset Current offset (from the beginning of the file) in bytes for the file opened on p_port.

Example

Let the file test.data contain the single line of text:

0123456789 test xyz

p = inFile("test.data") => port:"test.data"
fileTell(p) => 0
for(i 1 10 getc(p)) => t ; Skip first 10 characters
fileTell(p) => 10
fscanf(p "%s" s) => 1 ; s = "test" now
fileTell(p) => 15

Reference

inFile, isFile, fileSeek, outfile
fileTimeModified

fileTimeModified(  
  t_filename  
)  
=> x_time | nil

Description

Gets the time a given file was last modified.

The return value is an internal, numeric, representation of the time the named file was last modified (for example, the number of 1/100 seconds from January 1, 1970). The actual number, which is system-dependent, is derived from the underlying UNIX system.

Arguments

\[ t\_filename \]

Name of a file.

Value Returned

\[ x\_time \]

Last time \( t\_filename \) was modified.

\[ \text{nil} \]

No file with the given name was found.

Example

fileTimeModified( "~/.cshrc" )  
=> 787435470

Reference

getCurrentTime, timeToString, timeToTm
**fix**

```plaintext
fix(
    n_arg
)
=> x_result
```

**Description**

Returns the largest integer not larger than the given argument.

This function is equivalent to floor. See also “Type Conversion Functions (fix and float)” in the SKILL Language User Guide.

**Arguments**

- `n_arg`: Any number.

**Value Returned**

- `x_result`: The largest integer not greater than `n_arg`. If an integer is given as an argument, it returns the argument.

**Example**

```plaintext
fix(1.9)  => 1
fix(-5.6) => -6
fix(100)  => 100
```

**Reference**

`ceiling, fixp, floor, round`
**fixp**

```lisp
fixp(
    g_value
  ) => t | nil
```

**Description**

Checks if an object is an integer, that is, a fixed number.

The suffix `p` is usually added to the name of a function to indicate that it is a predicate function. This function is equivalent to `integerp`.

**Arguments**

- `g_value` Any SKILL object.

**Value Returned**

- `t` If `g_value` is an integer, a data type whose internal name is `fixnum`.
- `nil` If `g_value` is not an integer.

**Example**

```lisp
fixp(3) => t
fixp(3.0) => nil
```

**Reference**

- `fix`, `float`, `floatp`, `integerp`
float

float(
    n_arg
)
=> f_result

Description

Converts a number into its equivalent floating-point number.

Arguments

n_arg

Integer to be converted to floating-point. If you give a floating-point number as an argument, it returns the argument unchanged.

Value Returned

f_result

A floating-point number.

Example

float(3)  => 3.0  
float(1.2) => 1.2

Reference

fix, fixp, floatp
**floatp**

```plaintext
floatp(
    g_value
)
  => t | nil
```

**Description**

Checks if an object is a floating-point number. Same as `realp`.

The suffix `p` is usually added to the name of a function to indicate that it is a predicate function.

**Arguments**

- `g_value`: Any SKILL object.

**Value Returned**

- `t`: If `g_value` is a floating-point number, a data type whose internal name is `flonum`.
- `nil`: If `g_value` is not a floating-point number.

**Example**

- `floatp(3)` => `nil`
- `floatp(3.0)` => `t`

**Reference**

`fix, fixp, float, realp`
floor

floor(
    n_number
) => x_integer

Description

Returns the largest integer not larger than the given argument.

Arguments

n_number  Any number.

Value Returned

x_integer  Largest integer not larger than n_number.

Example

(floor -4.3) => -5
(floor 3.5)  => 3

Reference

ceiling, fix, round, truncate
for

```skill
for(s_loopVar
    x_initialValue
    x_finalValue
    g_expr1
    [ g_expr2 ... ]
) => t
```

**Description**

Evaluates the sequence `g_expr1, g_expr2 ...` for each loop variable value, beginning with `x_initialValue` and ending with `x_finalValue`. This is a syntax form.

First evaluates the initial and final values, which set the initial value and final limit for the local loop variable named `s_loopVar`. Both `x_initialValue` and `x_finalValue` must be integer expressions. During each iteration, the sequence of expressions `g_expr1, g_expr2 ...` is evaluated and the loop variable is then incremented by one. If the loop variable is still less than or equal to the final limit, another iteration is performed. The loop terminates when the loop variable reaches a value greater than the limit. The loop variable must not be changed inside the loop. It is local to the `for` loop and would not retain any meaningful value upon exit from the `for` loop.

**Arguments**

- `s_loopVar` Name of the local loop variable that must not be changed inside the loop.
- `x_initialValue` Integer expression setting the initial value for the local loop variable.
- `x_finalValue` Integer expression giving final limit value for the loop.
- `g_expr1` Expression to evaluate inside loop.
- `g_expr2` Additional expression(s) to evaluate inside loop.

**Value Returned**

- `t` This construct always returns `t`. 
Example

```plaintext
sum = 0
for( i 1 10
  sum = sum + i
  printf("%d\n" sum))
=> t ; Prints 10 numbers and returns t.
```

Reference

`foreach`
forall

forall(
    s_formalVar
    l_valueList
    g_predicateExpr )
=> t / nil

forall( 
    s_key
    o_table
    g_predicateExpr )
=> t | nil

Description

Checks if \( g\_predicateExpr \) evaluates to non-nil for every element in \( l\_valueList \). This is a syntax form.

Verifies that an expression remains true for every element in a list. The \texttt{forall} function can also be used to verify that an expression remains true for every key/value pair in an association table. The syntax for association table processing is provided in the second syntax statement.

Arguments

\begin{align*}
\texttt{s\_formalVar} & \quad \text{Local variable usually referenced in } g\_predicateExpr. \\
\texttt{l\_valueList} & \quad \text{List of elements that are bound to } s\_formalVar \text{ one at a time.} \\
\texttt{g\_predicateExpr} & \quad \text{A SKILL expression that usually uses the value of } s\_formalVar. \\
\texttt{s\_key} & \quad \text{Key portion of the table entry.} \\
\texttt{o\_table} & \quad \text{Association table containing the entries to be processed.}
\end{align*}

Value Returned

\begin{align*}
\texttt{t} & \quad \text{If } g\_predicateExpr \text{ evaluates to non-nil for every element in } l\_valueList \text{ or for every key in an association table.}
\end{align*}
nil Otherwise.

Example

forall( x '1 2 3 4) (x > 0) )=> t
forall( x '1 2 3 4) (x < 4) )=> nil
forall(key myTable (and (stringp key)(stringp myTable[key])))
=> t

Returns t if each key and its value in the association table are of the type string.

Reference

exists
**foreach**

foreach(
    s_formalVar
g_exprList
g_expr1
    [ g_expr2 ... ]
)
=> l_valueList / l_result

foreach(
    (s_formalVar1...
    s_formalVarN)
g_exprList1...
g_exprListN
g_expr1
    [ g_expr2 ... ]
)
=> l_valueList / l_result

foreach(
    s_formalVar
g_exprTable
g_expr1
    [ g_expr2 ... ]
)
=> o_valueTable | l_result

**Description**

Evaluates one or more expressions for each element of a list of values. This is a syntax form.

foreach( s_formalVar g_exprList g_expr1 [ g_expr2 ... ] )
=> l_valueList / l_result

The first syntax form evaluates g_exprList, which returns a list l_valueList. It then assigns the first element from l_valueList to the formal variable s_formalVar and executes the expressions g_expr1, g_expr2 ... in sequence. The function then assigns the second element from l_valueList and repeats the process until l_valueList is exhausted.

foreach( (s_formalVar1...s_formalVarN) g_exprList1... g_exprListN g_expr1 [ g_expr2 ... ] )
=> l_valueList / l_result

The second syntax form of foreach can iterate over multiple lists to perform vector operations. Instead of a single formal variable, the first argument is a list of formal variables.
followed by a corresponding number of expressions for value lists and the expressions to be evaluated.

\[
\text{foreach}( s\_\text{formalVar} \ g\_\text{exprTable} \ g\_\text{expr1} [ \ g\_\text{expr2} \ldots ]) \\
=> o\_\text{valueTable} / l\_\text{result}
\]

The third syntax form of \text{foreach} can be used to process the elements of an association table. In this case, \text{s\_formalVar} is assigned each key of the association table one by one, and the body expressions are evaluated each iteration. The syntax for association table processing is provided in this syntax statement.

**Arguments**

\begin{align*}
\text{s\_formalVar} & \quad \text{Name of the variable.} \\
\text{s\_mappingFunction} & \quad \text{One of map, mapc, mapcan, mapcar, or maplist.} \\
\text{g\_exprList} & \quad \text{Expression whose value is a list of elements to assign to the formal variable \text{s\_formalVar}.} \\
\text{g\_expr1, g\_expr2} & \quad \text{Expressions to execute.} \\
\text{g\_exprTable} & \quad \text{Association table whose elements are to be processed.}
\end{align*}

**Value Returned**

\begin{align*}
\text{l\_valueList} & \quad \text{Value of the second argument, \text{g\_exprList}.} \\
\text{l\_result} & \quad \text{The result of the last expression evaluated.} \\
\text{o\_valueTable} & \quad \text{Value of \text{g\_exprTable}.}
\end{align*}

**Example**

\[
\text{foreach( x ' (1 2 3 4) println(x))} \\
1 ; \text{Prints the numbers 1 through 4.} \\
2 \\
3 \\
4 \\
=> (1 2 3 4) ; \text{Returns the second argument to \text{foreach}.}
\]

The next example shows \text{foreach} accessing an association table and printing each key and its associated data.

\[
\text{foreach(key myTable printf("%L : %L\n" key myTable[key]))}
\]
Example with more than one loop variable:

```lisp
(foreach (x y) '(1 2 3) '(4 5 6) (println x+y))
5
7
9
=> (1 2 3)
```

### Reference

- `mapc`, `mapcar`, `mapcan`, `forall`, `case`, `caseq`

### Errors and Warnings

The error messages from `foreach` might at times appear cryptic because some `foreach` forms get expanded to call the mapping functions `mapc`, `mapcar`, `mapcan`, and so forth.

### Advanced Usage

The `foreach` function typically expands to call `mapc`; however, you can also request that a specific mapping function be applied by giving the name of the mapping function as the first argument to `foreach`. Thus, `foreach` can be used as an extremely powerful tool to construct new lists.

**Note:** Mapping functions are not accepted when this form is applied to association tables.

```lisp
foreach( mapcar x '(1 2 3) (x >1))=> (nil t t)
foreach( mapcan x '(1 2 3) if((x > 1) ncons(x))) => (2 3)
foreach( maplist x '(1 2 3) length(x)) => (3 2 1)
```
**fprintf**

```plaintext
fprintf(
    p_port
    t_formatString
    [ g_arg1 ... ]
)
=> t
```

**Description**

Writes formatted output to a port.

The `fprintf` function writes formatted output to the port given as the first argument. The optional arguments following the format string are printed according to their corresponding format specifications.

`printf` is identical to `fprintf` except that it does not take the `p_port` argument and the output is written to `poport`.

Output is right justified within a field by default unless an optional minus sign “-” immediately follows the `%` character, which will then be left justified. To print a percent sign, you must use two percent signs in succession. You must explicitly put \n in your format string to print a newline character and \t for a tab.

**Common Output Format Specifications**

<table>
<thead>
<tr>
<th>Format Specification</th>
<th>Type(s) of Argument</th>
<th>Prints</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>fixnum</td>
<td>Integer in decimal radix</td>
</tr>
<tr>
<td>%o</td>
<td>fixnum</td>
<td>Integer in octal</td>
</tr>
<tr>
<td>%x</td>
<td>fixnum</td>
<td>Integer in hexadecimal</td>
</tr>
<tr>
<td>%f</td>
<td>flonum</td>
<td>Floating-point number in the style [-]ddd.ddd</td>
</tr>
<tr>
<td>%e</td>
<td>flonum</td>
<td>Floating-point number in the style [-]d.ddde[-]ddd</td>
</tr>
<tr>
<td>%g</td>
<td>flonum</td>
<td>Floating-point number in style f or e, whichever gives full precision in minimum space</td>
</tr>
<tr>
<td>%s</td>
<td>string, symbol</td>
<td>Prints out a string (without quotes) or the print name of a symbol</td>
</tr>
</tbody>
</table>
The \texttt{t\_formatString} argument is a conversion control string containing directives listed in the table above. The \%L, \%P, and \%B directives ignore the width and precision fields.

\begin{verbatim}
%[-][width][.precision]conversion\_code
[-] = left justify
[width] = minimum number of character positions
[.precision] = number of characters to be printed
conversion\_code
\end{verbatim}

**Arguments**

- \texttt{p\_port} \hspace{1cm} Output port to write to.
- \texttt{t\_formatString} \hspace{1cm} Characters to be printed verbatim, intermixed with format specifications prefixed by the \% sign.
- \texttt{g\_arg1} \hspace{1cm} The arguments following the format string are printed according to their corresponding format specifications.

**Value Returned**

\texttt{t} \hspace{1cm} Prints the formatted output and returns \texttt{t}.

**Example**

```skill
p = outfile("power.out") => port:"power.out"
for(i 0 15 fprintf(p "%20d %-20d\n" 2**i 3**i)) => t
close( p)
```
At this point the `power.out` file has the following contents.

```
1 1
2 3
4 9
8 27
16 81
32 243
64 729
128 2187
256 6561
512 19683
1024 59049
2048 177147
4096 531441
8192 1594323
16384 4782969
32768 14348907
```

**Reference**

`close`, `fscanf`, `scanf`, `sscanf`, `outfile`, `printf`
fscanf, scanf, sscanf

fscanf(
    p_inputPort
    t_formatString
    [ s_var1 ... ]
)
=> x_items | nil

scanf(
    t_formatString
    [ s_var1 ... ]
)
=> x_items | nil

sscanf(
    t_sourceString
    t_formatString
    [ s_var1 ... ]
)
=> x_items | nil

**Description**

The only difference between these functions is the source of input. **fscanf** reads input from a port according to format specifications and returns the number of items read in. **scanf** takes its input from **piport** implicitly. **scanf** only works in standalone SKILL when the **piport** is not the CIW. **sscanf** reads its input from a string instead of a port.

The results are stored into corresponding variables in the call. The **fscanf** function can be considered the inverse function of the **fprintf** output function. The **fscanf** function returns the number of input items it successfully matched with its format string. It returns **nil** if it encounters an end of file.

The maximum size of any input string being read as a string variable for **fscanf** is currently limited to 8K. Also, the function **lineread** is a faster alternative to **fscanf** for reading SKILL objects.

If an error is found while scanning for input, only those variables read before the error will be assigned.
The common input formats accepted by \texttt{fscanf} are summarized below.

### Common Input Format Specifications

<table>
<thead>
<tr>
<th>Format Specification</th>
<th>Type(s) of Argument</th>
<th>Scans for</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>fixnum</td>
<td>An integer</td>
</tr>
<tr>
<td>%f</td>
<td>flonum</td>
<td>A floating-point number</td>
</tr>
<tr>
<td>%s</td>
<td>string</td>
<td>A string (delimited by spaces) in the input</td>
</tr>
</tbody>
</table>

### Arguments

- \texttt{p\_inputPort}  
  Input port \texttt{fscanf} reads from. The input port cannot be the CIW for \texttt{fscanf}.

- \texttt{t\_sourceString}  
  Input string for \texttt{sscanf}.

- \texttt{t\_formatString}  
  Format string to match against in the reading.

- \texttt{s\_var1}  
  Name of variable to store results of read.

### Value Returned

- \texttt{x\_items}  
  The number of input items it successfully read in. As a side-effect, the items read in are assigned to the corresponding variables specified in the call.

- \texttt{nil}  
  It encounters an end of file.

### Example

\texttt{fscanf( p "\%d \%f" i d )}

Scans for an integer and a floating-point number from the input port \texttt{p} and stores the values read in the variables \texttt{i} and \texttt{d}, respectively.

Assume a file \texttt{testcase} with one line:

\texttt{hello 2 3 world}
$x = \text{infile("testcase")}$ => \text{port:"testcase"}
$fscanf(x\ "\%s\ \%d\ \%d\ \%s\ a\ b\ c\ d\ ) => 4$
\text{(list\ a\ b\ c\ d\ )} => \text{("hello"\ 2\ 3\ "world")}$

Reference

\text{fprintf, lineread}
funcall

funcall(
    slu_func
    [ arg ... ]
) => g_result

Description

Applies the given function to the given arguments.

The first argument to funcall must be either the name of a function or a lambda/nlambda/macro expression or a function object. The rest of the arguments are to be passed to the function.

The arguments arg ... are bound to the formal arguments of slu_func according to the type of function. For lambda functions the length of arg should match the number of formal arguments, unless keywords or optional arguments exist. For nlambda and macro functions, arg are bound directly to the single formal parameter of the function.

Note: If slu_func is a macro, funcall evaluates it only once, that is, it expands it and returns the expanded form, but does not evaluate the expanded form again (as eval does).

Arguments

slu_func Name of the function.

arg Arguments to be passed to the function.

Value Returned

g_result The result of applying the function to the given arguments.

Example

funcall( 'plus 1 2 ) ; Apply plus to its arguments.
=> 3

procedure( sum3(x y z) funcall( 'plus x y z)
=> sum3
sum3(1 2 3) ;Define a procedure
=> 6
Reference
eval, apply
funobj

funobj(  
x_id  
)  
=> U_functionObject

Description

Returns the function object designated by the given object ID.

It signals an error if the argument is not a valid function object ID.

Arguments

x_id  

The ID of a function object that appears in its print representation.

Value Returned

U_functionObject  

Function object whose ID is x_id. An error is signaled if no match is found.

Example

F = lambda( ( x y ) x+y ) => funobj:0x1e3688
eq( funobj( 0x1e3688 ) F ) => t

This example assigns a function object to the variable F. Extract the ID from the print representation by inspection and pass it to the funobj function. Using the eq function demonstrates that the return value is the original function object.

Reference

envobj
**gc**

gc(
    [ t_string ]
)
=> nil

**Description**

Forces a garbage collection. This function is also called by the system.

Garbage collection (gc) refers to the process in which SKILL locates storage cells that are no longer needed (thus the term garbage) and recycles them by putting them back on the free storage list. Garbage collection is also called by the system. Garbage collection is transparent to SKILL users and to users of applications built on top of SKILL.

You can turn on the printing of garbage collection messages by setting the _gcprint variable to t (that is, _gcprint=t). Garbage collection can be turned off at any time by setting the gcdisable variable to t. To enable garbage collection again, you can restore gcdisable to its previous value. You can force a garbage collection at any time by calling the gc function.

**Caution**

*Because some applications turn off garbage collection during their execution, you should be careful about enabling it. Corrupted data can result.*

**Arguments**

**t_string**

File into which additional information is dumped.

**Value Returned**

**nil**

Always returns nil.

**Example**

gc( ) => nil
Reference

gcsummary, needNCells
gensym

gensym(
    [ S_arg ]
) => s_result

Description

Returns a new symbol based on the input argument.

The new symbol's print name is the result of concatenating the printed representation of the argument, or “G” if no argument is given, and the printed (decimal) representation of a number. The returned new symbol is unique in the sense that it does not exist at the time this function is called.

Arguments

S_arg            String or symbol to be concatenated into a new symbol. If not supplied, the default value is G.

Value Returned

s_result        New unique symbol.

Example

gensym() => G5
gensym("test") => test6
test7 = 10 => 10 ;test7 exists now.
gensym('test) => test8 ;test7 is skipped.
gensym() == gensym() => nil ;Always returns nil.

Reference

concat, symbolp, symeval, symstrp
**geqp**

```
geqp(  
    n_num1  
    n_num2  
  )  
=> t | nil
```

**Description**

This predicate function checks if the first argument is greater than or equal to the second argument. Prefix form of the \( \geq \) operator.

**Arguments**

- **n_num1**
  - Number to be checked.

- **n_num2**
  - Number against which \( n\_num1 \) is checked.

**Value Returned**

- **t**
  - \( n\_num1 \) is greater than or equal to \( n\_num2 \).

- **nil**
  - \( n\_num1 \) is less than \( n\_num2 \).

**Example**

```
geqp(2 2)  => t  
geqp(-2 2) => nil  
geqp(3 2.2) => t
```

**Reference**

`greaterp, leqp, lessp`
get

get(
   sl_id
   S_name
)
=> g_result | nil

Description

Returns the value of a property in a property list (including disembodied property list), association table, structure, and database object.

Used in conjunction with putprop, where putprop stores the property and get retrieves it.

Arguments

sl_id          Symbol or disembodied property list.
S_name         Name of the property you want the value of.

Value Returned

g_result       Value of S_name in the sl_id property list.
nil             The named property does not exist.

Example

putprop( 'chip 8 'pins ) => 8

Assigns the property pins to a value of 8 to the symbol chip.

get( 'chip 'pins ) => 8
chip.pins => 8
x = '(nil a 3 b 4) ;a disembodied property list
x->a => 3
get(x 'a) => 3

Reference

plist, putprop
get_filename

get_filename(
    p_port
)
=> s_result

Description
Returns the file name of a port.

Arguments

p_port A port object.

Value Returned

x_result The file name of the port.

Examples

aPort => port:"inFile"
get_filename( aPort ) => "inFile"
get_pname

get_pname(  
    s_arg  
)  
=> t_result

Description

Returns the print name of a symbol as a string.

This function is useful for converting symbols to strings. If you just want to print the name of a symbol, you do not need to use this function. This function is equivalent to symbolToString.

Arguments

s_arg  
A symbol.

Value Returned

t_result  
Print name of the symbol.

Example

get_pname( 'a' )  
=> "a"
get_pname(concat("Cell_" 123))  
=> "Cell_123"

Reference

concat, get_string, stringToSymbol, symbolToString
get_string

get_string(
    S_arg
) => t_result

Description

Converts the argument to a string if it is a symbol. Otherwise it returns the string itself.

Arguments

S_arg String or symbol.

Value Returned

t_result Of the argument is a string, returns the argument itself. If the argument is a symbol, returns the print name as a string.

Example

get_string('xyz) => "xyz"
get_string("xyz") => "xyz"

Reference

concat, get_pname, symbolToString
getc

cetc(
    [ p_inputPort ]
)
=> s_char

Description

Reads and returns a single character from an input port. Unlike the C library, the getc and getchar SKILL functions are totally unrelated.

The input port arguments for both gets and getc are optional. If the port is not given, the functions take their input from piport.

Arguments

p_inputPort Input port; if not given, function defaults to piport.

Value Returned

s_char Single character from the input port in symbol form. If the character returned is a non-printable character, its octal value is stored as a symbol.

Example

In the following assume the file test1.data has its first line read as:

#This is the data for test1
p = infile("test1.data") => port:"test1.data"
getc(p) => \
getc(p) => T
getc(p) => h

Reference

gets
getchar

getchar(
    S_arg
    x_index
  )
=> s_char | nil

Description

Returns an indexed character of a string or the print name if the string is a symbol. Unlike the C library, the getc and getchar SKILL functions are totally unrelated.

Arguments

S_arg  Character string or symbol.

x_index  Number corresponding to an indexed point in S_arg.

Value Returned

s_char  Single character symbol corresponding to the character in S_arg indexed by x_index.

nil  If x_index is less than 1 or greater than the length of the string.

Example

guchar("abc" 2) => b
guchar("abc" 4) => nil

Reference

nindex, parseString, strlen, substring
getCurrentTime

getCurrentTime(
  
) => tTimeString

**Description**

Returns a string representation of the current time.

**Arguments**

None.

**Value Returned**

`tTimeString` Current time in the form of a string. The format of the string is `month day hour:minute:second year`.

**Example**

```plaintext
getCurrentTime( )=> "Jan 26 18:15:18 1994"
```

This format is also used by the `compareTime` function.

**Reference**

`compareTime`
getd

getd(
    s_functionName
)
=> g_definition | nil

Description

Returns the function binding for a function name.

Note: This function is not needed in SKILL++ because functions are treated as regular values. Therefore you can simply use variable reference syntax to access any function binding.

Arguments

s_functionName 
Name of the function.

Value Returned

g_definition 
If the function is defined in SKILL, returns the function object that the procedure function associates with a symbol.

If the function is primitive, the binary definition is printed (see example below).

nil 
No function definition exists.

Example

getd( 'alias ') => nlambda:alias

The function is primitive.

getd( 'edit ') => funobj:0x24b478

The function is written in SKILL.

Reference

alias, bcdp, putd
getDirFiles

getDirFiles(
  S_name
)
=> l_strings

Description

Returns a list of the names of all files and directories, including . and .., in a directory.

Uses the current SKILL path for relative paths. Note that a path which is anchored to current
directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name  
Name of the directory in either string or symbol form.

Value Returned

l_strings  
List of names of all files and directories in a given directory name 
(including . and ..).

Signals an error if the directory does not exist or is inaccessible.

Example

getDirFiles(car(getInstallPath()))=> ("." ".." "bin" "cdsuser" "etc" "group" 
"include" "lib" "pvt" "samples" "share" "test" "tools" "man" "local" )

Reference

getInstallPath, getSkillPath, isDir
getFnWriteProtect

getFnWriteProtect (  
    s_name  
)  
  => t | nil

Description

Checks if the given function is write-protected.

The value is \texttt{t} if \texttt{s\_name} is write-protected; \texttt{nil} otherwise.

Arguments

\textit{s\_name} 
Name of the function.

Value Returned

\texttt{t} 
The function is write protected.

\texttt{nil} 
The function is not write protected.

Signals an error if the function is not defined.

Example

getFnWriteProtect ( 'strlen' ) => t

Reference

getd, setFnWriteProtect
getFunType

getFunType(
    u_functionObject
) => s_functionObject_type

Description

Returns a symbol denoting the function type for a given function object.

Possible function types include lambda, nlambda, macro, syntax, or primop.

Arguments

u_functionObject A function object.

Value Returned

s_functionObject_type

Possible return values include lambda, nlambda, macro, syntax, or primop.

Example

getFunType( getd( 'sin ') ) => lambda
getFunType( lambda( (x y) x+y ) ) => lambda
getFunType( getd( 'breakpt ') ) => nlambda
getFunType( getd( 'if ') ) => syntax
getFunType( getd( 'plus ') ) => primop

Reference

defmacro, getd, lambda, mprocedure, nprocedure – SKILL mode only, procedure
getInstallPath

getInstallPath(
  
) => 1_string

Description

Returns the absolute path of the Cadence DFII installation directory where the DFII products are installed on your system as a list of a single string.

Arguments

None.

Value Returned

1_string

Returns the installation path as a list of a single string.

Example

getInstallPath() => ("/usr5/cds/5.0")

Reference

getSkillPath, getWorkingDir, prependInstallPath, cdsGetInstPath
getLogin

getLogin(
)
  => t_loginName

Description

Returns the user's login name as a string.

Arguments

None.

Value Returned

t_loginName          Returns the user's login name as a string.

Example

getLogin
=> "fred"
getPrompts

getPrompts()
   => l_strings

Description

Returns the current values of the first level and second level prompt text strings, respectively.

The first prompt text string is the first level prompt that represents the topmost top-level prompt, while the second one indicates the second level prompt which is used whenever a nested top-level is entered.

Arguments

None.

Value Returned

l_strings

The current values of the first level and second level prompt text strings. The result is a list where the first element is the first level prompt and the second element is the second level prompt specified by setPrompts.

Example

skill> getPrompts()
   ("> " "<d> ")
CIW> getPrompts()
   ("> " ")

Default prompts for the SKILL interpreter and CIW, respectively.

Reference

setPrompts
**getq**

```lisp
getq(
   sl_id
   S_name
)
=> g_result | nil

getq(
   sl_id->s_name
)
=> g_result | nil
```

**Description**

Returns the value of a property in a property list. Same as `get` except that the second argument is not evaluated. This is a syntax form.

Used in conjunction with `putprop`, where `putprop` stores the property and `getq` retrieves it.

**Arguments**

- `sl_id` Symbol or disembodied property list.
- `S_name` Name of the property you want the value of.

**Value Returned**

- `g_result` Value of `s_name` in the `sl_id` property list.
- `nil` The named property does not exist.

**Example**

```lisp
putprop( 'chip 8 'pins ) => 8
```

Assigns the property pins to a value of 8 to the symbol chip.

```lisp
getq( 'chip pins ) => 8
chip.pins => 8
chip1 = list(nil 'pins 10) => (nil pins 10)
chip1->pins => 10
```
Reference

get, getgg, plist, putprop
getqq

getqq(s_id
    S_name
  )
=> g_result | nil

getqq(sl_id.sl_name
  )
=> g_result | nil

Description

Returns the value of a property in a symbol’s property list. Same as get except that neither argument is evaluated. This is a syntax form.

Used in conjunction with putprop, where putprop stores the property and getqq retrieves it.

Arguments

s_id Symbol to get a property from.
S_name Name of the property you want the value of.

Value Returned

g_result Value of the property S_name in the property list of s_id.
nil The named property does not exist.

Example

putprop( 'chip 8 'pins ) => 8
Assigns the property pins to a value of 8 to the symbol chip.

getqq( chip pins ) => 8
chip.pins => 8
Reference

get, getq, plist, putprop
**getTempDir**

```s>Your code here</s>
``` 

**Description**

Returns the system temp directory as a string.

**Arguments**

None.

**Value Returned**

`$t_TempDir` The name of your current temp directory.

**Example**

```s>Your code here</s>`
```
gets

gets(s_variableName
    [ p_inputPort ]
)
=> t_string | nil

Description
Reads a line from the input port and stores the line as a string in the variable. This is a macro.

The string is also returned as the value of gets. The terminating newline character of the line becomes the last character in the string.

Arguments

s_variableName         Variable to store input string in.
p_inputPort            Name of input port; piport is used if none is given.

Value Returned

t_string               Returns the input string when successful.
nil                    When EOF is reached.

Example

Assume the test1.data file has the following first two lines:

```
#This is the data for test1
0001 1100 1011 0111
```

```
p = infile("test1.data")
gets(s p)          => "#This is the data for test1\n"
gets(s p)          => "0001 1100 1011 0111\n"
s              => "0001 1100 1011 0111\n"
```

Reference

getc, getchar, infile
getShellEnvVar

getShellEnvVar(
    t_UnixShellVariableName
)
=> t_value | nil

Description

Returns the value of a UNIX environment variable, if it has been set.

