Cadence SKILL Language Reference

Product Version ICADVM20.1
October 2020

Contents

Preface

This manual covers the core features of the Cadence SKILL language and its application programming interface (API). It introduces SKILL language to new users and encourages them to use sound SKILL programming methods.

This manual is intended for the following users:

- Programmers beginning to program in SKILL language
- CAD developers (internal users and customers) who have experience in SKILL programming
- CAD integrators

This preface contains the following topics:

Scope

Unless otherwise noted, the functionality described in this guide can be used in both mature node (for example, IC6.1.8) and advanced node and methodologies (for example, ICADVM20.1) releases.

Label	Meaning

(ICADVM20.1 Only)	Features supported only in the ICADVM20.1 advanced nodes and advanced methodologies releases.
(IC6.1.8 Only)	Features supported only in mature node releases.

Licensing Requirements

SKILL uses **Cadence Design Framework II** license (License Number 111), which is checked out at the launch of the `skill` executable or the workbench.

For information on licensing in the Cadence SKILL Language, see the *Virtuoso Software Licensing and Configuration User Guide*.

Related Documentation

What's New

- *Cadence SKILL Language What's New*

Installation, Environment, and Infrastructure

- *Cadence Installation Guide*
- *Virtuoso Design Environment SKILL Reference*
- *Cadence Application Infrastructure User Guide*
- *Virtuoso Software Licensing and Configuration Guide*

Other SKILL Books

- *Cadence SKILL IDE User Guide*
- *Cadence SKILL Development Reference*
- *Cadence SKILL Language User Guide*
- *Cadence Interprocess Communication SKILL Reference*
- *Cadence SKILL++ Object System Reference*

Additional Learning Resources

Video Library

The Video Library on the Cadence Online Support website provides a comprehensive list of videos on various Cadence products.

To view a list of videos related to a specific product, you can use the Filter Results feature available in the pane on the left. For example, click the *Virtuoso Layout Suite* product link to view a list of videos available for the product.

You can also save your product preferences in the Product Selection form, which opens when you click the *Edit* icon located next to *My Products*.

Virtuoso Videos Book

You can access certain videos directly from Cadence Help. To learn more about the related features and to access the list of available videos, see Virtuoso Videos.

Rapid Adoption Kits

Cadence provides a number of Rapid Adoption Kits that demonstrate how to use Virtuoso applications in your design flows. These kits contain design databases and instructions on how to run the design flow.

In addition, Cadence offers the following training courses on the SKILL programming language:

- SKILL Language Programming Introduction
- SKILL Language Programming
- Advanced SKILL Language Programming

To explore the full range of training courses provided by Cadence in your region, visit Cadence Training or write to training_enroll@cadence.com.

Note: The links in this section open in a separate web browser window when clicked in Cadence Help.

Help and Support Facilities

Virtuoso offers several built-in features to let you access help and support directly from the software.

- The Virtuoso *Help* menu provides consistent help system access across Virtuoso tools and applications. The standard Virtuoso *Help* menu lets you access the most useful help and support resources from the Cadence support and corporate websites directly from the CIW or any Virtuoso application.
- The Virtuoso Welcome Page is a self-help launch pad offering access to a host of useful knowledge resources, including quick links to content available within the Virtuoso installation as well as to other popular online content.

  The Welcome Page is displayed by default when you open Cadence Help in standalone mode from a Virtuoso installation. You can also access it at any time by selecting *Help – Virtuoso Documentation Library* from any application window, or by clicking the *Home* button on the Cadence Help toolbar (provided you have not set a custom home page).

For more information, see Getting Help in *Virtuoso Design Environment User Guide.*

Customer Support

For assistance with Cadence products:

- Contact Cadence Customer Support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions and to any queries about the latest software updates and training needs. For more information, visit https://www.cadence.com/support.
- Log on to Cadence Online Support

  Customers with a maintenance contract with Cadence can obtain the latest information about various tools at https://support.cadence.com.

Feedback about Documentation

You can contact Cadence Customer Support to open a service request if you:

- Find erroneous information in a product manual
- Cannot find in a product manual the information you are looking for

- Face an issue while accessing documentation by using Cadence Help

You can also submit feedback by using the following methods:

- In the Cadence Help window, click the *Feedback* button and follow instructions.
- On the Cadence Online Support Product Manuals page, select the required product and submit your feedback by using the *Provide Feedback* box.

Understanding Cadence SKILL

Cadence SKILL is a high-level, interactive programming language based on the popular artificial intelligence language, Lisp. It lets you customize and extend your design environment. Using SKILL, you can validate the steps of your algorithm incrementally before incorporating them into a larger program.

For more information about the SKILL language, see Getting Started in the *SKILL Language User Guide*.

Using SKILL Code Examples

The SKILL APIs in this user manual are explained with illustrative code examples.

You can copy these examples from the manual and paste them directly into the Command Interpreter Window (CIW) or use the code in non-graphical SKILL mode.

Sample SKILL Code

The following code sample shows the syntax of a SKILL API that accepts three arguments.

axlGetRunStatus

```
axlGetRunStatus(
    t_sessionName                                    Required argument
    [ ?optionName t_optionName ]                     Optional keyword argument
    [ ?historyName t_historyName ]                   Optional keyword argument
    )
    => l_statusValues                                Return value
```

The first argument `t_sessionName` is a required argument, where `t` signifies the data type of the argument. The second and third arguments `?optionName t_optionName` and `?historyName t_historyName` are optional keyword arguments (identified by a question mark), which are specified in name-value pairs and can be placed in any order during the function call.

The return value is the value that the SKILL API returns after evaluating the expression. In this case, it is a list of status values, *`l_statusValues`*.

Example

```
axlSession=axlGetWindowSession( hiGetCurrentWindow())
=> "session0"
axlGetRunStatus("session0" ?historyName "Interactive.10" ?optionName "tests")
=> (1 2)
```

Return value

Value of the session name argument

Question mark and argument name before the value of the keyword argument

Question mark and argument name before the value of the keyword argument

Accessing API Help

Quick reference information for SKILL APIs is available from the CIW and the SKILL API Finder. To access the reference information for a particular SKILL API, do one of the following:

- Type `help <function_name>` in the CIW.
- Type `startFinder ( [ ?funcName t_functionName ] )` in the CIW.
- Start the SKILL API Finder from the CIW by choosing *Tools – Finder* or type `cdsFinder` on the UNIX command line.

  In the *Search in* field of the displayed Cadence SKILL API Finder window, type the SKILL API name for which you want to display the help information and click *Go*.

  The matches for the searched SKILL API appear in the *Results* area.

  To view the complete documentation of the searched SKILL API, select the API name in the *Results* area and click the *More Info* button. The complete documentation of the selected SKILL API appears in a new Cadence Help window.

Typographic and Syntax Conventions

The following typographic and syntax conventions are used in this manual.

text	Indicates names of manuals, menu commands, buttons, and fields.
`text`	Indicates text that you must type exactly as presented. Typically used to denote command, function, routine, or argument names that must be typed literally.
`z_argument`	Indicates text that you must replace with an appropriate argument value. The prefix (in this example, *`z_`*) indicates the data type the argument can accept and must not be typed.
|	Separates a choice of options.
`{ }`	Encloses a list of choices, separated by vertical bars, from which you **must** choose one.
`[ ]`	Encloses an optional argument or a list of choices separated by vertical bars, from which you **may** choose one.
`[ ?argName` *`t_arg`* `]`	Denotes a *key argument*. The question mark and argument name must be typed as they appear in the syntax and must be followed by the required value for that argument.
`...`	Indicates that you can repeat the previous argument. Used with brackets to indicate that you can specify zero or more arguments. Used without brackets to indicate that you must specify at least one argument.
`,...`	Indicates that multiple arguments must be separated by commas.
`=>`	Indicates the values returned by a Cadence® SKILL® language function.
`/`	Separates the values that can be returned by a Cadence SKILL language function.

If a command-line or SKILL expression is too long to fit within the paragraph margins of this document, the remainder of the expression is moved to the next line and indented. In code excerpts, a backslash (\ ) indicates that the current line continues on to the next line.

Identifiers Used to Denote Data Types

Data type identifiers are used to indicate the type of value required by an API argument. These data types are denoted by a single letter that is prefixed to the argument label and is separated from the argument by an underscore; for example, *t* is the data type in *t_viewName*. Data types and underscores are used only as identifiers; they must not be typed when specifying the argument in a function.

Prefix	Internal Name	Data Type
a	array	array
A	amsobject	AMS object
b	ddUserType	DDPI object
B	ddCatUserType	DDPI category object
C	opfcontext	OPF context
d	dbobject	Cadence database object (CDBA)
e	envobj	environment
f	flonum	floating-point number
F	opffile	OPF file ID
g	general	any data type
G	gdmSpecIlUserType	generic design management (GDM) spec object
h	hdbobject	hierarchical database configuration object
I	dbgenobject	CDB generator object
K	mapiobject	MAPI object
l	list	linked list
L	tc	Technology file time stamp
m	nmpIIUserType	nmpII user type
M	cdsEvalObject	cdsEvalObject
n	number	integer or floating-point number
o	userType	user-defined type (other)
p	port	I/O port
q	gdmspecListIlUserType	gdm spec list

Prefix	Internal Name	Data Type
r	defstruct	defstruct
R	rodObj	relative object design (ROD) object
s	symbol	symbol
S	stringSymbol	symbol or character string
t	string	character string (text)
T	txobject	transient object
u	function	function object, either the name of a function (symbol) or a lambda function body (list)
U	funobj	function object
v	hdbpath	hdbpath
w	wtype	window type
sw	swtype	subtype session window
dw	dwtype	subtype dockable window
x	integer	integer number
y	binary	binary function
&	pointer	pointer type

For more information, see *Cadence SKILL Language User Guide*.

1

List Functions

append

```
append(
     l_list1
     l_list2
     )
     => l_result

     append(
     o_table
     g_assoc
     )
     => o_table

     append(
     o_table1
     o_table2
     )
     => o_newTable
```

Description

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

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

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

The `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 *Cadence SKILL Language User Guide* for more details.

Arguments

l_list1	List of elements to be added to a list.
l_list2	List of elements to be added.
o_table	Association table to be updated.
g_assoc	Key/value pairs to be added to the association table.

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 *`l_list`* with *`g_arg`* added as the last element of the list.

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

Arguments

`l_list`	List to which *`g_arg`* is added.
`g_arg`	Argument to be added to the end of *`l_list`*.

Value Returned

`l_result`	Returns a copy of *`l_list`* with *`g_arg`* attached to the end.

Example

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

Like `append`, `append1` duplicates the top-level list cells of *`l_list`*.

Reference

append

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 caaaar, caaadr, caadar, caaddr, caar, caddar, cadddr, cadr, cdaaar, cdaadr, cdaar, cdadar, cdaddr, cdadr, cdar, cddaar, cddadr, cddar, cdddar, cddddr, cdddr, cddr, caaar, caadr, cadar, caddr, cdadr, cadaar, cadadr, caddar, cadddr, cdaaar, cdaadr, cdadar, cdaddr, cddaar, cddadr, cdddar, and cddddr.

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

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

`caaar` is equivalent to `car( car( car( l_list)))`.

```
caadr('(((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 `car( cdr( l_list))`.

Reference

car, 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 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

Value Returned

l_list	A list of elements.
g_result	Returns the first element in a list. `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

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, cddr, ...

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, append1

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`:

```
constar( 1 2 3 nil ) => (1 2 3)
```

The last argument must be a list:

```
constar( 'x 1 2 )
*Error* constar: the last arg must be a list - 2
```

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

```
cons(1 cons(2 cons(3 '(a b c)))) => (1 2 3 a b c)
constar( 1 2 3 '(a b c)) => (1 2 3 a b c)
```

copy

```
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

```
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.

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

`z` and `x` are not the same list.

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. `dtpr(nil)` returns `nil`.

Example

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

Reference

listp, pairp

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(
     l_tconc
     l_list
     )
     => l_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

length

```
length(
     laot_arg
     )
     => x_result / 0
```

Description

Determines the length of a list, array, association table, or string.

The time taken to compute the length depends on the type of object. For example,

List	Time taken to compute the length of a list is proportional to the number of items in the list.
Array	Time taken for computing the length of an array is constant.
Association table	Time taken for computing the length of an association table is constant.
String	Time taken to compute the length of a string is proportional to the number of characters in the string.

Arguments

laot_arg	SKILL list, array, association table, or string.

Value Returned

x_result	Length of the *laot_arg* object. (The length is either the number of elements in the list, string, or array or the number of key/value pairs in the association table).
0	*laot_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

length("hello")                => 5
declare(a[11])
length( a )                    => 11
```

```
myTable = makeTable( "atable" 0) => table:atable
myTable[ 'one] = "blue"          => "blue"
myTable[ "two"] = '(red)         => (r e d)
length(myTable)                  => 2
```

Reference

list

lindex

```
lindex(
     l_list
     g_element
     [ ?all g_all ]
     )
     => x_result / l_result / nil
```

Description

Returns the index number of the given element in *l_list*.

Arguments

l_list	A list of elements.
g_element	The element to be searched in *l_list*.
`?all` *g_all*	Specifies whether to print the index number for all occurrences of *g_element*.

Value Returned

x_result	The index number of *g_element* in *l_list* when `?all` is either `nil` or not specified.
l_result	The list of index numbers for all occurrences of *g_element* in *l_list* when `?all` is set to `t`.
`nil`	Returns `nil`, if the given element is not found in *l_list*.

Example

```
lindex('(1 2 3 4) 2)
   => 2
lindex('(1 4 6 7 4 8 4) 4 ?all t)
   =>(2 5 7)
lindex('(1 4 6 7 4 8 4) 4 ?all nil)
   =>2
```

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`. `listp(nil)` returns `t`.
`nil`	Otherwise.

Example

```
listp('(1 2 3))  => t
listp( nil )     => t
listp( 1 )       => nil
```

Reference

list

nconc

```
nconc(
     l_arg1
     l_arg2
     [ l_arg3 ... ]
     )
     => l_result
```

Description

Equivalent to a destructive `append` where the first argument is modified.

This results in `nconc` being much faster than `append` but not 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

lconc， tconc

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

list

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

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

nth

nthelem

```
nthelem(
     x_index1
     l_list
     )
     => g_result / nil
```

Description

Returns the indexed element of the list, assuming a one-based index.

Thus `nthelem(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_index1* 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

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

Reference

nth

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

listp

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)
```

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 = '("a" "b" "d" "f")  =>  ("a" "b" "d" "f")
remd( "a" y)                =>  ("b" "d" "f")
y                           =>  ("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 = '("a" "b" "d" "f")      =>  ("a" "b" "d" "f")
y=remd("a" y)               =>  ("b" "d" "f")
y                           =>  ("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

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 `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

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

```
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, remove, remq

removeListDuplicates

```
removeListDuplicates(
     l_list
     )
     => l_newList
```

Description

Removes duplicate entries from a SKILL list and returns a new list with the duplicates removed.

Arguments

l_list	A SKILL list.

Value Returned

l_newList	Copy of *l_list* with all duplicates removed.

Example

```
removeListDuplicates("a" 1 "a" 2 "a" 3 "a" 4)
=> ("a" 1 2 3 4)
```

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 x d f x g)) => (a b d f g)
```

Reference

remd, remove

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)
```

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

rplacd, setcar, setcdr

rplacd

```
rplacd(
     l_arg1
     l_arg2
     )
     => 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                    => (a d e f)
```

The `cdr` of `x` is replaced by the second argument.

Reference

rplaca, setcar, setcdr

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

rplacd, rplaca, 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

rplacd, setcar, rplaca

tailp

```
tailp(
     l_arg1
     l_arg2
     )
     => l_arg1 / nil
```

Description

Returns `arg1` if a list cell `eq` to `arg1` is found by `cdr` down `arg2` zero or more times, `nil` otherwise.

Because `eq` is being used for comparison `l_arg1` must point to a tail list in `l_arg2` for this predicate to return a non-`nil` value.

Arguments

l_arg1	A list.
l_arg2	Another list, which can contain *l_arg1* as its tail.

Value Returned

l_arg	If a list cell `eq` to *l_arg1* is found by `cdr`'ing down *l_arg2* zero or more times.
`nil`	Otherwise.

Example

```
y = '(b c)
z = cons( 'a y )     => (a b c)
tailp( y z )         => (b c)
tailp( '(b c) z )    => nil
```

`nil` was returned because `'(b c)` is not `eq` the `cdr( z )`.

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.

Arguments

`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

lconc

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

append1, lconc, list, ncons, tconc

xCoord

```
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

```
xValue = 300
yValue = 400
aCoordinate = xValue:yValue => ( 300 400 )
xCoord( aCoordinate ) => 300
yCoord( aCoordinate ) => 400
```

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
```

2

Data Structure

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.
`nil`	Otherwise.

Example

```
declare(x[10])
arrayp(x)   => t
arrayp('x) => nil
```

Reference

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.

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 `eq` to compare *`g_key`* with the car fields of the lists in *`alist`*. `assoc` uses `equal`. `assv` uses `eqv`.

The association list, *`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 *`l_alist`* whose car field is *`g_key`* and return that list. `assq` uses `eq` to compare *`g_key`* with the car fields of the lists in *`l_alist`*. `assv` uses `eqv`. `assoc` uses `equal`.

Arguments

g_key	An arbitrary object as the search key.
l_alist	Association list. Must be a list of lists.

Value Returned

l_association	The returned list is always an element of *`l_alist`*.
`nil`	If no list in *`l_alist`* has *`g_key`*, as its `car`.

Example

```
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)
```

Reference

eq, equal, eqv

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. Indexes 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 indexes 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

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 _*<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 returns `t`.

Example

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

```
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

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

Arguments

g_object	A data object.
S_name	Name of the structure to be tested for.

Value Returned

`t`	If *g_object* is an instance of `defstruct` *S_name*.
`nil`	Otherwise.

Example

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

Reference

defstruct, `printstruct`

defvar

```
defvar(
    s_varName
    [ g_value ]
    )
    => g_value / nil
```

Description

Defines a global variable and assigns it a value. You can also use the `defun` or `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

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

Assigns `x` a value of 3.

Reference

defprop, set, setq

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

makeVector

```
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. `Vector` 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

```
V = makeVector( 3 0 )    => array[3]:1955240
V[0]                     => 0
V[1]                     => 0
V[2]                     => 0
```

setarray

```
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

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

type, typep

```
type(
     s_object
     )
     => s_type

     typep(
     s_object
     )
     => s_type
```

Description

Returns a symbol whose name denotes the type of a SKILL object. The functions `type` and `typep` are identical.

Arguments

s_object	A SKILL object.

Value Returned

s_type	Symbol whose name denotes the type of *s_object*.

Example

```
type( 'foo )     => symbol
typep( "foo" )   => string
type( 12 )       => fixnum
typep ("12" )    => string
```

Reference

fixp, floatp, numberp, portp, stringp, symbolp

vector

```
vector(
     g_value ...
     )
     => 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, makeVector

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

3

Data Operator Functions

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

strcmp, strncmp

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, member, memq, memv

copy_<name>

```
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

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

Reference

copyDefstructDeep, make_<name>, `printstruct`

copyDefstructDeep

```
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

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

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

m2 = _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>, `printstruct`

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, database object, and a standard object (instance of a user defined subclass of standardObject). `get` has no infix operator syntax.

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

getSG

```
getSG(
     g_obj
     S_prop
     )
     => g_propValue
```

Description

Evaluates and then retrieves the value of the specified attribute or property. It is a lambda implementation of `getSGq()`.

Arguments

g_obj	Specifies the name of an object
S_prop	Specifies the name of the attribute or property for which you want to retrieve the value

Value Returned

g_propValue	The value of the property

Example

In the following example, `getSG()` evaluates the `tbl_list` argument and then retrieves its value.

```
tbl_list = list( (Table 'a nil)
    Table( 'b nil)
    Table( 'c nil))
setSG( tbl_list 41 'x)
=> '(41 41 41)

getSG( tbl_list 'x)
 => '(41 41 41)
```

getq

```
getq(
     sl_id
     S_name
     )
     => g_result / nil

     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.

`getq` corresponds to `->` as an LHS infix operator. So, `obj->prop` is equivalent to `getq(obj prop).` For more information, see Appendix 17, "Mapping Symbols to Values"

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

```
putprop( 'chip 8 'pins ) => 8
```

Assigns the property pins to a value of 8 to the symbol chip.

```
getq( 'chip pins )           => 8
chip.pins                    => 8
chip1 = list(nil 'pins 10)   => (nil pins 10)
chip1->pins                  => 10
```

Reference

get, getqq, plist, putprop

getqq

```
getqq(
     s_id
     S_name
     )
     => g_result / nil

     sl_id.S_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.

`getqq` corresponds to `.` as an LHS infix operator. So, `obj.prop` is equivalent to `getqq(obj prop)`. For more information, see Appendix 17, "Mapping Symbols to Values".

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

importSkillVar

```
importSkillVar(
     s_variable ...
     )
     => t / nil
```

Description

(SKILL++ mode) 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 returns `nil` 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

`t`	All variables were imported successfully.
`nil`	One or more variables failed to import.

Note: If the variables are not imported, a warning message displays.

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++.

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

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 = _myStruct(?a 3 ?b 2 ?c 1) => array[5]:3436504
m2 = copy_myStruct(m1) => array[5]:3436168
```

Reference

copy_<name>, copyDefstructDeep, printstruct

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
```

plist

```
plist(
     s_symbolName
     )
     => l_propertyList / nil
```

Description

Returns the property list associated with a symbol.

From time to time, it is useful to print 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

```
a.x = 10
a.y = 20
println(plist('a))
(y 20 x 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

popf

```
popf(
     g_place
     )
     => g_result
```

Description

A pop that uses the `setf` function. It returns the value for `g_place` that is removed.

Arguments

g_place	Place to be modified.

Value Returned

g_result	The value for `g_place` that is removed.

Example

```
a = '((4 1) 2 3)
popf(car(a) )
=> 4
a == '((1) 2 3)
```

References

setf, pushf

postArrayDec

```
postArrayDec(
     g_array
     g_index
     )
     => n_oldValue
```

Description

Takes an array or an associated table element with an index `g_index`, decrements its value by one, stores the new value back into the array, and returns the original value. Prefix form of `s--`.

If the associated table element is not a number or `g_index` is not valid, it returns an error.

Arguments

g_array	An array or an associated table.
g_index	An index in the array or an associated table.

Value Returned

n_oldValue	Original value of the element.

Example

```
a = vector(1 2 34)
array@0x8382028
postArrayDec(a 2)
=> 34
postArrayDec(a -4)
*Error* setarray: array index out of bounds -
postArrayDec(a -4)
```

Reference

postArrayInc, postArraySet, postArrayDec, preArrayInc, preArraySet

postArrayInc

```
postArrayInc(
     g_array
     g_index
     )
     => n_oldValue
```

Description

Takes an array or an associated table element with an index `g_index`, increments its value by one, stores the new value back into the array, and returns the original value. Prefix form of `s++.`

If the associated table element is not a number or `g_index` is not valid, it returns an error.

Arguments

g_array	An array or an associated table.
g_index	An index in the array or an associated table.

Value Returned

n_oldValue	Original value of the element.

Example

```
a = vector(1 2 34)
array@0x8382028
postArrayInc(a 2)
=> 34
a[2]
=> 35
postArrayInc(a -4)
*Error* setarray: array index out of bounds -
postArrayInc(a -4)
```

Reference

postArrayDec, postArraySet, preArrayDec, preArrayInc, preArraySet

postArraySet

```
postArraySet(
     g_array
     g_index
     n_modifier
     )
     => n_oldValue
```

Description

Takes an array or an associated table element with an index `g_index`, adds an n_modifier value to its original value, stores the new value back into the array, and returns the original value.

If the associated table element is not a number or `g_index` is not valid, it returns an error.

Arguments

g_array	An array or an associated table.
g_index	An index in the array or an associated table.
n_modifier	Value that should be added to the element.

Value Returned

n_oldValue	Original value of the element.

Example

```
a = vector(1 2 34)
array@0x8382028
postArraySet(a 2 3)
=> 34
postArraySet(a -4 9)
*Error* setarray: array index out of bounds -
postArraySet(a -4 9)
```

Reference

postArrayDec, postArrayInc, preArrayDec, preArrayInc, preArraySet

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

```
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

```
s = 2
postincrement( s ) => 2
s => 3

s = 2.2
postincrement( s ) => 2.2
s => 3.2
```

Reference

postdecrement, predecrement, preincrement

preArrayDec

```
preArrayDec(
     g_array
     g_index
     )
     => n_newValue
```

Description

Takes an array or an associated table element with an index `g_index`, decrements its value by one, stores the new value back into the array, and returns the updated value. Prefix form of `--s`.

If the associated table element is not a number or `g_index` is not valid, it returns an error.

Arguments

g_array	An array or an associated table.
g_index	An index in the array or an associated table.

Value Returned

n_newValue	New value of the element.

Example

```
a = vector(1 2 34)
array@0x8382028
preArrayDec(a 2)
=> 33
preArrayDec(a -4)
*Error* setarray: array index out of bounds -
preArrayDec(a -4)
```

Reference

postdecrement, predecrement, preincrement

preArrayInc

```
preArrayInc(
     g_array
     g_index
     )
     => n_newValue
```

Description

Takes an array or an associated table element with an index `g_index`, increments its value by one, stores the new value back into the array, and returns the updated value. Prefix form of `++s`.

If the associated table element is not a number or `g_index` is not valid, it returns an error.

Arguments

g_array	An array or an associated table.
g_index	An index in the array or an associated table.

Value Returned

n_newValue	New value of the element.

Example

```
a = vector(1 2 34)
array@0x8382028
preArrayInc(a 2)
=> 35
preArrayInc(a -4)
*Error* setarray: array index out of bounds -
preArrayInc(a -4)
```

Reference

postdecrement, predecrement, preincrement

preArraySet

```
preArraySet(
     g_array
     g_index
     n_modifier
     )
     => n_newValue
```

Description

Takes array or an associated table element with an index `g_index`, adds an n_modifier value to its original value, stores the new value back into the array, and returns the updated value.

If the associated table element is not a number or `g_index` is not valid, it returns an error.

Arguments

g_array	An array or an associated table.
g_index	An index in the array or an associated table.
n_modifier	The value that should be added to the element.

Value Returned

n_newValue	New value of the element i.e, (g_array [`g_index`] + `n_modifier`)

Example

```
a = vector(1 2 34)
array@0x8382028
preArraySet(a 2 3)
=> 37
preArraySet(a -4 9)
*Error* setarray: array index out of bounds -
preArraySet(a -4 9)
```

Reference

postdecrement, predecrement, preincrement

predecrement

```
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

```
s = 2
predecrement( s ) => 1
s => 1

s = 2.2
predecrement( s ) => 1.2
s => 1.2
```

Reference

postdecrement, predecrement, 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

```
s = 2
preincrement( s ) => 3
s => 3

s = 2.2
preincrement( s ) => 3.2
s => 3.2
```

Reference

postdecrement, predecrement

pushf

```
pushf(
     g_obj
     g_place
     )
     => g_newPlaceValue
```

Description

A push that uses the `setf` function. It modifies the contents of the original storage location.

Arguments

g_obj	New value to be pushed.
g_place	Place to be modified with the new value.

Value Returned

g_newPlaceValue	New value.

Example

```
a = list((list 1)  2 3)
pushf(4 (car a))
=> a == '((4 1) 2 3)
```

References

setf, popf

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. However, `putprop` has no infix operator syntax.

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

     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.

`putpropq` corresponds to `->` `=` as an assignment operator. So, `obj->prop = value` is equivalent to `putpropq(obj value prop)`. For more information, see Appendix 17, "Mapping Symbols to Values".

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

     s_id.S_name = g_value
     => g_value
```

Description

Adds properties to symbols. Identical to `putprop` except that *`sl_id`* and *`S_name`* are not evaluated. If the property already exists, the old value is replaced with a new one.

`putpropqq` corresponds to `. =` as an assignment operator. So, `obj.prop = value` is equivalent to `putpropqq(obj value prop)`. For more information, see Appendix 17, "Mapping Symbols to Values".

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)
```

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

rotatef

```
rotatef(
     [ gplace1 ]
     [ gplace2 ]
     .....
     [ gplacen ]
     )
     => g_newPlaceValues
```

Description

Modifies the value of each place by rotating the values from one place to another in a cyclic order.

Arguments

gplace1...gplacen

Values to be rotated.

Value Returned

g_newPlaceValues New values.

Example

```
a=1 b=2 c=3
rotatef(a b c)
=> a=b b=c c=a,
```

Now,

```
a=2 b=3 c=1
```

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 *g_newValue*.

Arguments

s_variableName	Symbol that is evaluated.
g_newValue	Value to set symbol to.
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	Returns *g_newValue*.

Example

```
y = 'a  => a    ; Sets y to the constant a.
set (y 5) => 5    ; Sets the value of y to 5.
y       => a
a       => 5
```

Reference

setq

setf

```
setf(
     g_place
     g_value
     )
     => g_result

     setf(
     g_place := g_value
     => g_result
```

Description

Assigns a new value to an existing storage location, destroying the value that was previously in that location. `setf` is the same as the assignment (`:=`) operator. This is a syntax form.

The `setf` function uses special expander functions, defined as `setf_<helper>`. For a list of the helper functions, see setf Helper Functions.

Arguments

g_place	Specifies the storage location
g_value	Specifies the new value

Value Returned

g_result	Returns the updated result

Example

```
x = '(a b c d e)
setf( (car x) 42);; here x changes to (42 b c d e)
=> (42 b c d e)

x = '(a b c d e)
(car x) := 42
x => (42 b c d e)
```

References

pushf, popf

setf_*<helper>*

```
setf_<helper>(
     g_new
     [ g_cell ]
     )
     => g_result
```

Description

An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with the expander name. For a list of the helper functions, see setf Helper Functions.

Arguments

g_new	New value to be set for `g_cell`.
g_cell	Cell to be modified.

Value Returned

g_result	Result of the corresponding `setf` operation.

Example

The following is an example of the helper function for `getSkillPath`:

```
defun(setf_getSkillPath (new)
  if(listp(new)
    setSkillPath(new)
    setSkillPath(list(new)))) ; alters the skill path with setf
setf(getSkillPath()   "/home/user/temp")  ; now skill path changed to "/home/user/
temp"
```

setguard

```
setguard(
     s_symbol
     g_guard
     )
     => u_guard
```

Description

Mainly enforces disciplined use of a symbol as a global variable by associating it with a guarding function that is either a symbol that identifies the name of the guarding function or a lambda form (just like the first argument to the `apply` function). If the guarding function is `nil`, the symbol is unguarded. The guarding function is called with two arguments whenever a new value is assigned to the symbol: the symbol and the value to be assigned to it. The result of the guarding function determines the `setguard` return value that gets assigned to the symbol.

Note: The guarding function associated with a guarded symbol is triggered whenever a new value is assigned to that symbol by way of the `setq` (or `set`) function. Neither a lambda binding nor a let binding will cause the guarding function to be called (see examples below).

Arguments

s_symbol	Symbol to be associated with the guarding function.
g_guard	Guarding function to be associated with the symbol.

Value Returned

u_guard	Either a symbol that identifies the name of the guarding function or a function object.

Example

```
> procedure( myPortGuard(varName newValue)
        if( portp(newValue)
        then
            newValue
        else
            printf("Only port values can be assigned to `%s'\n" varName)
            symeval(varName)
        )
  )
myPortGuard
```

```
> setguard('poport 'myPortGuard)
myPortGuard
> poport = nil
Only port values can be assigned to `poport'
port:"*stdout*"
> poport = 123
Only port values can be assigned to `poport'
port:"*stdout*"
> setguard( 'myStringVar
        lambda((varName newValue)
            if(stringp(newValue)
            then
                newValue
            else
                printf("Only strings can be assigned to `%s'\n" varName)
                symeval(varName)
            )
        ) ; lambda
  ) ; setguard
> myStringVar = "default"
"default"
> myStringVar = 123
Only strings can be assigned to `myStringVar'
"default"
> myStringVar = nil
Only strings can be assigned to `myStringVar'
"default"
;; A lambda binding will not trigger the guard
> ((lambda (myStringVar) (println 'hello)) nil)
hello
nil
;; A let binding will also not trigger the guard
> let( ((myStringVar 123))
        println(myStringVar)
  )
123
nil
;; This s the symbol `myStringVar' unguarded
> setguard('myStringVar nil)
nil
> myStringVar = 123
123
```

Reference

apply, lambda, let, set, setq

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

```
setplist( 'chip '(pins 8 power 5) )     => (pins 8 power 5)
plist( 'chip )                          => (pins 8 power 5)
chip.power                              => 5
```

Reference

getq, getqq, plist, putpropq, putpropqq, remprop

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`*. 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

setSG

```
setSG(
     g_obj
     S_prop
     g_value
     )
     =>g_propValue
```

Description

Evaluates and then sets the value for the specified attribute or property. It is a lambda implementation of `setSGq()`.

Arguments

g_obj	Specifies the name of an object
S_prop	Specifies the name of the attribute or property for which you want to set the value
g_value	Specifies the value you want to set

Value Returned

g_propValue	The set value of the property

Example

In the following example, `setSG()` evaluates the `tbl_list` argument and then sets its value.