Arguments

\( t_{\text{UnixShellVariableName}} \)

Name of the UNIX shell environment variable.

Value Returned

\( t_{\text{value}} \)

Value of named UNIX environment variable.

\( \text{nil} \)

No environment variable with the given name has been set.

Example

getShellEnvVar("SHELL") => "/bin/csh"

Returns the current value of the SHELL environment variable.

Reference

setShellEnvVar
getSkillPath

getSkillPath(
  )
  => l_strings | nil

Description

Returns the current SKILL path.

The SKILL path is used in resolving relative paths for some SKILL functions. See "/O and File Handling" in the SKILL Language User Guide.

Arguments

None.

Value Returned

l_strings Directory paths from the current SKILL path setting. The result is a list where each element is a path component as specified by setSkillPath.

nil The last call to setSkillPath gave nil as its argument.

Example

setSkillPath('"." "~" "~/cpu/test1")
=> ("~/cpu/test1")
getSkillPath() => ("." "~" "~/cpu/test1")

The example below shows how to add a directory to the beginning of your search path (assuming a directory "~/lib").

setSkillPath(cons("~/lib" getSkillPath()))
=> ("~/lib" "~/cpu/test1")
getSkillPath()
=> ("~/lib" "." "~" "~/cpu/test1")

Reference

setSkillPath
getSkillVersion

getSkillVersion(
)
=> t_version

Description

Returns the version of the SKILL that is currently running.

Arguments

None.

Value Returned

\textit{t\_version} \hspace{1cm} \text{Version of the SKILL that is currently running.}

Example

\texttt{getSkillVersion()}

\begin{verbatim}
=> "SKILL04.20"
\end{verbatim}

Reference

\texttt{getVersion}
getVarWriteProtect - SKILL mode only

getVarWriteProtect(s_name)

=> t | nil

Description

Checks if a variable is write-protected. Does not work in SKILL++ mode. In SKILL++ mode, use getFnWriteProtect instead.

Arguments

s_name Name of the variable to check.

Value Returned

t The variable is write-protected.
nil Otherwise.

Example

x = 5
getVarWriteProtect( 'x' ) => nil

Returns nil if the variable x is not write protected.

Reference

getFnWriteProtect, setVarWriteProtect - SKILL mode only
**getVersion**

getVersion( 
    [ g_opt ]
) => t_version

**Description**

Returns the version number of the Cadence software you are currently using.

**Arguments**

* g_opt

Optional argument. If this argument is given, the subversion number of the Cadence software currently using is returned. By default, the full version number, including hotfix version, of the Cadence software currently using is returned.

**Value Returned**

* t_version

String identifying the version/subversion of the program you are running.

**Example**

getVersion() => "@(#)CDS: icfb.exe version 5.0.0 08/14/2002 17:52 (cds11612) $"
getVersion( 'subVer ) => "sub-version 5.0.0.36.72"

**Reference**

getSkillVersion
getWarn

getWarn(
)
=> t_warning

Description

Returns the buffered warning if it has not already been printed.

Arguments

None.

Value Returned

\textit{t\_warning} \quad The warning message that would have been printed if it had not been intercepted by the call to getWarn.

Example

\begin{verbatim}
procedure( testWarn( @key ( getLastWarn nil ) )
    warn("This is warning %d\n" 1 ) ;;; print previous warning
    warn("This is warning %d\n" 2 ) ;;; and buffer new one.
    warn("This is warning %d\n" 3 )
    when( getLastWarn
        thrownAwayWarn = getWarn( ) ;;; throw away last warning
        nil ;;; return nil
    )
    ; when
    ; procedure
\end{verbatim}

The \texttt{testWarn} function intercepts the last warning message and stores it in a global variable \texttt{thrownAwayWarn} if \texttt{t} is passed in, and lets the system print all the warnings if \texttt{nil} is given as an argument. Use of the getWarn( ) function makes it possible to throw away a warning message, if desired.

\begin{verbatim}
testWarn( ?getLastWarn t)
=> nil
*WARNING* This is warning 1
*WARNING* This is warning 2
\end{verbatim}

\texttt{testWarn( ?getLastWarn nil)}

\begin{verbatim}
testWarn( ?getLastWarn nil)
=> nil
*WARNING* This is warning 1
\end{verbatim}

Returns \texttt{nil}. The system prints the first two warnings and the third is intercepted and stored in global variable \texttt{thrownAwayWarn}. 
Returns nil. The system prints all the queued warnings.

Note that the return value may be interleaved with the warning message output. The following example shows how the actual output can appear in the CIW.

```skill
testWarn( ?getLastWarn t)  
*WARNING* This is warning 1 
*WARNING* This is warning 2 
=> nil 

testWarn( ?getLastWarn nil) 
*WARNING* This is warning 1 
*WARNING* This is warning 2 
=> nil 
*WARNING* This is warning 3
```

**Reference**

`print`, `warn`
getWorkingDir

getWorkingDir(
    )
    => t_currentDir

Description

Returns the current working directory as a string.

The result is put into a ~/prefixed form if possible by testing for commonality with the current user's home directory. For example, ~/test would be returned in preference to /usr/mnt/user1/test, assuming that the home directory for user1 is /usr/mnt/user1 and the current working directory is /usr1/mnt/user1/test.

Arguments

None.

Value Returned

t_currentDir           The name of your current working directory.

Example

getWorkingDir() => "~/project/cpu/layout"

Reference

changeWorkingDir
**go**

```go
go(
    s_label
)
```

**Description**

Transfers control to the statement following the label argument. This is a syntax form.

The `go` statement is only meaningful when it is used inside a `prog` statement. Control can be transferred to any labelled statement inside any `progs` that contain the `go` statement, but cannot be transferred to labelled statements in a `prog` that is not active at the time the `go` statement is executed. Generally, using `go` is considered poor programming style when higher level control structures such as `foreach` and `while` can be used.

**Arguments**

- `s_label`: Label you want to transfer control to inside a `prog`.

**Value Returned**

None.

**Example**

The following example demonstrates how to use the `go` function form in a simple loop structure.

```go
procedure( testGo( data )
    prog( ()
        start
            print( car( data ))
            data = cdr( data )
            if( data go( start )) ; go statement to jump to start.
    ))
    testGo( '(a b c))
    abc ; Prints the variable data.
    => nil ; Returns nil.
```

**Reference**

- `prog`, `foreach`, `return`, `while`
greaterp

greaterp(
    n_num1
    n_num2
)
=> t | nil

Description

This predicate function checks if the first argument is greater than the second argument. Prefix form of the > operator.

Arguments

n_num1  Number to be checked.
n_num2  Number against which n_num1 is checked.

Value Returned

t  n_num1 is greater than n_num2.
nil  n_num1 is less than or equal to n_num2.

Example

greaterp(2 2)  => nil
greaterp(-2 2) => nil
greaterp(3 2.2) => t

Reference

ggeq, leq, lessp
help

help(
    [ S_name ]
) => t | nil

Description

Retrieves and prints the cdsFinder documentation strings for the given function name (a symbol). If the given name is a string, it is interpreted as a regular expression, and the entire cdsFinder database is searched for functions whose name or documentation string contains or matches the given string. Help is an nlambda function.

Arguments

S_name Name to search for.

Value Returned

t The given function name is found in the cdsFinder.

nil No match is found for S_name.

Example

help nonexist => nil
help scanf
Prints the following and returns t.

fscanf( p_inputPort t_formatString [s_var1 ...] )
scanf( t_formatString [s_var1 ...] )
sscanf( t_sourceString t_formatString [s_var1 ...] )

The only difference between these functions is the source of input. fscanf reads input from a port according to format specifications and returns the number of items read in. scanf takes its input from piport implicitly. scanf only works in standalone SKILL when the piport is not the CIW. sscanf reads its input from a string instead of a port.

=> t
help println
Prints the following and returns t.
println( g_value [p_outputPort] ) => nil

Prints a SKILL object using the default format for the data type of the value, then prints a newline character.
=> t
help "read"

Prints the following and returns t.
fscanf, scanf, sscanf, getWarn, infile, instring, ipcReadProcess, ipcWaitForProcess, isReadable, lineread, linereadstring, load, loadstring, outfile, pp, putpropq, putpropqq, read, readTable, readstring
=> t
help "match nowhere"
=> nil
if

if(
  g_condition
g_thenExpression
  [ g_elseExpression ]
)
=> g_result

if(
  g_condition
  then g_thenExpr1 ...
  [ else g_elseExpr1 ... ]
)
=> g_result

Description

Selectively evaluates two groups of one or more expressions. This is a syntax form.

The if form evaluates g_condition, typically a relational expression, and executes g_thenExpression if the condition is true (that is, its value is non-nil); otherwise, g_elseExpression is executed. The value returned by if is the value of the corresponding expression evaluated. The if form can therefore be used to evaluate expressions conditionally.

if( g_condition then g_thenExpr1 ... [ else g_elseExpr1 ... ] )
=> g_result

The second form of if uses the keywords then and else to group sequences of expressions for conditional execution. If the condition is true, the sequence of expressions between then and else (or the end of the if form) is evaluated, with the value of the last expression evaluated returned as the value of the form. If the condition is nil instead, the sequence of expressions following the else keyword (if any) is evaluated instead. Again, the value of the last expression evaluated is returned as the value of the form.

Arguments

g_condition
  Any SKILL expression.

g_thenExpression
  Any SKILL expression.

g_elseExpression
  Any SKILL expression.
Value Returned

\[ g\_result \]

The value of \( g\_thenExpression \) if \( g\_condition \) has a non-nil value. The value of \( g\_elseExpression \) is returned if the above condition is not true.

Example

\[
\begin{align*}
x &= 2 \\
\text{if( } (x > 5) \text{ 1 0) } \\
&\Rightarrow 0 \\
\text{a = "polygon"}
\end{align*}
\]

; Returns 0 because \( x \) is less than 5.

\[
\begin{align*}
a &= "polygon"
\text{if( } (a == "polygon") \text{ print(a) )}
\end{align*}
\]

"polygon" ; Prints the string polygon.

\[
\Rightarrow \text{ nil} \\
\text{; Returns the result of print.}
\]

\[
\begin{align*}
x &= 5 \\
\text{if( } x \text{ "non-nil" "nil" ) }
\end{align*}
\]

; Returns "non-nil" because \( x \) was not nil. If \( x \) was nil then "nil" would be returned.

\[
\begin{align*}
x &= 7 \\
\text{if( } (x > 5) \text{ then 1 else 0) } \\
&\Rightarrow 1 \\
\end{align*}
\]

; Returns 1 because \( x \) is greater than 5.

\[
\begin{align*}
\text{if( } (x > 5) \\
\text{ then println("x is greater than 5") } \\
&x + 1 \\
\text{ else print("x is less ") } \\
&x - 1)
\end{align*}
\]

\( x \) is greater than 5 ; Printed if \( x \) was 7.

\[
\Rightarrow 8 \\
\text{; Returned 8 if } x \text{ was 7.}
\]

Reference

\textit{cond, for, foreach, unless, while}
importSkillVar - SKILL++ mode

importSkillVar(
    s_variable ...
)
=> nil

Description

Tells the compiler that the given variable names should be treated as SKILL global variables in SKILL++ code.

All global SKILL functions are automatically accessible from SKILL++ code, but not the SKILL variables. This form tells the compiler that the given variable names should be treated as SKILL global variables in SKILL++ code.

This function has no effect if there is already a SKILL++ global variable of the same name defined. Also remember that local variables can use the same name and always take precedence.

Note: This only means that the variables will be accessed as SKILL globals, NOT that they will follow SKILL's dynamic scope rule in SKILL++ code.

Arguments

s_variable Variable to be treated as SKILL global variables in SKILL++ code.

Value Returned

nil Always returns nil. This function is for side-effect only.

Example

> q = 1
=> 1
> toplevel 'ils
ILS-<2> q
*Error* eval: unbound variable - q
ILS-<2> importSkillVar( q )
=> 1
ILS-<2> q
=> 1
This example shows assigning a value to the global variable $q$ in SKILL mode and then importing the variable into SKILL++.
index

index(  
    t_string1
    S_string2
  )
  => t_result | nil

Description

Returns a string consisting of the remainder of string1 beginning with the first occurrence of string2.

Arguments

\[ t_{-}string1 \]  String to search for the first occurrence of S_string2.

\[ S_{-}string2 \]  String to search for in t_string1.

Value Returned

\[ t_{-}result \]  If S_string2 is found in t_string1, returns a string equal to the remainder of t_string1 that begins with the first character of S_string2.

\[ \text{nil} \]  If S_string2 is not found.

Example

index( "abc" 'b' )    => "bc"
index( "abcdabce" "dab" ) => "dabce"
index( "abc" "cba" )    => \( \text{nil} \)
index( "dandelion" "d" ) => "dandelion"

Reference

rindex, strcmp, strncmp
infile

infile(
    S_fileName
  )
  => p_inport | nil

Description

Opens an input port ready to read a file. Always remember to close the port when you are done.

The file name can be specified with either an absolute path or a relative path. In the latter case, current SKILL path is used if it's not nil. Note that a path which is anchored to current directory, for example, ./, ../, or ../../../, etc., is not considered as a relative path.

Note: Always remember to close the port when you are done.

Arguments

S_fileName Name of the file to be read; it can be either a string or a symbol.

Value Returned

p_inport Port opened for reading the named file.

nil The file does not exist or cannot be opened for reading.

Example

in = infile("~/test/input.il") => port:"~/test/input.il"

If such a file exists and is readable.

infile("myFile") => nil

If myFile does not exist according to the current setting of the SKILL path or exists but is not readable.

close(in) => t
Reference

close, isFileName, isReadable, outfile, portp
inportp

inportp(
    g_obj
)
=> t | nil

Description

Checks if an object is an input port.

Note: An input port may be closed, so if inportp returns t, that does not guarantee a successful read from the port.

Arguments

g_obj Any SKILL object.

Value Returned

t The given object is an input port.

nil Otherwise.

Example

(inportp piport) => t
(inportp poport) => nil
(inportp 123) => nil

Reference

outportp
inScheme

inScheme(
    g_form
) => g_result

Description

Evaluates a form as top-level SKILL++ code, disregarding the surrounding evaluation context.

Arguments

g_form
Form to be evaluated as top-level SKILL++ code.

Value Returned

g_result
Result of the evaluation.

Example

(inScheme
    (define myVar 100)) => myVar

Defines a SKILL++ global variable, even if this code appears inside a SKILL file.

Reference

inSkill
inSkill

inSkill(
  g_form
) => g_result

Description

Evaluates a form as top-level SKILL code, disregarding the surrounding evaluation context.

Arguments

g_form

Form to be evaluated as top-level SKILL code.

Value Returned

g_result

Result of the evaluation.

Example

(inSkill
  skillVar = 100) => 100

Sets a SKILL global variable, even if this code appears inside a SKILL++ file.

Reference

inScheme
**instring**

```skill
instring(
    t_string
) => p_port
```

**Description**

Opens a string for reading, just as `infile` would open a file.

An input port that can be used to read the string is returned. *Always remember to close the port when you are done.*

**Arguments**

- `t_string`  
  Input string opened for reading.

**Value Returned**

- `p_port`  
  Port for the input string.

**Example**

```skill
s = "Hello World!"
p = instring(s) => "Hello World!"
scanf(p "%s %s" a b) => 2
  a => "Hello"
b => "World!"
close(p) => t
```

**Reference**

- `gets`
- `infile`
integerp

integerp(
    g_obj
)
=> t | nil

Description

Checks if an object is an integer. This function is the same as fixp.

Arguments

g_obj
      Any SKILL object.

Value Returned

t         The given object is an integer.

nil        Otherwise.

Example

(integerp 123) => t
(integerp "123") => nil

Reference

fixp
**intToChar**

```skill
intToChar(
    x_ascii
) => s_char
```

**Description**

Returns the single-character symbol whose ASCII code is the given integer value.

**Arguments**

- **x_ascii**
  
  ASCII code.

**Value Returned**

- **s_char**
  
  Symbol of single-character whose ASCII code is `x_ascii`.

**Example**

```skill
intToChar(66) => B
```

**Reference**

- `charToInt`
isCallable

```currently
isCallable(
  s_function
)
  => t | nil
```

**Description**

Checks if a function is defined or is autoloadable from a context.

**Arguments**

```currently
s_function
```
Name of a function.

**Value Returned**

```currently
t
```
The specified function is defined or is autoloadable.

```currently
nil
```
The specified function is not defined or is not autoloadable.

**Example**

```currently
isCallable( 'car) => t
procedure( myFunction( x ) x+1)
isCallable('myFunction) => t
```

**Reference**

bcdp, getd, load, putd
**isDir**

``` SKILL 
isDir( 
    S_name
    [ tl_path ]
)
=> t | nil
```

**Description**

Checks if a path exists and if it is a directory name.

When `S_name` is a relative path, the current SKILL path is used if it’s non-nil. Note that a path which is anchored to current directory, for example, `./`, `../`, or `../../..`, etc., is not considered as a relative path.

**Arguments**

- **S_name**
  Path you want to check.

- **tl_path**
  List of paths that overrides the SKILL path.

**Value Returned**

- **t**
  The name exists and it is the name of a directory.

- **nil**
  The name exists and is not the name of a directory or `S_name` does not exist at all.

**Example**

``` SKILL 
isDir("DACLib") => t 
isDir("triadc") => nil
```

Assumes `DACLib` is a directory and `triadc` is a file under the current working directory and the SKILL path is nil.

``` SKILL 
isDir("test") => nil
```

Result if `test` does not exist.
Reference

getSkillPath, isFile, isWritable
isExecutable

isExecutable(
  S_name
  [ tl_path ]
)
=> t | nil

Description

Checks if you have permission to execute a file or search a directory.

A directory is executable if it allows you to name that directory as part of your path in searching files. It uses the current SKILL path for relative paths. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name
Name of the file or directory you want to check for execution/search permission.

tl_path
List of paths that overrides the SKILL path.

Value Returned

t
If you have permission to execute the file or search the directory specified by S_name.

nil
The directory does not exist or you do not have the required permissions.

Example

isExecutable("/bin/ls")    => t
isExecutable("/usr/tmp")  => t
isExecutable("attachFiles") => nil

Result if attachFiles does not exist or is non-executable.
Reference

isDir, isFile, isReadable, isWritable
isFile

isFile(
    S_name
    [ tl_path ]
) => t | nil

Description

Checks if a file exists and that it is not a directory.

Identical to isFileName, except that directories are not viewed as (regular) files. Uses the current SKILL path for relative paths. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name Path you want to check.

tl_path List of paths that overrides the SKILL path.

Value Returned

t The S_name file exists.
nil The S_name file does not exist.

Example

isFile( "DACLib") => nil

Assumes DACLib is a directory and triadc is a file in the current working directory and the SKILL path is nil. A directory is not viewed as a file in this case.

isFile( "triadc") => t
isFile( ".cshrc" list( "." ".~" )) => t

Reference

isDir, isFileName, getSkillPath
isFileEncrypted

isFileEncrypted(
  \( S\_name \)
) => t | nil

Description

Checks if a file exists and is encrypted.

Similar to `isFile`, except that it returns \( t \) only if the file exists and is encrypted. Uses the current SKILL path for relative paths. Note that a path which is anchored to current directory, for example, \( ./ \), \( ../ \), or \( ../../.. \), etc., is not considered as a relative path.

Arguments

\( S\_name \) File you want to check.

Value Returned

\( t \) The \( S\_name \) file exists and is encrypted.

\( \text{nil} \) The \( S\_name \) file does not exist or is not encrypted.

Example

isFileEncrypted( "~/testfns.il") => nil
encrypt( "~/testfns.il" "~/testfns.ile"
isFileEncrypted( "~/testfns.ile") => t

Reference

encrypt, `getSkillPath`, `isFile`
isFileName

isFileName(
    S_name
    [ tl_path ]
)
=> t | nil

Description

Checks if a file or directory exists.

The file name can be specified with either an absolute path or a relative path. In the latter case, current SKILL path is used if it’s not nil. Only the presence or absence of the name is checked. If found, the name can belong to either a file or a directory. isFileName differs from isFile in this regard. Note that a path which is anchored to current directory, for example, ./, ../../../, or ../../../, etc., is not considered as a relative path.

Arguments

S_name Path you want to check.

tl_path List of paths to override the SKILL path.

Value Returned

t The S_name path exists.

nil The S_name path does not exist.

Example

Suppose DACLib is a directory and triadc is a file in the current working directory and the SKILL path is nil.

isFileName("DACLib") => t

A directory is just a special kind of file.

isFileName("triadc") => t
isFileName("triad1") => nil

Result if triad1 does not exist in current working directory.
isFileName( "cshrc" list("." "~")) => t

Reference

isDir, isFile, getSkillPath
isInfinity

isInfinity(  
    f_flownum  
)  
=> t | nil

Description

Checks if the given flownum argument represents infinity (positive or negative).

Arguments

f_flownum A floating-point number.

Value Returned

t If f_flownum is infinity (positive or negative).

nil Otherwise.

Example

plus_inf = 2.0 * 1e999
isInfinity (plus_inf) => t
isInfinity (987.65) => nil
**isLargeFile**

```markdown
isLargeFile(
  S_name
  [ tl_path ]
)
=> t | nil
```

**Description**

Checks if a file is a large file (with size greater than 2GB).

The file name can be specified with either an absolute path or a relative path. In the latter case, the current SKILL path is searched if it's not nil. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

The SKILL path can be overridden by specifying `tl_path`.

**Arguments**

- **S_name**: Name of the file you want to check.
- **tl_path**: List of paths to override the SKILL path.

**Value Returned**

- **t**: The `S_name` file has a size greater than 2GB.
- **nil**: The `S_name` file has a size less than or equal to 2GB.

**Example**

```markdown
fileLength( "largeFile" ) => 3072000000
isLargeFile( "largeFile" ) => t
```

**Reference**

- fileLength, isDir, isFile, isFileName
isLink

```lisp
isLink(
    S_name
    [ tl_path ]
)
=> t | nil
```

**Description**

Checks if a path exists and if it is a symbolic link.

When `S_name` is a relative path, the current SKILL path is used if it’s non-nil. Note that a path which is anchored to current directory, for example, `.`, `../`, or `../../..`, etc., is not considered as a relative path.

**Arguments**

- `S_name`: Path you want to check.
- `tl_path`: List of paths that override the SKILL path.

**Value Returned**

- `t`: The name exists and it is a symbolic link.
- `nil`: The name exists and is not a symbolic name or if `S_name` does not exist at all.

**Example**

```lisp
isLink("/usr/bin")=> nil
isLink("/usr/spool")=> t       ;Assuming it’s a link to /var/spool
```

**Reference**

`isFile`, `isDir`
isMacro

isMacro(
    s_symbolName
)
=> t | nil

Description
Checks if the given symbol denotes a macro.

Arguments
s_symbolName    Symbol to check.

Value Returned
t    The given symbol denotes a macro.
nil   Otherwise.

Example
(isMacro 'plus)  => nil
(isMacro 'defmacro) => t

Reference
defmacro
isNaN

isNaN(
    \_f_flownum
)
=> t | nil

Description

Checks if the given flownum argument represents NaN (not-a-number), nil otherwise.

Arguments

f_flownum A floating-point number.

Value Returned

t If f_flownum is NaN.
nil Otherwise.

Example

nan = 0.0 * 2.0 * 1e999
isNaN (nan) => t
isNaN (123.456) => nil
isReadable

isReadable(
  S_name
  [ tl_path ]
)
=> t | nil

Description

Checks if you have permission to read a file or list a directory. Uses the current SKILL path for relative paths. Note that a path which is anchored to current directory, for example, ./, ../../../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name  Name of a file or directory you want to know your access permissions on.

tl_path  List of paths to override the SKILL path.

Value Returned

t  If S_name exists and you have permission to read it (for files) or list the contents (for directories).

nil  The file does not exist or does exist, but you do not have permission to read it.

Example

isReadable("./") => t

Result if current working directory is readable.

isReadable("~/DACLib") => nil

Result if "~/DACLib" is not readable or does not exist.

Reference

infile, isExecutable, isFile, isWritable
isWritable

isWritable(
    S_name
    [ tl_path ]
)
=> t | nil

Description

Checks if you have permission to write to a file or update a directory. Uses the current SKILL path for relative paths. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

S_name Name of a file or directory you want to find out your write permission on.

tl_path List of paths to search that overrides the SKILL path.

Value Returned

t If S_name exists and you have permission to write or update it.

nil The file does not exist or does exist, but you do not have permission to read it.

Example

isWritable("/tmp") => t
isWritable("~/test/out.1") => nil

Result if out.1 does not exist or there is no write permission to it.

Reference

isExecutable, isFile, isReadable
**lambda**

\[
\text{lambda}(
\quad ( s\_formalArgument )
\quad g\_expr1 ...
\quad )
\quad \Rightarrow U\_result
\]

**Description**

Defines a function without a name. This is a syntax form.

The keywords `lambda` and `nlambda` allow functions to be defined without having names. This is useful for writing temporary or local functions. In all other respects `lambda` is identical to the `procedure` form.

**Arguments**

- `s_formalArgument` Formal argument for the function definition.
- `g_expr1` SKILL expression to be evaluated when the function is called.

**Value Returned**

- `U_result` A function object.

**Example**

\[
(\text{lambda}( (x \; y) \; \; \; x + y ) \; \; 5 \\; 6) \\
\Rightarrow 11
\]

**Reference**

- `apply`, `nlambda` - SKILL mode only, `nprocedure` - SKILL mode only, `putd`, `procedure`
last

last (
  _l_arg
)
=> _l_result

Description

Returns the last list cell in a list.

Arguments

_l_arg  List of elements.

Value Returned

_l_result  Last list cell (not the last element) in _l_arg.

Example

last( '(a b c) ) => (c)
z = '(1 2 3)
last(z) => (3)
last( '(a b c (d e f))) => ((d e f))

Reference

car, cdr, list, listp
lconc

lconc(
   _tconc
   _list
)
=> _result

Description

Uses a tconc structure to efficiently splice a list to the end of another list.

See the example below.

Arguments

_l_tconc A tconc structure that must initially be created using the tconc function.

_l_list List to be spliced onto the end of the _tconc structure.

Value Returned

_l_result Returns _l_tconc, which must be a tconc structure, with the list _l_list spliced in at the end.

Example

x = tconc(nil 1) ; x is initialized ((1) 1)
lconc(x '(2 3 4)) ; x is now ((1 2 3 4) 4)
lconc(x nil) ; Nothing is added to x.
lconc(x '(5)) ; x is now ((1 2 3 4 5) 5)
x = car( x ) ; x is now (1 2 3 4 5)

Reference

append, tconc
**leftshift**

```plaintext
leftshift(
  x_val
  x_num
)
=> x_result
```

**Description**

Returns the integer result of shifting a value a specified number of bits to the left. Prefix form of the `<<` arithmetic operator. Note that `leftshift` is logical (that is, vacated bits are 0-filled).

**Arguments**

- `x_val` Value to be shifted.
- `x_num` Number of bits `x_val` is shifted.

**Value Returned**

`x_result` Result of the operation.

**Example**

```plaintext
leftshift(7 2) => 28  
leftshift(10 1) => 20
```

**Reference**

`rightshift`
### length

```
length(
    lao_arg
)
=> x_result | 0
```

### Description

Determines the length of a list, array, or association table.

### Arguments

- **lao_arg**: SKILL list, array, or association table.

### Value Returned

- **x_result**: Length of the `lao_arg` object. (The length is either the number of elements in the list or array or the number of key/value pairs in the association table).

- **0**: `lao_arg` is `nil` or an empty array or table.

### Example

- `length( '(a b c d) )` => 4
- `z = '(1 2 3)` => (1 2 3)
- `length( z )` => 3
- `declare(a[11])` => 11
- `length( a )` => 11
- `myTable = makeTable( "atable" 0)` => table:atable
- `myTable[ 'one' ] = "blue"` => "blue"
- `myTable[ "two"] = '(red)` => (r e d)
- `length(myTable)` => 2

### Reference

- `declare`, `list`, `makeTable`, `strlen`
leqp

leqp(
    n_num1
    n_num2
)
=> t | nil

Description

This predicate function checks if the first argument is less than or equal to the second argument. Prefix form of the <= operator.

Arguments

n_num1            Number to be checked.
n_num2            Number against which n_num1 is checked.

Value Returned

t            n_num1 is less than or equal to n_num2.
nil            n_num1 is greater than n_num2.

Example

leqp(2.2)  => t
leqp(-2 2) => t
leqp(3 2.2) => nil

Reference

ggeqp, greaterp, lessp
**lessp**

`lessp(n_num1 n_num2)`

=> t | nil

**Description**

This predicate function checks if the first argument is less than the second argument. Prefix form of the `<` operator.

**Arguments**

- `n_num1`: Number to be checked.
- `n_num2`: Number against which `n_num1` is checked.

**Value Returned**

- `t`: `n_num1` is less than `n_num2`.
- `nil`: `n_num1` is greater than or equal to `n_num2`.

**Example**

```
lessp(2 2) => nil
lessp(-2 2) => t
lessp(3 2.2) => nil
```

**Reference**

`geqp`, `greaterp`, `leqp`
let - SKILL mode

let(
    l_bindings
    g_expr1 ...
  )
=> g_result

Description

Provides a faster alternative to prog for binding local variables only. This is a syntax form.

l_bindings is either a list of variables or a list of the form (s_variable g_value). The bindings list is followed by one or more forms to be evaluated. The result of the let form is the value of the last g_expr.

let is preferable to prog in all circumstances where a single exit point is acceptable, and where the go and label constructs are not required.

Arguments

l_bindings          Local variable bindings, can either be bound to a value or nil (the default).

g_expr1             Any number of expressions.

Value Returned

g_result            The result of the last expression evaluated.

Example

x = 5
let( ((x '(a b c)) y)
    println( y ) ; Prints nil.
  x)
=> (a b c) ; Returns the value of x.

procedure( test( x y )
    let( ((x 6) (z "return string"))
        if( (equal x y)
            then z
            else nil)))
test( 8 6 )
=> "return string"

; Call function test.
; z is returned because 6 == 6.

Reference

procedure, prog
let - SKILL++ mode

```
let(
    [ s_var ]
    (  
      (s_var1  
      s_initExp1  
    )  
      (s_var2  
      s_initExp2  
    )  ...  
    ) body  
)  
=> g_result
```

**Description**

Declares a lexical scope in SKILL++ mode. This includes a collection of local variables, as well as body expressions to be evaluated. This becomes a named let if the optional `s_var` is given.

let, letseq and letrec give SKILL++ a block structure. The syntax of the three constructs is similar, but they differ in the regions they establish for their variable bindings.

- In a `let` expression, the initial values are computed before any of the variables become bound.
- In a `letseq` expression, the bindings and evaluations are performed sequentially.
- In a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

Use the `let` form to declare a collection of local variables. You can provide an initialization expression for each variable. The order of evaluation of the initialization expressions is unspecified. Each variable has the body of the `let` expression as its lexical scope. This means that the initialization expressions should not make cross-references to the other local variables.

In SKILL++ mode, local `defines` can appear at the beginning of the body of a `let`, `letseq`, or `letrec` form.
Arguments

\( s_{\text{var}} \)
When the optional \( s_{\text{var}} \) is given, this becomes a named let. A named let is just like an ordinary let except that \( s_{\text{var}} \) is bound within the body to a function whose formal arguments are the bound variables and whose body is \( body \).

\( s_{\text{var1}} \)
Name of local variable. The variables are bound to fresh locations holding the result of evaluating the corresponding \( initExp \).

\( s_{\text{initExp}} \)
Expression evaluated for the initial value. The \( initExps \) are evaluated in the current environment (in some unspecified order).

\( body \)
A sequence of one or more expressions. The expressions in \( (body) \) are evaluated sequentially in the extended environment. Each local variable binding has \( body \) as its scope.

Value Returned

\( g_{\text{result}} \)
Value of the last expression of \( body \).

Example

```plaintext
let( (( x 2 ) ( y 3 ))
  x*y
) => 6

let( (( x 2 ) ( y 3 ))
  let( (( z 4 ))
    x + y + z
  ) ; let
) ; let
=> 9

let( (( x 2 ) ( y 3 ))
  let( (( x 7 ))
    foo lambda( ( z ) x + y + z )
  ) ; let
) ; let
=> 10 ;not 15

let( ((x 2) (y 3))
define( f(z) x*z+y)
f(5)
```
Reference

begin - SKILL++ mode, define - SKILL++ mode, letrec - SKILL++ mode, letseq - SKILL++ mode
letrec - SKILL++ mode

letrec(
  (  
    s_var1
    s_initExp1
  )
  (  
    s_var2
    s_initExp2
  ) ...
  )
body) => g_result

Description

A letrec expression can be used in SKILL++ mode only. All the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. Use letrec to declare recursive local functions.

Recursive let form. Each binding of a variable has the entire letrec expression as its scope, making it possible to define mutually recursive procedures.

Use letrec when you want to declare recursive local functions. Each initialization expression can refer to the other local variables being declared, with the following restriction: each initialization expression must be executable without actually accessing any of those variables.

For example, a lambda expression satisfies this restriction because its body gets executed only when called, not when it's defined.

Arguments

s_var Name of a local variable. The variables are bound to fresh locations holding undefined values. Each variable is assigned to the result of the corresponding initExp.

s_initExp1 Expressions evaluated for the initial value. The initExps are evaluated in the resulting environment (in some unspecified order).
**body**

A sequence of one or more expressions. The expressions in body are evaluated sequentially in the extended environment.

**Value Returned**

\( g\_result \)

Value of the last expression of body.

**Example**

```plaintext
letrec(
  ;; variable list
  ( f
    lambda( ( n )
      if( n > 0 then n*f(n-1) else 1
    ) ; if
    ) ; lambda
  ) ; f
) ; variable list
f( 5 )
);
letrec

=> 120
```

This example declares a single recursive local function. The local function \( f \) computes the factorial of its argument. The letrec expression returns the factorial of 5.

**Reference**

begin - SKILL++ mode, define - SKILL++ mode, let - SKILL++ mode, letseq - SKILL++ mode
letseq - SKILL++ mode

```scheme
letseq(
(  
  (  
    s_var1
    initExp1
  )  
  (  
    s_var2
    initExp2
  ) ...  
)  
body)
=> g_result
```

Description

A `letseq` expression can be used in `SKILL++ mode only`. The bindings and evaluations are performed sequentially.

Use `letseq` to control the order of evaluation of the initialization expressions. `letseq` is similar to `let`, but the bindings are performed sequentially from left to right, and the scope of a binding indicated by `var1 initExp1` is that part of the `letseq` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

This form is equivalent to a corresponding sequence of nested `let` expressions. It is also equivalent to `let*` is the standard Scheme syntax. This function is equivalent of `let\*()` but it is strongly recommended using this function over `let\*()`.

Arguments

- `s_var` Name of a local variable. Each variable is assigned to the result of the corresponding `initExp`.
- `initExp` Expressions evaluated for the initial value. The `initExps` are evaluated sequentially in the environments that result from previous bindings.
- `body` A sequence of one or more expressions.
Value Returned

\[ g_{\text{result}} \]

Value of the last expression of \( body \).

Example

\[
\text{letseq( ( ( x 1 ) ( y x+1 ) ) } \\
\hspace{1cm} \text{y} \hspace{1cm} \text{letseq} \\
\hspace{2cm} \Rightarrow \text{2}
\]

The code above is a more convenient equivalent to the code below in which you control the sequence explicitly by the nesting.

\[
\text{let( ( ( x 1 ) } \\
\hspace{1cm} \text{let( ( ( y x+1 ) ) } \\
\hspace{2cm} \text{y} \\
\hspace{3cm} ) \\
\hspace{2cm} ) \\
\hspace{1cm} )}
\]

Reference

begin - SKILL++ mode, define - SKILL++ mode, let - SKILL++ mode, letrec - SKILL++ mode
lineread

lineread(
    [ p_inputPort ]
)
=> t | nil | l_results

Description

Parses the next line in the input port into a list that you can further manipulate. It is used by the interpreter’s top level to read in all input and understands SKILL and SKILL++ syntax.

Only one line of input is read in unless there are still open parentheses pending at the end of the first line, or binary infix operators whose right-hand argument has not yet been supplied, in which case additional input lines are read until all open parentheses have been closed and all binary infix operators satisfied. The symbol t is returned if lineread reads a blank input line and nil is returned at the end of the input file.

Arguments

p_inputPort
Input port. The default is piport.

Value Returned

t
If the next line read in is blank.

nil
If the input port is at the end of file.

l_results
Otherwise returns a list of the objects read in from the next (logical) input line

Example

lineread(piport) ; Reads in the next input expression
f 1 2 +
3
=> (f 1 (2 + 3)) ; First input line of the file being read
; Second input line
lineread(piport)
f(a b c)
=> ((f a b c)) ; Another input line of the file
; Returns a list of input objects
Reference

gets, infile, linereadstring
linereadstring

linereadstring(
    t_string
)
    => g_value | nil

Description

Executes lineread on a string and returns the first form read in. Anything after the first form is ignored.

Arguments

$t_string$ Input string.

Value Returned

$g_value$ The first form (line) read in from the argument string.

nil No form is read (that is, the argument string is all spaces).

Example

linereadstring "abc" => (abc)
linereadstring "f a b c" => (f a b c)
linereadstring "x + y" => ((x + y))
linereadstring "f a b c\n g 1 2 3" => (f a b c)

In the last example, only the first form is read in.

Reference

evalstring, gets, instring, lineread
list

list(
  [ g_arg1
    g_arg2 ... ]
) => l_result | nil

Description

Creates a list with the given elements.

Arguments

g_arg1 Element to be added to a list.
g_arg2 Additional elements to be added to a list

Value Returned

l_result List whose elements are g_arg1, g_arg2, and so on.
nil No arguments are given.

Example

list(1 2 3) => (1 2 3)
list('a 'b 'c) => (a b c)

Reference

car, cdr, cons, listp, tconc
**listp**

```
listp(
    g_value
)
=> t | nil
```

**Description**

Checks if an object is a list.

The suffix `p` is usually added to the name of a function to indicate that it is a predicate function.

**Arguments**

`g_value`  
A data object.

**Value Returned**

`t`  
If `g_value` is a list, a data type whose internal name is also `list`. Note that `listp(nil)` returns `t`.

`nil`  
Otherwise.

**Example**

```
listp(['1 2 3]) => t
listp( nil ) => t
listp( 1 ) => nil
```

**Reference**

`atom`, `list`, `null`
**listToVector**

```plaintext
listToVector(
    l_list
) => a_vectorArray
```

**Description**

Returns a vector (array) filled with the elements from the given list.

A vector is represented by an array.

**Arguments**

- `l_list` A list whose elements will be stored in consecutive entries in the vector.

**Value Returned**

- `a_vectorArray` Vector filled with the elements from the given list.

**Example**

```plaintext
V = listToVector( '( 1 2 3 ) ) => array[3]:1954920
V[0] => 1
V[1] => 2
V[2] => 3
```

**Reference**

- `declare`, `vector`, `makeVector`, `vectorToList`
**load**

```perl
load(
    t_fileName
    [ t_password ]
) => t
```

**Description**

Opens a file, repeatedly calls `lineread` to read in the file, immediately evaluating each form after it is read in. Uses the file extension to determine the language mode (`.il` for SKILL and `.ils` for SKILL++) for processing the language expressions contained in the file. For a SKILL++ file, the loaded code is always evaluated in the top level environment.

It closes the file when end of file is reached. Unless errors are discovered, the file is read in quietly. If `load` is interrupted by pressing `Control-c`, the function skips the rest of the file being loaded.

This function uses the file extension to determine the language mode (`.il` for SKILL and `.ils` for SKILL++) for processing the language expressions contained in the file.

SKILL has an autoload feature that allows applications to load functions into SKILL on demand. If a function being executed is undefined, SKILL checks to see if the name of the function (a symbol) has a property called `autoload` attached to it. If the property exists, its value, which must be either a string or an expression that evaluates to a string, is used as the name of a file to be loaded. The file should contain a definition for the function that triggered the autoload. Execution proceeds normally after the function is defined.

**Arguments**

- **t_fileName**
  - File to be loaded. Uses the file name extension to determine the language mode to use.
  - Valid values:
    - `'ils` Means the file contains SKILL++ code.
    - `'il` Means the file contains SKILL code.

- **t_password**
  - Password, if `t_fileName` is an encrypted file.
Value Returned

t

The file is successfully loaded.

Example

```skill
load( "testfns.il" ) ; Load file testfns.il
fn.autoload = "myfunc.il" ; Declares an autoload property.
fn(1)
```

`fn` is undefined at this point, so this call triggers an autoload of `myfunc.il`, which contains the definition of `fn`. The function call `fn(1)` is then successfully performed.

```skill
fn(2) ; fn is now defined and executes normally.
```

You might have an application partitioned into two files. Assume that `UtilsA.il` contains classic SKILL code and `UtilsB.ils` contains SKILL/SKILL++ code. The following example loads both files appropriately.

```skill
procedure( trLoadSystem() 
    load( "UtilsA.il" ) ;;; SKILL code
    load( "UtilsB.ils" ) ;;; SKILL++ code
) ; procedure
```

Reference

`include`, `loadContext`, `loadi`, `lineread`
**loadi**

```javascript
loadi(
    t_fileName
    [ t_password ]
) => t
```

**Description**

Identical to `load`, except that `loadi` ignores errors encountered during the load, prints an error message, and then continues loading.

Opens the named file, repeatedly calls `lineread` to read in the file, immediately evaluates each form after it is read in, then closes the file when end of file is reached. Unlike `load`, `loadi` ignores errors encountered during the load. Rather than stopping, `loadi` causes an error message to be printed and then continues to end of file. Otherwise, `loadi` is the same as `load`.

**Arguments**

- `t_fileName`  
  File to be loaded, with the proper extension to specify the language mode.

- `t_password`  
  Password, if `t_fileName` is an encrypted file.

**Value Returned**

`t`  
Always returns `t`.

**Example**

```javascript
loadi( "testfns.il" )
```

Loads the `testfns.il` file.

```javascript
loadi( "/tmp/test.il"
```

Loads the `test.il` file from the `tmp` directory.
Reference
encrypt, include, load, lineread
loadstring

loadstring(
    t_string
    [ s_langMode ]
) => t

Description

Opens a string for reading, then parses and executes expressions stored in the string, just as load does in loading a file.

**Note:** loadstring is different from evalstring in two ways: (1) it uses lineread mode, and (2) it always returns t if it evaluates successfully.

Arguments

- **t_string**: Input string to be evaluated.
- **s_langMode**: Must be a symbol.

Valid values:

- 'ils: Means the file contains SKILL++ code.
- 'il: Means the file contains SKILL code.

Value Returned

- **t**: When t_string has been successfully read in and evaluated.

  Signals an error if t_string is not a string, or contains ill-formed SKILL expressions.

Example

```
loadstring "1+2" => t
loadstring "procedure( f(y) x=x+y )" => t
loadstring "x=10\n f 20\n f 30" => t
x => 60
```
Reference

evalstring, instring, load, gets
log

log(
    n_number
)
=> f_result

Description

Returns the natural logarithm of a floating-point number or integer.

Arguments

n_number
Floating-point number or integer.

Value Returned

f_result
Natural logarithm of the value passed in.

If the value of n_number is not a positive number, an error is signaled.

Example

log( 3.0 ) => 1.098612

Reference

exp, sqrt
log10

log10(
    n_number
)
=> f_result

Description

Returns the base 10 logarithm of a floating-point number or integer.

Arguments

n_number  Floating-point number or integer.

Value Returned

f_result  Base 10 logarithm of the value passed in.

If the value of n_number is not a positive number, an error is signaled.

Example

log10( 10.0 ) => 1.0
log10( -20.0 )
*Error* log10: argument must be positive - -20

Reference

log, sqrt
lowerCase

lowerCase(
    S_string
)
=> t_result

Description

Returns a string that is a copy of the given argument with uppercase alphabetic characters replaced by their lowercase equivalents.

If the parameter is a symbol, the name of the symbol is used.

Arguments

S_string Input string or symbol.

Value Returned

t_result Copy of S_string in lowercase letters.

Example

lowerCase("Hello World!") => "hello world!"

Reference

upperCase
**make_<name>**

```
make_<name>(
    ...,
) => r_defstruct
```

**Description**

Creates an instance of a `defstruct` specified by `<name>`.

**Arguments**

... Initial values for structure elements (slots).

**Value Returned**

`r_defstruct` Copy of the given instance

**Example**

```#
defstruct(myStruct a b c) => t
m1 = make_myStruct(\ ?a 3 \ ?b 2 \ ?c 1) => array[5]:3436504
m2 = copy_myStruct(m1) => array[5]:3436168
```

**Reference**

`copy <name>`, `copyDefstructDeep`, `defstruct`, `printstruct`, `defstructp`
makeTable

makeTable(
    S_name
    [  
        g_default_value ]
) => o_table

Description

Creates an empty association table.

Arguments

S_name  
Print name (either a string or symbol) of the new table.

g_default_value  
Default value to be returned when references are made to keys that are not in the table. If no default value is given, the system returns unbound if the key is not defined in the table.

Value Returned

o_table  
The new association table.

Example

myTable = makeTable("atable1" 0) => table:atable1  
myTable[1] => 0

If you specify a default value when you create the table, the default value is returned if a nonexistent key is accessed.

myTable2 = makeTable("atable2") => table:atable2  
myTable2[1] => unbound

If you do not specify a default value when you create the table, the symbol unbound is returned if an undefined key is accessed.

myTable[1] = "blue" => blue  
myTable["two"] = '(r e d) => (r e d)  
myTable["three"] = 'green => green

You can refer to and set the contents of an association table with the standard syntax for accessing array elements.
myTable['three'] => green

Reference

declare
**makeTempFileName**

makeTempFileName(
    _S_nameTemplate_
) => _t_name_

**Description**

Appends a string suffix to the last component of a path template so that the resulting composite string does not duplicate any existing file name.

That is, it checks that such named file does not exist. SKILL path is not used in this checking.

**Note:** Successive calls to `makeTempFileName` return different results only if the first name returned is actually used to create a file in the same directory before a second call is made.

The last component of the resultant path is guaranteed to be no more than 14 characters. If the original template has a long last component it is truncated from the end if needed. Also, any trailing X’s (uppercase only) are removed from the template before the new string suffix is appended. You are encouraged to follow the convention of placing temporary files in the `/tmp` directory on your system.

**Arguments**

_S_nameTemplate_ Template file name as a string or a symbol.

**Value Returned**

_t_name_ Path that can be used to create a file or directory.

**Example**

d = makeTempFileName("/tmp/testXXXX") => "/tmp/testa00324"

Trailing X’s (uppercase only) are removed.

createDir(d) => t

The name is used this time.

makeTempFileName("/tmp/test") => "/tmp/testb00324"

A new name is returned this time.
**makeVector**

```skill
makeVector(
    x_size
    [ g_init_val ]
) => a_vectorArray
```

**Description**

Creates an array (vector) with the specified number of elements, and optionally initializes each entry.

Allocates a vector of `x_size` number of entries. `makeVector` initializes each entry in the vector with `g_init_val`. The default value of `g_init_val` is the symbol `unbound`.

**Arguments**

- `x_size` Size of the vector to be allocated.
- `g_init_val` Initial value of each entry of the vector to be allocated.

**Value Returned**

- `a_vectorArray` Array of the given size.

**Example**

```skill
V = makeVector( 3 0 ) => array[3]:1955240
V[0] => 0
V[1] => 0
V[2] => 0
```

**Reference**

- `listToVector`
map

map(
    u_func
    l_arg1
    [ l_arg2 ... ]
)
=> l_arg1

Description

Applies the given function to successive sublists of the argument lists and returns the first argument list. All of the lists should have the same length. This function is not the same as the standard Scheme map function. To get the behavior of the standard Scheme map function, use mapcar instead.

Note: This function is usually used for its side effects, not its return value (see mapc).

Caution

This function is not the same as the standard Scheme map function. To get the behavior of the standard Scheme map function, use mapcar instead.

Arguments

u_func Function to apply to successive sublists. Must be a function that accepts lists as arguments.

l_arg1 Argument list.

l_arg2 Additional argument lists, which must be the same length as l_arg1.

Value Returned

l_arg1 The first argument list.

Example

map( 'list '(1 2 3) '(9 8 7) )
=> (1 2 3)
No interesting side effect.

```
map( '(lambda (x y) (print (append x y))) '(1 2 3) '(9 8 7) )
(1 2 3 9 8 7) (2 3 8 7) (3 7)
=> (1 2 3)
```

Prints three lists as a side effect and returns the list (1 2 3).

**Reference**

*apply*, *foreach*, *mapc*, *mapcar*, *mapcan*, *maplist*
mapc

mapc(
    u_func
    l_arg1
    [ l_arg2 ... ]
)
=> l_arg1

Description

Applies a function to successive elements of the argument lists and returns the first argument list. All of the lists should have the same length. mapc returns l_arg1.

mapc is primarily used with a u_func that has side effects, because the values returned by the u_func are not preserved. u_func must be an object acceptable as the first argument to apply and it must accept as many arguments as there are lists. It is first passed the car of all the lists given as arguments. The elements are passed in the order in which the lists are specified. The second elements are passed to u_func, and so on until the last element.

Arguments

u_func Function to apply to argument lists.
l_arg1 Argument list.
l_arg2 Additional argument lists, which must be the same length as l_arg1.

Value Returned

l_arg1 The first argument list.

Example

mapc( 'list '(1 2 3) '(9 8 7) ) => (1 2 3)
mapc( '(lambda (x y) (print (list x y))) '(1 2 3) '(9 8 7) )
(1 9) (2 8) (3 7) => (1 2 3)

Prints three lists as a side effect and returns the list (1 2 3).
SKILL Language Reference
SKILL Language Functions

Reference

foreach, map, mapcar, mapcan, maplist
mapcan

mapcan(
    u_func
    l_arg1
    [ l_arg2 ... ]
)
=> l_result

Description

Applies a function to successive *elements* of the argument lists and returns the result of appending these intermediate results. All of the lists should have the same length.

Specifically, a function is applied to the *car* of all the argument lists, passed in the same order as the argument lists. The second elements are processed next, continuing until the last element is processed. The result of each call to *u_func* must be a list. These lists are concatenated using *nconc* and the resulting list of all the concatenations is the result of *mapcan*. The argument *u_func* must accept as many arguments as there are lists.

Arguments

- *u_func* Function to apply to argument lists.
- *l_arg1* Argument list.
- *l_arg2* Additional argument lists, which must be the same length as *l_arg1*.

Value Returned

- *l_result* List consisting of the concatenated results.

Example

```
mapcan( 'list '(1 2 3) '(a b c) )
=> (1 a 2 b 3 c)
mapcan( (lambda (n) (and (plusp n) (list n))) '(1 -2 3 -4 5))
=> (1 3 5)
```
Reference

map, mapc, mapcan, mapcar, maplist, nconc
mapcar

mapcar(
   u_func
   l_arg1
   [ l_arg2 ... ]
)
=> l_result

Description

Applies a function to successive elements of the argument lists and returns the list of the corresponding results. All of the lists should have the same length.

The values returned from successive calls to u_func are put into a list using the list function.

Arguments

u_func Function to be applied to argument lists. The result of each call to u_func can be of any data type.

l_arg1 Argument list.

l_arg2 Additional argument lists, which must be the same length as l_arg1.

Value Returned

l_result A list of results from applying u_func to successive elements of the argument list.

Example

mapcar( 'plus ' '(1 2 3) ' '(9 8 7) )
=> (10 10 10)

mapcar( 'list ' '(a b c) ' '(1 2 3) ' '(x y z) )
=> ((a 1 x) (b 2 y) (c 3 z))

mapcar( 'lambda (x) plus (x 1) ' '(2 4 6) )
=> (3 5 7)
Reference

list, map, mapc, mapcan, maplist
maplist

maplist(
      u_func
     l_arg1
     [ l_arg2 ... ]
  )
=> l_result

Description

Applies a function to successive sublists of the argument lists and returns a list of the corresponding results. All of the lists should have the same length.

The returned values of the successive function calls are concatenated using the function list.

Arguments

u_func Function to be applied to argument lists. Must accept lists as arguments. The result of calling u_func can be of any data type.

l_arg1 Argument list.

l_arg2 Additional argument lists, which must be the same length as l_arg1.

Value Returned

l_result A list of the results returned from calling u_func on successive sublists of the argument list.

Example

maplist( 'length '(1 2 3) )
=> (3 2 1)

maplist( 'list '(a b c) '(1 2 3) )
=> (((a b c)(1 2 3))((b c)(2 3))((c)(3)))
Reference

list, map, mapc, mapcar, mapcan
max

max(
  n_num1
  n_num2
  [ n_num3 ... ]
) => n_result

Description

Returns the maximum of the values passed in. Requires a minimum of two arguments.

Arguments

n_num1  First value to check.
n_num2  Next value to check.
n_num3  Additional values to check.

Value Returned

n_result  Maximum of the values passed in.

Example

max(3 2 1) => 3
max(-3 -2 -1) => -1

Reference

abs, min, numberp
**measureTime**

```skill
measureTime(
    g_expression ...
) => l_result
```

**Description**

Measures the time needed to evaluate an expression and returns a list of four numbers. This is a syntax form.

- The first number is the amount of user CPU time in seconds devoted to the process.
- The second number is the amount of CPU time used by the kernel for the process.
- The third and most significant number is the total elapsed time it took to evaluate the expression in seconds.
- The fourth number is the number of page faults that occurred during the evaluation of the expression.

**Arguments**

- `g_expression` Expression(s) to be evaluated and timed.

**Value Returned**

- `l_result` Returns the elapsed time and number of page faults to evaluate `g_expression`.

**Example**

```skill
myList = nil ; Initializes the variable myList.
measureTime( for( i 1 10000 myList = cons(i myList) ) ) => (0.4 0.05 0.4465 0)
```

Result indicates that it took .4 seconds and 0 page faults to build a list from 1 to 10,000 using `cons`.

```skill
myList = nil ; Initializes the variable myList.
measureTime( for( i 1 1000 myList = append1(myList i) ) ) => (5.04 0.03 5.06 0)
```
Result indicates that it took 5 seconds and 0 page faults to build a list from 1 to 1000 using `append1`.

**Reference**

`compareTime`, `getCurrentTime`
member, memq, memv

member(
    g_obj
    l_list
)
=> l_sublist | nil

Description

Returns the largest sublist of l_list whose first element is g_obj. For comparison, member uses the equal function, memq uses the eq function, and memv uses eqv.

memq should only be used when comparing symbols and lists. See eq for restrictions on when eq based comparisons can be used.

Note: It is faster to convert a string to a symbol using concat in conjunction with memq than to simply use member, which performs a comparison using equal which is slower, especially for large lists. These functions return a non-nil value if the first argument matches a member of the list passed in as the second argument.

Arguments

g_obj Element to be searched for in l_list.
l_list List to search.

Value Returned

l_sublist The part of l_list beginning with the first match of g_obj.
nil If g_obj is not in the top level of l_list.

Example

x = "c"
member( x '("a" "b" "c" "d") ) => ("c" "d")
memq( 'c ' (a b c d c d) ) => (c d c d)
memq( concat( x ) ' (a b c d ) ) => (c d)
memv( 1.5 ' (a 1.0 1.5 "1.5") ) => (1.5 "1.5")
Reference

eg, equal, eqv, concat
min

\[
\text{min(}
\quad n\_num1
\quad n\_num2
\quad [\ n\_num3 \ldots \ ]
\quad )
\quad \Rightarrow \ n\_result
\]

Description

Returns the minimum of the value passed in. Requires a minimum of two arguments.

Arguments

\begin{align*}
\text{n\_num1} & & \text{First value to check}. \\
\text{n\_num2} & & \text{Next value to check}. \\
\text{n\_num3} & & \text{Additional values to check}. \\
\end{align*}

Value Returned

\begin{align*}
\text{n\_result} & & \text{Minimum of the values passed in}. \\
\end{align*}

Example

\begin{align*}
\text{min}(1 \ 2 \ 3) & \Rightarrow 1 \\
\text{min}(-1 \ -2.0 \ -3) & \Rightarrow -3.0 \\
\end{align*}

Reference

\begin{align*}
\text{abs, max, numberp} \\
\end{align*}
minus

minus(
    n_op
)
=> n_result

Description

Returns the negative of a number. Prefix form of the – unary operator.

Arguments

n_op  A number.

Value Returned

n_result  Negative of the number.

Example

minus( 10 ) => -10
minus( -1.0 ) => 1.0
minus( -0 ) => 0
minusp

minusp(  
    n_num
  )
  => t | nil

Description

Checks if a value is a negative number. Same as negativep.

Arguments

n_num    Number to check.

Value Returned

t          If n_num is a negative number.
nil       Otherwise.

Example

minusp( 3 ) => nil
minusp( -3 ) => t

Reference

evenp, negativep, numberp, oddp, onep, plusp, zerop
mod

mod(
    x_integer1
    x_integer2
 )
 => x_result

Description

Returns the integer remainder of dividing two integers. The remainder is either zero or has the sign of the dividend.

This function is equivalent to remainder.

Arguments

x_integer1   Dividend.

x_integer2   Divisor.

Value Returned

x_result   Integer remainder of the division. The sign is determined by the dividend.

Example

mod(4 3) => 1

Reference

fixp, modulo, remainder
modf

\[
\text{modf}( \quad f\_{\text{flonum1}} \quad f\_{\text{flonum2}} \quad ) \\
\Rightarrow f\_{\text{result}}
\]

**Description**

Returns the floating-point remainder of the division of \( f\_{\text{flonum1}} \) by \( f\_{\text{flonum2}} \).

**Arguments**

\[
\begin{align*}
\text{f\_{flonum1}} & \quad \text{A floating-point number (Dividend).} \\
\text{f\_{flonum2}} & \quad \text{A floating-point number (Divisor).}
\end{align*}
\]

**Value Returned**

\[
\begin{align*}
f\_{\text{result}} & \quad \text{Floating-point remainder of the division.} \\
\text{The sign is determined by the dividend.}
\end{align*}
\]

**Example**

\[
\begin{align*}
\quad \text{;; Sign is determined by the dividend} \\
\quad \text{modf(-10.1 10.0) => -0.1} \\
\quad \text{modf(10.1 -10.0) => 0.1}
\end{align*}
\]
modulo

modulo(
    x_integer1
    x_integer2
)
=> x_integer

Description

Returns the remainder of dividing two integers. The remainder always has the sign of the divisor.

The remainder (mod) and modulo functions differ on negative arguments. The remainder is either zero or has the sign of the dividend if you use the remainder function. With modulo the return value always has the sign of the divisor.

Arguments

x_integer1  Dividend.

x_integer2  Divisor.

Value Returned

x_integer  The remainder of the division. The sign is determined by the divisor.

Example

modulo( 13 4)  => 1
remainder( 13 4)  => 1

modulo( -13 4)  => 3
remainder( -13 4)  => -1

modulo( 13 -4)  => -3
remainder( 13 -4)  => 1

modulo( -13 -4)  => -1
remainder( -13 -4)  => -1
Reference

mod, remainder
mprocedure

mprocedure(
    s_macroName(
        s_formalArgument
    )
    g_expr1 ...
)
=> s_funcName

Description

Defines a macro with the given name that takes a single formal argument. This is a syntax form.

The body of the macro is a list of expressions to be evaluated one after another. The value of the last expression evaluated is considered the result of macro expansion and is evaluated again to get the actual value of the macro call.

When a macro is called, s_formalArgument is bound to the entire macro call form, that is, a list with the name of the macro as its first element followed by the unevaluated arguments to the macro call.

Macros in SKILL are completely general in that a macro body can call any other function to build an expression that is to be evaluated again.

**Note:** A macro call within a function definition is expanded only once, when the function is compiled. For this reason, be cautious when defining macros. Make sure they are purely functional, that is, side-effects free. You can use expandMacro to verify the correct behavior of a macro definition.