```
tbl_list = list( (Table 'a nil)
     Table( 'b nil)
     Table( 'c nil))
setSG( tbl_list 41 'x)
=> '(41 41 41)
```

symbolp

```
symbolp(
     g_value
     )
     => t / 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.
`nil`	Otherwise.

Example

```
symbolp( 'foo)              => t
symbolp( "foo")             => nil
symbolp( concat("foo"))     => t
```

Reference

concat, stringp

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

4

Type Conversion Functions

charToInt

```
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

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

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

Reference

intToChar

intToChar

```
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

```
intToChar( 66)
=> B
```

Reference

charToInt

listToVector

```
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

```
V = listToVector( '( 1 2 3 ) ) => array[3]:1954920
V[0] => 1
V[1] => 2
V[2] => 3
V[3]
*Error* arrayref: array index out of bounds - V[3]
```

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
```

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"
```

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

timeToString(793561559)
=> "Feb 23 09:45:59 1995"

stringToTime("Feb 23 09:45:59 1995")
=> 793561559
```

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\""
```

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))
```

timeToString

```
timeToString(
     x_time
     )
     => t_time / nil
```

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.
`nil`	Returns `nil` if a negative argument is passed.

Example

```
valTime=fileTimeModified( "~/.cshrc" )
timeToString(valTime)
=> "Feb 23 09:45:59 1995"

timeToString(-valTime)
=> nil
```

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:

```
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

```
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
)

x->tm_mon
=>1
```

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
```

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( Vector( 3 "Hi"))
=> (3 "Hi")
```

5

String Functions

blankstrp

```
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 blank or is an empty string.
`nil`	If there are non-space characters.

Example

```
blankstrp( "")
=> t
blankstrp( " ")
=> t
blankstrp( "a string")
=> nil
```

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 *t_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

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

```
getchar("abc" 2)  => b
getchar("abc" 4)  => nil
```

Reference

nindex, parseString, strlen, substring

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*.
nil	If *S_string2* is not found.

Example

```
index( "abc" 'b )            => "bc"
index( "abcdabce" "dab" )    => "dabce"
index( "abc" "cba" )         => nil
index( "dandelion" "d")      => "dandelion"
```

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

lsprintf

```
lsprintf(
     t_formatString
     [ g_arg1 ... ]
     )
     => t_string
```

Description

Returns a string according to the provided format. `lsprintf` is a lambda version of the `sprintf` function that can be used as an argument with `apply` or `funcall`.

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

t_formatString	Specifies the format string
g_arg1	Specifies the arguments following the format string that are printed corresponding to their format specifications.

Value Returned

t_string	Returns the formatted string

Example

```
let( (format( "%d %d %s %L\n")
    printf_style_args( (list 42 41 "hello" (list "world"))))
    apply( 'lsprintf format printf_style_args))
=>"42 41 hello (\"world\")\n"
```

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

getchar, substring

outstringp

```
outstringp(
     g_port
     )
     => t / nil
```

Description

Checks whether the specified value is an outstring port.

Arguments

g_port	The value to be checked.

Value Returned

`t`	If the given value is an outstring port.
`nil`	If the given value is not an outstring port.

Example

```
p = outstring()
outstringp(p)
=> t
```

parseString

```
parseString(
     S_string
     [ S_breakCharacters ]
     [ g_insertEmptyString ]
     )
     => 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. If the third optional argument *g_insertEmptyString* is provided, insert (“ “) into the result list at each occurrence of *S_breakCharacters*. It generates the list of strings so that if the *S_breakCharacters* has a single character then the generated string is:

```
buildString( parseString( string delimiter t) delimiter)
```

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.

```
parseString( "-abc-def--ghi-" "-" )
=> ("abc"  "def"  "ghi")
```

Splits the string at each occurrence of the delimiter character "-".

```
parseString( "-abc-def--ghi-" "-" t )
=> (""  "abc"  "def"  ""  "ghi"  "")
```

Inserts an empty string at each occurrence of the delimiter character "-".

Reference

buildString, linereadstring, strcat, strlen, stringp

pcreCompile

```
pcreCompile(
     t_pattern
     [ x_options ]
     )
     => o_comPatObj / nil
```

Description

Compiles a regular expression string pattern (`t_pattern`) into an internal representation that you can use in a `pcreExecute` function call. The compilation method is PCRE/Perl-compatible. You can use a second (optional) argument to specify independent option bits for controlling pattern compilation. You can set and unset the `PCRE_CASELESS`, `PCRE_MULTILINE`, `PCRE_DOTALL`, and `PCRE_EXTENDED` independent option bits from within the pattern. The content of the options argument specifies the initial setting at the start of compilation. You can set the `PCRE_ANCHORED` option at matching time and at compile time.

Note: PCRE stands for Perl Compatible Regular Expressions. The PCRE library contains functions that implement Perl-compatible regular expression pattern matching. You can visit `http://www.pcre.org` for more information.

Arguments

t_pattern	String containing regular expression string to be compiled.
x_options	Optional) Independent option bits that affect the compilation. You can specify zero or more of these options symbolically using the `pcreGenCompileOptBits` SKILL function.

Valid Values:

PCRE_CASELESS / 0x00000001

Equivalent to setting `?caseLess t` using the pcreGenCompileOptBits SKILL function.

PCRE_MULTILINE / 0x00000002

Equivalent to setting `?multiLine t` using the pcreGenCompileOptBits SKILL function.

PCRE_DOTALL / 0x00000004

Equivalent to setting `?dotAll t` using the pcreGenCompileOptBits SKILL function.

PCRE_EXTENDED / 0x00000008

Equivalent to setting `?extended t` using the pcreGenCompileOptBits SKILL function.

PCRE_ANCHORED / 0x00000010

Equivalent to setting `?anchored t` using the pcreGenCompileOptBits SKILL function.

PCRE_DOLLAR_ENDONLY / 0x00000020

Equivalent to setting `?dollar_endonly t` using the pcreGenCompileOptBits SKILL function.

PCRE_UNGREEDY / 0x00000200

Equivalent to setting `?ungreedy t` using the pcreGenCompileOptBits SKILL function.

PCRE_NO_AUTO_CAPTURE / 0x00001000

Equivalent to setting `?no_auto_capture t` using the pcreGenCompileOptBits SKILL function.

PCRE_FIRSTLINE / 0x00040000

Equivalent to setting `?firstline t` using the pcreGenCompileOptBits SKILL function.

Value Returned

o_comPatObj	Data object containing the compiled pattern.
`nil`	Pattern compilation failed. An error message indicating the cause of the failure appears.

Example

```
comPat1 = pcreCompile( "\\Qabc\\$xyz\\E" ) => pcreobj@0x27d0fc
pcreExecute( comPat1 "abc\\$xyz" )         => t

comPat2 = pcreCompile( "sam | Bill | jack | alan | bob" ) => pcreobj@0x27d108
pcreExecute( comPat2 "alan" )    => t

comPat3 = pcreCompile( "z{1,5}" ) => pcreobj@0x27d120
pcreExecute( comPat3 "zzzzz" ) => t

comPat4 = pcreCompile( "/\\*.*?\\*/" ) => pcreobj@0x27d12c
pcreExecute( comPat4 "/* first command */ not comment /* second comment */" ) => t

comPat5 = pcreCompile( "^[a-z][0-9a-z]*" pcreGenCompileOptBits(?caseLess t) )
=> pcreobj@0x27d138
pcreExecute( "AB12cd" ) => t

comPat6 = pcreCompile( "[a-z" ) => *Error* pcreCompile: compilation failed at
offset 4: missing terminating ] for character class
nil
```

Reference

pcreExecute, pcreGenCompileOptBits

pcreExecute

```
pcreExecute(
     o_comPatObj
     S_subject
     [ x_options ]
     )
     => t / nil
```

Description

Matches the subject string or symbol (*S_subject*) against a previously compiled pattern set up by the last `pcreCompile` call (*o_comPatObj*). The matching algorithm is PCRE/Perl-compatible. You can use a third (optional) argument to specify independent option bits for controlling pattern matching. You can use this function in conjunction with `pcreCompile` to match several subject strings or symbols against a single pattern.

Arguments

o_comPatObj	Data object containing the compiled pattern returned from a previous `pcreCompile` call.
S_subject	Subject string or symbol to be matched. If it is a symbol, its print name is used.
x_options	(Optional) Independent option bits that affect pattern matching. You can specify zero or more of these options symbolically using the `pcreGenExecOptBits` SKILL function.

Valid Values:

`PCRE_ANCHORED`	Equivalent to setting `?anchored t` using the `pcreGenExecOptBits` SKILL function.
`PCRE_NOTBOL`	Equivalent to setting `?notbol t` using the `pcreGenExecOptBits` SKILL function.
`PCRE_NOTEOL`	Equivalent to setting `?noteol t` using the `pcreGenExecOptBits` SKILL function.
`PCRE_NOTEMPTY`	Equivalent to setting `?notempty t` using the `pcreGenExecOptBits` SKILL function.
`PCRE_PARTIAL`	Equivalent to setting `?partial t` using the `pcreGenExecOptBits` SKILL function.

Value Returned

`t`	A match is found.
`nil`	No match. You can see the error message associated with this matching failure by calling `pcrePrintLastMatchErr`.

Example

```
comPat1 = pcreCompile( "[12[:^digit:]]" ) => pcreobj@0x27d150
pcreExecute( comPat1 "abc" )          => t

comPat2 = pcreCompile( "((?i)ab)c" ) => pcreobj@0x27d15c
pcreExecute( comPat2 "aBc" )          => t

comPat3 = pcreCompile( "\\d{3}" ) => pcreobj@0x27d168
pcreExecute( comPat3 "789" )       => t

comPat4 = pcreCompile( "(\\D+|<\\d+>)*[!?]" ) => pcreobj@0x27d174
pcreExecute( comPat4 "Hello World!" )          => t
```

```
comPat5 = pcreCompile( "^\\d?\\d(jan | feb | mar | apr | may | jun)\\d\\d$/" )
=> pcreobj@0x27d180
pcreExecute( comPat5 "25jun3" ) => nil
pcreExecute( comPat5 "25jun3" pcreGenExecOptBits(?anchored t) ) => nil
pcreExecute( comPat5 "25jun3" pcreGenExecOptBits(?partial t) ) => t
```

Reference

pcreCompile, pcreExecute, pcreGenExecOptBits

pcreGenCompileOptBits

```
pcreGenCompileOptBits(
     [ ?caseLess  g_setCaseLessp ]
     [ ?multiLine g_setMultiLinep ]
     [ ?dotAll    g_setDotAllp ]
     [ ?extended  g_setExtendedp ]
     [ ?anchored  g_setAnchoredp ]
     [ ?dollar_endonly g_setDollarEndonlyp ]
     [ ?ungreedy  g_setUngreedyp ]
     [ ?no_auto_capture g_setNoAutoCapturep ]
     [ ?firstline g_setFirstlinep ]
     )
     => x_resultOptBits
```

Description

Generates bitwise inclusive OR—`bor()`—of zero or more independent option bits that affect compilation so that you can specify them symbolically in the `pcreCompile` function. If you call `pcreGenCompileOptBits` with no arguments, the function returns a zero (options have their default settings).

Arguments

`?setCaseLessp` *`g_setCaseLessp`*

When not `nil`, letters in the pattern match both upper and lower case letters. Setting this bit is equivalent to using Perl's `/i` option. You can change this setting within a pattern using `(?i)`..

`?setMultiLinep` *`g_setMultiLinep`*

When not `nil`, each newline in the subject string defines a line of characters for which the start-of-line metacharacter (`^`) matches at the start of the line and the end-of-line metacharacter (`$`) matches at the end of the line.

By default, PCRE treats the subject string as a single line of characters, even if it contains newlines, such that the start-of-line metacharacter (`^`) matches only at the start of the string and the end-of-line metacharacter (`$`) matches only at the end of the string, or before a terminating newline (unless `PCRE_DOLLAR_ENDONLY` is set).

`?setDotAllp` *`g_setDotAllp`*

When not `nil`, a dot metacharater in the pattern matches all characters, including newlines. Without it, newlines are excluded.

Setting this bit is equivalent to using Perl's `/s` option. You can change this setting within a pattern using `(?s)`. A negative class such as `[^a]` always matches a newline character, independent of whether this bit is set or not.

`?setExtendedp` *`g_setExtendedp`*

When not `nil`, PCRE ignores whitespace data characters in the pattern except when they are escaped or inside a character class.

Whitespace does not include the VT character (code 11).

PCRE also ignores characters between an unescaped `#` outside a character class and the next newline character, inclusive.

Setting this bit is equivalent to using Perl's `/x` option. You can change this setting within a pattern using `(?x)`.

You can use this setting to include comments (data characters only) inside complicated patterns.

You may not use whitespace characters in special character sequences in a pattern, such as `(?(` which introduces a conditional subpattern.

`?setAnchoredp` *`g_setAnchoredp`*

When not `nil`, PCRE constrains the match to the first matching point in the subject string. You can achieve this same effect using appropriate constructs in the pattern itself.

`?setDollarEndonlyp` *`g_setDollarEndonlyp`*

When not `nil`, a dollar metacharacter in the pattern matches at the end of the subject string only. Without this option, a dollar metacharacter also matches immediately before the final character if it is a newline (but not before any other newlines). PCRE ignores this setting if you specify `PCRE_MULTILINE`.

`?setUngreedyp` *`g_setUngreedyp`*

When not `nil`, PCRE inverts the greed of quantifiers so that they are not greedy by default. You can force a quantifier to become greedy by putting `?` after it. You can change this setting within a pattern using `(?U)`.

`?setNoAutoCapturep` *`g_setNoAutoCapturep`*

When not `nil`, If you set this bit, you are disabling the use of numbered capturing parentheses in a pattern. Any opening parenthesis that is not followed by `?` behaves as if it were followed by `?:` but you can still use named parentheses for capturing (and they acquire numbers in the usual way).

`?setFirstlinep` *`g_setFirstlinep`*

When not `nil`, PCRE requires an unanchored pattern to match before or at the first newline character in the subject string; the matched text may continue over the newline.

Value Returned

`x_resultOptBits` Bitwise inclusive OR— `bor()`—of zero or more independent option bits that affect pattern compilation.

Example

```
comPat1 = pcreCompile( "^abc$"
pcreGenCompileOptBits(?dollar_endonly t ?multiLine t) ) = > pcreobj@0x27d060
pcreExecute( comPat1 "abc\ndef")
=> t

pcreMatchAssocList("^[a-z][0-9]*$"
'((abc "ascii") ("123" "number") ("yy\na123"  "alphanum") (a12z "ana"))
pcreGenCompileOptBits(?multiLine t) pcreGenExecOptBits( ?notbol t) )
=> (("yy\na123" "alphanum"))
```

Reference

pcreCompile, pcreExecute, pcreGenExecOptBits, pcreMatchAssocList

pcreGenExecOptBits