Arguments

- **s_macroName**
  Name of the macro function.

- **s_formalArgument**
  Formal arguments for the macro definition.

- **g_expr1**
  A SKILL expression.

Value Returned

- **s_funcName**
  Name of the macro defined.
Example

```lisp
mprocedure( whenNot(callForm)
   'if !,(cadr callForm) then ,(cddr callForm))
=> whenNot
expandMacro( 'whenNot x>y z=f(y) x*z)
=> if(!(x>y) then (z=f(y)) (x*z))
whenNot(1>2 "Good")
=> "Good"
```

Reference

`defmacro`
nconc

nconc(
  l_arg1
  l_arg2
  [ l_arg3 ... ]
)
=> l_result

Description

Equivalent to a destructive append where the first argument is actually modified.

This results in nconc being much faster than append but not quite as fast as tconc and lconc. Thus nconc returns a list consisting of the elements of l_arg1, followed by the elements of l_arg2, followed by the elements of l_arg3, and so on. The cdr of the last list cell of l_arg_i is modified to point to l_arg_{i+1}. Thus caution must be taken because if nconc is called with the l_arg_i two consecutive times it can form an infinite structure where the cdr of the last list cell of l_arg_i points to the car of l_arg_i.

Use the nconc function principally to reduce the amount of memory consumed. A call to append would normally duplicate the first argument whereas nconc does not duplicate any of its arguments, thereby reducing memory consumption.

Arguments

l_arg1 List of elements.

l_arg2 List elements concatenated to l_arg1.

l_arg3 Additional lists.

Value Returned

l_result The modified value of l_arg1.

Example

x = '(a b c)
nconc( x '(d)) ; x is now (a b c d)
nconc( x '(e f g) ) ; x is now the list (a b c d e f g)
nconc( x x ) ; Forms an infinite structure.
This forms an infinite list structure (a b c d e f g a b c d e f g ...).

Reference

append, cdr, lcons, tcons
ncons

ncons(
  g_element
)
=> l_result

Description

Builds a list containing an element. Equivalent to cons( g_element nil ).

Arguments

g_element Element to be added to the beginning of an empty list.

Value Returned

l_result A list with g_element as its single element.

Example

ncons( 'a ) => (a)
z = '(1 2 3) => (1 2 3)
ncons( z ) => ((1 2 3))

Reference

cons, list
needNCells

needNCells(
    {s_cellType | S_userType}
    x_cellCount
)
=> t | nil

Description

Ensures that there is enough memory available for the specified number of SKILL objects (cells).

If necessary, more memory is allocated. The name of the user type can be passed in as a string or a symbol, however internal types like list or fixnum must be passed in as symbols.

Arguments

s_cellType Objects of type cellType.
S_userType Objects of type userType.
x_cellCount Number of objects.

Value Returned

t Enough memory is available.
nil Otherwise.

Example

needNCells( 'list 1000 ) => t

Guarantees there will always be 1000 list cells available in the system.

Reference

gc, summary
The `negativep` function checks if a value is a negative number. It is similar to the `minusp` function.

**Arguments**

- `n_num` - Number to check.

**Value Returned**

- `t` - `n_num` is a negative number.
- `nil` - Otherwise.

**Example**

```skill
negativep( 3 ) => nil
negativep( -3 ) => t
```

**Reference**

- `evenp`, `minusp`, `numberp`, `oddp`, `onep`, `plusp`, `zerop`
neq

neq(
    g_arg1
    g_arg2
)
=> t | nil

Description

Checks if two arguments are not identical using the eq function and returns t if they are not. That is, g_arg1 and g_arg2 are tested to see if they are at the same address in memory.

Arguments

g_arg1
Any SKILL object.

g_arg2
Any SKILL object.

Value Returned

t            If g_arg1 and g_arg2 are not eq.

nil          Otherwise.

Example

a = 'dog
neq( a 'dog ) => dog
neq( a 'cat ) => t

z = '(1 2 3)
zeq(z z) => (1 2 3)
zeq('(1 2 3) z) => nil
neq('(1 2 3) z) => t

Reference

eq, equal, nequal
nequal

nequal(
    g_arg1
    g_arg2
)
=> t | nil

Description

Checks if two arguments are *not* logically equivalent using the `equal` function and returns `t` if they are not.

`g_arg1` and `g_arg2` are only equal if they are either `eqv` or they are both lists/strings and their contents are the same.

Arguments

`g_arg1`    Any SKILL object.
`g_arg2`    Any SKILL object.

Value Returned

`t`        If `g_arg1` and `g_arg2` are not equal.
`nil`      Otherwise.

Example

```
x = "cow"            => "cow"
nequal( x "cow" )   => nil
nequal( x "dog" )   => t

z = '(1 2 3)         => (1 2 3)
nequal( z z)        => nil
nequal('(1 2 3) z)  => nil
```

Reference

`neq`, `equal`
**newline**

```skill
newline([ p_outputPort ])
=> nil
```

**Description**

Prints a newline (\n) character and then flushes the output port.

**Arguments**

- `p_outputPort` Output port. Defaults to poport, the standard output port.

**Value Returned**

- `nil` Prints a newline and then returns nil.

**Example**

```skill
print("Hello") newline() print("World!")
"Hello"
"World!"
=> nil
```

**Reference**

- `drain`, `fprintf`, `outfile`
**nindex**

**nindex**

t_string1
S_string2 )
=> x_result | nil

**Description**

Finds the symbol or string, S_string2, in t_string1 and returns the character index, starting from one, of the first point at which the S_string2 matches part of t_string1.

**Arguments**

**t_string1**

String you want to search for S_string2.

**S_string2**

String you want to find occurrences of in t_string1.

**Value Returned**

**x_result**

Index corresponding to the point at which S_string2 matches part of t_string1. The index starts from one.

**nil**

No character match.

**Example**

nindex( "abc" 'b ) => 2
nindex( "abcdabce" "dab" ) => 4
nindex( "abc" "cba" ) => nil

**Reference**

getattr, index, substring
nlambda - SKILL mode only

```
nlambda(
    (  
        s_formalArgument
    )
    g_expr1 ...
)
=> u_result
```

**Description**

 Allows `nlambda` functions to be defined without having names. In all other respects, `nlambda` is identical to `nprocedure`. This is a syntax form that is not supported in SKILL++ mode.

Allowing `nlambda` functions to be defined without having names is useful for writing temporary or local functions. In all other respects `nlambda` is identical to `nprocedure`.

An `nlambda` function should be declared to have a single formal argument. When evaluating an `nlambda` function, SKILL collects all the actual argument expressions unevaluated into a list and binds that list to the single formal argument. The body of the `nlambda` can selectively evaluate the elements of the argument list.

In general, it is preferable to use `lambda` instead of `nlambda` because `lambda` is more efficient. In most cases, `nlambdas` can be easily replaced by macros (and perhaps helper functions).

**Arguments**

- **s_formalArgument** Formal argument for the function definition.
- **g_expr1** SKILL expressions to be evaluated when the function is called.

**Value Returned**

- **u_result** A function object.

**Example**

```
putd('foo nlambda((x) println(x))) => funobj:0x309128
```
apply( nlambda((y) foreach(x y printf(x))) '("Hello" "World\n"))
HelloWorld
=> ("Hello" "World\n")

Reference

apply, lambda, nprocedure - SKILL mode only, procedure, putd
**not**

```
not (  
g_obj  
)  
=> t | nil
```

**Description**

Same as the `!` operator. Returns `t` if the object is `nil`, and returns `nil` otherwise.

**Arguments**

- `g_obj` Any SKILL object.

**Value Returned**

- `t` If `g_obj` is `nil`.
- `nil` Otherwise.

**Example**

```
(not nil)  => t
(not 123)  => nil
(not t)    => nil
```

**Reference**

`null`
nprocedure - SKILL mode only

nprocedure(
    s_funcName(
        s_formalArgument
    )
    g_expr1 ...
)
=> s_funcName

Description

Defines an nlambda function with a function name and a single formal argument. This is a syntax form that is not supported in SKILL++ mode.

The body of the procedure is a list of expressions to be evaluated one after another. The value of the last expression evaluated is returned as the value of the function. There must be no white space separating the s_funcName and the open parenthesis of the list containing s_formalArgument.

An nlambda function defined by nprocedure differs from a lambda function defined by procedure in that an nlambda function does not evaluate its arguments; it binds the whole actual argument list to its single formal argument. lambda functions, on the other hand, evaluate each argument in the actual argument list and bind them one by one to each formal argument on the formal argument list. It is recommended that procedure be used over nprocedure whenever possible, in part because procedure is faster and also offers better type checking.

In general, it is preferable to use lambda instead of nlambda because lambda is more efficient.

Arguments

s_funcName Name of newly defined function.
s_formalArgument Formal argument for the function definition.
g_expr1 SKILL expressions to be evaluated when the function is called.

Value Returned

s_funcName Returns the name of the function defined.
Example

procedure( printarg(x) println(x))
=> printarg

Defines a lambda function.

nprocedure( nprintarg(x) println(x))
=> nprintarg

Defines an nlambda function.

y = 10
=> 10
printarg(y * 2)
20
=> nil

Calls a lambda function. Prints the value 20. println returns nil.

nprintarg(y * 2)
((y * 2))
=> nil

Calls an nlambda function. Prints a list of the unevaluated arguments. println returns nil.

Reference

lambda, nlambda - SKILL mode only, procedure
nth

nth(
    x_index0
    l_list
  )
=> g_result | nil

Description

Returns an index-selected element of a list, assuming a zero-based index.

Thus nth(0 l_list) is the same as car(l_list). The value nil is returned if
x_index0 is negative or is greater than or equal to the length of the list.

Arguments

x_index0  Index of the list element you want returned.

l_list     List of elements.

Value Returned

g_result   Indexed element of l_list, assuming a zero-based index

nil         If x_index0 is negative or is greater than or equal to the length
            of the list.

Example

nth( 1 '(a b c) ) => b
z = '(1 2 3) => (1 2 3)
nth(2 z) => 3
nth(3 z) => nil

Reference

car, list, nthcdr, nthelem
nthcdr

nthcdr(   
  x_count
  l_list
)
=> l_result

Description

Applies cdr to a list a given number of times.

Arguments

x_count Number of times to apply cdr to l_list.

l_list List of elements.

Value Returned

l_result Result of applying cdr to l_list, x_count number of times.

Example

nthcdr( 3 '(a b c d)) => (d)
z = '(1 2 3)
nthcdr(2 z) => (3)
nthcdr(-1 z) => (nil 1 2 3)

If x_count is less than 0, then cons(nil l_list) is returned.

Reference

cdr, cons, nth
**ntelement**

```
ntelement(
   x_index1
   l_list
)
=> g_result | nil
```

**Description**

Returns the indexed element of the list, assuming a one-based index.

Thus `ntelement(1 l_list)` is the same as `car(l_list)`.

**Arguments**

- `x_index1` Index of the element of `l_list` you want returned.
- `l_list` List of elements.

**Value Returned**

- `g_result` The `x_index` element of `l_list`.
- `nil` If `x_index1` is less than or equal to 0 or is greater than the length of the list.

**Example**

```
ntelement(1 '(a b c)) => a
z = '(1 2 3)
nntelement(2 z) => 2
```

**Reference**

`car`, `nth`
null

null(  
    g_value  
)  
=> t | nil

Description

Checks if an object is equal to nil.

null is a type predicate function.

Arguments

g_value  

A data object.

Value Returned

t  

If g_value is equal to nil.

nil  

Otherwise.

Example

null( 3  )  => nil
null('()  )  => t
null( nil)  => t

Reference

atom, listp
numberp

numberp(
  g_value
)
=> t | nil

Description

Checks if a data object is a number, that is, either an integer or floating-point number.

The suffix \( p \) is usually added to the name of a function to indicate that it is a predicate function.

Arguments

\( g\_value \) A data object.

Value Returned

\( t \) The data object is a number.

\( \text{nil} \) Otherwise.

Example

numberp( 3 ) => t
numberp('isASymbol) => nil
numberp( 3.5) => t

Reference

\( \text{fixp, floatp} \)
numOpenFiles

numOpenFiles(
)  
=> ( x_current x_maximum )

Description

Returns the number of files now open and the maximum number of files that a process can open. The numbers are returned as a two-element list.

Arguments

None.

Value Returned

x_current  Number of files that are currently open.

x_maximum  Maximum number of files that a process can open. This is usually platform-dependent.

Example

numOpenFiles()  => (6 64)

Result is system-dependent.

f = infile("/dev/null")  => port:"/dev/null"
numOpenFiles()  => (7 64)

One more file is open now.

Reference

close, infile, outfile
oddp

oddp ( x_num )
   => t | nil

Description

Checks if the value of an integer is odd.

oddp is a predicate function.

Arguments

x_num  An integer.

Value Returned

t  If x_num is an odd integer.

nil  Otherwise.

Example

oddp( 7 )
 => t
oddp( 8 )
 => nil

Reference

evenp, fixp, integerp, minusp, onep, plusp, zerop
onep

onep(  
    \( n\_num \)  
  )  
  => \( t \mid nil \)

Description

Checks if a value is equal to one.

onep is a predicate function.

Arguments

\( n\_num \) Number to check.

Value Returned

\( t \) If \( n\_num \) is equal to one.

\( nil \) Otherwise.

Example

onep( 1 )  
=> \( t \)
onep( 7 )  
=> \( nil \)
onep( 1.0 )  
=> \( t \)

Reference

evenp, minusp, numberp, oddp, plusp, zerop
openportp

openportp(
  g_obj
)
  => t | nil

Description

Checks if the given argument is a port object and it is open (for input or output), nil otherwise.

Arguments

g_obj Any SKILL object.

Value Returned

t If g_obj is a port and it is open for input or output.
nil Otherwise.

Example

(portp ip = (infile "inFile")) => t
(portp op = (outfile "outFile")) => t
(openportp ip) => t
(openportp op) => t
(close ip) => t
(openportp ip) => nil
(close op) => t
(openportp op) => nil
or

or(
    g_arg1
    g_arg2

    [ g_arg3... ]
)

=> nil | g_val

Description

Evaluates from left to right its arguments to see if the result is non-nil. As soon as an argument evaluates to non-nil, or returns that value without evaluating the rest of the arguments. If all arguments except the last evaluate to nil, or returns the value of the last argument as the result of the function call. Prefix form of the || binary operator.

Arguments

- **g_arg1**: First argument to be evaluated.
- **g_arg2**: Second argument to be evaluated.
- **g_arg3**: Optional additional arguments to be evaluated.

Value Returned

- **nil**: All arguments evaluate to nil.
- **g_val**: Value of the argument that evaluates to non-nil, or the value of the last argument if all the preceding arguments evaluate to nil.

Example

- `or(t nil) => t`
- `or(nil t) => t`
- `or(18 12) => 18`

Reference

- and, band, bband, bnor, bnot, bor, bxnor, bxor, not
otherp

otherp(              
   g_value           
)
=> t | nil

Description
Checks if an object is a user type object, such as an association table or a window.
The suffix p is usually added to the name of a function to indicate that it is a predicate function.

Arguments

-g_value A data object.

Value Returned

-t If g_value is a user type object.
-nil Otherwise.

Example

otherp(3.0) => nil
otherp(makeTable("table1" nil)) => t

Reference

type, typep
## outfile

(outfile(
    S_fileName
    [ t_mode ]
    [ g_openHiddenFile ]
)
=> p_outport | nil

### Description

Opens an output port ready to write to a file.

The file can be specified with either an absolute path or a relative path. If a relative path is given and the current SKILL path setting is not nil, all directory paths from SKILL path are checked in order, for that file. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path. If found, the system overwrites the first updatable file in the list. If no updatable file is found, it places a new file of that name in the first writable directory.

If the optional `g_openHiddenFile` argument (which is intended to be used on Windows only) is specified, the system will be forced to open a Windows hidden file. The `g_openHiddenFile` must be used for opening existing Windows hidden files only. If the named Windows hidden file does not exist (including the current SKILL path), outfile will fail. In addition, the `t_mode` option must also be specified (to either `w` or `a` only) if `g_openHiddenFile` is given.

### Arguments

- **S_fileName**
  
  Name of the file to open or create.

- **t_mode**
  
  If the mode string `t_mode` is specified, the file is opened in the mode requested. If `t_mode` is `a`, an existing file is opened in append mode. If it is `w`, a new file is created for writing (any existing file is overwritten). The default is `w`.

- **g_openHiddenFile**
  
  If specified to non-nil, the named Windows hidden file is forced to open. This argument must be used for Windows hidden files only.
Value Returned

$p_{outport}$  An output port ready to write to the specified file.

$nil$  If the named file cannot be opened for writing or the named
Windows hidden file does not exist (including the current SKILL path).

An error is signaled if an illegal mode string is supplied.

Example

$p = \text{outfile("/tmp/out.il" "w")}$  => port:"/tmp/out.il"
outfile("/bin/ls")  => nil

outfile( "aHiddenFile" "w" t)

To force opening a Windows hidden file $t_{mode}$ must also be specified.

Reference

close, drain, getSkillPath, infile
**outportp**

`outportp(  
g_obj  
)  
=> t | nil`

**Description**

Checks if an object is an output port.

**Note:** An output port may be closed, so if `outportp` returns `t`, that does not guarantee a successful write to the port.

**Arguments**

`g_obj` Any SKILL object.

**Value Returned**

`t` The given object is an output port.

`nil` Otherwise.

**Example**

`(outportp poport)  => t  
(outportp piport)  => nil  
(outportp 123)  => nil`

**Reference**

`inportp`
pairp

pairp(
  g_obj
)
=> t | nil

Description

Checks if an object is a cons object, that is, a non-empty list.

This function is equivalent to dtpr.

Arguments

g_obj

Any SKILL object.

Value Returned

t
  g_obj is a cons object.

nil
  g_obj is not a cons object.

Example

(pairp nil) => nil
(pairp 123) => nil
(pairp '(1 2)) => t

Reference

dtpr, listp, null
parseString

parseString(
    S_string
    [ S_breakCharacters ]
)
=> l_strings

Description

Breaks a string into a list of substrings with break characters.

Returns the contents of S_string broken up into a list of words. If the optional second argument, S_breakCharacters, is not specified, the white space characters, \t\f\r\n\v, are used as the default.

A sequence of break characters in S_string is treated as a single break character. By this rule, two spaces or even a tab followed by a space is the same as a single space. If this rule were not imposed, successive break characters would cause null strings to be inserted into the output list.

If S_breakCharacters is a null string, S_string is broken up into characters. You can think of this as inserting a null break character after each character in S_string.

No special significance is given to punctuation characters, so the “words” returned by parseString might not be grammatically correct.

Arguments

S_string String to be parsed.
S_breakCharacters List of individual break characters.

Value Returned

l_strings List of strings parsed from S_string.

Example

parseString( "Now is the time" ) => ("Now" "is" "the" "time")

Space is the default break character
parseString( "prepend" "e" ) => ("pr" "p" "nd" )

`e` is the break character.

parseString( "feed" "e" ) => ("f" "d" )

A sequence of break characters in `S_string` is treated as a single break character.

parseString( "~/exp/test.il" "./" ) => ("~" "exp" "test" "il" )

Both . and / are break characters.

parseString( "abc de" "" ) => ("a" "b" "c" " d" "e" )

The single space between c and d contributes " " in the return result.

Reference

buildString, linereadstring, strcat, strlen, stringp
### plist

```plaintext
plist(
    s_symbolName
)
=> l_propertyList | nil
```

#### Description

Returns the property list associated with a symbol.

From time to time, it is useful to print out the entire property list attached to a given symbol and see what properties have been assigned to the symbol.

#### Arguments

- **s_symbolName**
  
  Name of the symbol.

#### Value Returned

- **l_propertyList**
  
  Property list for the named symbol.

- **nil**
  
  If there is no property list for the named symbol.

#### Example

```plaintext
a.x = 10
a.y = 20
println(plist('a))
(y 20 × 10)
=> nil
```

Prints the property list attached to the symbol a. Returns `nil`, the result of `println`. Notice that a single quote is used in this example. You can think of this as passing in the name of the symbol rather than its value.

#### Reference

- `putprop`
- `setplist`
plus

plus(
    n_op1
    n_op2
    [ n_op3 ... ]
)
=> n_result

Description

Returns the result of adding one or more operands to the first operand. Prefix form of the + arithmetic operator.

Arguments

n_op1  First number to be added.
n_op2  Second number to be added.
n_op3  Optional additional numbers to be added.

Value Returned

n_result  Sum of the numbers.

Example

plus(5 4 3 2 1) => 15
plus(-12 -13) => -25
plus(12.2 13.3) => 25.5

Reference

xplus
plusp

plusp(n_num)

=> t | nil

Description

Checks if a value is a positive number.

plusp is a predicate function.

Arguments

n_num

Floating-point number or integer.

Value Returned

t

If n_num is a positive number.

nil

Otherwise.

Example

plusp(-209.623472)
=> nil
plusp(209.623472)
=> t

Reference

evenp, minusp, numberp, oddp, onep, zerop
portp

```
portp( 
    g_value
)
=> t | nil
```

**Description**

Checks if an object is an input or output port.

The suffix `p` is usually added to the name of a function to indicate that it is a predicate function.

**Arguments**

`g_value` A data object.

**Value Returned**

- `t` If `g_value` is an input or output port, whose type name is `port`.
- `nil` Otherwise.

**Example**

```
portp( piport )  => t
portp( 3.0 )     => nil
```

**Reference**

`infile, outfile`
postdecrement

postdecrement(
    s_var
)
=> n_result

Description

Takes a variable, decrements its value by one, stores the new value back into the variable, and returns the original value. Prefix form of $s--$. The name of the variable must be a symbol and the value must be a number.

Arguments

s_var Variable representing a number.

Value Returned

n_result Original value of the variable.

Example

s = 2
postdecrement( s ) => 2
s => 1
s = 2.2
postdecrement( s ) => 2.2
s => 1.2

Reference

postincrement, predecrement, preincrement
postincrement

```skill
postincrement(
   s_var
)
=> n_result
```

**Description**

Takes a variable, increments its value by one, stores the new value back into the variable, and returns the original value. Prefix form of `s++`. The name of the variable must be a symbol and the value must be a number.

**Arguments**

- `s_var` Variable representing a number.

**Value Returned**

- `n_result` Original value of the variable.

**Example**

```skill
s = 2
postincrement( s ) => 2
s => 3
s = 2.2
postincrement( s ) => 2.2
s => 3.2
```

**Reference**

- `postdecrement`
- `prederecrement`
- `preincrement`
pprint

pprint(
  g_value
  [ p_outputPort ]
)
=> nil

Description

Identical to print except that it pretty prints the value whenever possible.

The pprint function is useful, for example, when printing out a long list where print simply prints the list on one (possibly huge) line but pprint will limit the output on a single line and produce a multiple line printout if necessary. This makes the output much more readable.

pprint does not work the same as the pp function. pp is an nlambda and only takes a function name whereas pprint is a lambda and takes an arbitrary SKILL object.

Arguments

  g_value          Any SKILL value to be printed.

  p_outputPort    Output port to print to. Default is poport.

Value Returned

  nil              Prints the argument value (to the given port).

Example

pprint '(1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j k)
(1 2 3 4
  5
  6 7 8 9
  0
  a b c d e
  f g h i
  j k
)
=> nil

Reference

pp, print
**predecrement**

```plaintext
predecrement(
    s_var
)
=> n_result
```

**Description**

Takes a variable, decrements its value by one, stores the new value back into the variable, and returns the new value. Prefix form of --s. The name of the variable must be a symbol and the value must be a number.

**Arguments**

`s_var` Variable representing a number.

**Value Returned**

`n_result` Decremented value of the variable.

**Example**

```plaintext
s = 2
predecrement( s ) => 1
s => 1
s = 2.2
predecrement( s ) => 1.2
s => 1.2
```

**Reference**

[postdecrement](#), [postincrement](#), [preincrement](#)
**preincrement**

```
preincrement(
    s_var
  )
=> n_result
```

**Description**

Takes a variable, increments its value by one, stores the new value back into the variable, and returns the new value. Prefix form of `++s`. The name of the variable must be a symbol and the value must be a number.

**Arguments**

`s_var` Variable representing a number.

**Value Returned**

`n_result` Incremented value of the variable.

**Example**

```plaintext
s = 2
preincrement( s ) => 3
s => 3
s = 2.2
preincrement( s ) => 3.2
s => 3.2
```

**Reference**

`postdecrement`, `postincrement`, `predecrement`
prependInstallPath

prependInstallPath(
    S_name
) => t_string

Description

Prepends the Cadence installation path to a file or directory and returns the resulting path as a string.

Possibly adds a slash (/) separator if needed. The typical use of this function is to compute one member of a list passed to setSkillPath.

Arguments

S_name File or directory name to append to the installation path. If a symbol is given, its print name is used.

Value Returned

t_string String formed by prepending the installation path to the argument path.

Example

getInstallPath() => "(/usr5/cds/4.2)"
Assume this is your install path.
prependInstallPath( "etc/context" ) => "/usr5/cds/4.2/etc/context"

A slash (/) is added.
prependInstallPath( "/bin" ) => "/usr5/cds/4.2/bin"
setSkillPath( list("." prependInstallPath("bin")
    prependInstallPath("etc/context")) )
=> nil
getSkillPath()
=> ("." "/usr5/cds/4.2/bin" "/usr5/cds/4.2/etc/context")

Reference

getInstallPath, getSkillPath, setSkillPath
print

print(
    g_value
    [ p_outputPort ]
) => nil

Description

Prints a SKILL object using the default format for the data type of the value.

For example, strings are enclosed in double quotes. Same as println, except no newline character is printed.

Arguments

\[
\begin{align*}
g\_value & \quad \text{Any SKILL object.} \\
p\_outputPort & \quad \text{Output port to print to. Default is poport.}
\end{align*}
\]

Value Returned

nil \quad \text{Always returns nil after printing out the object supplied.}

Example

print("hello")
"hello"
=> nil

Reference

pprint, println, printlev
**printf**

```c
printf(
    t_formatString
    [ g_arg1 ... ]
)
=> t
```

**Description**

Writes formatted output to `poport`.

The optional arguments following the format string are printed according to their corresponding format specifications. Refer to the "Common Output Format Specifications" table on the `fprintf` manual page.

`printf` is identical to `fprintf` except that it does not take the `p_port` argument and the output is written to `poport`.

**Arguments**

- `t_formatString` Characters to be printed verbatim, intermixed with format specifications prefixed by the % sign.
- `g_arg1` Arguments following the format string are printed according to their corresponding format specifications.

**Value Returned**

`t` Prints the formatted output and returns `t`.

**Example**

```bash
x = 197.9687 => 197.9687
printf("The test measures %10.2f\n" x)
Prints the following line to `poport` and returns `t`.
The test measures 197.97.
=> t
```
Reference

`fprintf`, `println`
printlev

printlev(
    g_value
    x_level
    x_length
    [ p_outputPort ]
)
=> nil

Description

Prints a list with a limited number of elements and levels of nesting.

Lists are normally printed in their entirety no matter how many elements they have or how deeply nested they are. Applications have the option, however, of setting upper limits on the number of elements and the levels of nesting shown when printing lists. These limits are sometimes necessary to control the volume of interactive output because the SKILL top-level automatically prints the results of expression evaluation. Limits can also protect against the infinite looping on circular lists possibly created by programming mistakes.

Two integer variables, print length and print level (specified by x_length and x_level), control the maximum number of elements and the levels of nesting that are printed. List elements beyond the maximum specified by print length are abbreviated as “...” and lists nested deeper than the maximum level specified by print level are abbreviated as &. Both print length and print level are initialized to nil (meaning no limits are imposed) by SKILL, but each application is free to set its own limits.

The printlev function is identical to print except that it takes two additional arguments specifying the maximum level and length to be used in printing the expression.

Arguments

- **g_value**: Any SKILL value.

- **x_level**: Specifies the level of nesting that you want to print; lists nested deeper than the maximum level specified are abbreviated as “&”.

- **x_length**: Specifies the length (or maximum number of elements) you want to print. List elements beyond the maximum specified here are abbreviated as “...”.

- **p_outputPort**: Output port. Default is poport.
Value Returned

nil

Prints the argument value and then returns nil.

Example

List = '(1 2 (3 (4 (5))) 6)
=> '(1 2 (3 (4 (5))) 6)
printlev(List 100 2)
(1 2 ...)
=> nil

printlev(List 3 100)
(1 2 (3 (4 &)) 6)
=> nil

printlev(List 3 3 p) ; Assumes port p exists.
(1 2 (3 (4 &)) ...)
=> nil

printlev(List 3 3 p) ; Prints to port p.
(1 2 (3 (4 &)) ...)
=> nil

Reference

list, print
println

println(
  g_value
  [ p_outputPort ]
)
=> nil

Description

Prints a SKILL object using the default format for the data type of the value, then prints a newline character.

A newline character is automatically printed after printing \texttt{g\_value}. \texttt{println} flushes the output port after printing each newline character.

Arguments

\texttt{g\_value} \hspace{1cm} \text{Any SKILL value.}

\texttt{p\_outputPort} \hspace{1cm} \text{Port to be used for output. The default is poport.}

Value Returned

\texttt{nil} \hspace{1cm} \text{Prints the given object and returns \texttt{nil}.}

Example

\texttt{for( i 1 3 println( "hello" )) \hspace{1cm} ;Prints hello three times.}
\texttt{"hello"}
\texttt{"hello"}
\texttt{"hello"}
\texttt{=> t \hspace{1cm} ;for always returns t}

Reference

\texttt{drain, print, newline}
procedure

procedure(
    s_funcName(
        l_formalArglist
    )
    g_expr1 ...
)
=> s_funcName

Description

Defines a function using a formal argument list. The body of the procedure is a list of expressions to evaluate.

The body of the procedure is a list of expressions to be evaluated one after another when s_funcName is called. There must be no white space between procedure and the open parenthesis that follows, nor between s_funcName and the open parenthesis of l_formalArglist. However, for defun there must be white space between s_funcName and the open parenthesis. This is the only difference between the two functions. defun has been provided principally so that you can make your code appear more like other LISP dialects.

Expressions within a function can reference any variable on the formal argument list or any global variable defined outside the function. If necessary, local variables can be declared using the let or prog functions.

Arguments

s_funcName
    Name of the function you are defining.

l_formalArglist
    Formal argument list.

g_expr1
    Expression or expressions to be evaluated when s_funcName is called.

Value Returned

s_funcName
    Name of the function being defined.
ARGUMENT LIST PARAMETERS

Several parameters provide flexibility in procedure argument lists. These parameters are referred to as @ ("at") options. The parameters are @rest, @optional, and @key.

@rest Option

The @rest option allows an arbitrary number of arguments to be passed into a function. Let’s say you need a function that takes any number of arguments and returns a list of them in reverse order. Using the @rest option simplifies this task.

**Note:** The name of the parameter following @rest is changeable. The r has been used for convenience.

```plaintext
procedure( myReverse( @rest r )
    reverse( r ))
=> myReverse
myReverse( 'a 'b 'c )
=> (c b a)
```

@optional Option

The @optional option gives you another way to specify a flexible number of arguments. With @optional, each argument on the actual argument list is matched up with an argument on the formal argument list. If you place @optional in the argument list of a procedure definition, any argument following it is considered optional.

You can provide any optional argument with a default value. Specify the default value using a default form. The default form is a two-member list. The first member of this list is the optional argument’s name. The second member is the default value.

The default value is assigned only if no value is assigned when the function is called. If the procedure does not specify a default value for a given argument, nil is assigned.

The following is an outline of a procedure that builds a box of a certain length and width.

```plaintext
procedure(buildbox(length width @optional (xcoord 0)
    (ycoord 0) color)
    .
) .
```

Both length and width must be specified when this function is called. However, the color and the coordinates of the box are declared as optional parameters. If only two parameters are specified, the optional parameters are given their default values. For xcoord and
Examine the following calls to buildbox and their return values:

buildbox(1 2); Builds a box of length 1, width 2; at the coordinates (0,0) with the default color nil
buildbox(3 4 5.5 10.5); Builds a box of length 3, width 4; at coordinates (5.5,10.5) with the default color nil
buildbox(3 4 5 5 ’red); Builds a box of length 3, width 4; at coordinates (5,5) with the default color red.

As illustrated in the above examples, @optional relies on order to determine what actual arguments are assigned to each formal argument. When relying on order is too lengthy or inconvenient, another “at” sign parameter, @key, provides an alternative.

@key Option

@key and @optional are mutually exclusive; they cannot appear in the same argument list. The @key option lets you specify the expected arguments in any order.

For example, examine the following function:

procedure(setTerm(@key (deviceType 'unknown) (baudRate 9600) (keyClick )))

If you call setTerm without arguments (that is, setTerm()), deviceType is set to unknown, baudRate to 9600, and keyClick to nil. Default forms work the same as they do for @optional. To specify a keyword for an argument (for example, deviceType, baudRate, and keyClick in the above function), precede the keyword with a question mark (?).

To set the baudRate to 4800 and the keyClick to ON, the call is:

setTerm(?baudRate 4800 ?keyClick 'ON)
    ; This sets baudRate and keyClick. Because nothing
    ; was specified for deviceType, it is set to its default,
    ; unknown.
setTerm(?keyClick 'ON ?baudRate 4800) ; Does exactly
    ; the same as above.

In summary, there are two standard forms that procedure argument lists follow:

procedure(functionname([var1 var2 ...]
    [@optional opt1 opt2 ...]
    [@rest r])
    .
procedure(functionname([var1 var2 ...])
   [@key key1 key2 ...]
   [@rest r])

Example

procedure( cube(x) x**3 ) ; Defines a function to compute the
=> cube ; cube of a number using procedure.
cube( 3 ) => 27
defun( cube (x) x**3 ) ; Defines a function to compute the
defun => cube ; cube of a number using defun.

The following function computes the factorial of its positive integer argument by recursively
calling itself.

procedure( factorial(x)
   if( (x == 0) then 1
      else x * factorial(x - 1))
=> factorial
defun( factorial (x)
   if( (x == 0) then 1
      else x * factorial( x - 1))
defun => factorial
factorial( 6 )
defun => 720

Reference

defun, let - SKILL mode, nprocedure - SKILL mode only, prog
procedurep

procedurep(
    g_obj
)  
=> t | nil

Description

Checks if an object is a procedure, or function, object.

A procedure may be a function object defined in SKILL or SKILL++, or system primitives. Note that symbols are not considered procedures even though they may have function bindings.

Arguments

g_obj  
Any SKILL object.

Value Returned

t  
The argument is a procedure, or function, object.

nil  
Otherwise.

Example

(procedurep 123 )  => nil
(procedurep (getd 'plus))  => t
(procedurep 'plus)  => nil

Reference

defun, isCallable, lambda, procedure
prog

 prog( l_localVariables  
  [  
   [ s_label ] 
   g_exprl  
  ] ...  
)  
=> g_result | nil

Description

Allows for local variable bindings and permits abrupt exits on control jumps. This is a syntax form.

The first argument to prog is a list of variables declared to be local within the context of the prog. The expressions following the prog are executed sequentially unless one of the control transfer statements such as go or return is encountered. A prog evaluates to the value of nil if no return statement is executed and control simply “falls through” the prog after the last expression is executed. If a return is executed within a prog, the prog immediately returns with the value of the argument given to the return statement.

Any statement in a prog can be preceded by a symbol that serves as a label for the statement. Unless multiple return points are necessary or you are using the go function, a faster construct for binding local variables, let, should be used over prog.

Arguments

l_localVariables List of variables local to prog.

s_label Labels a statement inside a prog; labels can be defined only for statements at the top level. Statements nested inside another statement cannot be labelled unless the surrounding statement is itself a prog.

g_exprl Any SKILL expression to be evaluated inside the prog.

Value Returned

g_result Value of the return statement if one is used.
nil

Otherwise always returns nil.

Example

```plaintext
x = "hello"
=> "hello"
prog( (x y) ; Declares local variables x and y.
    x = 5 ; Initialize x to 5.
    y = 10 ; Initialize y to 10.
    return( x + y )
)
=> 15
x
=> "hello" ; The global x keeps its original value.
```

Reference

let - SKILL mode, go, procedure, progn
prog1

prog1(
    g_expr1
    [ g_expr2 ... ]
)
=> g_result

Description

Evaluates expressions from left to right and returns the value of the first expression. This is a syntax form.

Arguments

g_expr1 Any SKILL expression.

g_expr2 Any SKILL expression.

Value Returned

g_result Value of the first expression, g_expr1.

Example

prog1(
    x = 5
    y = 7
)
=> 5

Returns the value of the first expression.

Reference

prog, prog2, progn
prog2

prog2(
  g_expr1
  g_expr2
  [ g_expr3... ]
)
=> g_result

Description

Evaluates expressions from left to right and returns the value of the second expression. This is a syntax form.

Arguments

  g_expr1                                      First SKILL expression.
  g_expr2                                      Second SKILL expression.
  g_expr3                                      Additional SKILL expressions.

Value Returned

  g_result                                    Value of the second expression, g_expr2.

Example

prog2(
  x = 4
  p = 12
  x = 6 )
=> 12

Returns the value of the second expression.

Reference

  prog, prog1, progn
progn
progn(
  g_expr1 ...
)
=> g_result

Description
Evaluates expressions from left to right and returns the value of the last expression. This is a
syntax form.
progn is useful for grouping a sequence of expressions into a single expression. As a
shorthand notation for progn, use braces ({ }) to group multiple expressions into a single
expression.

Arguments

$g_{expr1}$

Any SKILL expression.

Value Returned

$g_{result}$

Value of the last expression evaluated.

Example
progn(
  println("expr 1")
  println("expr 2")
) "expr 1" "expr 2" => nil

The value of println is nil. The following example uses braces.
{
  println("expr 1")
  println("expr 2")
  2 + 3
}"expr 1" "expr 2" 5

Reference

begin - SKILL mode, let - SKILL mode, prog, prog1, prog2
**putd**

```lisp
putd(
    s-functionName
    u-functionDef
)
=> u-functionDef
```

**Description**

Assigns a new function binding, which must be a function, a `lambda` expression, or `nil`, to a function name. If you just want to define a function, use `procedure` or `defun`.

Assigns the function definition of `u-functionDef` to `s-functionName`. This is different from `alias`, which does a macro expansion when evaluated. You can undefine a function name by setting its function binding to `nil`. A function name can be write-protected by the system to protect you from unintentional name collisions, in which case you cannot change the function binding of that function name using `putd`.

**Note:** If you just want to define a function, use `procedure` or `defun`.

**Arguments**

- **s-functionName**
  - Name of the function.

- **u-functionDef**
  - New function binding, which must be a binary function, a `lambda` expression, or `nil`.

**Value Returned**

- **u-functionDef**
  - Function definition, which is either a binary function or a SKILL expression.

**Example**

```lisp
putd( 'mySqrt getd( 'sqrt ))
=> lambda:sqrt
```

Assigns the function `mySqrt` the same definition as `sqrt`.

```lisp
putd( 'newFn 'lambda( () println( "This is a new function" )))
=> funobj:0x3cf088
```
Assigns the symbol `newFn` a function definition that prints the string `This is a new function when called`.

**Reference**

`alias`, `getd`, `lambda`
putprop

putprop(
   sl_id
   g_value
   S_name
)
=> g_value

Description

Adds properties to symbols or disembodied property lists.

If the property already exists, the old value is replaced with a new one. The putprop function is a lambda function, which means all of its arguments are evaluated.

Arguments

sl_id                    Symbol or disembodied property list.

g_value                  Value of the named property.

S_name                   Name of the property.

Value Returned

g_value                  The value of the named property.

Example

putprop('s 1+2 'x) => 3

Sets the property x on symbol s to 3.

Reference

get, putpropq, putpropqq
putpropq

putpropq(
    sl_id
    g_value
    S_name
)
=> g_value

outpropq(
    sl_id->s_name = g_value
)
=> g_value

Description

Adds properties to symbols or disembodied property lists. Identical to putprop except that
s_name is not evaluated. If the property already exists, the old value is replaced with a new
one.

Arguments

sl_id        Symbol or disembodied property list.
g_value     Value of the named property.
S_name      Name of the property.

Value Returned

g_value     Value of the named property.

Example

putpropq('s 1+2 x) => 3
y = 'x => x
y->x = 1+2 => 3

Both examples are equivalent expressions that set the property x on symbol s to 3.
Reference

get, putprop, putpropqq
putpropqq

putpropqq(
  s_id
  g_value
  S_name
)
=> g_value

outpropqq(
  s_id.s_name = g_value
)
=> g_value

Description

Adds properties to symbols. Identical to putprop except that $s_id$ and $s_name$ are not evaluated. If the property already exists, the old value is replaced with a new one.

Arguments

$s_id$  Can only be a symbol.
$g_value$  Value of the named property.
$S_name$  Name of the property.

Value Returned

$g_value$  Value of the named property.

Example

putpropqq(s 1+2 x)  => 3
s.x = 1+2  => 3

Both examples are equivalent expressions that set the property $x$ on symbol $s$ to 3.

Reference

get, putprop, putpropq
quote

quote(
  g_expr
)
=> g_result

Description

Returns the name of the variable or the expression. Prefix form of the ' operator. Quoting is used to prevent expressions from being evaluated.

Arguments

g_expr Variable or expression.

Value Returned

g_result Name of the variable or expression.

Example

(quote a) => a
(quote (f a b)) => (f a b)
**quotient**

```plaintext
quotient(
    n_op1
    n_op2
    [ n_op3 ... ]
) => n_result
```

**Description**

Returns the result of dividing the first operand by one or more operands. Prefix form of the `/` arithmetic operator.

**Arguments**

- `n_op1` Dividend.
- `n_op2` Divisor.
- `n_op3` Optional additional divisors for multiple divisions.

**Value Returned**

- `n_result` Result of the operation.

**Example**

```plaintext
quotient(5 4 3 2 1) => 0
quotient(-10 -2) => 5
quotient(10.8 -2.2) => -4.909091
```

**Reference**

*quotient*
random

random(
    [ x_number ]
)
=> x_result

Description

Returns a random integer between zero and a given number minus one.

If you call random with no arguments, it returns an integer that has all of its bits randomly set.

Arguments

x_number An integer.

Value Returned

x_result Random integer between zero and x_number minus one.

Example

random( 93 )
=> 26

Reference

srandom
range

range(
    n_num1
    n_num2
) => l_result

Description

Returns a list whose first element is `n_num1` and whose tail is `n_num2`. Prefix form of the `:` operator.

Arguments

`n_num1`  
First element of the list.

`n_num2`  
Tail of the list.

Value Returned

`l_result`  
Result of the operation.

Example

L = range(1 2) => (1 2)
car(L) => 1
cdr(L) => (2)
L = range(1.1 3.3) => (1.1 3.3)
car(L) => 1.1
cdr(L) => (3.3)

Reference

`cdr`
read

read(
    [ p_inputPort ]
) => g_result | nil | t

Description

Parses and returns the next expression from an input port.

Returns the next expression regardless of how many lines the expression takes up - even if there are other expressions on the same line. If the next line is empty, returns t. If the port is positioned at end of file, then it returns nil.

Arguments

p_inputPort Input port. Default is piport.

Values Returned

g_result The object read in.
nil When the port is at the end of file.
t If an empty line is encountered.

Example

Suppose the file SkillSyntaxFile.il contains the following expressions. Note that a blank line follows the second expression:

define( x 1 )
define( y 2 )
procedure( add( x y ) x+y )

myPort = infile( "SkillSyntaxFile.il" )
    => port:SkillSyntaxFile.il"
read( myPort ) => define(x 1)
read( myPort ) => define(y 2)
read( myPort ) => t
read( myPort ) => procedure((add x y) (x + y ) )
read( myPort ) => nil
close( myPort ) => t
Reference

lineread
readstring

readstring(
    t_string
)
=> g_result | nil

Description

Returns the first expression in a string. Subsequent expressions in the string are ignored. The expression is not processed in any way.

Arguments

t_string String to read.

Value Returned

g_result The object read in.

nil When the port is at the end of the string.

Example

readstring("fun( 1 2 3 ) fun( 4 5 )") => ( fun 1 2 3 )

The first example shows normal operation.

readstring("fun(" )
fun(
^
SYNTAX ERROR found at line 1 column 4 of file *string*
*Error* lineread/read: syntax error encountered in input
*WARNING* (include/load): expression was improperly terminated.

The second example shows the error message if the string contains a syntax error.

EXPRESSION = 'list( 1 2 )
=> list(1 2)
EXPRESSION == readstring( sprintf( nil "%L" EXPRESSION ))
=> t

The third example illustrates that readstring applied to the print representation of an expression, returns the expression.
Reference

linereadstring
readTable

readTable(
  S_fileName
  o_table
)
=> t | nil

Description
Reads and appends the contents of a file to an existing association table.

Prerequisites
The file submitted must have been created with the writeTable function so that the contents are in a usable format.

Arguments

S_fileName File name (either a string or symbol) from which to read the data.
o_table Association table to which the file contents are appended.

Value Returned
t The data is read and appended.
nil Otherwise.

Example

myTable = makeTable("table1") => table:table1
myTable2 = makeTable("table2") => table:table2
myTable["three"] = 3 => 3
writeTable("table.out" myTable) => t
readTable("table.out" myTable2) => t

Reference
makeTable, writeTable
realp

realp(
    g_obj
) => t | nil

Description

Checks if a value is a real number. Same as floatp.

Arguments

g_obj Any SKILL object.

Value Returned

t Argument is a real number.

nil Argument is not a real number.

Example

realp( 2789987) => nil
realp( 2789.987) => t

Reference

floatp, integerp, fixp
regExitAfter

regExitAfter(
    s_name
)
=> t | nil

Description

Registers the action to be taken after the exit function has performed its bookkeeping tasks but before it returns control to the operating system.

Arguments

s_name
Name of the function that is to be added to the head of the list of functions to be performed after the exit function.

Value Returned

t
The function is added to the list of functions.

nil
Otherwise.

Example

 procedure( foo( @rest args)
    println( "After proc being executed")
)regExitAfter( 'foo) => t

Reference

clearExitProcs, exit, regExitBefore, remExitProc
**regExitBefore**

`regExitBefore(`

  `s_name`

`) => t`

**Description**

Registers the action to be taken before the `exit` function is executed. If the function registered returns the `ignoreExit` symbol, the exit is aborted.

**Arguments**

`s_name` Name of the function that is to be added to the head of the list of functions to be executed before the `exit` function.

**Value Returned**

t Always.

**Example**

```skill
procedure( foo() println( "Aborting exit") 'ignoreExit)
=> foo
regExitBefore('foo)
=> t
exit ;Does not exit.
"Aborting exit"

procedure( foo() println( "Exiting")
=> foo
exit ;Exits program.
"Exiting"
```

**Reference**

clearExitProcs, exit, regExitAfter, remExitProc
remainder

remainder(
    x_integer1
    x_integer2
)
=> x_integer

Description

Returns the remainder of dividing two integers. The remainder is either zero or has the sign of the dividend. Same as mod.

The remainder and modulo functions differ on negative arguments. The remainder is either zero or has the sign of the dividend if you use the remainder function. With modulo the return value always has the sign of the divisor.

Arguments

x_integer1 Dividend.

x_integer2 Divisor.

Value Returned

x_integer Remainder of dividing x_integer1 by x_integer2. The sign is determined by the sign of x_integer1.

Example

modulo( 13 4) => 1
remainder( 13 4) => 1
modulo( -13 4) => 3
remainder( -13 4) => -1
modulo( 13 -4) => -3
remainder( 13 -4) => 1
modulo( -13 -4) => -1
remainder( -13 -4) => -1

Reference

mod, modulo
remd

remd(
    g_x
    l_arg
)
=> l_result

Description

Removes all top-level elements equal to a SKILL object from a list. This is a destructive removal, which means that the original list itself is modified. Therefore, any other reference to that list will also see the changes.

remd uses equal for comparison.

Caution

This is a destructive removal. The original list itself will be modified except for the first element from the original list. Therefore, any other reference to that list will also see the changes. See example 3 where the same variable is used to hold the updated list.

Arguments

g_x        Any SKILL object to be removed from the list.
l_arg      List from which to remove g_x.

Value Returned

l_result   Returns l_arg modified so that all top-level elements equal to g_x are removed.

Example 1

y = '("a" "b" "x" "d" "f")' => ('"a" "b" "x" "d" "f")
remd('"x" y) => ('"a" "b" "d" "f")
y => ('"a" "b" "d" "f"')
Example 2

The first element from the original list will not be modified in-place.

\[
y = \text{'("a" "b" "d" "f")} \Rightarrow \text{'("a" "b" "d" "f")}
\]
\[
\text{remd( "a" y)} \Rightarrow \text{'("b" "d" "f")}
\]
\[
y \Rightarrow \text{'("a" "b" "d" "f")}
\]

Note the original list, \(y\), is not modified.

Example 3

In order to remove the first element from the original list, use the same variable (that holds the original list) to hold the updated list.

\[
y = \text{'("a" "b" "d" "f")} \Rightarrow \text{'("a" "b" "d" "f")}
\]
\[
y=\text{remd("a" y)} \Rightarrow \text{'("b" "d" "f")}
\]
\[
y \Rightarrow \text{'("b" "d" "f")}
\]

Reference

remdq, remove, remq
remdq

remdq(
  g_x
  l_arg
)
=> l_result

Description

Removes all top-level elements that are identical to a SKILL object using eq from a list. This is a destructive removal, which means that the original list itself is modified. Therefore, any other reference to that list will also see the changes.

remdq uses eq instead of equal for comparison.

Caution

This is a destructive removal, which means that the original list itself is modified. Therefore, any other reference to that list will also see the changes.

Arguments

<table>
<thead>
<tr>
<th>g_x</th>
<th>Any SKILL object to be removed from the list.</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_arg</td>
<td>List from which to remove g_x.</td>
</tr>
</tbody>
</table>

Value Returned

| l_result | Returns l_arg modified so that all top-level elements eq to g_x are removed. |

Example

| y = '(a b x d f x g) => (a b x d f x g) |
| remdq('x y) => (a b d f g) |
| y => (a b d f g) |
Reference
remd, remove, remq
remExitProc

remExitProc(        
    s_name            
  )             
=> t

Description

Removes a registered exit procedure.

When SKILL exits, the function is not called.

Prerequisites

The exit procedure must have been previously registered with the \textit{regExitBefore} or \textit{regExitAfter} function.

Arguments

\textit{s\_name}                  Name of the registered exit procedure to be removed.

Value Returned

\textit{t}                               Always.

Example

remExitProc( 'endProc) => t

Reference

\textit{exit, regExitAfter, regExitBefore}
remove

remove(
    g_x
    l_arg
)
=> l_result

remove(
    g_key
    o_table
)
=> g_value

Description

Returns a copy of a list with all top-level elements equal to a SKILL object removed. Can also be used to remove an entry from an association table, in which case the removal is destructive, that is, any other reference to the table will also see the changes.

remove uses equal for comparison.

remove can also be used with an association table to identify and remove an entry corresponding to the key specified in the function.

Arguments

\[ g_x \] Any SKILL object to be removed from the list.

\[ l_arg \] List from which to remove \[ g_x \].

\[ g_key \] Key or first element of the key/value pair.

\[ o_table \] Association table containing the key/value pairs to be processed.

Value Returned

\[ l_result \] Copy of \[ l_arg \] with all top-level elements equal to \[ g_x \] removed.

\[ g_value \] Value associated with the key that is removed.
Example

```skill
remove( "x" '("a" "b" "x" "d" "f")
=> ("a" "b" "d" "f")

myTable = makeTable("myTable" -1)
=> table:myTable ;default is -1

myTable["two"]=2
=> 2

remove("two" myTable)
=> 2 ; permanently removed from table

myTable["two"]
=> -1 ; the default value
```

Reference

remd, remq
remprop

remprop(
  sl_id
  S_name
)
=> l_result | nil

Description

Removes a property from a property list and returns the property's former value.

Arguments

sl_id                Symbol or disembodied property list.
S_name               Property name.

Value Returned

l_result             Former value of the property as a single element list.
nil                   The property does not exist.

Example

putprop( 'chip 8 'pins ) => 8

Assigns the property pins to chip.

get( 'chip 'pins ) => 8
remprop( 'chip 'pins ) => (8)

Removes the property pins from chip.

get( 'chip 'pins) => nil

Reference

get, putprop
remq

remq(
   g_x
   l_arg
)
=> l_result

Description

Returns a copy of a list with all top-level elements that are identical to a SKILL object removed. Uses `eq`.

Arguments

`g_x`  
Any SKILL object to be removed from the list.

`l_arg`  
List from which to remove `g_x`.

Value Returned

`l_result`  
A copy of `l_arg` with all top-level elements `eq` to `g_x` removed.

Example

`remq('x '(a b d f x g)) => (a b d f g)`

Reference

`remd`, `remdg`, `remove`
renameFile

renameFile(
    S_old
    S_new
)
=> t | nil

Description:

The `renameFile()` function changes the name of a file or directory. The `S_old` argument points to the pathname of the file or directory to be renamed. The `S_new` argument points to the new pathname of the file or directory. If the SKILL path is nil, `renameFile()` would search the current directory. Otherwise, the SKILL path would be searched first for `S_old`. Note that a path which is anchored to current directory, for example, `./`, `../`, or `../../..`, etc., is not considered as a relative path.

Arguments:

`S_old` Points to the pathname of the file or directory to be renamed.

`S_new` Points to the new pathname of the file or directory.

Value Returned

`t` File or directory is successfully re-named.

`nil` If `S_old` path does not exist.

Example

renameFile( "/usr/oldname" "/usr/newName" ) => t
renameFile( "/usr/old" "/usr/new" ) => nil ;if old does not exist.
renameFile( "old" "new" ) ;if old is a file while new is a directory
*Error* renameFile: is a directory
**return**

```
return(
  [ g_result ]
)
=> g_result | nil
```

**Description**

Forces the enclosing prog to exit and returns the given value. The return statement has meaning only when used inside a prog statement.

Both go and return are not purely functional in the sense that they transfer control in a non-standard way. That is, they don’t return to their caller.

**Arguments**

`g_result` Any SKILL object.

**Value Returned**

The enclosing prog statement exits with the value given to return as the prog’s value. If return is called with no arguments, nil is returned as the enclosing prog’s value.

**Example**

```skill
procedure( summation(l)
  prog( (sum temp)
    sum = 0
    temp = l
    while( temp
      if( null(car(temp))
        then
          return(sum)
        else
          sum = sum + car(temp)
          temp = cdr(temp)
    }
  )
)
)
```

Returns the summation of previous numbers if a nil is encountered.

```
summation( '(1 2 3 nil 4))
=> 6 ; 1+2+3
```
summation( '(1 2 3 4))
=> nil ; prog returns nil if no explicit return)

Reference

nlambda - SKILL mode only, go, prog
reverse

reverse(
  \ l_arg
)
=> \ l_result

Description

Returns a copy of the given list with the elements in reverse order.

Because this function copies the list, it uses a lot of memory for large lists.

Arguments

\ l_arg
A list.

Value Returned

\ l_result
A new list with the elements at the top level in reverse order.

Example

reverse( '(1 2 3) ) => (3 2 1)
reverse( '(a b (c d) e) ) => '(e (c d) b a)

Reference

append, sort
**rexCompile**

```lang
rexCompile(
    t_pattern
) => t | nil
```

**Description**

Compiles a regular expression string pattern into an internal representation to be used by succeeding calls to `rexExecute`.

This allows you to compile the pattern expression once using `rexCompile` and then match a number of targets using `rexExecute`; this gives better performance than using `rexMatchp` each time.

**Arguments**

`t_pattern` Regular expression string pattern.

**Value Returned**

`t` The given argument is a legal regular expression string.

`nil` Signals an error if the given pattern is ill-formed or not a legal expression.

**Example**

```lang
rexCompile("^[a-zA-Z]+") => t
rexCompile("\\([a-z]+\)\.\1") => t
rexCompile("\\(\[a-z]*\)\1\$") => t
rexCompile("[ab]") => *Error* rexCompile: Missing ] - ";[ab"
```

**Reference**

`rexExecute`, `rexMatchp`, `rexSubstitute`
Pattern Matching of Regular Expressions

In many applications, you need to match strings or symbols against a pattern. SKILL provides a number of pattern matching functions that are built on a few primitive C library routines with a corresponding SKILL interface.

A pattern used in the pattern matching functions is a string indicating a regular expression. Here is a brief summary of the rules for constructing regular expressions in SKILL:

Rules for Constructing Regular Expressions

<table>
<thead>
<tr>
<th>Synopsis</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Any ordinary character (not a special character listed below) matches itself. A dot matches any character.</td>
</tr>
<tr>
<td></td>
<td>A backslash when followed by a special character matches that character literally. When followed by one of &lt;, &gt;, (, ), and 1,...,9, it has a special meaning as described below.</td>
</tr>
<tr>
<td>\</td>
<td>A backslash followed by a digit \n matches the contents of the \n th register from the current regular expression.</td>
</tr>
<tr>
<td>[c...]</td>
<td>A nonempty string of characters enclosed in square brackets (called a set) matches one of the characters in the set. If the first character in the set is ^, it matches a character not in the set. A shorthand S-E is used to specify a set of characters S up to E, inclusive. The special characters ] and - have no special meaning if they appear as the first character in a set.</td>
</tr>
<tr>
<td>*</td>
<td>A regular expression of any of the forms above, followed by the closure character * matches zero or more occurrences of that form.</td>
</tr>
<tr>
<td>+</td>
<td>Similar to *, except it matches one or more times.</td>
</tr>
<tr>
<td>(...)</td>
<td>A regular expression wrapped as ( form ) matches whatever form matches, but saves the string matched in a numbered register (starting from one, can be up to nine) for later reference.</td>
</tr>
<tr>
<td>\n</td>
<td>A backslash followed by a digit \n matches the contents of the \n th register from the current regular expression.</td>
</tr>
</tbody>
</table>
| <...\>  | A regular expression starting with a \< and/or ending with a \\
 restricts the pattern matching to the beginning and/or the end of a word. A word defined to be a character string can consist of letters, digits, and underscores. |
| rs       | A composite regular expression rs matches the longest match of r followed by a match for s. |
| ^, $     | A ^ at the beginning of a regular expression matches the beginning of a string. A $ at the end matches the end of a string. Used elsewhere in the pattern, ^ and $ are treated as ordinary characters. |
How Pattern Matching Works

The mechanism for pattern matching

- Compiles a pattern into a form and saves the form internally.
- Uses that internal form in every subsequent matching against the targets until the next pattern is supplied.

The `rexCompile` function does the first part of the task, that is, the compilation of a pattern. The `rexExecute` function takes care of the second part, that is, actually matching a target against the previously compiled pattern. Sometimes this two-step interface is too low-level and awkward to use, so functions for higher-level abstraction (such as `rexMatchp`) are also provided in SKILL.

Avoiding Null and Backslash Problems

- A null string (""") is interpreted as no pattern being supplied, which means the previously compiled pattern is still used. If there was no previous pattern, an error is signaled.
- To put a backslash character (\) into a pattern string, you need an extra backslash (\) to escape the backslash character itself.

For example, to match a file name with dotted extension .il, the pattern “^[a-zA-Z]+\.*il$” can be used, but “^[a-zA-Z]\.*il$” gives a syntax error. However, if the pattern string is read in from an input function such as `gets` that does not interpret backslash characters specifically, you should not add an extra backslash to enter a backslash character.
rexExecute

rexExecute(
    S_target
)
=> t | nil

Description

Matches a string or symbol against the previously compiled pattern set up by the last rexCompile call.

This function is used in conjunction with rexCompile for matching multiple targets against a single pattern.

Arguments

S_target String or symbol to be matched. If a symbol is given, its print name is used.

Value Returned

t A match is found.

nil Otherwise.

Example

rexCompile("^[a-zA-Z][a-zA-Z0-9]*") => t
rexExecute('Cell123) => t
rexExecute("123 cells") => nil

Target does not begin with a-z/A-Z

rexCompile("\([a-z]+\)[a-zA-Z0-9]*") => t
rexExecute("abc.bc") => t
rexExecute("abc.ab") => nil

Reference

rexCompile, rexMatchp, rexSubstitute
**rexMagic**

```ruby
rexMagic(
    [ g_state ]
) => t | nil
```

**Description**

Turns on or off the special interpretation associated with the meta-characters in regular expressions.