```
pcreGenExecOptBits(
     [ ?anchored g_setAnchoredp ]
     [ ?notbol   g_setNotbolp ]
     [ ?noteol   g_setNoteolp ]
     [ ?notempty g_setNotemptyp ]
     [ ?partial  g_setPartialp ]
     )
     => x_resultOptBits
```

Description

Generates bitwise inclusive OR—`bor()`—of zero or more independent option bits that affect pattern matching so that you can specify them symbolically in the `pcreExecute` function. If you call `pcreGenExecOptBits` with no arguments, the function returns a zero (options have their default settings).

Arguments

`?setAnchoredp` *`g_setAnchoredp`*

When not `nil`, PCRE constrains the match to the first matching point in the `pcreExecute` function.
If you compiled a pattern using the `PCRE_ANCHORED` option, or if the pattern was anchored by virtue of its contents, then it must also be anchored at matching time.

`?setNotbolp` *`g_setNotbolp`*

When not `nil`, the first character of the subject string is not the beginning of a line such that the circumflex metacharacter `^` does not match before it. If you enable this option without setting the `PCRE_MULTILINE` option (at compile time), the circumflex metacharacter never results in a match.
This option affects the behavior of `^` only; it does not affect the behavior of `\A`.

`?setNoteolp` *`g_setNoteolp`*

When not `nil`, the end of the subject string is not the end of a line such that the dollar sign metacharacter `$` does not match it nor does it match a newline character immediately before it (except if you have set the `PCRE_MULTILINE` option). If you enable this option without setting the `PCRE_MULTILINE` option (at compile time), the dollar sign metacharacter never results in a match.
This option affects the behavior of `$` only; it does not affect the behavior of `\Z` or `\z`.

`?setNotemptyp` *`g_setNotemptyp`*

When not `nil`, an empty string is not a valid match. PCRE attempts to match any alternatives in the pattern. If all the alternatives match the empty string, the entire match fails.
For example, if you do not set this option, when PCRE applies the following sequence to a string that does not begin with `a` or `b`, it matches the empty string at the start of the subject:

```
a?b?
```

If you set this option, an empty string is not a valid match; PCRE searches further into the string for occurrences of `a` or `b`.

`?setPartialp` *`g_setPartialp`*

When not `nil`, the function returns `PCRE_ERROR_PARTIAL` instead of `PCRE_ERROR_NOMATCH` in the case of a partial match. A partial match occurs when PCRE encounters the end of a subject string before it can match the complete pattern. You may not use this option with all patterns. The following restrictions apply:

You may not specify quantified atom matches to search for repeated single characters or repeated single metasequences where the maximum quantity is greater than one. However, you may specify quantifiers with any values after parentheses. For example:

Use `(a){2,4}` instead of `a{2,4}`.
Use `(\d)+` instead of `\d+`.

Value Returned

`x_resultOptBits` Bitwise inclusive OR— `bor()`—of zero or more independent option bits that affect pattern matching.

Example

```
comPat = pcreCompile( "^\\d?\\d(jan | feb | mar | apr | may | jun)\\d\\d$/" )
=> pcreobj@0x27d0d8
pcreExecute( comPat "25jun3" pcreGenExecOptBits(?partial t) )
=> t

pcreMatchAssocList("^[a-z][0-9]*$"
'((abc "ascii") ("123" "number") ("yy\na123"  "alphanum") (a12z "ana"))
pcreGenCompileOptBits(?multiLine t) pcreGenExecOptBits( ?notbol t) )
=> (("yy\na123" "alphanum"))
```

Reference

pcreCompile, pcreExecute, pcreGenCompileOptBits, pcreMatchAssocList

pcreGetRecursionLimit

```
pcreGetRecursionLimit()
     => x_value
```

Description

Returns the PCRE maximum recursion depth (stack depth) that is set by the `pcreSetRecursionLimit()` function. The default value is 10000000.

Arguments

None.

Value Returned

x_value	Maximum recursion depth for the PCRE match algorithms.

Example

```
pcreGetRecursionLimit()
=> 10000000
```

pcreListCompileOptBits

```
pcreListCompileOptBits()
     => t
```

Description

Displays information about the options used with `pcreGenCompileOptBits`. See the description of `pcreGenCompileOptBits` for more information.

Arguments

None.

Value Returned

`t`	Returns `t`.

Reference

pcreGenCompileOptBits

pcreListExecOptBits

```
pcreListExecOptBits()
     => t
```

Description

Displays information about the options used with `pcreGenExecOptBits`. See the description of `pcreGenExecOptBits` for more information.

Arguments

None.

Value Returned

`t`	Returns `t`.

Reference

pcreGenExecOptBits

pcreMatchAssocList

```
pcreMatchAssocList(
     g_pattern
     l_subjects
     [ x_compOptBits ]
     [ x_execOptBits ]
     )
     => l_results / nil / error message(s)
```

Description

Matches the keys of an association list of subjects (strings or symbols) against a regular expression pattern (`g_pattern`) and returns an association list of those elements that match. The keys are the first elements of each key/value pair in the association list. You can use optional arguments to specify independent option bits for controlling pattern compiling and matching. The compiling and matching algorithms are PCRE/Perl-compatible.

The specified regular expression pattern overwrites the previously-compiled pattern and is used for subsequent matching until you provide a new pattern. The function reports any errors in the given pattern.

You can set and unset the `PCRE_CASELESS`, `PCRE_MULTILINE`, `PCRE_DOTALL`, and `PCRE_EXTENDED` independent option bits from within the pattern. The content of the options argument specifies the initial setting at the start of compilation. You can set the `PCRE_ANCHORED` option at matching time and at compile time.

Note: If `pcreObject` is specified as the `g_pattern`, `pcreMatchAssocList` skips pattern compilation and ignores `x_compOptBits`.

Arguments

g_pattern	String containing regular expression string to be compiled or a `pcreObject`.
l_subjects	Association list whose keys are strings or symbols.
x_compOptBits	(Optional) Independent option bits that affect the compilation. Valid values for this argument are the same as those for the *x_options* argument to the `pcreCompile` SKILL function.
x_execOptBits	(Optional) Independent option bits that affect pattern matching. Valid values for this argument are the same as those for the *x_options* argument to the `pcreExecute` SKILL function.

Value Returned

l_results	Association list of elements from the subject association list whose keys match the pattern.
`nil`	No keys in the subject association list match the pattern.
`error message(s)`	Zero or more error messages that appear if the function fails for any reason, if the subject association list is not valid, or if the pattern compilation fails (indicating the cause of the failure).

Example

```
pcreMatchAssocList( "^[a-z][0-9]*$"
'((abc "ascii") ("123" "number") (a123 "alphanum")
(a12z "ana")) )
=> ((a123 "alphanum"))
```

```
pcreMatchAssocList("^[a-z][0-9]*$"
'((abc "ascii") ("123" "number") ("yy\na123"  "alphanum") (a12z "ana"))
pcreGenCompileOptBits(?multiLine t) pcreGenExecOptBits( ?notbol t) )
=> (("yy\na123" "alphanum"))
```

```
pcreMatchAssocList( "box[0-9]*" '(square circle "cell9" "123") ) =>
*Error* pcreMatchAssocList: element in the list given as argument #2 is not a valid
association because its car() (taken as a key) is not either a symbol or a string
- square
```

Reference

pcreCompile, pcreExecute, pcreGenCompileOptBits, pcreGenExecOptBits

pcreMatchList

```
pcreMatchList(
     g_pattern
     l_subjects
     [ x_compOptBits ]
     [ x_execOptBits ]
     )
     => l_results / nil / error message(s)
```

Description

Matches a list of subjects (strings or symbols) against a regular expression pattern (`g_pattern`) and returns a list of those elements that match. You can use optional arguments to specify independent option bits for controlling pattern compiling and matching. The compiling and matching algorithms are PCRE/Perl-compatible.

The specified regular expression pattern overwrites the previously-compiled pattern and is used for subsequent matching until you provide a new pattern. The function reports any errors in the given pattern.

You can set and unset the `PCRE_CASELESS`, `PCRE_MULTILINE`, `PCRE_DOTALL`, and `PCRE_EXTENDED` independent option bits from within the pattern. The content of the options argument specifies the initial setting at the start of compilation. You can set the `PCRE_ANCHORED` option at matching time and at compile time.

Note: If `pcreObject` is specified as the `g_pattern`, `pcreMatchList` skips pattern compilation and ignores `x_compOptBits`.

Arguments

g_pattern	String containing regular expression string to be compiled or a `pcreObject`.
l_subjects	List of subject strings or symbols to be matched against the regular expression string. If it is a symbol, its print name is used.
x_compOptBits	(Optional) Independent option bits that affect the compilation. Valid values for this argument are the same as those for the *x_options* argument to the `pcreCompile` SKILL function.
x_execOptBits	(Optional) Independent option bits that affect pattern matching. Valid values for this argument are the same as those for the *x_options* argument to the `pcreExecute` SKILL function.

Value Returned

l_results	List of strings, symbols, or PCRE objects from the subject list that match the pattern.
`nil`	No matches or match failure.
`error message(s)`	Zero or more error messages that appear if the function fails for any reason, if the subject list is not valid, or if the pattern compilation fails (indicating the cause of the failure).

Example

```
pcreMatchList( "^[a-z][0-9]*" '(a01 x02 "003" aa01 "abc") )
=> (a01 x02 aa01 "abc")

pcreMatchList( "^[a-z][0-9][0-9]*" '(a001 b002 "003" aa01 "abc") )
=> (a001 b002)

pcreMatchList( "box[0-9]*" '(square circle "cell9" "123") )
=> nil

pcreMatchList("^[a-z][0-9][0-9]*" '("12\na001" b002)
pcreGenCompileOptBits(?multiLine t) pcreGenExecOptBits( ?notbol t) )
=> ("12\na001")

pcreMatchList("^[a-z][0-9]*" '(abc 123)) =>
*Error* pcreMatchList: element in the list given as argument #2 must be either a
symbol or a string - 123

pcreMatchList( "^[a-z][0-9]*$" '((abc "ascii") (a123 "alphanum")) ) =>
*Error* pcreMatchList: element in the list given as argument #2 must be either a
symbol or a string - (abc "ascii")
```

Reference

pcreCompile, pcreExecute, pcreGenCompileOptBits, pcreGenExecOptBits

pcreMatchp

```
pcreMatchp(
     g_pattern
     S_subject
     [ x_compOptBits ]
     [ x_execOptBits ]
     )
     => t / nil
```

Description

Checks to see whether the subject string or symbol (*S_subject*) matches the specified regular expression pattern (*g_pattern*). You can use optional arguments to specify independent option bits for controlling pattern compiling and matching. The compiling and matching algorithms are PCRE/Perl-compatible. For greater efficiency when matching a number of subjects against a single pattern, you should use pcreCompile and pcreExecute.

The specified regular expression pattern overwrites the previously-compiled pattern and is used for subsequent matching until you provide a new pattern. The function reports any errors in the given pattern.

You can set and unset the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED independent option bits from within the pattern. The content of the options argument specifies the initial setting at the start of compilation. You can set the PCRE_ANCHORED option at matching time and at compile time.

Note: If pcreObject is specified as the g_pattern, pcreMatchp skips pattern compilation and ignores x_compOptBits.

Arguments

g_pattern	String containing regular expression string to be compiled or a `pcreObject`.
S_subject	Subject string or symbol to be matched. If it is a symbol, its print name is used.
x_compOptBits	(Optional) Independent option bits that affect the compilation. Valid values for this argument are the same as those for the *x_options* argument to the `pcreCompile` SKILL function.
x_execOptBits	(Optional) Independent option bits that affect pattern matching. Valid values for this argument are the same as those for the *x_options* argument to the `pcreExecute` SKILL function.

Value Returned

`t`	A match is found. A message appears if you have any errors in the regular expression pattern.
`nil`	No match. An error message indicating the cause of the matching failure appears.

Example

```
pcreMatchp( "[0-9]*[.][0-9][0-9]*" "100.001" ) => t
pcreMatchp( "[0-9]*[.][0-9]+" ".001" )  => t
pcreMatchp( "[0-9]*[.][0-9]+" "." )     => nil
pcreMatchp( "[0-9" "100" ) =>
*Error* pcreCompile: compilation failed at offset 4: missing terminating ] for
character class nil
pcreMatchp( "((?i)rah)\\s+\\1" "rah rah" ) => t
pcreMatchp( "^[0-9]+" "abc\n123\nefg"
pcreGenCompileOptBits(?multiLine t) pcreGenExecOptBits( ?notbol t) )
=> t
```

Reference

pcreCompile, pcreExecute, pcreGenCompileOptBits, pcreGenExecOptBits

pcreObjectp

```
pcreObjectp(
     g_arg
     )
     => t / nil
```

Description

Checks to see whether the given argument is a `pcreObject` or not.

Arguments

g_arg	A value to be checked.

Value Returned

`t`	`g_arg` is a pcreObject.
`nil`	`g_arg` is not a pcreObject.

Example

```
a = pcreCompile("abc[0-9]+")
=> pcreobj@0x83b8018
(pcreObjectp a)
=> t
(pcreObjectp 9)
=> nil
```

Reference

pcreCompile

pcrePrintLastMatchErr

```
pcrePrintLastMatchErr(
     o_patMatchObj
     )
     => t / nil
```

Description

Prints the error message associated with the last failed matching operation (that is, when `pcreExecute` returns `nil`).

Argument

o_patMatchObj	Data object containing information from a previously failed pattern comilation/ matching operation.

Value Returned

`t`	Prints the error message associated with the last failed matching operation and returns `t`.
`nil`	No previously failed matching operation.

Example

```
comPat = pcreCompile( "[0-9]*[.][0-9]+" ) => pcreobj@0x27d060
pcreExecute( comPat "123" ) => nil
pcrePrintLastMatchErr( comPat ) =>
The subject string did not match the compiled pattern.
```

```
pcreExecute( comPat "123" pcreGenCompileOptBits(?caseLess t) ) => nil
pcrePrintLastMatchErr( comPat ) =>
An unrecognized bit was set in the options argument.
```

Reference

pcreCompile, pcreExecute, pcreGenCompileOptBits, pcreGenExecOptBits

pcreReplace

```
pcreReplace(
     o_comPatObj
     t_source
     t_replacement
     x_index
     [ x_options ] )
     => t_result / t_source
```

Description

Replaces one or all occurrences of a previously-compiled regular expression in the given source string with the specified replacement string. The integer index indicates which of the matching substrings to replace. If the index is less than or equal to zero, the function applies the replacement string to all matching substrings. You can use an optional argument to specify independent option bits for controlling pattern matching. The matching algorithm is PCRE/Perl-compatible.

Arguments

o_comPatObj	Data object containing the compiled pattern returned from a previous `pcreCompile` call.
t_source	Source string to be matched and replaced.
t_replacement	Replacement string. You can use pattern tags in this string (see `pcreSubstitute`).
x_index	Integer index indicating which of the matching substrings to replace. If the index is less than or equal to zero, the function applies the replacement string to all matching substrings.
x_options	(Optional) Independent option bits that affect pattern matching. Valid values for this argument are the same as those for the *x_options* argument to the `pcreExecute` SKILL function.

Value Returned

t_result	Copy of the source string with the specified replacement (determined by the integer index).
t_source	Original source string if no match was found.

Example

```
comPat1 = pcreCompile( "[0-9]+" ) => pcreobj@0x27d258
pcreReplace( comPat1 "abc-123-xyz-890-wuv" "(*)" 0 )
=> "abc-(*)-xyz-(*)-wuv"
pcreReplace( comPat1 "abc-123-xyz-890-wuv" "(*)" 1 )
=> "abc-(*)-xyz-890-wuv"
pcreReplace( comPat1 "abc-123-xyz-890-wuv" "(*)" 2 )
=> "abc-123-xyz-(*)-wuv"
pcreReplace( comPat1 "abc-123-xyz-890-wuv" "(*)" 3 )
=> "abc-123-xyz-890-wuv"

comPat2 = pcreCompile( "xyz" ) => pcreobj@0x27d264
pcreReplace( comPat2 "xyzzyxyzz" "xy" 0 ) => "xyzyxyz"
```

Reference

pcreCompile

pcreSetRecursionLimit

```
pcreSetRecursionLimit(
     x_maxDepth
     )
     => t
```

Description

Sets the maximum recursion depth for SKILL/PCRE match algorithms. The maximum recursion depth needs to be set for systems that have a low stack depth, in order to prevent crashes while using SKILL PCRE functions.

Arguments

x_maxDepth	Maximum recursion depth for the PCRE match algorithms.

Value Returned

`t`	The maximum recursion depth for the PCRE match algorithms is set.

Example

```
pcreSetRecursionLimit(1000)
=> t

pt = pcreCompile("sam | Bill| jack | alan| bob")
=> pcreobj@0x1df55020

pcreExecute(pt "myString")
=> nil
```

pcreSubpatCount

```
pcreSubpatCount(
     o_pcreObj
     )
     => x_count
```

Description

Counts the subpatterns in a PCRE pattern.

Argument

o_pcreObj	A PCRE compile object, produced by the `pcreCompile` function.

Value Returned

x_count	The number of subpatterns in a PCRE pattern. If there are no subpatterns in the PCRE pattern, it returns 0. *x_count* is a fixnum value.

Example

```
p1 = pcreCompile("(a)(b)(c)(d)") ;compile a pcre with 4 subpatterns
pcreSubpatCount(p1)
=> 4
```

pcreSubstitute

```
pcreSubstitute(
     [o_pcreObject]
     t_string
     )
     => t_result / nil
```

Description

If `o_pcreObject` is not provided, `pcreSubstitute` copies the input string and substitutes all pattern tags in it using the corresponding matched strings from the last `pcreExecute`/ `pcreMatch*` operation.

If `o_pcreObject` is provided, `pcreSubstitute` copies the input string and substitutes all pattern tags in it using the corresponding matched strings from the last `pcreExecute` operation that used the given `o_pcreObject`.

Pattern tags are of the form `\n`, where `n` is `0-9`. `\0` (or `&`) refers to the string that matched the entire regular expression; `\k` refers to the string that matched the pattern wrapped by the `k`th backslash (`...\`) in the regular expression.

If `o_pcreObject` is provided, pattern tag can also have the next form `\{x_num}`, where `x_num` is a positive integer. This refers to the string that matches the pattern by the `x_num(th)` backslash (`...\`) in the regular expression which has been compiled to `o_pcreObject`. The matched string will be taken from the last string which was matched by `pcreExecute` using `o_pcreObject`.

Argument

o_pcreObject	An object that was used in `pcreExecute`.
t_string	Argument string to which the function applies the substitution.

Value Returned

t_result	Copy of the argument with the specified substitutions.
`nil`	The last string matching operation failed (none of the pattern tags are meaningful).

Example

```
comPat = pcreCompile( "([a-z]+)\\.\\1" ) => pcreobj@0x27d048
pcreExecute( comPat "abc.bc" )
=> t
pcreSubstitute( "*\\0*" )
=> "*bc.bc*"
pcreSubstitute( "The matched string is: \\1" )
=> "The matched string is: bc"
r = pcreCompile("x[0-9]")
=> pcreobj@0x81ca018
pcreExecute(r "x1")
=> t
str1 = "\\0fff\\1ffff\\2fffff"
"\\0fff\\1ffff\\2fffff"
pcreSubstitute(str1)
=> "x1ffffffffffff"
pcre = pcreCompile("(a)(b+)([as]+)(q)(w)(r*)(t)(u)(i)(h)(k)(b).*")
=> pcreobj@0x83bb018
pcre1 = pcreCompile("0x([0-9]+)")
=> pcreobj@0x83bb034
pcreExecute(pcre "abbbasasssqwtuihkbdddd")
=> t
pcreExecute(pcre1 "0x333")
=> t
(for i 0 12
    str = (if i < 10 (sprintf nil "\\%d" i) (sprintf nil "\\{%d}" i))
    (printf "pcreSubstitute(pcre '%s') == '%L'\n" str pcreSubstitute(pcre str))
)
pcreSubstitute(pcre '\0') == '"abbbasasssqwtuihkbdddd"'
```

```
pcreSubstitute(pcre '\1') == '"a"'
pcreSubstitute(pcre '\2') == '"bbb"'
pcreSubstitute(pcre '\3') == '"asasss"'
pcreSubstitute(pcre '\4') == '"q"'
pcreSubstitute(pcre '\5') == '"w"'
pcreSubstitute(pcre '\6') == '""'
pcreSubstitute(pcre '\7') == '"t"'
pcreSubstitute(pcre '\8') == '"u"'
pcreSubstitute(pcre '\9') == '"i"'
pcreSubstitute(pcre '\{10}') == '"h"'
pcreSubstitute(pcre '\{11}') == '"k"'
pcreSubstitute(pcre '\{12}') == '"b"'
t
pcreSubstitute("the last pcreExecute was called - &")
=>"the last pcreExecute was called - 0x333"
```

Reference

pcreCompile, pcreExecute

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

rexCompile

```
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.

Note: `rexCompile` does not support the extended regular expression syntax. To parse such regular expressions, you can use the `pcre` (Perl Compatible Regular Expressions) functions (such as `pcreCompile`) instead.

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

```
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, pcreCompile

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

Synopsis	Meaning
c	Any ordinary character (not a special character listed below) matches itself.
.	A dot matches any character.
\	A backslash when followed by a special character matches that character literally. When followed by one of <, >, (,), and 1,...,9, it has a special meaning as described below.
[c...]	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.
*	A regular expression of any of the forms above, followed by the closure character * matches zero or more occurrences of that form.
+	Similar to *, except it matches one or more times.
\(...\)	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.
\n	A backslash followed by a digit *n* matches the contents of the *n*th register from the current regular expression.
\<...\>	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, 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.

Note: Calls to `rexMatchp` reset the pattern set up by `rexCompile`. If any calls to `rexMatchP` have been made, `rexExecute` will not match the pattern set by `rexCompile`.

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]+\\)\\.\\1")          => t
rexExecute("abc.bc")                      => t
rexExecute("abc.ab")                      => nil
```

Reference

rexCompile, rexMatchp, rexSubstitute, pcreCompile

rexMagic

```
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]+"

rexMagic(nil) ;; switch off
rexSubstitute("[&]")=> "[&]"
```

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

```
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 ] - "[0-9"
```

Reference

rexCompile, rexExecute

rexReplace

```
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

```
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( "xyzzyxyzz" "xy" 0)
                                   => "xyzyxyz" ; no rescanning!

rexCompile("^teststr")
rexReplace("teststr_a" "bb" 0) => "bb_a"
rexReplace("teststr_a" "bb&" 0) => "b teststr_a"
rexReplace("teststr_a" "[&]" 0) => "[teststr]_a"
```

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

rexCompile("^teststr") => t
s="teststr_1"
rexExecute(s)
rexSubstitute("&") => "teststr"
rexSubstitute("[&]") => "[teststr]"
```

Reference

rexCompile, rexExecute, rexReplace

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

t_string1	String to search for the last occurrence of *S_string2*.
S_string2	String or symbol to search for.

Value Returned

t_result	Remainder of *t_string1* starting with last match of *S_string2*.
`nil`	There is no match.

Example

```
rindex( "dandelion" "d") => "delion"
```

Reference

nindex

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.

Note: `sprintf` is a syntax form and should not be used as an argument to `apply` or `eval`.

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"
```

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

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 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

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

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

parseString, substring, strcat, strcmp, strncmp, stringp

strncat

```
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)         => "abcdef"
strncat( "abcd" "efghijk" 5)       => "abcdefghi"
```

Reference

parseString, strcat, strcmp, strncmp, substring, stringp

strncmp

```
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 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

```
strncmp( "abc" "ab" 3)  => 1
strncmp( "abc" "de" 4)  => -1
strncmp( "abc" "ab" 2)  => 0
```

Reference

strcmp

strpbrk

```
strpbrk(
     t_str1
     t_str2
     )
     => t_subStr / nil
```

Description

Returns a substring of the first occurence in *t_str1* of any character from the string pointed to by *t_str2*

Arguments

t_str1	Specifies the string that you need to scan
t_str2	Specifies the pattern that you need to match

Value Returned

t_substr	Returns a substring of the first occurence of any character specified in *t_str2*
nil	Returns nil if no occurence of any character from *t_str2* is found in *t_str1*

Example

```
s="world"
strpbrk(s "o")
=> "orld"
strpbrk(s "sssssl")
=>"ld"
strpbrk(s "ss")
=> nil
strpbrk("WORLD" "world")
=> nil
strpbrk("WORLD" " ")
=> nil
```

subst

```
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

```
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

upperCase

```
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

```
upperCase("Hello world!") => "HELLO WORLD!"
```

Reference

lowerCase

6

Arithmetic Functions

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

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

atof

```
atof(
     t_string [t]
     )
     => 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.
`t`	If *`t_string`* includes any non-numerical characters, this argument enforces that `nil` is returned.

Value Returned

`f_result`	The floating-point value represented by *`t_string`*.
`nil`	If *`t_string`* includes any non-numerical characters.

Example

```
atof("123")             => 123.0
atof("abc")             => nil
atof("123.456")         => 123.456
atof("123abc")          => 123.0
atof("12.01.01")        => 12.01
atof("12.01.01" t)      => nil
```

Reference

atoi

atoi

```
atoi(
     t_string [t]
     )
     => 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.
`t`	If *`t_string`* includes any non-numeric characters, this argument enforces that `nil` is returned.

Value Returned

`x_result`	The integer value represented by *`t_string`*.
`nil`	If *`t_string`* includes any non-numeric characters.

Example

```
atoi("123")          => 123
atoi("abc")          => nil
atoi("123.456")      => 123
atoi("123abc")       => 123
atoi("12.01.01")     => 12.01
atoi("12.01.01" t)   => nil
```

Reference

atof

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

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.

Name	Meaning
E	The base of natural logarithms. (*e*)
LOG2E	The base-2 logarithm of *e*
LOG10E	The base-10 logarithm of *e*
LN2	The natural logarithm of 2.
LN10	The natural logarithm of 10.
PI	The ratio of the circumference of a circle to its diameter. (π)
PI_OVER_2	$\pi/2$
PI_OVER_4	$\pi/4$
ONE_OVER_PI	$1/\pi$
TWO_OVER_PI	$2/\pi$
TWO_OVER_SQRTPI	
SQRT_TWO	(The positive square root of 2.)
SQRT_POINT_FIVE	(The positive square root of 1/2.)
INT_MAX	The maximum value of a SKILL integer.

Name	Meaning
INT_MIN	The minimum value of a SKILL integer. **Note:** The minimum value of a SKILL integer is -2147483648. The minimum literal value which may appear in a program is -2147483647.
DBL_MAX	The maximum value of a SKILL double.
DBL_MIN	The minimum value of a SKILL double.
SHRT_MAX	The maximum value of a SKILL "short" integer.
SHRT_MIN	The minimum value of a SKILL "short" integer.

Value Returned

s_id	Returns the symbol ID.

Example

```
defMathConstants('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

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

evenp

```
evenp(
     g_general
     )
     => t / nil
```

Description

Checks if a number is an even integer.

Arguments

g_general	Number to check.

Value Returned

t	If *g_general* is an even integer.
nil	Otherwise.

Example

```
evenp( 59 )
=> nil
evenp( 60 )
=> t
evenp( 2.0 )
=> nil              ; Number is even, but not an integer.
```

Reference

minusp, oddp, onep, plusp, zerop

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

asin, atan, cos, log, sin

expt

```
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. If expt(0,0) is specified, the value returned is 1.0, indicating no error.

Example

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

fix

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

Description

Returns the largest integer not larger than the given argument.

Note: If the given floating point argument `n_arg` is greater than the maximum integer value `INT_MAX`, a warning message displays and the `INT_MAX` value is returned. Similarly, if the floating point argument `n_arg` is less than the minimum integer value `INT_MIN`, a warning message displays and the `INT_MIN` value is returned.

This function is equivalent to `floor`. See also "Type Conversion Functions (fix and float)" in the *Cadence 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

```
fix(1.9)       => 1
fix(-5.6)      => -6
fix(100)       => 100
fix(4.1 * 100)=> 409

fix(1.111111e10)
*WARNING* (fix): Input value 11111110000.000000 is out of range. Using the maximum
integer value allowed (2147483647) instead. Check your code to ensure that all input
values and calculations have been correctly specified.
              =>2147483647

fix(-1.1234e20)
*WARNING* (fix): Input value -112340000000000000000.000000 is out of range. Using
the minimum integer value allowed (-2147483648) instead. Check your code to ensure
that all input values and calculations have been correctly specified.
              =>-2147483648
```

Reference

ceiling, fixp, floor, round

fixp

```
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

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

Reference

fix, float, floatp, integerp

fix2

```
fix2(
     n_value
     )
     => x_result / nil
```

Description

This function is a version of the `fix` function that works for rounding issue in floating-point calculations. The function returns the largest integer not larger than the given argument.

For more information, see "Comparing Floating-Point Numbers" in the "Arithmetic and Logical Expressions" chapter of the *SKILL Language User Guide*.

Arguments

n_value	Any number.

Value Returned

x_result	Returns the largest integer not larger than the given argument.
`nil`	If *n_value* is not an integer.

Example

```
fix2(4.1 * 100)
=> 410
```

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

```
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

int

```
int(
     g_value
     )
     => x_result
```

Description

Rounds off the number value to the nearest integer. The `int` function works as an overloadable arithmetic operator adopted from DFII to the SKILL Core language. The argument (*g_value*) is specified on the number class (numberp arguments).

Arguments

g_value	Specifies the number value you want to round off.

Value Returned

x_result	Returns the nearest integer

Example

```
int(2.7)
=>2
int(.7)
=>0
```

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
```

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
```

leftshift

```
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. `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

```
leftshift(7 2)  => 28
leftshift(10 1) => 20
```

Reference

rightshift

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

max

```
max(
     n_num1
     [ n_num2 ... ]
     )
     => n_result
```

Description

Returns the maximum of the values passed in. Requires a minimum of one argument.

Arguments

n_num1	First value to check.
n_num2	Additional values to check.

Value Returned

n_result	Maximum of the values passed in.

Example

```
max(6)          => 6
max(3 2 1)          => 3
max(-3 -2 -1)       => -1
```

Reference

abs, min, numberp

min

```
min(
    n_num1
    [ n_num2 ... ]
    )
    => n_result
```

Description

Returns the minimum of the values passed in. Requires a minimum of one argument.

Arguments

n_num1	First value to check.
n_num2	Additional values to check.

Value Returned

n_result	Minimum of the values passed in.