By default the meta-characters (^, $, *, +, \, [, ], etc.) in a regular expression are interpreted specially. However, this “magic” can be explicitly turned off and on programmatically by this function. If no argument is given, the current setting is returned. Users of **vi** will recognize this as equivalent to the **set magic/set nomagic** commands.

**Arguments**

- **g_state**
  - nil turns off the magic of the meta-characters. Anything else turns on the magic interpretation.

**Value Returned**

- **t** The current setting.
- **nil** The given argument.

**Example**

```
rexCompile( "^[0-9]+" ) => t
rexExecute( "123abc" ) => t
rexSubstitute( "got: \0") => "got: 123"
rexMagic( nil ) => nil
rexCompile( "^[0-9]+" ) => t  # recompile w/o magic
rexExecute( "123abc" ) => nil
rexExecute( "**^[0-9]+!**") => t
rexSubstitute( "got: \0") => "got: \0"
rexMagic( t ) => t
rexSubstitute( "got: \0") => "got: ^[0-9]+"
```
Reference

rexCompile, rexSubstitute, rexReplace
rexMatchAssocList

rexMatchAssocList(
  t_pattern
  l_targets
)
 => l_results | nil

Description

Returns a new association list created out of those elements of the given association list whose key matches a regular expression pattern. The supplied regular expression pattern overwrites the previously compiled pattern and is used for subsequent matching until the next new pattern is provided.

l_targets is an association list, that is, each element on l_targets is a list with its car taken as a key (either a string or a symbol). This function matches the keys against t_pattern, selects the elements on l_targets whose keys match the pattern, and returns a new association list out of those elements.

Arguments

t_pattern Regular expression pattern.

l_targets Association list whose keys are strings and/or symbols.

Value Returned

l_results New association list of elements that are in l_targets and whose keys match t_pattern.

nil If no match is found. Signals an error if the given pattern is ill-formed.

Example

rexMatchAssocList("^[a-z][0-9]*$" 
  '((abc "ascii") ("123" "number") (a123 "alphanum") 
   (a12z "ana")))
 => ((a123 "alphanum"))
Reference

`rexCompile`, `rexExecute`, `rexMatchp`, `rexMatchList`
rexMatchList

rexMatchList(t_pattern, l_targets)

=> l_results | nil

Description

Creates a new list of those strings or symbols in the given list that match a regular expression pattern. The supplied regular expression pattern overwrites the previously compiled pattern and is used for subsequent matching until the next new pattern is provided.

Arguments

t_pattern Regular expression pattern.
l_targets List of strings and/or symbols to be matched against the pattern.

Value Returned

l_results List of strings (or symbols) that are on l_targets and found to match t_pattern.
nil If no match is found. Signals an error if the given pattern is ill-formed.

Example

rexMatchList("^[a-z][0-9]" '(a01 x02 "003" aa01 "abc")
=> (a01 x02 aa01 "abc")
rexMatchList("^[a-z][0-9][0-9]" '(a001 b002 "003" aa01 "abc")
=> (a001 b002)
rexMatchList("box[0-9]" '(square circle "cell9" "123")
=> nil

Reference

rexCompile, rexExecute, rexMatchAssocList, rexMatchp
**rexMatchp**

`rexMatchp(
  t_pattern
  S_target
)
=> t | nil`

### Description

Checks to see if a string or symbol matches a given regular expression pattern. The supplied regular expression pattern overwrites the previously compiled pattern and is used for subsequent matching until the next new pattern is provided.

This function matches `S_target` against the regular expression `t_pattern` and returns `t` if a match is found, `nil` otherwise. An error is signaled if the given pattern is ill-formed. For greater efficiency when matching a number of targets against a single pattern, use the `rexCompile` and `rexExecute` functions.

### Arguments

- **t_pattern**: Regular expression pattern.
- **S_target**: String or symbol to be matched against the pattern.

### Value Returned

- **t**: A match is found. Signals an error if the given pattern is ill-formed.

### Example

```ruby
rexMatchp("[0-9]*[.]\[0-9][0-9]*" "100.001") => t
rexMatchp("[0-9]*[.]\[0-9]+" ".001") => t
rexMatchp("[0-9]*[.]\[0-9]+" ".") => nil
rexMatchp("[0-9]*[.]\[0-9][0-9]*" "10." => nil
rexMatchp("[0-9]*" "100")
*Error* rexMatchp: Missing ] - 
```

### Reference

rexCompile, rexExecute
rexReplace

```ruby
rexReplace(
    t_source
    t_replacement
    x_index
)
=> t_result
```

**Description**

Returns a copy of the source string in which the specified substring instances that match the last compiled regular expression are replaced with the given string.

Scans the source string `t_source` to find all substring(s) that match the last regular expression compiled and replaces one or all of them by the replacement string `t_replacement`. The argument `x_index` tells which occurrence of the matched substring is to be replaced. If it’s 0 or negative, all the matched substrings will be replaced. Otherwise only the `x_index` occurrence is replaced. Returns the source string if the specified match is not found.

**Arguments**

- `t_source` Source string to be matched and replaced.
- `t_replacement` Replacement string to be used. Pattern tags can be used in this string (see `rexSubstitute`).
- `x_index` Specifies which of the matching substrings to replace. Do a global replace if it’s <= 0.

**Value Returned**

- `t_result` Copy of the source string with specified replacement or the original source string if no match was found.

**Example**

```ruby
rexCompile( "[0-9]+" ) => t
rexReplace( "abc-123-xyz-890-wuv" "(*)" 1) => "abc-(*)-xyz-890-wuv"
```
rexReplace( "abc-123-xyz-890-wuv" "(*)" 2) => "abc-123-xyz-(*)-wuv"

rexReplace( "abc-123-xyz-890-wuv" "(*)" 3) => "abc-123-xyz-890-wuv"
rexReplace( "abc-123-xyz-890-wuv" "(*)" 0) => "abc-(*)-xyz-(*)-wuv"

rexCompile( "xyz" ) => t
rexReplace( "xyzzxyzz" "xy" 0) => "xyzyxyz" ; no rescanning!

Reference

rexCompile, rexExecute, rexMatchp, rexSubstitute
rexSubstitute

rexSubstitute(
    t_string
)
  => t_result | nil

Description

Substitutes the pattern tags in the argument string with previously matched (sub)strings.

Copies the argument string and substitutes all pattern tags in it by their corresponding matched strings in the last string matching operation. The tags are in the form of \n, where n is 0-9. \0 (or '&') refers to the string that matched the entire regular expression and \k refers to the string that matched the pattern wrapped by the k'th (...) in the regular expression.

Arguments

| t_string  | Argument string to be substituted. |

Value Returned

| t_result | Copy of the argument with all the tags in it being substituted by the corresponding strings. |
| nil      | The last string matching operation failed (and none of the pattern tags are meaningful). |

Example

rexCompile( "[a-z]+\((0-9)+\)" ) => t
rexExecute( "abc123" ) => t
rexSubstitute( "\0*" ) => "*abc123*
rexSubstitute( "The matched number is: \1" ) => "The matched number is: 123"
rexExecute( "123456" ) => nil ; match failed
rexSubstitute( "-\0-" ) => nil

Reference

rexCompile, rexExecute, rexReplace
rightshift

rightshift(
    x_val
    x_num
)
=> x_result

Description

Returns the integer result of shifting a value a specified number of bits to the right. Prefix form of the >> arithmetic operator. Note that rightshift is logical (that is, vacated bits are 0-filled).

Arguments

x_val Value to be shifted.

x_num Number of bits x_val is shifted.

Value Returned

x_result Result of the operation.

Example

rightshift(7 2) => 1
rightshift(10 1) => 5

Reference

leftshift
rindex

rindex(
    t_string1
    S_string2
) => t_result | nil

Description

Returns a string consisting of the remainder of string1 beginning with the last occurrence of string2.

Compares two strings. Similar to index except that it looks for the last (that is, rightmost) occurrence of the symbol or string S_string2 in string t_string instead of the first occurrence.

Arguments

<table>
<thead>
<tr>
<th>t_string1</th>
<th>String to search for the last occurrence of S_string2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_string2</td>
<td>String or symbol to search for.</td>
</tr>
</tbody>
</table>

Value Returned

<table>
<thead>
<tr>
<th>t_result</th>
<th>Remainder of t_string1 starting with last match of S_string2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>There is no match.</td>
</tr>
</tbody>
</table>

Example

rindex( "dandelion" "d") => "delion"

Reference

index, nindex
round

round(  
    n_arg  
)  
=>  x_result

Description

Rounds a floating-point number to its closest integer value.

Arguments

n_arg  Floating-point number.

Value Returned

x_result  Integer whose value is closest to n_arg.

Example

round(1.5)  =>  2  
round(-1.49)  =>  -1  
round(1.49)  =>  1

Reference

fix, float
rplaca

rplaca(
   l_arg1
   g_arg2
)
=> l_result

Description

Replaces the first element of a list with an object. This function does not create a new list; it alters the input list. Same as setcar.

Caution

This is a destructive operation, meaning that any other reference to the list will also see the change.

Arguments

l_arg1 A list.
g_arg2 Any SKILL object.

Value Returned

l_result Modified l_arg1 with the car of l_arg1 replaced by g_arg2.

Example

x = '(a b c)
rplaca( x 'd ) => (d b c)
x => (d b c)

The car of x is replaced by the second argument.

Reference

car, rplacd, setcar, setcdr
rplacd

\[ rplacd( \quad l\_arg1
l\_arg2
) \quad => \quad l\_result \]

Description

Replaces the tail of a list with the elements of a second list. This function does not create a new list; it alters the input list. Same as setcdr.

Caution

This is a destructive operation, meaning that any other reference to the list will also see the changes.

Arguments

- **l\_arg1**: List that is modified.
- **l\_arg2**: List that replaces the cdr of **l\_arg1**.

Value Returned

- **l\_result**: Modified **l\_arg1** with the cdr of the list **l\_arg1** replaced with **l\_arg2**.

Example

\[
x = '(a b c)
rplacd( x '(d e f)) => (a d e f)
x \quad => \quad (a d e f)
\]

The cdr of \( x \) is replaced by the second argument.

Reference

- **cdr**, **rplaca**, **setcar**, **setcdr**
schemeTopLevelEnv

schemeTopLevelEnv(
)
  => e_envobj

Description

Returns the top level SKILL++ environment as an environment object.

Arguments

None.

Value Returned

e_envobj  The top level SKILL++ environment object.

Example

schemeTopLevelEnv() => envobj:0x1ad018

Reference

envobj, theEnvironment - SKILL++ mode only
set

set(
    s_variableName
    g_newValue
    [ e_environment ]
) => g_result

Description

Sets a variable to a new value. Similar to setq but the first argument for set is evaluated.

The set function is similar to the setq function, but unlike setq, the first argument for set is evaluated. This argument must evaluate to a symbol, whose value is then set to \texttt{g\_newValue}.

Arguments

\texttt{s\_variableName} \quad Symbol that is evaluated.

\texttt{g\_newValue} \quad Value to set symbol to.

\texttt{e\_environment} \quad If this argument is given, SKILL++ semantics is assumed. The forms entered will be evaluated within the given (lexical) environment.

Value Returned

\texttt{g\_result} \quad Returns \texttt{g\_newValue}.

Example

\begin{verbatim}
y = 'a => a ; Sets y to the constant a.
set y 5 => 5 ; Sets the value of y to 5.
y => a
a => 5
\end{verbatim}

Reference

setq
**setarray**

```skill
setarray(
    a_array
    x_index
    g_value
)
=> g_value

setarray(
    o_table
    g_key
    g_value
)
=> g_value
```

**Description**

Assigns the given value to the specified element of an array or to the specified key of a table. Normally this function is invoked implicitly using the array-subscription syntax, such as, `x[i] = v`.

Assigns `g_value` to the `x_index` element of `a_array`, or adds the association of `g_value` with `g_key` to `o_table`, and returns `g_value`. Normally this function is invoked implicitly using the array-subscription syntax, such as, `x[i] = v`.

**Arguments**

- **a_array**
  - An array object.
- **x_index**
  - Index of the array element to assign a value to. Must be between 0 and one less than the size of the array.
- **g_key**
  - Any SKILL value.
- **g_value**
  - Value to be assigned to the specified array element or table entry.

**Value Returned**

- **g_value**
  - Value assigned to the specified array element or table entry.
Example

```
declare(myar[8]) => array[8]:3895304
myar[0] => unbound
setarray(myar 0 5) => 5
myar[0] => 5
setarray(myar 8 'hi)
```

Signals an array bounds error.

```
setarray(myar
  (plus 1 2)       ; assigns element 3 the value 8.
  (plus 3 5))     => 8
mytab = makeTable('myTable) => table:myTable
setarray(mytab 8 4) => 4
mytab[8] => 4
mytab[9] = 3 => 3       ; same as setarray(mytab 9 3)
mytab[9] => 3
```

Reference

arrayref, declare
setcar

setcar( 
    l_arg1 
    g_arg2 
  )
 => l_result

Description

Replaces the first element of a list with an object. Same as rplaca.

Caution

This is a destructive operation, meaning that any other reference to the list will also see the change.

Arguments

l_arg1 A list.
g_arg2 A SKILL object.

Value Returned

l_result Modified l_arg1 with the car of l_arg1 replaced by g_arg2.

Example

x = '(a b c) => (a b c)
setcar( x 'd ) => (d b c)
x => (d b c)

The car of x is replaced by the second argument.

Reference

car, rplaca, rplacd, setcdr
setcdr

setcdr(
    l_arg1
    l_arg2
  )
=> l_result

Description

Replaces the tail of a list with the elements of a second list. Same as rplacd.

Caution

This is a destructive operation, meaning that any other reference to the list will also see the change.

Arguments

l_arg1 List that is modified.

l_arg2 List that replaces the cdr of l_arg1.

Value Returned

l_result Modified l_arg1 with the cdr of the list l_arg1 replaced with l_arg2.

Example

x = '(a b c)
setcdr( x '(d e f)) => (a d e f)
x => (a d e f)

The cdr of x is replaced by the second argument.

Reference

cdr, rplaca, rplacd, setcar
setFnWriteProtect

setFnWriteProtect(
        s_name
    )
    => t | nil

Description

Prevents a named function from being redefined.

If `s_name` has a function value, it can no longer be changed. If it does not have a function value but does have an autoload property, the autoload is still allowed. This is treated as a special case so that all the desired functions can be write-protected first and autoloaded as needed.

Arguments

`s_name` Name of the function.

Value Returned

t The function is now write protected.

nil If the function is already write protected.

Example

Define a function and set its write protection so it cannot be redefined.

```skill
procedure( test() println( "Called function test" ) )
setFnWriteProtect( 'test ) => t
procedure( test() println( "Redefine function test" ) )
*Error* def: function name already in use and cannot be redefined - test
setFnWriteProtect( 'plus ) => nil
```

Returns nil because the `plus` function is already write protected.

Reference

`getFnWriteProtect`, `setVarWriteProtect` - SKILL mode only
setof

setof(
  s_formalVar
  l_valueList
  g_predicateExpression
)
=> l_result

setof(
  s_formalVar
  o_table
  g_predicateExpression
)
=> l_result

Description

Returns a new list containing only those elements in a list or the keys in an association table that satisfy an expression. This is a syntax form.

The setof form can also be used to identify all keys in an association table that satisfy the specified expression.

Arguments

s_formalVar               Local variable that is usually referenced in
                          g_predicateExpression.

l_valueList               List of elements that are bound to s_formalVar one at a time.

g_predicateExpression    SKILL expression that usually uses the value of s_formalVar.

o_table                   Association table whose keys are bound to s_formalVar one
                          at time.

Value Returned

l_result                  New list containing only those elements in l_valueList that
                          satisfy g_predicateExpression, or list of all keys that
                          satisfy the specified expression.
Example

```
setof( x '(1 2 3 4) (x > 2) ) => (3 4)
setof( x '(1 2 3 4) (x < 3) ) => (1 2)
myTable = makeTable("atable" 0) => table:atable
myTable["a"]="first" => "first"
myTable["b"]=2 => 2
setof(key myTable (and (stringp key)(stringp myTable[key])))) => ("a")
```

Reference

exists, foreach
setplist

setplist(  
    s_atom
    l_plist
)
=> l_plist

Description

Sets the property list of an object to a new property list; the old property list attached to the object is lost.

⚠️ Caution

*Users are strongly discouraged from using setplist because it might remove vital properties being used by the system or other applications.*

Arguments

- **s_atom**: A symbol.
- **l_plist**: New property list to attach to *s_atom*.

Value Returned

- **l_plist**: New property list for *s_atom*; the old property list is lost.

Example

```scheme
setplist( 'chip '(pins 8 power 5) ) => (pins 8 power 5)
plist( 'chip ) => (pins 8 power 5)
chip.power => 5
```

Reference

- `getg`, `getgg`, `plist`, `putprop`, `putpropq`, `remprop`
**setPrompts**

```skill
setPrompts(
    s_prompt1
    s_prompt2
)
=> t | nil
```

**Description**

Sets the prompt text string for the CIW. The first prompt is used to indicate the topmost top-level. The second prompt is used whenever a nested top-level is entered.

The text string for `s_prompt2` should always be the `%d` format string, which behaves the same as the `printf()` format string, such that the nesting level of a nested top-level will be shown as it deepens.

**Note:** Changing prompts in some applications can seriously interfere with their functioning; be very careful using this function.

**Arguments**

- `s_prompt1` Prompt text string.
- `s_prompt2` Prompt text string.

**Value Returned**

- `t` The prompt has been set.
- `nil` Returns `nil` and issues an error message if the prompt is not changed.

**Example**

```
> setPrompts("-> " "%d" ")
t
~> toplevel( 'ils ')
ILS-<2>> toplevel( 'ils ')
ILS-<3>>
```

Sets the topmost top-level to `->` and the nested top-level to `<%d>`:

```
> setPrompts("-> " "%s" ")
*Error* setPrompts: setPrompts expected %d not %s in prompt --
"%s"
```
%s is an illegal format string.

Reference
getPrompts
setq

setq(  
    s_variableName  
    g_newValueExp  
)  
=> g_result

setq(  
    s_variableName = g_newValue  
)  
=> g_result

Description

Sets a variable to a new value. setq is the same as the assignment (=) operator. This is a syntax form.

The symbol s_variableName is bound to the value of g_newValueExp. Note that the first argument to setq is not evaluated but the second one is.

Arguments

s_variableName  Variable to be bound.
g_newValueExp  Expression to be evaluated and bound to s_variableName.

Value Returned

g_result  Evaluated result of g_newValueExp is returned.

Example

x = 5  => 5  
Assigns the value 5 to the variable x.

setq( x 5 )  => 5  
Assigns the value 5 to the variable x.
y = 'a  => a  
Assigns the symbol a to the variable y.
Reference

set
**setqbitfield1**

```skill
setqbitfield1(
    s_var
    x_val
    x_bitPosition
)
=> x_result
```

**Description**

Sets a value into a single bit in the bit field specified by the variable `s_var`, stores the new value back into the variable, and then returns the new value. Prefix form of the `<>=` operator.

**Arguments**

- `s_var` Variable representing the bit field whose value is to be changed.
- `x_val` New value of the bit.
- `x_bitPosition` Position of the bit whose value you are changing.

**Value Returned**

- `x_result` New value of `s_var`.

**Example**

```
x = 0b1001
setqbitfield1(x 1 1) => 11
x => 11
setqbitfield1(x 1 2) => 15
x => 15
```

**Reference**

- `bitfield1`, `bitfield`, `setqbitfield`
setqbitfield

setqbitfield(
    s_var
    x_val
    x_msb
    x_lsb
)
  => x_result

Description

Sets a value into a set of bits in the bit field specified by the variable s_var, stores the new value back into the variable, and then returns the new value. Prefix form of the <:= operator.

Arguments

s_var        Variable representing the bit field whose value is to be changed.
x_val        New value of the bit.
x_msb        Leftmost bit of the set of bits whose value is to be changed.
x_lsb        Rightmost bit of the set of bits whose value is to be changed.

Value Returned

x_result     New value of s_var.

Example

x = 0
setqbitfield(x 0b1001 3 0) => 9
x => 9
setqbitfield(x 1 2 1) => 11
x => 11
setqbitfield(x 0 3 2) => 3
x => 3

Reference

bitfield1, bitfield, setqbitfield1
**setShellEnvVar**

setShellEnvVar(
    t_UnixShellVariableExpr
)
=> t | nil

**Description**

Sets the value of a UNIX environment variable to a new value.

**Arguments**

`t_UnixShellVariableExpr`

Name of the UNIX shell environment variable and the new value, separated by an equals sign.

**Value Returned**

`t`  If the shell environment variable was set.

`nil`  If the shell environment variable was not set.

**Example**

setShellEnvVar("PWD=/tmp")  => t

Sets the parent working directory to the `/tmp` directory.

getShellEnvVar("PWD")  => "/tmp"

Gets the parent working directory.

**Reference**

csh, getShellEnvVar, sh, shell
**setSkillPath**

```
setSkillPath(
    {tl_paths | nil }
) => l_strings | nil
```

**Description**

Sets the internal SKILL path used by some file-related functions in resolving relative path names.

You can specify the directory paths either in a single string, separated by spaces, or as a list of strings. The system tests the validity of each directory path as it puts the input into standard form. If all directory paths exist, it returns `nil`.

If any path does not exist, a list is returned in which each element is an invalid path. Note that

- The directories on the SKILL path are always searched for in the order you specified in `tl_paths`.
- Even if a path does not exist (and hence appears in the returned list) it remains on the new SKILL path.

The use of the SKILL path in other file-related functions can be effectively disabled by calling `setSkillPath` with `nil` as the argument.

**Arguments**

- `tl_paths` Directory paths specified either in a single string or list of strings.
- `nil` Turns off the use of the SKILL path.

**Value Returned**

- `l_strings` List of directory paths that appear in the `tl_paths` argument but do not actually exist.
- `nil` If all directory paths exist.
Example

```
setSkillPath('("." "~" "~/cpu/test1")
 => nil ; If "~/cpu/test1" exists.
 => ("~/cpu/test1") ; If "~/cpu/test1" does not exist.
```

The same task can be done with the following call that puts all paths in one string.

```
setSkillPath(". ~ ~/cpu/test1")
```

Reference

`getSkillPath`, `prependInstallPath`
setVarWriteProtect - SKILL mode only

setVarWriteProtect(
    s_name
) => t | nil

Description

Sets the write-protection on a variable to prevent its value from being updated. Does not work in SKILL++ mode.

Use this function in SKILL mode only when the variable and its contents are to remain constant.

- If the variable has a value, it can no longer be changed.
- If the variable does not have a value, it cannot be used.
- If the variable holds a list or other data structure as its value, it is assumed that the contents will not be changed. If you try to update the contents, the behavior is unspecified.

In SKILL++ mode, use setFnWriteProtect instead.

Arguments

s_name Name of variable to be write-protected.

Value Returned

t Variable is write protected.

nil Variable was already write protected.

Example

y = 5 ; Initialize the variable y.
setVarWriteProtect( 'y )=> t ; Set y to be write protected.
setVarWriteProtect( 'y )=> nil ; Already write protected.
y = 10 ; y is write protected.
*Error* setq: Variable is protected and cannot be assigned to - y
Reference

getFnWriteProtect, getVarWriteProtect - SKILL mode only, setFnWriteProtect
**sh, shell**

```
sh(
   [ t_command ]
)
=> t | nil

shell(
   [ t_command ]
)
=> t | nil
```

**Description**

Starts the UNIX Bourne shell `sh` as a child process to execute a command string.

If the `sh` function is called with no arguments, an interactive UNIX shell is invoked that prompts you for UNIX command input (available only in nongraphic applications).

**Arguments**

`t_command` Command string.

**Value Returned**

`t` If the exit status of executing the given shell command is 0.

`nil` Otherwise.

**Example**

```
shell( rm /tmp/junk)
```

Removes the `junk` file from the `/tmp` directory and returns `t` if it is removed successfully.

**Reference**

`csh`, `getShellEnvVar`, `setShellEnvVar`
simplifyFilename

simplifyFilename(  
  t_name  
  [  g_dontResolveLinks  ]  
)  
=> t_result

Description

Expands the name of a file to its full path.

Returns the fully expanded name of the file t_name. Tilde expansion is performed, “.”/” and “../” are compressed, and redundant slashes are removed. By default, symbolic links are also resolved, unless the second (optional) argument g_notResolveLinks is specified to non-nil.

If t_name is not absolute, the current working directory is prefixed to the returned file name.

Arguments

  t_name File to be fully expanded.
  g_dontResolveLinks If specified to non-nil, symbolic links are not resolved.

Value Returned

  t_result Fully expanded name of the file.

Example

simplifyFilename("~/test") => "/usr/mnt/user/test"

Assumes the user’s home directory is /usr/mnt/user.

simplifyFilename("/tmp/fileName" t) => "/tmp/fileName"

Assumes /tmp/fileName is a symbolic link of /tmp/fileName.real.

Reference

isDir, isFileName, prependInstallPath
**sin**

```
sin(
    n_number
) => f_result
```

**Description**

Returns the sine of a floating-point number or integer.

**Arguments**

- `n_number`
  
  Floating-point number or integer.

**Value Returned**

- `f_result`
  
  Sine of `n_number`.

**Example**

```
sin(3.14/2) => 0.9999997
sin(3.14159/2) => 1.0
```

Floating point results from evaluating the same expressions may be machine dependent.

**Reference**

- `acos`, `asin`, `cos`
sort

sort(
  l_data
  u_comparefn
)
=> l_result

Description

Sorts a list according to a comparison function; defaults to an alphabetical sort when
u_comparefn is nil. This function does not create a new list. It returns the altered input
list. This is a destructive operation. The l_data list is modified in place and no new storage
is allocated. Pointers previously pointing to l_data may not be pointing at the head of the
sorted list.

Sorts the list l_data according to the sort function u_comparefn. u_comparefn( g_x
  g_y ) returns non-nil if g_x can precede g_y in sorted order, nil if g_y must precede
g_x. If u_comparefn is nil, alphabetical order is used. The algorithm currently
implemented in sort is based on recursive merge sort.

Caution

The l_data list is modified in place and no new storage is allocated.
Pointers previously pointing to l_data may not be pointing at the head of
the sorted list.

Arguments

l_data      List of objects to be sorted.
u_comparefn Comparison function to determine which of any two elements
            should come first.

Value Returned

l_result    l_data sorted by the comparison function u_comparefn.
Example

y = '(c a d b) => (a b c d)
(sort y nil) => (c d) ;no longer points to head of list
y => (c d)
y = '(c a d b)
y = (sort y nil) => (a b c d)
y => (a b c d) ;reassignment points y to sorted list.

Reference

alphalessp, lessp, sortcar
sortcar

sortcar(
  l_data
  u_comparefn
)
=> l_result

Description

Similar to sort except that only the car of each element in a list is used for comparison by the sort function. This function does not create a new list. It returns the altered input list.

This function also sorts l_data based on the function u_comparefn.

⚠️ Caution

The l_data list is modified in place and no new storage is allocated. Pointers previously pointing to l_data might not be pointing at the head of the sorted list.

Arguments

l_data List of objects to be sorted.
u_comparefn Comparison function to determine which of any two elements should come first.

Value Returned

l_result l_data sorted by the comparison function u_comparefn.

Example

sortcar( '((4 four) (3 three) (2 two)) 'lessp )
=> ((2 two) (3 three) (4 four)
sortcar( '((d 4) (b 2) (c 3) (a 1)) nil )
=> ((a 1) (b 2) (c 3) (d 4))
Reference

sort
sprintf

```
sprintf(
   {s_Var | nil }
   t_formatString
   [ g_arg1 ... ]
)
=> t_string
```

**Description**

Formats the output and assigns the resultant string to the variable given as the first argument. This is a syntax form.

Refer to the “Common Output Format Specifications” table on the `fprintf` manual page. If `nil` is specified as the first argument, no assignment is made, but the formatted string is returned.

**Arguments**

- **s_Var** Variable name.
- **nil** `nil` if no variable name.
- **t_formatString** Format string.
- **g_arg1** Arguments following the format string are printed according to their corresponding format specifications.

**Value Returned**

- **t_string** Formatted output string.

**Example**

```
sprintf(s "Memorize %s number %d!" "transaction" 5)
=> "Memorize transaction number 5!"
s
=> "Memorize transaction number 5!"
p = outfile(sprintf(nil "test%d.out" 10))
=> port:"test10.out"
```
Reference

fprintf, fscanf, scanf, sscanf, printf
sqrt

sqrt(n_number) => f_result

**Description**

Returns the square root of a floating-point number or integer.

**Arguments**

*n_number*  
Floating-point number or integer.

**Value Returned**

*f_result*  
Square root of the value passed in.

If the value of *n_number* is not a positive number, an error is signaled.

**Example**

```
sqrt( 49 ) => 7.0
sqrt( 43942 ) => 209.6235
```
**srandom**

`srandom(`

    `x_number`

`)=> t`

**Description**

Sets the seed of the random number generator to a given number.

**Arguments**

`x_number` An integer.

**Value Returned**

`t` Always.

**Example**

`srandom( 89 ) => t`

**Reference**

`random`
**sstatus**

```{}
sstatus(
    s_name
    g_switchValue
)
=> g_switchValue
```

**Description**

Sets the internal system variable named to a given value. This is a syntax form.

The internal variables are typically Boolean switches that accept only the Boolean values of `t` and `nil`. Efficiency and security are the reasons why these system variables are stored as internal variables that can only be set by `sstatus`, rather than as SKILL variables you can set directly.

**Internal System Variables**

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoReload</td>
<td>If <code>t</code>, the debugger will try to auto-reload a file that is not loaded under <code>debugMode</code> when the user tries to single step into the code defined by that file. Note: this may not work correctly for SKILL++ functions defined using assignment.</td>
<td><code>nil</code></td>
</tr>
<tr>
<td>debugMode</td>
<td>Debug mode provides more information for debugging SKILL programs. Allows you to redefine write-protected SKILL functions.</td>
<td><code>nil</code></td>
</tr>
<tr>
<td>errsetTrace</td>
<td>Prints errors and stacktrace information that is normally suppressed by <code>errset</code>.</td>
<td><code>nil</code></td>
</tr>
<tr>
<td>fullPrecision</td>
<td>If <code>t</code>, unformatted print functions (<code>print</code>, <code>println</code>, <code>printlev</code>) print floating point numbers in full precision (usually 16 digits); otherwise, the default is about 7 digits of precision.</td>
<td><code>nil</code></td>
</tr>
<tr>
<td>integermode</td>
<td>When on (default is off), the parser translates all arithmetic operators into calls to functions that operate only on <code>fixnums</code>. This results in small execution time savings and makes sense only for compute-intensive tasks whose inner loops are dominated by integer arithmetic calculations.</td>
<td><code>nil</code></td>
</tr>
</tbody>
</table>
**SKILL Language Functions**

### Internal System Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>mergemode</td>
<td>When on (default), arithmetic expressions are merged by the parser whenever possible into a minimum number of function calls and therefore run somewhat faster because most of the arithmetic functions such as plus, difference, times, and quotient can accept a variable number of arguments.</td>
<td>t</td>
</tr>
<tr>
<td>printinfix</td>
<td>Printing of arithmetic expressions and function calls in infix notation is turned off (on) if the second argument is nil (t).</td>
<td>t</td>
</tr>
<tr>
<td>writeProtect</td>
<td>When on, all functions being defined have their write protection set to t so they cannot be redefined. When off, all functions being defined for the first time are not write-protected and thus can be redefined. When developing SKILL code, be sure this switch is set to off.</td>
<td>nil</td>
</tr>
<tr>
<td>stacktraceDump</td>
<td>Prints the local variables when an error occurs if sstatus( stacktrace t) is set. Toggle on/off with t / nil.</td>
<td>nil</td>
</tr>
<tr>
<td>stacktrace</td>
<td>Prints stack frames every time an error occurs. Toggle on/off with t / nil, or set the number of frames to display.</td>
<td>0</td>
</tr>
<tr>
<td>sourceTracing</td>
<td>If t, the debugger will try to print out the corresponding source location at stop/breakpoints (as well as in stack tracing). A file must be loaded in when debugMode is set to t in order to get its source line numbers. The source forms printed are truncated to fit on one line.</td>
<td>nil</td>
</tr>
<tr>
<td>traceArgs</td>
<td>If set to non-nil, the system will save the evaluated arguments of function calls, which can then be displayed in the stacktrace.</td>
<td>nil</td>
</tr>
</tbody>
</table>

Setting `debugMode` or tracing functions (using `tracef`) will no longer turn on `traceArgs` automatically. The default behavior is to turn off this switch because it is very expensive to keep the evaluated arguments around all the time.

**Note:** turning on this switch could slow down the execution speed significantly.
Arguments

$s_name$ Name of internal system variable.

$g_switchValue$ New value for internal system variable, usually $t$ or $nil$.

Value Returned

$g_switchValue$ The second argument to $sstatus$.

Example

$sstatus(\ debugMode\ t ) \ =>\ t$  
Turns on debug mode.

$sstatus(\ integermode\ t ) \ =>\ t$  
Turns on integer mode.

$sstatus(\ stacktraceDump\ t ) \ =>\ t$  
Prints the local variables when an error occurs if $sstatus(\ stacktrace\ t )$ is set.

$sstatus(\ stacktrace\ 6 ) \ =>\ 6$  
Prints the first six stack frames every time an error occurs.

Reference

$status$
status

status(
    s_name
)
=> g_switchValue

Description

Returns the value of the internal system variable named. This \texttt{nlambda} function also works in SKILL++ mode.

See the \texttt{sstatus} function for a list of the internal system variables.

Arguments

\begin{description}
\item[$s\_name$] Name of internal system variable.
\end{description}

Value Returned

\begin{description}
\item[$g\_switchValue$] Status of the internal system variable, usually either \texttt{t} or \texttt{nil}.
\end{description}

Example

\begin{verbatim}
status( debugMode ) => t
\end{verbatim}

Checks the status of \texttt{debugMode} and returns \texttt{t} if \texttt{debugMode} is on.

The \texttt{status} function gets a switch. The \texttt{sstatus} function sets a switch.

\begin{verbatim}
status debugMode ; read the current value of the switch => nil
sstatus debugMode t ; set the value of the switch to new value => t
status debugMode => t
\end{verbatim}

Reference

\texttt{sstatus}
strcat

strcat(
    S_string1
    [ S_string2 ... ]
)
=> t_result

Description

Takes input strings or symbols and concatenates them.

Arguments

S_string1  S_string2  ...  
One or more input strings or symbols.

Value Returned

Value Returned

{t_result}  
New string containing the contents of all input strings or symbols 
S_string1, S_string2, ..., concatenated together. 
The input arguments are left unchanged.

Example

strcat( 'ab "xyz" ')  =>  "abxyz"
strcat( "l" "ab" "ef" )  =>  "labef"

Reference

buildString, concat, strncat, strcmp, strncmp, substring
**strcmp**

```
strcmp(
    t_string1
    t_string2
)
=> 1 | 0 | -1
```

**Description**

Compares two argument strings alphabetically.

Compares the two argument strings `t_string1` and `t_string2` and returns an integer greater than, equal to, or less than zero depending on whether `t_string1` is alphabetically greater, equal to, or less than `t_string2`. To simply test if the contents of two strings are the same, use the `equal` function.

**Arguments**

- `t_string1`: First string to be compared.
- `t_string2`: Second string to be compared.

**Value Returned**

- `1`: `t_string1` is alphabetically greater than `t_string2`.
- `0`: `t_string1` is alphabetically equal to `t_string2`.
- `-1`: `t_string1` is alphabetically less than `t_string2`.

**Example**

```
strcmp( "abc" "abb" ) => 1
strcmp( "abc" "abc" ) => 0
strcmp( "abc" "abd" ) => -1
```

**Reference**

- `equal`, `strncmp`
stringp

stringp(  
    g_value
  )
=> t | nil

Description

Checks if an object is a string.

The suffix `p` is usually added to the name of a function to indicate that it is a predicate function.

Arguments

`g_value` A data object.

Value Returned

`t` `g_value` is a string.

`nil` Otherwise.

Example

stringp( 93)  => nil
stringp( "93")  => t

Reference

`listp`, `symbolp`
stringToFunction

```
stringToFunction(
    t_string
    [ s_langMode ]
) => u_function
```

Description

Wraps and converts a string of SKILL code into a parameterless SKILL function.

 Parses the given string argument and wraps the result with a parameterless `lambda`, then compiles the entire form into a function object. The returned function can later be `applied` with better performance than direct evaluation using `evalstring`.

Arguments

- `t_string` String representing some SKILL code.
- `s_langMode` Must be a symbol.

Valid values:

- `'ils` Treats the string as SKILL++ code.
- `'il` Treats the string as SKILL code.

Value Returned

- `u_function` Parameterless function equivalent to evaluating the string
  `(lambda() t_string)`.

Example

```
f = stringToFunction("1+2") => funobj:0x220038
apply(f nil) => 3
```

Reference

`evalstring`, `apply`
stringToSymbol

stringToSymbol(
    t_string
)
=> s_symbolName

Description

Converts a string to a symbol of the same name.

Arguments

t_string
  String to convert to a symbol.

Value Returned

s_symbolName
  Symbol for the given string.

Example

y = stringToSymbol( "test"
) => test
sprintf(nil "%L" y)
=> "test"

Reference

concat, symbolToString
stringToTime

stringToTime(
    t_time
)
  => x_time

Description

Given a date and time string, returns an integer time value representation. The time argument
must be in the format as returned by the timeToString function, such as: Dec 28
16:57:06 1996.

All time conversion functions assume local time, not GMT time.

Arguments

:t_time
  String indicating a time and date in this format: "Dec 28
16:57:06 1996". Same as format returned by timeToString
or getCurrentTime.

Value Returned

:x_time
  Integer time value.

Example

fileTimeModified( "~/cshrc" )
=> 793561559
TimeString(793561559)
=> "Feb 23 09:45:59 1995"
stringToTime("Feb 23 09:45:59 1995")
=> 793561559

Reference

getCurrentTime, timeToString, timeToTm, tmToTime
strlen

strlen(
    t_string
)
=> x_length

Description

Returns the number of characters in a string.

Arguments

  t_string  String length you want to obtain.

Value Returned

  x_length    Length of t_string.

Example

strlen( "abc" ) => 3
strlen( "\007" ) => 1 ; Backslash notation used.

Reference

  index, parseString, substring, strcat, strcmp, strncmp, stringp
strncat

```lisp
strncat(
    t_string1
    t_string2
    x_max
)
=> t_result
```

**Description**

Creates a new string by appending a maximum number of characters from `t_string2` to `t_string1`.

Concatenates input strings. Similar to `strcat` except that at most `x_max` characters from `t_string2` are appended to the contents of `t_string1` to create the new string. `t_string1` and `t_string2` are left unchanged.

**Arguments**

- `t_string1`: First string included in the new string.
- `t_string2`: Second string whose characters are appended to `t_string1`.
- `x_max`: Maximum number of characters from `t_string2` that you want to append to the end of `t_string1`.

**Value Returned**

- `t_result`: The new string; `t_string1` and `t_string2` are left unchanged.

**Example**

```
strncat( "abcd" "efghi" 2)  => "abcdefgh"
strncat( "abcd" "efghijk" 5) => "abcdefghi"
```

**Reference**

- `parseString`
- `strcat`
- `strcmp`
- `strncmp`
- `substring`
- `stringp`
**strncmp**

```c
strncmp(
    t_string1
    t_string2
    x_max
)
=> 1 | 0 | -1
```

**Description**

Compares two argument strings alphabetically only up to a maximum number of characters.

Similar to `strcmp` except that only up to `x_max` characters are compared. To simply test if the contents of two strings are the same, use the `equal` function.

**Arguments**

- `t_string1` : First string to be compared.
- `t_string2` : Second string to be compared.
- `x_max` : Maximum number of characters in both strings to be compared.

**Value Returned**

For the first specified number of characters:

- `1` : `t_string1` is alphabetically greater than `t_string2`
- `0` : `t_string1` is alphabetically equal to `t_string2`.
- `-1` : `t_string1` is alphabetically less than `t_string2`.

**Example**

```c
strncmp( "abc" "ab" 3) => 1
strncmp( "abc" "de" 4) => -1
strncmp( "abc" "ab" 2) => 0
```
Reference

`equal`, `strcmp`
**sub1**

```sub1(n_number)
  => n_result```

**Description**

Subtracts one from a floating-point number or integer.

**Arguments**

- `n_number` Floating-point number or integer.

**Value Returned**

- `n_result` `n_number` minus one.

**Example**

```sub1( 59 )
=> 58```

**Reference**

- `add1`
**subst**

```plaintext
subst(
    g_x
    g_y
    l_arg
)
=> l_result
```

**Description**

Substitutes one object for another object in a list.

**Arguments**

- **g_x**  
  Object substituted.

- **g_y**  
  Object substituted for.

- **l_arg**  
  A list.

**Value Returned**

- **l_result**  
  Result of substituting **g_x** for all equal occurrences of **g_y** at all levels in **l_arg**.

**Example**

```plaintext
subst( 'a 'b '(a b c) ) => (a a c)
subst('x 'y '(a b y (d y (e y)))) => (a b x (d x (e x )))
```

**Reference**

- [remd](#)
substring

substring(
    S_string
    x_index
    [ x_length ]
)
 => t_result | nil

Description

Creates a new substring from an input string, starting at an index point and continuing for a given length.

Creates a new substring from S_string with a starting point determined by x_index and length determined by an optional third argument x_length.

- If S_string is a symbol, the substring is taken from its print name.
- If x_length is not given, then all of the characters from x_index to the end of the string are returned.
- If x_index is negative the substring begins at the indexed character from the end of the string.
- If x_index is out of bounds (that is, its absolute value is greater than the length of S_string), nil is returned.

Arguments

S_string A string.

x_index Starting point for returning a new string. Cannot be zero.

x_length Length of string to be returned.

Value Returned

 t_result Substring of S_string starting at the character indexed by x_index, with a maximum of x_length characters.

 nil If x_index is out of bounds.
Example

substring("abcdef" 2 4) => "bcde"
substring("abcdef" 4 2) => "de"
substring("abcdef" -4 2) => "cd"

Reference

parseString
sxtd

sxtd(
    x_number
    x_bits
)
=> x_result

Description

Sign-extends the number represented by the rightmost specified number of bits in the given integer.

Sign-extends the rightmost \textit{x_bits} bits of \textit{x_number}. That is, sign-extends the bit field \textit{x_number}<x_bits - 1:0> with \textit{x_number}<x_bits - 1> as the sign bit.

Arguments

\textit{x_number} \hspace{1cm} \text{An integer.}

\textit{x_bits} \hspace{1cm} \text{Number of bits.}

Value Returned

\textit{x_result} \hspace{1cm} \text{\textit{x_number} with the rightmost \textit{x_bits} sign-extended.}

Example

sxtd( 7 4 ) => 7
sxtd( 8 4 ) => -8
sxtd( 5 2 ) => 5

Reference

zxtd
symbolp

\[
\text{symbolp}(\ g\_value\ ) \Rightarrow t | \text{nil}
\]

**Description**

Checks if an object is a symbol.

The suffix \( p \) is usually added to the name of a function to indicate that it is a predicate function.

**Arguments**

\( g\_value \)

A data object.

**Value Returned**

\( t \)

If \( g\_value \) is a symbol.

\( \text{nil} \)

Otherwise.

**Example**

\[
\begin{align*}
\text{symbolp( 'foo')} & \Rightarrow t \\
\text{symbolp( "foo")} & \Rightarrow \text{nil} \\
\text{symbolp( concat("foo")}) & \Rightarrow t
\end{align*}
\]

**Reference**

concat, stringp
symbolToString

symbolToString(
    s_symbolName
)
=> t_string

Description

Converts a symbol to a string of the same name. Same as `get_pname`.

Arguments

s_symbolName          Symbol to convert.

Value Returned

t_string              String with the same name as the input symbol.

Example

y = symbolToString( 'test2)
=> "test2"
sprintf(nil "%L" y)
=> "\"test2\""

Reference

`get_pname`, `stringToSymbol`
symeval

symeval(
    s_symbol
    [ e_environment ]
) => g_result

Description

Returns the value of the named variable.

symeval is slightly more efficient than eval and can be used in place of eval when you are sure that the argument being evaluated is indeed a variable name.

Arguments

s_symbol Name of the variable.

e_environment If this argument is given, SKILL++ semantics is assumed. The variable name will be looked up within the given (lexical) environment.

Value Returned

g_result Value of the named variable.

Example

x = 5
symeval( 'x ) => 5
symeval( 'y ) => unbound ;Assumes y is unbound.

Reference

eval
symstrp

symstrp(  
    g_value  
)  
=> t | nil

Description

Checks if an object is either a symbol or a string.

The suffix p is usually added to the name of a function to indicate that it is a predicate function.

Arguments

$g_value$  
A data object.

Value Returned

t  
If $g_value$ is either a symbol or a string.

nil  
Otherwise.

Example

symstrp( "foo" ) => t  
symstrp( 'foo' ) => t  
symstrp( 3 ) => nil

Reference

stringp, symbolp
**system**

system(
    t_command
) => x_result

**Description**

Spawns a separate UNIX process to execute a command.

**Arguments**

*t_command*  
Command to execute.

**Value Returned**

*x_result*  
The return code caused by executing the given UNIX command.

**Example**

```system( "date" )
Wed Dec 14 15:14:53 PST 1994
0
system( "daa" )
sh: daa: not found
1```

**Reference**

csh, sh, shell
tablep

tablep(
    g_object
) => t | nil

Description

Checks if an object is an association table.

Arguments

  g_object       A SKILL object.

Value Returned

  t             If g_object is an association table.

  nil           If g_object is not an association table.

Example

  myTable = makeTable("atable1" 0) => table:atable1
  tablep(myTable) => t
  tablep(9) => nil

Reference

  makeTable
tableToList

tableToList(  
    o_table
  )
  => l_assoc_list

Description

Converts the contents of an association table to an association list. Use this function interactively to look at the contents of a table.

Note: This function eliminates the efficiency that you gain from referencing data in an association table. Do not use this function for processing data in an association table. Instead, use this function interactively to look at the contents of a table.

Arguments

o_table  Association table to be converted.

Value Returned

l_assoc_list  Association list containing key/value pairs from the association table.

Example

myTable = makeTable( "table" 0)  => table:table
myTable[ "first"] = 1  => 1
myTable[ 'two'] = 2  => 2
tableToList(myTable)  => ((two 2)("first" 1))

Reference

makeTable, tablep
tailp

tailp(
  l_arg1
  l_arg2
)
=> l_arg1 | nil

Description

Returns \textit{arg1} if a list cell \textit{eq} to \textit{arg1} is found by \texttt{cdr} down \textit{arg2} zero or more times, \texttt{nil} otherwise.

Because \textit{eq} is being used for comparison \texttt{l_arg1} must actually point to a tail list in \texttt{l_arg2} for this predicate to return a non-\texttt{nil} value.

Arguments

\begin{itemize}
  \item \texttt{l_arg1} A list.
  \item \texttt{l_arg2} Another list, which can contain \texttt{l_arg1} as its tail.
\end{itemize}

Value Returned

\begin{itemize}
  \item \texttt{l_arg} If a list cell \textit{eq} to \texttt{l_arg1} is found by \texttt{cdr}ing down \texttt{l_arg2} zero or more times.
  \item \texttt{nil} Otherwise.
\end{itemize}

Example

\begin{verbatim}
y = '(b c)
z = cons( 'a y ) => (a b c)
tailp( y z ) => (b c)
tailp( '(b c) z ) => nil
\end{verbatim}

\texttt{nil} was returned because \texttt{'(b c)} is not \textit{eq} the \texttt{cdr( z )}.

Reference

\begin{itemize}
  \item \texttt{cdr}, \texttt{eq}
\end{itemize}
**tan**

\[
\tan(\ n\_\text{number} \ )
\]

\[\Rightarrow f\_\text{result}\]

**Description**

Returns the tangent of a floating-point number or integer.

**Arguments**

\[n\_\text{number}\]

Floating-point number or integer.

**Value Returned**

\[f\_\text{result}\]

Tangent of \(n\_\text{number}\).

**Example**

\[
\tan(\ 3.0\ ) \Rightarrow -0.1425465
\]

**Reference**

atan, cos, sin
**tconc**

`tconc(`
```
l_ptr
g_x
```
`)
`=> l_result`

**Description**

Creates a list cell whose **car** points to a list of the elements being constructed and whose **cdr** points to the last list cell of the list being constructed.

A **tconc** structure is a special type of list that allows efficient addition of objects to the end of a list. It consists of a list cell whose **car** points to a list of the elements being constructed with **tconc** and whose **cdr** points to the last list cell of the list being constructed. If **l_ptr** is **nil**, a new **tconc** structure is automatically created. To obtain the list under construction, take the **car** of the **tconc** structure.

**tconc** and **lconc** are much faster than **append** when adding new elements to the end of a list. The **append** function is much slower, because it traverses and copies the list to reach the end, whereas **tconc** and **lconc** only manipulate pointers.

`l_ptr`  
A **tconc** structure. Must be initialized to **nil** to create a new **tconc** structure.

`g_x`  
Element to add to the end of the list.

**Value Returned**

`l_result`  
Returns **l_ptr**, which must be a **tconc** structure or **nil**, with **g_x** added to the end.

**Example**

```
x = tconc(nil 1) ; x is now ((1) 1)
tconc(x 2) ; x is now ((1 2) 2)
tconc(x 3) ; x is now ((1 2 3) 3)
x = car(x) ; x is now (1 2 3)
```

**x now equals (1 2 3)**, the desired result.
Reference

append, car, cdr, lconc
theEnvironment - SKILL++ mode only

theEnvironment(
    [ u_funobj ]
) => e_environment | nil

Description

Returns the top level environment if called from a SKILL++ top-level. Returns the enclosing lexical environment if called within a SKILL++ function. Returns the associated environment if passed a SKILL++ function object. Otherwise returns nil.

- In SKILL++, there is a unique top-level environment that implicitly encloses all other local environments. If you do not pass the optional argument, when you call theEnvironment from a SKILL++ top-level, theEnvironment returns this environment. The schemeTopLevelEnv function also returns this environment.

- If you call theEnvironment from within a SKILL++ function and if you do not pass the optional argument, theEnvironment returns the enclosing lexical environment.

- If you are in debug mode, you can pass a closure to theEnvironment. A closure is another term for a function object returned by evaluating a SKILL++ lambda expression which abstractly, consists of two parts:
  - The code for the lambda expression.
  - The environment in which the free variables in the body are bound when the lambda expression is evaluated.

- If you call theEnvironment from a SKILL function and do not pass a closure, then theEnvironment function returns nil.

Arguments

* u_funobj
  
  Optional argument. Should be a SKILL++ closure.

Value Returned

* nil
  
  Returned when called from a SKILL function and you do not pass a SKILL++ closure as the optional argument.
**e_environment**

Either the top-level environment, or the enclosing environment, or the closure’s environment.

**Example**

\[ Z = \text{let}( ( x ) \]
\[ x = 3 \]
\[ \text{theEnvironment}() \]
\[ ) ; \text{let} \]
\[ => \text{envobj:0x1e0060} \]

Returns the environment that the \text{let} expression establishes. The value of \text{Z} is an environment in which \text{x} is bound to 3. Each time you execute the above expression, it returns a different environment object, as you can tell by observing the print representation.

\[ Z = \text{let}( ( ( x \text{theEnvironment})()) \]
\[ x \]
\[ ) \]
\[ => \text{envobj:0x2fc018} \]
\[ \text{eq( schemeTopLevelEnv( Z ) )} => t \]

Uses \text{theEnvironment} to illustrate that the variable initialization expressions in a \text{let} expression refer to the enclosing environment.

\[ V = \text{letrec}( ( ( x \text{theEnvironment})()) \]
\[ x \]
\[ ) \]
\[ => \text{envobj:0x33506c} \]
\[ \text{eq( schemeTopLevelEnv( V ) )} => \text{nil} \]
\[ \text{eq( V~>x V )} => t \]

Uses \text{theEnvironment} to illustrate that the variable initialization expressions in a \text{letrec} expression refers to the \text{letrec}’s environment.

\[ W = \text{let}( ( ( r 3 ) ( y 4 )) \]
\[ \text{let}( ( ( z 5 ) ( v 6 )) \]
\[ \text{theEnvironment}() \]
\[ ) ; \text{let} \]
\[ ) ; \text{let} \]
\[ => \text{envobj:0x456030c} \]
\[ W~>r => 3 \]
\[ W~>z => 5 \]
\[ W~>?? => ((z(5) (v 6)) ((r 3) y(4))) \]

Returns the environment that the nested \text{let} expressions establish. Notice that assigning it to the top-level variable \text{W} makes it persistent.

\[ Q = \text{letrec(} \]
\[ ( ;; \text{begin locals} \]
\[ ( X 6 ) \]
\[ ( \text{self} \]
\[ \text{lambda}( ( ) \]
\[ \text{theEnvironment}() \]
\[ ) ; \text{lambda} \]
\[ ) ; \text{self} \]
\[ ) ;; \text{end of locals} \]
Returns a function object which, in turn, returns its local environment.

Reference

schemeTopLevelEnv, envobj, funobj
times

```
times(
    n_op1
    n_op2
    [ n_op3 ... ]
)
=> n_result
```

**Description**

Returns the result of multiplying the first operand by one or more operands. Prefix form of the * arithmetic operator.

**Arguments**

- **n_op1**: First operand to be multiplied.
- **n_op2**: Second operand to be multiplied.
- **n_op3**: Optional additional operands to be multiplied.

**Value Returned**

- **n_result**: Result of the multiplication.

**Example**

```
times(5 4 3 2 1) => 120
.times(-12 -13) => 156
times(12.2 -13.3) => -162.26
```

**Reference**

xtimes
timeToString

timeToString(
   x_time
)
=> t_time

Description

Takes an integer UNIX time value, returns a formatted string that the value denotes. The string is always in a form like: Dec 28 16:57:06 1994.

Arguments

x_time Integer time value.

Value Returned

t_time Formatted string the value denotes.

Example

fileTimeModified( "~/.cshrc" )
=> 793561559
timeToString(793561559)
=> "Feb 23 09:45:59 1995"
stringToTime("Feb 23 09:45:59 1995")
=> 793561559

Reference

fileTimeModified, stringToTime, timeToTm
timeToTm

timeToTm(
    x_time
)
=> r_tm

Description

Given an integer time value, returns a tm structure.

r_tm is a defstruct similar to POSIX's tm struct:

```c
struct tm {
    int tm_sec; /* seconds after the minute: [0, 61] */
    int tm_min; /* minutes after the hour: [0, 59] */
    int tm_hour; /* hours after midnight: [0, 23] */
    int tm_mday; /* day of the month: [1, 31] */
    int tm_mon; /* month of the year: [0, 11] */
    int tm_year; /* year since 1900 */
    int tm_wday; /* days since Sunday: [0, 6] */
    int tm_yday; /* days since January: [0, 365] */
    int tm_isdst; /* daylight saving time flag: <0,0,>0*/
};
```

- Use x->?? to get all its fields.
- Use x->tm_sec and so forth to access individual fields.

All time conversion functions assume local time, not GMT time.

Arguments

x_time                    Integer time value.

Value Returned

r_tm                      A defstruct similar to POSIX's tm struct.

Example

defTimeModified( "~/.cshrc" )
=> 793561559

timeToString(793561559)
=> "Feb 23 09:45:59 1995"
x = timeToTm(793561559)
=>array[11]:1702872
x->??
(tm_sec 59 tm_min 45 tm_hour  
 9 tm_mday 23 tm_mon 1  
 tm_year 95 tm_wday 4 tm_yday  
 53 tm_isdst 0)

x->tm_mon
=>1

Reference

fileTimeModified, stringToTime, timeToString, tmToTime
tmToTime

tmToTime(
    r_tm
)
=> x_time

Description

Given a tm structure, returns the integer value of the time it represents.

r_tm is a defstruct similar to POSIX's tm struct:

```
struct tm {
    int     tm_sec; /* seconds after the minute: [0, 61] */
    int     tm_min; /* minutes after the hour: [0, 59] */
    int     tm_hour; /* hours after midnight: [0, 23] */
    int     tm_mday; /* day of the month: [1, 31] */
    int     tm_mon; /* month of the year: [0, 11] */
    int     tm_year; /* year since 1900 */
    int     tm_wday; /* days since Sunday: [0, 6] */
    int     tm_yday; /* days since January: [0, 365] */
    int     tm_isdst; /* daylight saving time flag: <0,0,>0*/
};
```

- Use x->?? to get all its fields.
- Use x->tm_sec and so forth to access individual fields.

All time conversion functions assume local time, not GMT time.

Arguments

r_tm A defstruct similar to POSIX's tm struct.

Value Returned

x_time Integer time value.

Example

fileTimeModified( "~/.cshrc" )
=> 793561559
timeToString(793561559)
=> "Feb 23 09:45:59 1995"
x = timeToTm(793561559)
=>array[11]:1702872
x->??
(tm_sec 59 tm_min 45 tm_hour
  9 tm_mday 23 tm_mon 1
  tm_year 95 tm_wday 4 tm_yday
  53 tm_isdst 0)
}
tmToTime(x)
=> 793561559

Reference

fileTimeModified, stringToTime, timeToString, timeToTm
**truncate**

```plaintext
truncate(
    n_number
)
=> x_integer
```

**Description**

Truncates a given number to an integer.

**Arguments**

- `n_number` Any SKILL number.

**Value Returned**

- `x_integer` `n_number` truncated to an integer.

**Example**

```plaintext
truncate( 1234.567) => 1234
round( 1234.567) => 1235
truncate( -1.7) => -1
```

**Reference**

- `ceiling`, `floor`, `round`
**type, typep**

```lisp
(type
    g_value
) => s_type | nil

typep(
    g_value
) => s_type | nil
```

**Description**

Returns a symbol whose name denotes the type of a data object. The functions `type` and `typep` are identical.

**Arguments**

- `g_value` A data object.

**Value Returned**

- `s_type` Symbol whose name denotes the type of `g_value`.
- `nil` Otherwise.

**Example**

```lisp
(type 'foo) => symbol
(typep "foo") => string
```

**Reference**

- `fixp`, `floatp`, `numberp`, `portp`, `stringp`, `symbolp`
unalias

unalias(
   \textit{s\_aliasName1} ...
)
=> \textit{l\_result}

\textbf{Description}

Undefines the aliases specified in an argument list and returns a list containing the aliases undefined by the call. This is \texttt{nlambda} function also works in SKILL++ mode.

\textbf{Caution}

\textit{Use alias for interactive command entry only and never in programs.}

\textbf{Arguments}

\textit{s\_aliasName1} \hspace{2cm} Symbol name of the alias.

\textbf{Value Returned}

\textit{l\_result} \hspace{2cm} List of the aliases removed.

\textbf{Example}

alias path getSkillPath => path

Aliases \texttt{path} to the \texttt{getSkillPath} function.
unalias path => (\texttt{path})

Removes \texttt{path} as an alias.

\textbf{Reference}

\texttt{alias}
unless

unless(
    g_condition
    g_expr1 ...
)
=> g_result | nil

Description

Evaluates a condition. If the result is true (non-nil), it returns nil; otherwise evaluates the body expressions in sequence and returns the value of the last expression. This is a syntax form.

The semantics of this function can be read literally as "unless the condition is true, evaluate the body expressions in sequence".

Arguments

g_condition Any SKILL expression.
g_expr1 Any SKILL expression.

Value Returned

g_result Value of the last expression of the sequence g_expr1 if g_condition evaluates to nil.
nil If g_condition evaluates to non-nil.

Example

x = -123
unless( x >= 0 println("x is negative") -x)
=> 123 ; Prints "x is negative" as side effect.
unless( x < 0 println("x is positive") x)
=> nil

Reference

cond, if, when
**upperCase**