Example

```
min(3)          => 3
min(1 2 3)          => 1
min(-1 -2.0 -3)     => -3.0
```

Reference

abs, max, numberp

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(
     g_general
     )
     => t / nil
```

Description

Checks if a value is a negative number. Same as `negativep`.

Arguments

g_general	Number to check.

Value Returned

`t`	If *g_general* 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

modf

```
modf(
     f_flonum1
     f_flonum2
     )
     => f_result
```

Description

Returns the floating-point remainder of the division of *f_flonum1* by *f_flonum2*.

Arguments

f_flonum1	A floating-point number (Dividend).
f_flonum2	A floating-point number (Divisor).

Value Returned

f_result	Floating-point remainder of the division. The sign is determined by the dividend.

Example

```
;; Sign is determined by the dividend
modf(-10.1 10.0) => -0.1
modf(10.1 -10.0) => 0.1
```

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

remainder

nearlyEqual

```
nearlyEqual(
    n_val1 n_val2
    [f_relTolerance [f_absTolerance]]
    )
    => t / nil
```

Description

Checks if one value (*n_val1*) is approximately equal to another value (*n_val2*).

Arguments

n_val1 n_val2	The values that need to be checked.
f_relTolerance	The relative tolerance or the amount of error allowed, relative to the larger absolute value of *n_val1* or *n_val2*. It must be greater than 0. The default tolerance is 1e-9, which ensures that the two values are the same within about 9 decimal digits. To set a tolerance of 5%, for example, the pass tolerance must be equal to 0.05.
f_absTolerance	The minimum absolute tolerance level that can be used for comparisons near zero.

Value Returned

`t`	*n_val1* is nearly equal to *n_val2*.
`nil`	Otherwise.

Example

```
nearlyEqual(0.7777777777777 0.7777777777777777777777) => t
nearlyEqual(0.7777777777777 0.7777777777777777777777 0.0) => nil
nearlyEqual(nan nan) => nil
nearlyEqual(infinity nan1) => nil
```

Reference

eq, equal, eqv

negativep

```
negativep(
     n_num
     )
     => t / nil
```

Description

Checks if a value is a negative number. Same as `minusp`.

Arguments

n_num	Number to check.

Value Returned

`t`	*n_num* is a negative number.
`nil`	Otherwise.

Example

```
negativep( 3 )    => nil
negativep( -3 )   => t
```

Reference

evenp, minusp, numberp, oddp, onep, plusp, zerop

oddp

```
oddp(
     g_value
     )
     => t / nil
```

Description

Checks if an object is an odd integer.

`oddp` is a predicate function.

Arguments

g_value	A SKILL object that is an integer.

Value Returned

`t`	If *g_value* is an odd integer.
`nil`	Otherwise.

Example

```
oddp( 7 )
=> t
oddp( 8 )
=> nil
```

Reference

evenp, fixp, integerp, minusp, onep, plusp, zerop

onep

```
onep(
     g_value
     )
     => t / nil
```

Description

Checks if the given object is equal to one.

`onep` is a predicate function.

Arguments

g_value	A SKILL object that is either a floating-point number or an integer.

Value Returned

`t`	If *g_value* is equal to one.
`nil`	Otherwise.

Example

```
onep( 1 )
=> t
```

```
onep( 7 )
=> nil
```

```
onep( 1.0 )
=> t
```

Reference

evenp, minusp, numberp, plusp, zerop

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(
     g_value
     )
     => t / nil
```

Description

Checks if the given object is a positive number.

`plusp` is a predicate function.

Arguments

g_value	A SKILL object that is either a floating-point number or an integer.

Value Returned

`t`	If *g_value* is a positive number.
`nil`	Otherwise.

Example

```
plusp( -209.623472)
=> nil
plusp( 209.623472)
=> t
```

Reference

evenp, minusp, oddp, onep, zerop

quotient

```
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

```
quotient(5 4 3 2 1) => 0
quotient(-10 -2)    => 5
quotient(10.8 -2.2) => -4.909091
```

Reference

xquotient

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

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

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

modulo

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. `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

round

```
round(
     n_arg
     )
     =>  x_result
```

Description

Rounds a floating-point number to its closest integer value.

Note: If the given floating point argument `n_arg` is greater than the maximum integer value `INT_MAX`, a warning message displays and the `INT_MAX` value is returned. Similarly, if the floating point argument `n_arg` is less than the minimum integer value `INT_MIN`, a warning message displays and the `INT_MIN` value is returned.

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
```

```
round(1.111111e10)
*WARNING* (round): Input value 11111110000.000000 is out of range. Using the
maximum integer value allowed (2147483647) instead. Check your code to ensure that
all input values and calculations have been correctly specified.
                   =>2147483647
```

```
round(-1.1234e20)
*WARNING* (round): Input value -112340000000000000000.000000 is out of range. Using
the minimum integer value allowed (-2147483648) instead. Check your code to ensure
that all input values and calculations have been correctly
specified.                  =>-2147483648
```

Reference

fix, float

round2

```
round2(
     n_arg
     )
     =>  x_result
```

Description

This function is a version of the `round` function that rounds the result in floating-point calculations to its closest integer value.

For more information, see "Type Conversion Functions (fix and float)" in the Arithmetic and Logical Expressions chapter of the *SKILL Language User Guide*.

Arguments

n_arg	A floating-point number.

Value Returned

x_result	Integer whose value is closest to *n_arg*.

Example

```
val=-0.2865
round(val/0.001)*0.001
=> -0.286
round2(val/0.001)*0.001
=> -0.287
```

sort

```
sort(
     l_data
     u_comparefn
     )
     => l_result
```

Description

Sorts a list according to the specified 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)
(sort y nil)
=> (a b c d)
```

```
y
=> (c d)  ;no longer points to head of list
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

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))

myList = list('(2 two) '(4 four) '(1 one) '(3 three))
newList = sortcar( copy(myList) 'lessp )
newList = ((1 one) (2 two) (3 three) (4 four))
myList = ((2 two) (4 four) (1 one) (3 three)) ;; not changed !!
```

Reference

sort

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

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

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

truncate

```
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

```
truncate( 1234.567)
=> 1234
round( 1234.567)
=> 1235
truncate( -1.7)
=> -1
```

Reference

ceiling, floor, round

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(
     x_op1
     x_op2
     [ x_opt3 ]
     )
     => x_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_op1	First operand to be added.
x_op2	Second operand to be added.
x_opt3	Optional additional operands to be added.

Value Returned

x_result	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
```

zerop

```
zerop(
     g_value
     )
     => t / nil
```

Description

Checks if an object is equal to zero.

`zerop` is a predicate function.

Arguments

g_value	A SKILL object that is either a floating-point number or an integer.

Value Returned

`t`	If *g_value* is equal to zero.
`nil`	Otherwise.

Example

```
zerop( 0 )
=> t
zerop( 7 )
=> nil
```

Reference

evenp, minusp, oddp, onep, plusp

zxtd

```
zxtd(
     x_number
     x_bits
     )
     => x_result
```

Description

Zero-extends the number represented by the rightmost specified number of bits in the given integer.

Zero-extends the rightmost *x_bits* bits of *x_number*. Executes faster than doing `x_number<x_bits - 1:0>`.

Arguments

x_number	An integer.
x_bits	Number of bits.

Value Returned

x_result	*x_number* with the rightmost *x_bits* zero-extended.

Example

```
zxtd( 8 3 )  => 0
zxtd( 10 2 ) => 2
```

7

Bitwise Operator Functions

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

bnor, bnot

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

, setqbitfield1,

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,

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

band, bnor, bnot

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

band, bnot

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

band, bnor

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

band, bnor, bnot

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

band, bnor, bnot

bxor

```
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

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

Reference

band, bnor, bnot

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

, bitfield, setqbitfield1

setqbitfield1

```
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

bitfield

8

Trigonometric Functions

asin

```
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

atan

```
atan(
     n_number
     )
     => f_result
```

Description

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

Arguments

n_number	Floating-point number or integer.

Value Returned

f_result	Arc tangent of *n_number*.

Example

```
atan(0.3) => 0.2914568
```

Reference

, atan2

atan2

```
atan2(
     n_y
     n_x
     )
     => f_result
```

Description

Computes the principal value of the arc tangent of *n_y*/*n_x*, using the signs of both arguments to determine the quadrant of the return value.

Arguments

n_y	Vertical coordinate value.
n_x	Horizontal coordinate value.
	n_y/*n_x* is the tangent of the required angle.

Value Returned

f_result

Arc tangent of `y/x` in the range `[-pi,pi]` radians. If both arguments are 0.0, 0.0 is returned.If `x` or `y` is `NaN`, `NaN` is returned. In IEEE754 mode, `atan2()` handles the following exceptional arguments according to ANSI/IEEE Std 754-1985:

`atan2(+0,x)` returns `+0` for `x>0` or `x=+0`

`atan2(+0,x)` returns `+pi` for `x<0` or `x=-0`

`atan2(y,+0)` returns `pi/2` for `y>0`

`atan2(y,+0)` returns `-pi/2` for `y<zatan2(+y,Inf)` returns `+0` for finite `y>0atan2(+Inf,x)` returns `+pi/2` for finite `xatan2(+y,-Inf)` returns `+pi` for finite `y>0atan2(+Inf,Inf)` returns `+pi/4atan2(+Inf,-Inf)` returns `+3pi/4`

Example

```
atan2(1 1) => 0.7853982
atan2(0 0) => 0.0
```

Reference

atan,

cos

```
cos(
     n_number
     )
     => f_result
```

Description

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

Arguments

n_number	Floating-point number or integer.

Value Returned

f_result	Cosine of *n_number*.

Example

```
cos(0.3)         => 0.9553365
cos(3.14/2)      => 0.0007963
```

Reference

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

asin

tan

```
tan(
     n_number
     )
     => f_result
```

Description

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

Arguments

n_number	Floating-point number or integer.

Value Returned

f_result	Tangent of *n_number*.

Example

```
tan( 3.0 ) => -0.1425465
```

Reference

atan, atan2

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

9

Logical and Relational Functions

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

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, bnand, bnor, bnot, bor, bxnor, bxor

compareTime

```
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

```
compareTime( "Apr 8 4:21:39 1991" "Apr 16 3:24:36 1991")
=> -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.

```
compareTime("Apr 16 3:24:36 1991" "Apr 16 3:14:36 1991")
=> 600
```

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

Reference

getCurrentTime

eq

```
eq(
     g_arg1
     g_arg2
     )
     => t / nil
```

Description

Checks addresses when testing for equality.

Returns `t` if *g_arg1* and *g_arg2* are the same (that is, are at the same address in memory). The `eq` function runs considerably faster than `equal` but should only be used for testing equality of symbols, shared lists, or small numeric values (in the range of -256 to +256). Using `eq` on types other than symbols, lists, or small numeric values will give unpredictable results and should be avoided.
For testing equality of numbers, strings, and lists in general, the `equal` function and not the `eq` function should be used. You can test for equality between symbols using `eq` more efficiently than using the `==` operator, which is the same as the `equal` function. If one argument of the `eq` function is a string, SKILL Lint prints an error suggesting that the `eqv` or `equal` function be used instead.

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.
`nil`	The two objects are not identical.

Example

```
x = 'dog
eq( x 'dog )        => t
eq( x 'cat )        => nil

y = 'dog
eq( x y )           => 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. 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

```
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_general1
     g_general2
     )
     => t / nil
```

Description

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

Arguments

g_general1	The first SKILL object.
g_general2	The second SKILL object.

Value Returned

`t`	*g_general1* and *g_general2* represent the same string 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
s1="world"
s2="world"
eqv(s1 s2)                     => t
```

Reference

eq, equal

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 `>=` 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

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

geqp, leqp, lessp

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

geqp, 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

member, memq, memv

```
member(
     g_obj
     g_arg
     )
     => l_sublist / t / nil
```

Description

Returns the largest sublist of *l_list* whose first element is *g_obj* or checks whether the key *g_obj* exists in the association table. 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 using `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* or key to be searched in the association table.
g_arg	A list or an association table.

Value Returned

l_sublist	The part of *l_list* or association table beginning with the first match of *g_obj*.
`t`	Returns `t` if the key *g_obj* exists in the association table.
`nil`	Returns `nil` if the key *g_obj* does not exist in the association table.

Example 1

```
x = "c"                            => "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")
```

Example 2

```
tb = makeTable("myTable")
tb[0]= 1
tb["skill"] = 2

member("skill" tb)
=> t
```

Reference

eq, equal, eqv, concat

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              => dog
neq( a 'dog )         => nil
neq( a 'cat )         => t

z = '(1 2 3)          => (1 2 3)
neq(z z)              => nil
neq('(1 2 3) z)       => t
```

Reference

eq, equal, eqv, 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
```

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
```

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.
`nil`	Otherwise.

Example

```
numberp( 3 )        => t
numberp('isASymbol)  => nil
numberp( 3.5)       => t
```

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
```

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 *x_bits* bits of *x_number*. That is, sign-extends the bit field `x_number<x_bits - 1:0>` with `x_number<x_bits - 1>` as the sign bit.

Arguments

x_number	An integer.
x_bits	Number of bits.

Value Returned

x_result	*x_number* with the rightmost *x_bits* sign-extended.

Example

```
sxtd( 7 4 )  => 7
sxtd( 8 4 )  => -8
sxtd( 5 2 )  => 5
```

Reference

zxtd

10

Flow Control Functions

case

```
case(
     g_keyForm
     l_clause1
     [ l_clause2 ... ]
     )
     => g_result / nil
```

Description

Branches to one of the clauses depending on the value of the given expression. caseq() evaluates g_keyForm and matches the resulting value sequentially against the clauses until it finds a match. Once a match is found it stops searching the clauses, evaluates the forms in the matching clause, and returns the resulting value. This is a syntax function.

Each l_clause is in turn a list of the form (g_keys g_expr1 [g_expr2 ...]) in which the first element, that is g_keys, is either an atom (that is, a scalar) of any data type or a list of keys (to be compared with the given expression). When using a list of keys, specify it as a list of one or more lists to distinguish it from a list of scalar keys, as shown in Example 2. If any of the keys matches the value from g_keyForm, that clause is selected. Keys are always treated as constants and are never evaluated.

The symbol t has special meaning as a key in that it matches anything. It acts as a catch-all and should be handled last to serve as a default case when no other match is found. To match the value t, use a list of t as the key.

Arguments

g_keyForm	An expression whose value is evaluated and tested for equality against the keys in each clause. A match occurs when either the selector is equal to the key or the selector is equal to one of the elements in the list of keys. 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.
l_clause1	An expression whose first element is an atom or list of atoms to be compared against the value of *g_keyForm*. The remainder of the *l_clause* is evaluated if a match is found. **Note**: Do not put quotes or use the list() function when specifying the lists in a clause.
l_clause2	Zero or more clauses of the same form as *l_clause1*.

Value Returned

g_resultb	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 1

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

Example 2

```
listofnums = list(2 4 6)
case( listofnums
 (( (1 2 3) ) 'onetwothree)
 (( (1 3 5)
```

```
    (7 9 11) ) 'odd)
 (( (2 4 6)
    (8 10 12) ) 'even)
 (t 'unknown))
=> even
```

Example 3

```
case( myBool
 (nil 'never)  ; this will never match, it is a list of no keys
 (( nil ) nil) ; matches nil
 (( t ) t)     ; matches t
 (t (error "Expected t or nil"))
)
```

Example 4

```
shapeType="line"
rectCount=0
labelOrLineCount=0
miscount=0
case( shapeType
("rect" ++rectCount println( "Shape is a rectangle" ))
(( "label" "line" ) ++labelOrLineCount println( "Shape is a line or a label" ))
(t ++miscount println( "Shape is miscellaneous" ))
) ; case
=> Shape is a line or a label
```

Example 5

```
procedure(migrateShape(shape "d")
    case(list(shape->layerName shape->purpose)
        ((("POLY" "test1"))
            shape->layerName = "CPO"
            shape->purpose = "drawing"
        )
        ((("Oxide" "drawing") ("abcd" "efgh"))
            println(shape->layerName)
        )
        ((("M1" "net"))
            shape->purpose = "label"
```

```
)
((("M2" "net"))
    shape->purpose = "label"
)
((("M3" "net"))
    shape->purpose = "label"
)
((("M4" "net"))
    shape->purpose = "label"
)
)
) ;
```

Reference

eq, equal

caseq

```
caseq(
     g_keyForm
     l_clause1
     [ l_clause2 ... ]
     )
     => g_result / nil
```

Description

Works like the case() function, but uses eq() to find a matching clause instead of the equal() function. The keys used with caseq() should therefore not be strings or lists. In case you want to use a string value or a list, SKILL recommends using the case() function. See eq for details on the difference between the eq() and equal() functions.

Arguments

g_keyForm	An expression whose value is evaluated and tested for equality against the comparators in each clause. 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.
l_clause1	An expression whose first element is an atom or list of atoms to be compared against the value of *g_keyForm*. 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

```
caseq(value
((nil) printf("Failed.\n"))
(indeterminate printf("Indeterminate.\n"))
((t) printf("Succeeded.\n"))
(t printf("Default.\n"))
))
```

catch

```
catch(
     s_tag
     g_form
     )
     => g_result
```

Description

Establishes a control transfer or a return point for the `throw` and `err` functions. The return point is identified with a *s_tag*. So, when a particular tag/exception is caught, `catch` evaluates *g_form*. If the forms execute normally (without error), the value of the last body form is returned from the `catch`. There can also be nested `catch` blocks and *s_tag* can be `t` (the value `t` the `catch` function catch any condition thrown by `throw`).