```skill
upperCase(
    S_string
) => t_result
```

**Description**

Returns a string that is a copy of the given argument with the lowercase alphabetic characters replaced by their uppercase equivalents.

If the parameter is a symbol, the name of the symbol is used.

**Arguments**

- **S_string**: Input string or symbol.

**Value Returned**

- **t_result**: Copy of `S_string` in uppercase letters.

**Example**

```skill
upperCase("Hello world!") => "HELLO WORLD!"
```

**Reference**

- `lowerCase`
vector

definition:
vector(g_value ...)

return:
a_vectorArray

Description

Returns a vector, or array, filled with the arguments in the given order. The `vector` function is analogous to the `list` function.

A vector is implemented as a SKILL array.

Arguments

g_value

Ordered list of values to be placed in an array.

Value Returned

a_vectorArray

Array filled with the arguments in the given order.

Example

V = vector(1 2 3 4) => array[4]:33394440
V[0] => 1
V[3] => 4

Reference

declare, list, listToVector, makeVector, vectorToList
vectorp

vectorp(
    g_value
)
=> t | nil

Description

Checks if an object is a vector. Behaves the same as arrayp.

The suffix p is usually added to the name of a function to indicate that it is a predicate function.

Arguments

g_value Any data object.

Value Returned

t If g_value is a vector object.
nil Otherwise.

Example

declare(x[10])
arrayp(x) => t
arrayp('x) => nil

Reference

declare, arrayp
vectorToList

vectorToList(
    a_vectorArray
)
=> l_list

Description

Returns a list containing the elements of an array.

Arguments

a_vectorArray     Vector to be converted.

Value Returned

l_list              List constructed from the given vector.

Example

vectorToList( vector( 1 2 3 ) )
=> ( 1 2 3 )
vectorToList( makeVector( 3 "Hi") )
=> ("Hi" "Hi""Hi")

Reference

declare, vector, listToVector, makeVector
vi, vii, vil

vi(
    [ S_fileName ]
) => t | nil

Description

Edits a file using the vi editor. This is an nlambda function. Edits the named file using the vi editor, and optionally includes (vii) or loads (vil) the file into SKILL after exiting the editor. These functions are just variants of ed, edi, and edl with explicit request for using the vi editor.

Arguments

S_fileName File to edit. If no argument is given, defaults to the previously edited file, or temp.il, if there is no previous file.

Value Returned

t If the operation was successfully completed.

nil If the file does not exit or there is an error condition.

Example

vil( "test.il" )
vi()

Reference

ed, edi, edl, edit
**warn**

```matlab
warn(
    t_formatString
    [ g_arg1 ... ]
)
=> nil
```

**Description**

Buffers a warning message with given arguments inserted using the same format specification as `sprintf`, `printf`, and `fprintf`.

After a function returns to the top level, the buffered warning message is printed in the Command Interpreter Window. Arguments to `warn` use the same format specification as `sprintf`, `printf`, and `fprintf`.

This function is useful for printing SKILL warning messages in a consistent format. You can also suppress a message with a subsequent call to `getWarn`.

**Arguments**

- `t_formatString` Characters to print verbatim in the warning message with format specifications prefixed by the percent (%) sign.
- `g_arg1 ...` Optional arguments following the format string, which are printed according to their corresponding format specifications.

**Value Returned**

`nil` Always returns `nil`.

**Example**

```matlab
arg1 = 'fail
warn( "setSkillPath: first argument must be a string or list of strings - %s\n" arg1)
=> nil

*WARNING* setSkillPath: first argument must be a string or list of strings - fail
```
Reference

fprintf, getWarn, printf, sprintf
when

when(
    g_condition
    g_expr1 ...
)
=> g_result | nil

Description

Evaluates a condition. If the result is non-nil, evaluates the sequence of expressions and returns the value of the last expression. This is a syntax form.

If the result of evaluating g_condition is nil, when returns nil.

Arguments

g_condition Any SKILL expression.
g_expr1 Any SKILL expression.

Value Returned

g_result Value of the last expression of the sequence g_expr1 if g_condition evaluates to non-nil.
nil If the g_condition expression evaluates to nil.

Example

x = -123
when( x < 0
    println("x is negative")
    -x)
=> 123 ;Prints "x is negative" as side effect.
when( x >= 0
    println("x is positive")
    x)
=> nil

Reference

cond, if, unless
which

which(  
    t_fileName
)
=> tFullPath | nil

Description

Returns the absolute path of the given context file, or regular file or directory.

The main usage of this function is to load prerequisite context files.

If t_fileName identifies a context file (that is with the .cxt extension), it looks under the standard contexts location (associated with the application in which this function is called), as well as common Cadence contexts directory, your_install_path/tools/dfII/etc/context, and user contexts location, youre_install_path/tools/dfII/local/context, for the presence of the context file.

If t_fileName identifies a regular file or directory, the current SKILL path is searched. Note that a path which is anchored to current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

 t_fileName    Name of a context file, or a regular file or directory that you want to get the absolute path.

Value Returned

 tFullPath      The absolute path of t_fileName.

 nil           If t_fileName is not found.

Example

Loading a prerequisite context file:
loadContext( which( "myPrereq.cxt" ) ) => t

Get the absolute path of a file:
which( ".cdsinit" ) => "/usr/michaelc/.cdsinit"
while

while(
  g_condition
  g_expr1 ...
)
=> t

Description

Repeatedly evaluates a condition and sequence of expressions until the condition evaluates to false. This is a syntax form.

Repeatedly evaluates \texttt{g\_condition} and the sequence of expressions \texttt{g\_expr1 \ldots} if the condition is true. This process is repeated until \texttt{g\_condition} evaluates to false (\texttt{nil}). Note that because this form always returns \texttt{t}, it is principally used for its side-effects.

Arguments

\begin{itemize}
  \item \texttt{g\_condition} \hspace{1em} \text{Any SKILL expression.}
  \item \texttt{g\_expr1} \hspace{1em} \text{Any SKILL expression.}
\end{itemize}

Value Returned

\begin{itemize}
  \item \texttt{t} \hspace{1em} \text{Always returns \texttt{t}.}
\end{itemize}

Example

\begin{verbatim}
i = 0
while( (i <= 10) printf("%d\n" i++) )
=> t
\end{verbatim}

Prints the digits 0 through 10.

Reference

\texttt{for, foreach}
write

write(
    g_value
    [ p_outputPort ]
)
=> nil

Description

Prints a SKILL object using the default format for the data type of the value.

For example, strings are enclosed in ". Same as print.

Arguments

- g_value: Any SKILL object.
- p_outputPort: Output port to print to. Default is pOutput.

Value Returned

- nil: Always returns nil, after it prints out the object supplied to it.

Example

for( i 1 3 write( "hello" )) ;Prints hello three times.
"hello""hello""hello"
=> t

Reference

display, pprint, print, println, printlev
writeTable

writeTable(
  S_fileName
  o_table
)
  => t | nil

Description

Writes the contents of an association table to a file with one key/value pair per line.

Note: This function is for writing basic SKILL data types that are stored in an association table. The function cannot write database objects or other user-defined types that might be stored in association tables.

Arguments

S_fileName  Name of the print file (either a string or symbol) to which the table contents are to be written.

o_table  Association table from which the data is accessed.

Value Returned

t  If the data is successfully written to the file.

nil  Otherwise.

Example

writeTable("inventory" myTable)  => t
writeTable(noFile myTable)  => nil

Reference

makeTable, readTable
xcons

xcons(
    l_list
    g_element
)
=> l_result

Description

Adds an element to the beginning of a list. Equivalent to cons but the order of the arguments is reversed.

Arguments

l_list
A list, which can be nil.

g_element
Element to be added to the beginning of l_list.

Value Returned

l_result
Returns a list.

Example

xcons( '(b c) 'a ) => ( a b c )

Reference

append, append1, cons, lconc, list, ncons, tconc
**xCoord**

```l_list
xCoord( l_list )
=> g_result
```

**Description**

Returns the first element of a list. **Does not modify the argument list.**

**Note:** The `xCoord` and `yCoord` functions are aliases for the `car` and `cadr` functions.

**Arguments**

- `l_list` A list of elements.

**Value Returned**

- `g_result` Returns the first element in a list.

**Example**

```text
xValue = 300
yValue = 400
aCoordinate = xValue:yValue => ( 300 400 )
xCoord( aCoordinate ) => 300
yCoord( aCoordinate ) => 400
```

**Reference**

- `car`
xdifference

xdifference(
    x_op1
    x_op2
    [ x_opt3 ]
) => x_result

Description

Returns the integer result of subtracting one or more operands from the first operand. xdifference is an integer-only arithmetic function while difference can handle integers and floating-point numbers. xdifference runs slightly faster than difference in integer arithmetic calculation.

Arguments

x_op1 OPERAND FROM WHICH ONE OR MORE OPERANDS ARE SUBTRACTED.

x_op2 OPERAND TO BE SUBTRACTED.

x_opt3 OPTIONAL ADDITIONAL OPERANDS TO BE SUBTRACTED.

Value Returned

x_result Result of the subtraction.

Example

xdifference(12 13) => -1
xdifference(-12 13) => -25

Reference

difference
xplus

\[
xplus(\quad x_{\text{op1}} \\
x_{\text{op2}} \\
\quad[\ x_{\text{opt3}} \ ] \\
\quad) \\
=> x_{\text{result}}
\]

Description

Returns the integer result of adding one or more operands to the first operand. xplus is an integer-only arithmetic function while plus can handle integers and floating-point numbers. xplus runs slightly faster than plus in integer arithmetic calculation.

Arguments

\[
x_{\text{op1}} \quad \text{First operand to be added.}
\]
\[
x_{\text{op2}} \quad \text{Second operand to be added.}
\]
\[
x_{\text{opt3}} \quad \text{Optional additional operands to be added.}
\]

Value Returned

\[
x_{\text{result}} \quad \text{Result of the addition.}
\]

Example

\[
xplus(12 \ 13) \ => \ 25 \\
xplus(-12 \ -13) \ => \ -25
\]

Reference

plus
xquotient

xquotient(
  x_op1
  x_op2
  [ x_opt3 ]
)
=> x_result

Description

Returns the integer result of dividing the first operand by one or more operands. xquotient is an integer-only arithmetic function while quotient can handle integers and floating-point numbers. xquotient runs slightly faster than quotient in integer arithmetic calculation.

Arguments

x_op1       Dividend.
x_op2       Divisor.
x_opt3      Optional additional divisors.

Value Returned

x_result     Result of the division.

Example

xquotient(10 2) => 5
xquotient(-10 -2) => 5

Reference

quotient
xtimes

xtimes(  
    x_op1  
    x_op2  
    [ x_opt3 ]  
)  
=> x_result

Description

Returns the integer result of multiplying the first operand by one or more operands. xtimes is an integer-only arithmetic function while times can handle integers and floating-point numbers. xtimes runs slightly faster than times in integer arithmetic calculation.

Arguments

x_op1       First operand to be multiplied.  
x_op2       Second operand to be multiplied.  
x_opt3      Optional additional operands to be multiplied.

Value Returned

x_result    Result of the multiplication.

Example

xtimes(12 13) => 156
xtimes(-12 -13) => 156

Reference

times
yCoord

yCoord(
    l_list
)
=> g_result

Description

Returns the tail of the list, that is, the list without its first element.

Note: The xCoord and yCoord functions are aliases for the car and cadr functions.

Arguments

l_list

A list of elements.

Value Returned

g_result

Returns the end of a list, or the list minus the first element.

Example

xValue = 300
yValue = 400
aCoordinate = xValue:yValue => ( 300 400 )
xCoord( aCoordinate ) => 300
yCoord( aCoordinate ) => 400

Reference

cdr
zerop

zerop(
    n_num
)
=> t | nil

Description

Checks if a value is equal to zero.
zerop is a predicate function.

Arguments

n_num          Number to check.

Value Returned

t          If n_num is equal to zero.
nil         Otherwise.

Example

zerop( 0 )
=> t
zerop( 7 )
=> nil

Reference

evenp, minusp, oddp, onep, plusp
zxtd

\[
\text{zxtd}( \ x\_\text{number} \\
\ x\_\text{bits} \\
) \\
\Rightarrow x\_\text{result}
\]

Description

Zero-extends the number represented by the rightmost specified number of bits in the given integer.

Zero-extends the rightmost \( x\_\text{bits} \) bits of \( x\_\text{number} \). Executes faster than doing \( x\_\text{number}<x\_\text{bits} - 1:0> \).

Arguments

\( x\_\text{number} \) An integer.

\( x\_\text{bits} \) Number of bits.

Value Returned

\( x\_\text{result} \) \( x\_\text{number} \) with the rightmost \( x\_\text{bits} \) zero-extended.

Example

\[
\text{zxtd}( 8 \ 3 ) \Rightarrow 0 \\
\text{zxtd}( 10 \ 2 ) \Rightarrow 2
\]

Reference

\( \text{sxt}\)
### Scheme/SKILL++ Equivalents Tables

The purpose of this appendix is to help users familiar with Scheme to get a jump start with SKILL++. All of Scheme's special (syntax) forms and functions are listed along with their SKILL++ equivalents.

The tables, which are divided into expressions, lexical structure, and functions, use these terms:

- **Same**
  - Means that this Scheme functionality is provided with the same name (syntax) and same behavior in SKILL++.

- **Supported**
  - Means that this Scheme functionality is provided, but it is implemented under a different name and/or is used somewhat differently. For example,
    - (1) In SKILL++, the Scheme function `make-vector` becomes `makeVector`.
    - (2) The global variable `piport` is used in place of the Scheme function `current-input-port`.

- **Infix only**
  - Means that the specific Scheme functionality is provided, but the given name can only be used as an infix operator in SKILL++. There is usually an equivalent function with a different name to which this infix operator can be mapped.

- **Unsupported**
  - Means that this Scheme functionality is not yet provided in current SKILL++.

See the following sections for more information:

- [Lexical Structure](#) on page 516
- [Expressions](#) on page 517
- [Functions](#) on page 518
## Lexical Structure

### Scheme/SKILL++ Equivalents Table – Lexical Structure

<table>
<thead>
<tr>
<th>Scheme</th>
<th>SKILL++</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean literals #t, #f</td>
<td>Supported.</td>
<td>Use t for #t, nil for #f.</td>
</tr>
<tr>
<td>Character literals #...</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>Simple numeric literals such as integers &amp; floats</td>
<td>Supported.</td>
<td>Use 0..., 0x..., and 0b... for #o..., #x..., and #b... (octal/hex/binary integers).</td>
</tr>
<tr>
<td>String literals &quot;...&quot;</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>Vector literals #(...)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>case-insensitive symbols</td>
<td>Unsupported.</td>
<td>Symbols in SKILL++ are always case-sensitive.</td>
</tr>
<tr>
<td>nil as a symbol</td>
<td>Unsupported.</td>
<td>In SKILL++, just as in SKILL, nil is not a symbol.</td>
</tr>
<tr>
<td>Special symbol constituent characters such as !, $, %, &amp;, *, /, &lt;, =, and so forth.</td>
<td>Unsupported.</td>
<td>Some of these are used for (infix) operators in SKILL++, others are illegal characters. ? is used for keyword prefix.</td>
</tr>
<tr>
<td>' (single quote)</td>
<td>Same.</td>
<td>Shorthand for quote.</td>
</tr>
<tr>
<td>' (back quote)</td>
<td>Same.</td>
<td>Shorthand for quasiquote in Scheme and for _backquote in SKILL++.</td>
</tr>
<tr>
<td>, (comma)</td>
<td>Same.</td>
<td>Shorthand for unquote in Scheme and for _comma in SKILL++.</td>
</tr>
<tr>
<td>,@</td>
<td>Same.</td>
<td>Shorthand for unquote-splicing in Scheme and for _commaAt in SKILL++.</td>
</tr>
</tbody>
</table>
## Expressions

### Scheme/SKILL++ Equivalents Table – Expressions

<table>
<thead>
<tr>
<th>Scheme</th>
<th>SKILL++</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(improper lists), such as (d ... . d)</td>
<td>Unsupported.</td>
<td>SKILL++ lists must end with nil.</td>
</tr>
<tr>
<td>(procedure calls), such as (f e ...)</td>
<td>Same.</td>
<td>Can be written as $f(e \ldots)$ in SKILL++ if $f$ is a symbol (variable).</td>
</tr>
<tr>
<td>(and e ...)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(begin e ...)</td>
<td>Same.</td>
<td>Equivalent to <code>progn</code> in SKILL++.</td>
</tr>
<tr>
<td>(case ((d ...) e ...) ... [(else e ...)])</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(cond (e ...) ... [(else e ...)])</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(define x e)</td>
<td>Same.</td>
<td>One can also use SKILL's <code>procedure</code> syntax to define functions in SKILL++.</td>
</tr>
<tr>
<td>(define (x v ...) body)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(do ((v e [e]) ... (e ...) e ...)</td>
<td>Same.</td>
<td>SKILL++ allows extended <code>if</code> syntax (with <code>then</code> and <code>else</code> keywords) as in SKILL.</td>
</tr>
<tr>
<td>(if e1 e2 e3)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(lambda (x ...) body)</td>
<td>Same.</td>
<td>Improper variable list such as (x ... . y) can't be used as formals in SKILL++. Use SKILL style <code>@rest</code>, <code>@optional</code> instead.</td>
</tr>
<tr>
<td>(let [x] ((v e) ...) body)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(let* ((v e) ...) body)</td>
<td>Supported.</td>
<td>Use <code>letseq</code> instead of <code>let*</code> in SKILL++.</td>
</tr>
<tr>
<td>(letrec ((v e) ...) body)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(or e ...)</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>(set! x e)</td>
<td>Supported.</td>
<td>Use <code>setq</code> or the infix <code>=</code> operator.</td>
</tr>
</tbody>
</table>
## Functions

### Scheme/SKILL++ Equivalents Table – Functions

<table>
<thead>
<tr>
<th>Scheme</th>
<th>SKILL++</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, *, /</td>
<td>Infix only.</td>
<td>Equivalent to functions <code>plus</code>, <code>difference</code>, <code>times</code>, and <code>quotient</code> in SKILL++.</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>Infix only.</td>
<td>Equivalent to functions <code>lessp</code>, <code>leqp</code>, <code>greaterp</code>, and <code>geqp</code> in SKILL++.</td>
</tr>
<tr>
<td>=</td>
<td>Supported.</td>
<td>Note that <code>=</code> is used as the infix assignment operator in SKILL++. For equality, use the infix operator <code>==</code> or function <code>equal</code>.</td>
</tr>
<tr>
<td>abs</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>acos</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>angle</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>Same.</td>
<td>Takes two arguments only.</td>
</tr>
<tr>
<td>apply</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>asin</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>assoc</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>assq</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>assv</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>atan</td>
<td>Same.</td>
<td>In SKILL++, <code>atan</code> takes one argument only; <code>atan2</code> takes two arguments.</td>
</tr>
<tr>
<td>boolean?</td>
<td>Supported.</td>
<td>Use <code>booleanp</code>.</td>
</tr>
<tr>
<td>car, cdr, caar, ..., cddddr</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>call-with-current-continuation</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>call-with-input-file</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>call-with-output-file</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>ceiling</td>
<td>Same.</td>
<td></td>
</tr>
</tbody>
</table>
### Scheme/SKILL++ Equivalents Table – Functions

<table>
<thead>
<tr>
<th>Scheme</th>
<th>SKILL++</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>char-&gt;integer</td>
<td>Unsupported.</td>
<td>True character type is not supported in SKILL++. However, single-character symbols can be used to simulate it. The function charToInt has the same effect on symbols.</td>
</tr>
<tr>
<td>char-alphabetic?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-ci&lt;=?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-ci&lt;?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-ci=?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-ci&gt;=?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-ci&gt;?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-downcase</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-lower-case?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-numeric?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-upcase</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-upper-case?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char-whitespace?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char&lt;=?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char&lt;?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char=?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char&gt;=?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char&gt;?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>char?</td>
<td>Unsupported.</td>
<td>Character type not supported.</td>
</tr>
<tr>
<td>close-input-port</td>
<td>Supported.</td>
<td>Use close.</td>
</tr>
<tr>
<td>close-output-port</td>
<td>Supported.</td>
<td>Use close.</td>
</tr>
<tr>
<td>complex?</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>cons</td>
<td>Same.</td>
<td>The second argument must be a list.</td>
</tr>
<tr>
<td>cos</td>
<td>Same.</td>
<td></td>
</tr>
</tbody>
</table>
## Scheme/SKILL++ Equivalents Table – Functions

<table>
<thead>
<tr>
<th>Scheme</th>
<th>SKILL++</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>current-input-port</td>
<td>Supported.</td>
<td>Use the <code>piport</code> global variable.</td>
</tr>
<tr>
<td>current-output-port</td>
<td>Supported.</td>
<td>Use the <code>poport</code> global variable.</td>
</tr>
<tr>
<td>denominator</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>display</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>eof-object?</td>
<td>Unsupported.</td>
<td>SKILL++ reader returns <code>nil</code> on EOF.</td>
</tr>
<tr>
<td>eq?</td>
<td>Supported.</td>
<td>Use <code>eq</code>.</td>
</tr>
<tr>
<td>equal?</td>
<td>Supported.</td>
<td>Use <code>equal</code>.</td>
</tr>
<tr>
<td>eqv?</td>
<td>Supported.</td>
<td>Use <code>eqv</code>.</td>
</tr>
<tr>
<td>even?</td>
<td>Supported.</td>
<td>Use <code>evenp</code>.</td>
</tr>
<tr>
<td>exact-&gt;inexact</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>exact?</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>expt</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td>Same.</td>
<td>Use <code>fix</code> or <code>floor</code>.</td>
</tr>
<tr>
<td>for-each</td>
<td>Supported.</td>
<td>Use <code>mapc</code>.</td>
</tr>
<tr>
<td>gcd</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>imag-part</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>inexact-&gt;exact</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>inexact?</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>input-port?</td>
<td>Supported.</td>
<td>Use <code>inportp</code>.</td>
</tr>
<tr>
<td>integer-&gt;char</td>
<td>Unsupported.</td>
<td>Character type not supported. Use <code>intToChar</code> for the same effect on symbols.</td>
</tr>
<tr>
<td>integer?</td>
<td>Supported.</td>
<td>Use <code>fixp</code> or <code>integerp</code>.</td>
</tr>
<tr>
<td>lcm</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>Same.</td>
<td>Works for both lists and vectors.</td>
</tr>
<tr>
<td>list</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>Scheme</td>
<td>SKILL++</td>
<td>Comment</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>list-&gt;vector</td>
<td>Supported.</td>
<td>Use listToVector.</td>
</tr>
<tr>
<td>list-ref</td>
<td>Supported.</td>
<td>Use nth.</td>
</tr>
<tr>
<td>list?</td>
<td>Supported.</td>
<td>Use listp.</td>
</tr>
<tr>
<td>log</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>magnitude</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>make-polar</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>make-rectangular</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>make-string</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>make-vector</td>
<td>Supported.</td>
<td>Use makeVector.</td>
</tr>
<tr>
<td>map</td>
<td>Supported.</td>
<td>Use mapcar instead. Note that map in SKILL++ behaves differently from map in standard Scheme.</td>
</tr>
<tr>
<td>max</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>member</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>memq</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>memv</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>modulo</td>
<td>Same.</td>
<td>modulo differs from mod in SKILL++, which is the same as remainder.</td>
</tr>
<tr>
<td>negative?</td>
<td>Supported.</td>
<td>Use minusp or negativep.</td>
</tr>
<tr>
<td>newline</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>Same.</td>
<td>New for SKILL++. Same as ! operator.</td>
</tr>
<tr>
<td>null?</td>
<td>Supported.</td>
<td>Use null.</td>
</tr>
<tr>
<td>number-&gt;string</td>
<td>Supported.</td>
<td>Use sprintf.</td>
</tr>
<tr>
<td>number?</td>
<td>Supported.</td>
<td>Use numberp.</td>
</tr>
<tr>
<td>numerator</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>odd?</td>
<td>Supported.</td>
<td>Use oddp.</td>
</tr>
<tr>
<td>open-input-file</td>
<td>Supported.</td>
<td>Use infile.</td>
</tr>
<tr>
<td>Scheme</td>
<td>SKILL++</td>
<td>Comment</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>open-output-file</td>
<td>Supported.</td>
<td>Use <code>outfile</code>.</td>
</tr>
<tr>
<td>output-port?</td>
<td>Supported.</td>
<td>Use <code>outportp</code>.</td>
</tr>
<tr>
<td>pair?</td>
<td>Supported.</td>
<td>Use <code>dtpr</code> or <code>pairp</code>.</td>
</tr>
<tr>
<td>peek-char</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>positive?</td>
<td>Supported.</td>
<td>Use <code>plusp</code>.</td>
</tr>
<tr>
<td>procedure?</td>
<td>Supported.</td>
<td>Use <code>procedurep</code>.</td>
</tr>
<tr>
<td>quotient</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>rational?</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>rationalize</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>Supported.</td>
<td>Or use <code>lineread</code>. Returns <code>nil</code> on EOF.</td>
</tr>
<tr>
<td>read-char</td>
<td>Unsupported.</td>
<td>Character type not supported. Use <code>getc</code> for similar effect.</td>
</tr>
<tr>
<td>real-part</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>real?</td>
<td>Supported.</td>
<td>Use <code>floatp</code> or <code>realp</code>.</td>
</tr>
<tr>
<td>remainder</td>
<td>Same.</td>
<td>Use <code>mod</code> or <code>remainder</code>.</td>
</tr>
<tr>
<td>reverse</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>round</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>set-car!</td>
<td>Supported.</td>
<td>Use <code>rplaca</code> or <code>setcar</code>.</td>
</tr>
<tr>
<td>set-cdr!</td>
<td>Supported.</td>
<td>Use <code>rplacd</code> or <code>setcdr</code>.</td>
</tr>
<tr>
<td>sin</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>Same.</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>Unsupported.</td>
<td></td>
</tr>
<tr>
<td>string-&gt;number</td>
<td>Supported.</td>
<td>Use <code>readstring</code>.</td>
</tr>
<tr>
<td>string-&gt;symbol</td>
<td>Supported.</td>
<td>Use <code>concat</code> or <code>stringToSymbol</code>.</td>
</tr>
<tr>
<td>string-append</td>
<td>Supported.</td>
<td>Use <code>strcat</code>.</td>
</tr>
<tr>
<td>string-ci&lt;=?</td>
<td>Unsupported.</td>
<td></td>
</tr>
</tbody>
</table>
# Scheme/SKILL++ Equivalents Table – Functions

<table>
<thead>
<tr>
<th>Scheme</th>
<th>SKILL++</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>string-ci&lt;?</td>
<td>Unsupported.</td>
<td>Use <code>strlen</code>.</td>
</tr>
<tr>
<td>string-ci&gt;?</td>
<td>Unsupported.</td>
<td>Use <code>getchar</code> for similar effect.</td>
</tr>
<tr>
<td>string-length</td>
<td>Supported.</td>
<td>Use <code>alphalessp</code> or <code>strcmp</code>.</td>
</tr>
<tr>
<td>string-ref</td>
<td>Unsupported.</td>
<td>Use <code>getchar</code> for similar effect.</td>
</tr>
<tr>
<td>string-set!</td>
<td>Unsupported.</td>
<td>Strings in SKILL++ are immutable.</td>
</tr>
<tr>
<td>string&lt;?</td>
<td>Supported.</td>
<td>Use <code>alphalessp</code> or <code>strcmp</code>.</td>
</tr>
<tr>
<td>string=?</td>
<td>Supported.</td>
<td>Use <code>alphalessp</code> or <code>strcmp</code>.</td>
</tr>
<tr>
<td>string&gt;=?</td>
<td>Supported.</td>
<td>Use <code>alphalessp</code> or <code>strcmp</code>.</td>
</tr>
<tr>
<td>string&gt;?</td>
<td>Supported.</td>
<td>Use <code>alphalessp</code> or <code>strcmp</code>.</td>
</tr>
<tr>
<td>string?</td>
<td>Supported.</td>
<td>Use <code>stringp</code>.</td>
</tr>
<tr>
<td>substring</td>
<td>Supported.</td>
<td>Argument values differ. SKILL++ uses <code>index</code> and <code>length</code>. Scheme standard uses <code>start</code> and <code>end</code>(index).</td>
</tr>
<tr>
<td>symbol-&gt;string</td>
<td>Supported.</td>
<td>Use <code>get_pname</code> or <code>symbolToString</code>.</td>
</tr>
<tr>
<td>symbol?</td>
<td>Supported.</td>
<td>Use <code>symbolp</code>.</td>
</tr>
<tr>
<td>tan</td>
<td>Same.</td>
<td>Use <code>symbolp</code>.</td>
</tr>
<tr>
<td>truncate</td>
<td>Same.</td>
<td>Use <code>symbolp</code>.</td>
</tr>
<tr>
<td>vector</td>
<td>Same.</td>
<td>Use <code>symbolp</code>.</td>
</tr>
<tr>
<td>vector-length</td>
<td>Supported.</td>
<td>Use <code>length</code>.</td>
</tr>
<tr>
<td>vector-&gt;list</td>
<td>Supported.</td>
<td>Use <code>vectorToList</code>.</td>
</tr>
<tr>
<td>vector-ref</td>
<td>Supported.</td>
<td>Use <code>arrayref</code> or the a[i] syntax.</td>
</tr>
<tr>
<td>vector-set!</td>
<td>Supported.</td>
<td>Use <code>setarray</code> or the a[i] = v syntax.</td>
</tr>
<tr>
<td>vector?</td>
<td>Supported.</td>
<td>Use <code>arrayp</code> or <code>vectorp</code>.</td>
</tr>
<tr>
<td>write</td>
<td>Same.</td>
<td>Use <code>symbolp</code>.</td>
</tr>
<tr>
<td>write-char</td>
<td>Unsupported.</td>
<td>Use <code>symbolp</code>.</td>
</tr>
<tr>
<td>zero?</td>
<td>Supported.</td>
<td>Use <code>zerop</code>.</td>
</tr>
</tbody>
</table>