Arguments

s_tag	Identifies the return point for the `throw` and `err` functions
g_form	Specifies the forms that are evaluated

Value Returned

g_result	Returns the value of the last form if the forms exit normally, otherwise, returns the values that are thrown if a `throw` or `err` occurs

Example

The following example describes a nested catch. The tag, '`wrongPlat`, is caught by the default handler `catch(t . . .).`

```
catch(t
     catch('issue1
          printf("Hello ")
               catch('issue2
                    when(cdsPlat() == "lnx86"
                         throw('wrongPlat printf("\n") nil); no exception is thrown on
non-lnx86 platforms
                         )
                         printf("world\n")
                    )
```

```
        )
    )

Hello
=> nil
```

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.

`cond` clauses can have one of the following forms:

- `( g_condition g_expr1 ... )` where *g_condition* is any expression
- `( g_condition )`
- Alternate clause, `"=> clause"` where *g_condition* => *u_expression*
- Else clause of the form (*else g_expr ...*), where the condition is replaced by the symbol `else`. This form is applicable only in SKILL++/Scheme mode.

Each clause is considered in succession. If *g_condition* evaluates to non-`nil` then processing stops and the value of the clause is used for the whole `cond` form. If one or more *g_expr* forms are given, they are evaluated in order and the value of the final *g_expr* is used as the value of the whole *cond* form. If no *g_expr* forms are given, the value of *g_condition* is used.

If the clause uses the alternate clause form, `"=> clause"`, *u_expression* must evaluate to a function, which is called with the value of *g_condition* as a single argument. The value returned by this function is used as the value of the whole `cond` form.

If an `else` clause is encountered, its *g_expr* forms are evaluated unconditionally and the value of the final *g_expr* is used as the value of the whole `cond` form.

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".
```

decode

```
decode(
     g_keyForm
     l_clause1
     [ l_clause2 ... ]
     )
     => g_result / nil
```

Description

Branches to one of the clauses depending on the value of the given expression. decode() evaluates `g_keyForm` and matches the resulting value sequentially against the clauses until it finds a match. Once a match is found it stops searching the clauses, evaluates the forms in the matching clause, and returns the resulting value.

Arguments

g_keyForm	An expression whose value is evaluated and tested for equality against the key in each clause. A match occurs when either the selector is equal to the key. If a match is found, the expressions in that clause and that clause only (that is, the first match) are executed.
l_clause1	A list of the form (*g_keys g_expr1* [*g_expr2* ...]) in which the first element, that is *g_keys*, is an atom (that is, a scalar) of any data type. If any of the keys matches the value from *g_keyForm*, the clause that contains it is selected. Keys are always treated as constants and are never evaluated.
l_clause2	Any other clauses in the same form as *l_clause1*.

Value Returned

g_result	Returns the value of the last expression evaluated in the matched clause.
`nil`	If there is no match.

Example 1

```
v = "test"
```

```
decode(v
            (1 nil)
            ("test" printf("test found\n"))
            (2 t)
            (t nil)
        )

=> 2
```

Reference

case, caseq

do

```
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 *`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 *`var1`*, *`var2`*, ... are bound to the above results. Reiterate from step one.

Example

By definition, the sum of the integers *1*, ..., *N* is the *Nth* triangular number. The following example finds the first triangular number greater than a given limit.

```
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
        )                     ; do
    )                         ; procedure

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

Reference

while

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_result	First tail of *l_valueList* whose `car` satisfies *g_predicateExpr*.
`nil`	If none of the elements in *l_valueList* can satisfy it.
`t`	Entry in an association table satisfies *g_predicateExpr*.

Example

```
exists( x '(1 2 3 4) (x > 1) )  => (2 3 4)
exists( x '(1 2 3 4) (x > 4) ) => nil


exists( key myTable (and (stringp key)
             (stringp myTable[key])))
=> t
```

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

Reference

car, cdr

existss

```
existss(
     s_formalVar
     l_valueList
     g_predicateExpr
     )
     => g_result

     existss(
     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. In the SKILL++ mode, this function always locally wraps the loop or iterator local variable (`s_formalVar`) in a `let` block while compiling the code. Local wrapping preserves the lexical scope of the loop variable. This function may work slower than its non-wrapped counterpart `exists`. 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_result	First tail of *l_valueList* whose `car` satisfies *g_predicateExpr*.
`nil`	If none of the elements in *l_valueList* can satisfy it.
`t`	Entry in an association table satisfies *g_predicateExpr*.

Example

```
(defun test_exists (x)
    existss( x (list x x+1 x+9) println(x))
    println(x)
    )
test_exists(9)

 => 9
10
18
9
nil
```

for

```
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 maximum value for the loop variable is INT_MAX-1. 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

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

Reference

foreach

fors

```
fors(
    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. In the SKILL++ mode, this function always locally wraps the loop or iterator local variable (s_loopVar) in a let() block while compiling the code. Local wrapping preserves the lexical scope of the loop variable. This function may work slower than its non-wrapped counterpart `for`. 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 maximum value for the loop variable is INT_MAX-1.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

```
(defun test_for (x)
    fors( x x+1 x+9 println(x))
    println(x)
    )
test_for(9)
 => 10
11
12
13
14
15
16
17
18
9
nil
```

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 `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

s_formalVar	Local variable usually referenced in *g_predicateExpr*.
l_valueList	List of elements that are bound to *s_formalVar* one at a time.
g_predicateExpr	A SKILL expression that usually uses the value of *s_formalVar*.
s_key	Key portion of the table entry.
o_table	Association table containing the entries to be processed.

Value Returned

`t`	If *g_predicateExpr* evaluates to non-`nil` for every element in *l_valueList* or for every key in an association table.
`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

foralls

```
foralls(
     s_formalVar
     l_valueList
     g_predicateExpr
     )
     => t / nil
     foralls(
     s_key
     o_table
     g_predicateExpr
     )
     => t / nil
```

Description

Checks if *`g_predicateExpr`* evaluates to non-`nil` for every element in *`l_valueList`*. In the SKILL++ mode, this function always locally wraps the loop or iterator local variable (*`s_formalVar`*) in a `let` block while compiling the code. Local wrapping preserves the lexical scope of the loop variable. This function may work slower than its non-wrapped counterpart `forall`. This is a syntax form.

Verifies that an expression remains true for every element in a list. The `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

s_formalVar	Local variable usually referenced in g_predicateExpr.
l_valueList	List of elements that are bound to s_formalVar one at a time.
g_predicateExpr	A SKILL expression that usually uses the value of s_formalVar.
s_key	Key portion of the table entry.
o_table	Association table containing the entries to be processed.

Value Returned

t	If g_predicateExpr evaluates to non-nil for every element in l_valueList or for every key in an association table.
nil	Otherwise.

Example

```
(defun test_forall (x)
    foralls( x (list x x+1 x+9) println(x))
    println(x)
    )
test_forall(9)
 => 9
9
nil
```

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.

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

The third syntax form of `foreach` can be used to process the elements of an association table. In this case, *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

s_formalVar	Name of the variable.
s_mappingFunction	One of `map`, `mapc`, `mapcan`, `mapcar`, or `maplist`.
g_exprList	Expression whose value is a list of elements to assign to the formal variable *s_formalVar*.
g_expr1, g_expr2	Expressions to execute.
g_exprTable	Association table whose elements are to be processed.

Value Returned

l_valueList	Value of the second argument, *g_exprList*.
l_result	The result of the last expression evaluated.
o_valueTable	Value of *g_exprTable*.

Example

```
foreach( x '(1 2 3 4) println(x))
1                   ; Prints the numbers 1 through 4.
2
3
4
=> (1 2 3 4)        ; Returns the second argument to foreach.
```

The next example shows `foreach` accessing an association table and printing each key and its associated data.

```
foreach(key myTable printf("%L : %L\n" key myTable[key]))
```

Example with more than one loop variable:

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

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.

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

```
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)
```

foreachs

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

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

     foreachs(
     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. In the SKILL++ mode, this function always locally wraps the loop or iterator local variable, *s_formalVar,* in a `let` block while compiling the code. Local wrapping preserves the lexical scope of the loop variable. This function may work slower than its non-wrapped counterpart `foreach`. This is a syntax form.

The first form shown in the syntax above 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.

The second form shown in the syntax above 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.

The third form shown in the syntax above can be used to process the elements of an association table. In this case, *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

s_formalVar	Name of the local loop variable that must not be changed inside the loop
s_mappingFunction	One of `map`, `mapc`, `mapcan`, `mapcar`, or `maplist`
g_exprList	Expression whose value is a list of elements to assign to the formal variable *s_formalVar*
g_expr1, g_expr2	Expressions to execute
g_exprTable	Association table whose elements are to be processed

Value Returned

l_valueList	Value of the second argument, *g_exprList*
l_result	The result of the last expression evaluated
o_valueTable	Value of *g_exprTable*

Example

```
toplevel('ils)
(defun test_foreach (x)
    foreachs( x (list x x+1 x+9) println(x))
    println(x)
    )
test_foreach(9)
 => 9
10
18
9
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.

```
if( g_condition g_thenExpression [ g_elseExpression ] )
=> g_result
```

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

```
x = 2
if( (x > 5) 1 0)
=> 0                    ; Returns 0 because x is less than 5.
a = "polygon"
if( (a == "polygon") print(a) )

"polygon"                ; Prints the string polygon.
=> nil                   ; Returns the result of print.

x = 5
if( x "non-nil" "nil" )
=> "non-nil"             ; Returns "non-nil" because x was not
                         ; nil. If x was nil then "nil" would be
                         ; returned.

x = 7
if( (x > 5) then 1 else 0)
=> 1                     ; Returns 1 because x is greater than 5.

if( (x > 5)
    then println("x is greater than 5")
        x + 1
    else print("x is less ")
        x - 1)

x is greater than 5      ; Printed if x was 7.
=> 8                     ; Returned 8 if x was 7.
```

Reference

cond, foreach, unless, while

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 labeled statement inside any `progs` that contain the `go` statement, but cannot be transferred to labeled statements in a `prog` that is not active at the time the `go` statement is executed. Usually, 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.

```
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

foreach, return, while

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)`.

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 destructively modified and concatenated so that 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.

The values returned from successive calls to `u_func` are put into a list using the `list` function. If the argument lists are of different lengths, the `mapcar` function iterates till the end of the shortest list.

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.

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( 'plus '(1 2 3 4) '(4 5) '(1 2 3 4 5 6) '(1 2 3))
=> (7 11)

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

map, mapc, mapcan, mapcon, maplist, nconc

mapcon

```
mapcon(
     u_func
     l_arg1
     [l_arg2]
     )
     => l_result
```

Description

Applies the function *u_func* to successive sublists of the lists and returns a concatenated list.

Arguments

u_func	Specifies the function to be applied to the given list. Must accept lists as arguments. The result of calling *u_func* can be of any data type.
l_arg1	Specifies the argument list to be processed
l_arg2	Additional argument lists, which must be the same length as *l_arg1*

Value Returned

l_result	Returns a concatenated list that results from calling the *u_func* on the cons cells of the given list

Example

```
mapcon((lambda (x)
(printf "x = %L\n" x)
(list (car x) (add1 (car x)))) '(1 2 3 4)); lambda: (u_func) with one argument
x = (1 2 3 4)
x = (2 3 4)
x = (3 4)
x = (4)
result: (1 2 2 3 3 4 4 5)
```

```
mapcon((lambda (x y) (printf "x = %L y = %L\n" x y)
```

```
  (list (car x) (add1 (car y))))
  '(1 2 3 4)
  '(4 3 2 1)
) ; lambda: (u_func) is with 2 arguments
x = (1 2 3 4) y = (4 3 2 1)
x = (2 3 4) y = (3 2 1)
x = (3 4) y = (2 1)
x = (4) y = (1)
result : (1 5 2 4 3 3 4 2)
```

mapinto

```
mapinto(
     l_resultSequence
     g_function
     ({l_sequences}*)
     )
     => l_resultSequence
```

Description

Applies `g_function` to the elements of `l_sequences` and destructively modifies the `l_resultSequence`. The first argument is a sequence that receives the results of the mapping. If `l_resultSequence` and the other argument sequences are not all of the same length, the mapping stops when the shortest of `l_resultSequence` or `l_sequences` is exhausted.

If `l_resultSequence` is longer than `l_sequences`, extra elements at the end of `l_resultSequence` are unchanged.

Note: If you specify `nil` as the `l_resultSequence`, no mapping is performed since `nil` is a sequence of length zero.

Arguments

l_resultSequence	A sequence that receives the results of the mapping.
g_function	Function (`symbol` or `funobj`) that takes as many arguments as there are sequences.
l_sequences	Several lists. Each element of these lists is used as an argument of `g_function`.

Value Returned

l_resultSequence	Updated first argument list.

Example

```
mapinto ('(1 2 3 4) 'plus )
=>(1 2 3 4)
mapinto (' (1 2 3 4) 'plus ())
=>(1 2 3 4)
```

```
a = '(1 2 3 4 5)
mapinto ( a 'plus '(1 1 1) '(1 1))
=>(2 2 3 4 5)
```

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

map, mapc, mapcan, mapcar, nconc

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

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

```
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, regExitBefore, remExitProc

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 `regExitBefore` or `regExitAfter` function.

Arguments

s_name	Name of the registered exit procedure to be removed.

Value Returned

`t`	Always.

Example

```
remExitProc( 'endProc) => t
```

Reference

exit, regExitBefore, regExitAfter

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

```
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, go

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")
```

setofs

```
setofs(
     s_formalVar
     l_valueList
     g_predicateExpression
     )
     => l_result

     setofs(
     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. In the SKILL++ mode, this function always locally wraps the loop or iterator local variable (*s_formalVar*) in a `let` block while compiling the code. Local wrapping preserves the lexical scope of the loop variable. This function may work slower than its non-wrapped counterpart `setof`. 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

```
(defun test_setof (x)
    setofs( x (list x x+1 x+9) println(x))
    println(x)
    )
test_setof(9)

 => 9
10
18
9
nil
```

throw

```
throw(
     s_tag
     g_value
     )
     =>
```

Description

Transfers the control back to the return point established in a catch block. The argument value is used as the value to be passed. The `throw` function should always be used inside `catch(. . . g_form . . .)`.

Arguments

s_tag	Specifies the return point in a catch block
g_value	Evaluates forms and saves the results. If the form produces multiple values, then all the values are saved. The saved results are returned as the value or values of catch.

Value Returned

Transfers the control back to the return point in a catch block

Example

```
catch( 'problem
    begin(
        throw( 'problem
                let( nil
                printf( "Caught %L\n" 1) 1
            )
                )
            2
        )
)
Caught 1  ;  message is printed . . .
= > 1  ;  value returned by throw()
```

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

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

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 `g_condition` and the sequence of expressions `g_expr1 ...` if the condition is true. This process is repeated until `g_condition` evaluates to false (`nil`). Because this form always returns `t`, it is principally used for its side-effects.

Arguments

g_condition	Any SKILL expression.
g_expr1	Any SKILL expression.

Value Returned

`t`	Always returns `t`.

Example

```
i = 0
while( (i <= 10) printf("%d\n" i++) )
=> t
```

Prints the digits 0 through 10.

Reference

foreach

11

Input Output Functions

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

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. The command sets the switch `fullPrecision` to `t` to retain floating point number precision while saving the file. It is not the same as encrypting the file because the representation of *`t_destFile`* is still in ASCII format. This process does not remove the source file.

Arguments

`t_sourceFile`	Name of the SKILL source file.
`t_destFile`	Name of the destination file.

Value Returned

`t`	Returns `t` when function executes successfully.
`error message`	Signals an error if problems are encountered compressing the file.

Example

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

Reference

`encrypt`

display

```
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. `poport` is the default.

Value Returned

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

Example

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

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

Reference

drain, print, write

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 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

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

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.

Example

```
edi( "~/myFile.il" )
```

Reference

ed, edit, edl

edit

```
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

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.

Example

```
edl( "/tmp/demo.il" )
```

Reference

ed, edi, edit

encrypt

```
encrypt(
     t_sourceFile
     t_destFile
     [ t_password ]
     )
     => t
```

Description

Encrypts a SKILL file and places the output into another file.

If a password is supplied, the same password must be given to the command used to reload the encrypted file.

Arguments

t_sourceFile	Name of the SKILL file you are encrypting.
t_destFile	Destination file you want the encrypted file to be placed in.
t_password	Optional password; you are asked for it before you can reload the encrypted file.

Value Returned

`t`	When the file has been encrypted and placed in *t_destFile*. Signals an error if you fail to name a destination file or give the name of a file already present.

Example 1

```
encrypt( "triadb" "myPlace" "option") => t
```

Encrypts the `triadb` file into the `myPlace` file with `option` as the password. Returns `t` if successful.

Example 2

```
encrypt("file.il" "file.ile") ; SKILL file
encrypt("file_sc.ils" "file_sc.ilse") ; SCHEME file
```

Reference

`compress`, `load`

expandMacroDeep

```
expandMacroDeep(
     g_form
     )
     => g_expandedForm
```

Description

This function recursively expands all macros specified in *g_form*.

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

```
expandMacroDeep(myFunction(1 2))
```

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, is not considered as a relative path.

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

isFile, isFileName

fileSeek

```
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
```

```
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, 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 seconds from January 1, 1970). The 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.
nil	No file with the given name was found.

Example

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

Reference

getCurrentTime

fprintf

```
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

Format Specification	Type(s) of Argument	Prints
%d	fixnum	Integer in decimal radix
%o	fixnum	Integer in octal
%x	fixnum	Integer in hexadecimal
%f	flonum	Floating-point number in the style [-]ddd.ddd
%e	flonum	Floating-point number in the style [-]d.ddde[-]ddd
%g	flonum	Floating-point number in style f or e, whichever gives full precision in minimum space **Note:** Qualifying %g with width may cause imprecise results to be printed.

Common Output Format Specifications

Format Specification	Type(s) of Argument	Prints
%s	string, symbol	Prints out a string (without quotes) or the print name of a symbol
%c	string, symbol	The first character
%n	fixnum, flonum	Number
%P	list	Point
%B	list	Box
%N	any	Prints an object in the old style, that is, does not call the `printself` function
%L	list	Default format for the data type Print behavior depends on the value of the printpretty variable: If printpretty is `nil`, this behaves like %N If printpretty is `non-nil` (default), %L uses `printself` for standard objects
%A	any	Prints any type of object using the `printself` representation

The *`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.

```
%[-][width][.precision]conversion_code
    [-]              = left justify
    [width]          = minimum number of character positions
    [.precision]     = number of characters to be printed
    conversion_code
```

Arguments

p_port	Output port to write to.
t_formatString	Characters to be printed verbatim, intermixed with format specifications prefixed by the `%` sign.
g_arg1	The arguments following the format string are printed according to their corresponding format specifications.

Value Returned

`t`	Prints the formatted output and returns `t`.

Example 1

```
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
```

Example 2

This example shows the use of `%A`, which calls the `printself` method.

```
defmethod(printself ((obj fixnum))
    sprintf(nil "FIXNUM{%d}" obj));;Defines the printself method
    printf("Print control A returns: %A\n" 42);; %A calls the printself method
=> Print control A returns: FIXNUM{42}
```

Example 3

This example shows the use of `%L`, which calls `printself` only for standard objects.

```
defmethod(printself ((obj fixnum))
    sprintf(nil "FIXNUM{%d}" obj));;Defines the printself method
    printf("Print control L returns: %L\n" 42)
=> Print control L returns: 42
```

Example 4

This example shows the use of `%L`, `%A`, and `%N` print controls with `printf` when printing standard objects. `%A` prints the same result as `%L` and `%N` does not call the `printself` method.

```
defclass(A () ());; Defines a class A
defmethod(printself ((obj A));; Defines the printself method
    sprintf(nil "OBJ_A{%L}" obj))
    printf("Print control L returns: %L\n" Instance('A))
    printf("Print control A returns: %A\n" Instance('A))
    printf("Print control N returns: %N\n" Instance('A))
=> Print control L returns: OBJ_A{stdobj@0x83bf024}
Print control A returns: OBJ_A{stdobj@0x83bf024}
Print control N returns: stdobj@0x83bf03c
```

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 main 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. Another difference is that whereas `sscanf` supports the width while reading floating-point numbers from the input string, `fscanf` and `scanf` do not.

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 `fscanf` are summarized below.

Common Input Format Specifications

Format Specification	Type(s) of Argument	Scans for
%d	fixnum	An integer
%f	flonum	A floating-point number
%s	string	A string (delimited by spaces) in the input

Arguments

p_inputPort	Input port `fscanf` reads from. The input port cannot be the CIW for `fscanf`.
t_sourceString	Input string for `sscanf`.
t_formatString	Format string to match against in the reading.
s_var1	Name of variable to store results of read.

Value Returned

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.
`nil`	It encounters an end of file.

Example

```
fscanf( p "%d %f" i d )
```

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

Assume a file `testcase` with one line:

```
hello 2 3 world
```

```
x = infile("testcase")=> port:"testcase"
fscanf( x "%s %d %d %s" a b c d )=> 4
(list a b c d) => ("hello" 2 3 "world")
```

Scans the given floating point number as `val1` (`1.23`) and `val2` (`4`) and returns the resulting number as `2` because two values were read.

```
s = "1.234"
sscanf(s "%4f%d" val1 val2)
```

Reference

fprintf, lineread

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"
```

getc

```
getc(
     [ 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

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

gets, getSkillPath

getOutstring

```
getOutstring(
     s_port)
     => t_string / nil
```

Description

Retrieves the content of the outstring port (while it is open).

Arguments

s_port	Specifies the outstring port from which the content needs to be retrieved

Value Returned

t_string	Returns the string read from the outstring port
nil	Returns `nil` if the string cannot be read from the outstring port

Example

```
s = outstring()
= >port:"*string*"
fprintf(s "Quick brown")
getOutstring(s)
=>"Quick brown"
fprintf(s " fox jumps")
getOutstring(s)
=> "Quick brown fox jumps"
 fprintf(s " over the lazy dog")
getOutstring(s)
=> "Quick brown fox jumps over the lazy dog"
close(s)
getOutstring(s)
=> nil
```

gets

```
gets(
     g_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. You can also specify `nil` instead of a variable name.
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. *s_variableName* stores the last value returned (that is, `nil`).

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")      => port:"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"


var = gets(nil port) -- read into var from port
var = gets(nil) -- read into var from <stdin>
```

Reference

getc, getchar, infile

include

```
include(
     t_file)
     => t / error
```

Description

Loads the file with name *t_file* in SKILL regardless of any errors in the file.

Arguments

t_file	Name of the file you want to load; it should be a string value.

Value Returned

t	The file loads sucessfully.
error	The file specified as *t_file* does not exist.

Example 1

```
include("./test.il")
t
```

Example 2

```
include("")
*WARNING* open : empty file name
*Error* include: can't access file - ""
```

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`. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

info

```
info(
     t_formatString
     [ g_args1... ]
     )
     => nil
```

Description

Prints the formatted output to poport according to the specification.

Arguments

r_formatString	Format specification string.
g_args	Arguments following the format string.

Value Returned

`nil`	Prints the argument value to poport.

Example1

```
info("Hello Skill") ; prints "Hello Skill"
Hello Skill
nil
```

Example2

```
info("value = %d" 42) ; prints value = 42
value = 42
nil
```

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

instring

```
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

```
s = "Hello World!"        => "Hello World!"
p = instring(s)           => port:"*string*"
fscanf(p "%s %s" a b)     => 2
a                         => "Hello"
b                         => "World!"
close(p)=> t
```

Reference

gets, infile

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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 se.

```
isFile( "triadc") => t
isFile( ".cshrc" list("." "~")) => t
```

Reference

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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.
`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

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. A path that is anchored to the current directory, for example, `./` , `../`, or `../../..`, and so on, 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

isFile, getSkillPath

isLargeFile

```
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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

```
fileLength( "largeFile" ) => 3072000000
isLargeFile( "largeFile" ) => t
```

Reference

fileLength, isFile, isFileName

isLink

```
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`. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

```
isLink("/usr/bin")=> nil
isLink("/usr/spool")=> t    ;Assuming it's a link to /var/spool
```

Reference

isFile

isPortAtEOF

```
isPortAtEOF(
     p_port
     )
     => t / nil
```

Description

Takes an input port and returns `t` if end-of-file (EOF) has previously been detected while reading the input port; it returns `nil` otherwise.

Arguments

p_port	Input port. This must be open, otherwise the function will return an error.

Value Returned

`t`	End-of-file (EOF) has previously been detected while reading the input port *p_port*.
`nil`	End-of-file (EOF) has not been reached yet.

Example

```
port = infile("input_file")
while(! isPortAtEOF(port)
    printf("%L\n" read(port))
)
close(port)
```

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

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 +                 ; First input line of the file being read
3                       ; Second input line
=> (f 1 (2 + 3))
lineread(piport)
f(a b c)                ; Another input line of the file
=> ((f a b c))          ; 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

load

```
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`/`.ile` for SKILL and `.ils`/`.ilse` for SKILL++) for processing the language expressions contained in the file. By default, the loaded code is evaluated in dynamic scoping. However, if the extension is `.ils`/`.ilse`, lexical scoping is used. 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.

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. The valid values are: ■ `'ils/'ilse`, which indicates that the file contains SKILL++ code. ■ `'il/'ils`, which indicates that the file contains SKILL code.
t_password	Password, if *t_fileName* is an encrypted file.

Value Returned

`t`	The file is successfully loaded.

Example

```
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.

```
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.

```
procedure( trLoadSystem()
    load( "UtilsA.il" )    ;;; SKILL code
    load( "UtilsB.ils" )   ;;; SKILL++ code
    )                     ; procedure
```

Reference

`include`, `loadContext`, loadi, lineread

loadi

```
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

```
loadi( "testfns.il" )
```

Loads the `testfns.il` file.

```
loadi( "/tmp/test.il")
```

Loads the `test.il` file from the `tmp` directory.

Reference

`encrypt`, `include`, load, lineread

loadPort

```
loadPort(
     p_port
     [?langMode g_langMode]
     [?password g_password]
     [?ignoreErrors g_ignoreErrors])
     )
     => t
```

Description

Loads a SKILL file from *p_port*.

Arguments

p_port An input (SKILL) port.

`?langMode` *g_langMode*

Specifies the language mode to use regardless of the original file extension
Valid values:

`'ils`: Loads the file in SKILL++ mode

`'il`: Loads the file in SKILL mode

Default value: `'il`

`?password` *g_password*

Password, if the file is encrypted

`?ignoreErrors` *g_ignoreErrors*

If specified, ignores errors during load

Value Returned

`t` Always returns `t`.

Example

```
loadPort( myPort ?langMode 'ils )
```

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	File to be loaded. Uses the file name extension to determine the language mode to use. The valid values are: ■ `'ils`, which indicates that the file contains SKILL++ code. ■ `'il`, which indicates that 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

outstring

```
outstring(
    )
    => p_openedPort / nil
```

Description

Takes no arguments and returns an opened output port for strings (or an outport). After a port is opened, it can be used with functions, such as `fprintf`, `println`, and `close` that write to an output port. You need to use the `getOutstring` function to retrieve the content of output port (while it is open).

You can use the `close` function to close the output port.

Arguments

Takes no arguments

Value Returned

p_openedPort	The opened output port
`nil`	Returns `nil` if the port cannot be opened

Example

```
s = outstring() ; string port opened for output
=> port:s
fprintf(s "the value is %d" 1) ; fprintf into string
getOutstring(s)
=> the value is 1
close(s)
getOutstring(s)
=> nil
```

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 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.

newline

```
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

```
print("Hello") newline() print("World!")
"Hello"
"World!"
=> nil
```

Reference

drain, fprintf, outfile

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

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
```

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. A path that is anchored to the current directory, for example, `./` , `../`, or `../../..`, and so on, 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 openning 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 = 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

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

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` prints the list on one (possibly huge) line but `pprint` limits the output on a single line and produces a multiple line printout if necessary. This output is 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

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

g_value	Any SKILL object.
p_outputPort	Output port to print to. Default is `poport`.

Value Returned

`nil`	Always returns `nil` after printing out the object supplied.

Example

```
print("hello")
"hello"
=> nil
```

Reference

pprint, println, printlev

printf

```
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

```
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 &)) ...)             ; Prints to port p.
=> 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 *g_value*. `println` flushes the output port after printing each newline character.

Arguments

g_value	Any SKILL value.
p_outputPort	Port to be used for output. The default is `poport`.

Value Returned

`nil`	Prints the given object and returns `nil`.

Example

```
for( i 1 3 println( "hello" ))     ;Prints hello three times.
"hello"
"hello"
"hello"
=> t                              ;for always returns t
```

Reference

drain, print, newline

putc

```
putc(
     x_symbol
     p_port
     )
=> s_symbol
```

Description

Puts the *x_symbol* to *p_port* (to complement getc function)

Arguments

x_symbol	Symbol number
p_port	An output port

Value Returned

s_symbol	The symbol that was put

Example

```
putc(1  poport)
  => \001
```

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. 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

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

makeTempFileName, writeTable

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*. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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.

Note: If you do not have sufficient privileges to rename a file or directory, the `renameFile()` function throws an error (neither returns `t` nor `nil`). You can use the `errset()` function to handle such exceptional situations. For more information on the `errset()` function, see The errset Function in the *Cadence SKILL Language User Guide*.

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
renameFile( "/usr/old" "/usr/new" ) ; if you do not have permissions to rename old
*Error* renameFile: permission denied
```

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

isFileName

simplifyFilenameUnique

```
simplifyFilenameUnique(
     t_path
     )
     => t_fullPath / error message
```

Description

Returns the full path for the given *t_path* without links and a trailing slash / at the end of the result string. The function returns an error if the given *t_path* is incorrect.

Arguments

t_path	Path to a directory or file.

Value Returned

t_fullPath	Full path for the given *t_path* without links and a trailing slash / at the end.

Example

```
;The example below illustrates the difference between the simplifyFilename and
simplifyFilenameUnique functions
simplifyFilename(".////")
=> "/home/user1/"
simplifyFilenameUnique(".////")
=> "/home/user1"
```

truename

```
truename(
     t_string
     )
     => t_truename
```

Description

Tries to find the specified file (`t_string`) and returns the full path to the file.

It uses the current SKILL path for relative paths. A path that is anchored to the current directory, for example, ./, ../, or ../../.., and so on, is not considered as a relative path.

Arguments

t_string	A string specifying the file name.

Value Returned

t_truename	The truename or full path of the specified file.

Example

```
getSkillPath()
=> ("." "~")
setSkillPath(append1(getSkillPath() "~/skill"))
=> nil
getSkillPath()
=> ("." "~" "~/skill")
getWorkingDir()
=> "/home/skillproj/work"
truename("./runtest")
=> "/home/skillproj/work/runtest"
truename("mycode.il")
=> "/home/skillproj/skill/mycode.il"
truename(".cshrc")
=> "/home/skillproj/.cshrc"
truename("~/old/code.il")
=> nil ; this file/directory does not exist
```

Reference

which

which

```
which(
     t_fileName
     )
     => t_fullPath / 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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, is not considered as a relative path.

Note: *t_fileName* should be a simple file or directory name, and should not contain directory separators.

Arguments

t_fileName	Name of a context file, or a regular file or directory that you want to get the absolute path.

Value Returned

t_fullPath	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/deeptik/.cdsinit"
```

Reference

truename

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 `poport`.

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

makeTempFileName, readTable

12

Core Functions

arglist

```
arglist(
     g_function
     )
     => l_argumentList
```

Description

Returns the number and types of arguments expected for a function. Also checks if the specified function is a binary object,

This function is useful for determining how many arguments a function takes and what they are.

If the function is read-protected, the arguments are still returned. If the function is a primitive (binary), the argument list is based on the type template for the function specified. If the function is defined in SKILL, the argument list in the function definition is returned.

Arguments

g_function	Name of the function or the symbol whose argument list you want to see.

Value Returned

l_argumentList	Number and types of arguments for *g_function*.

Example

```
arglist('rexMatchp) => ( t_string S_stringSymbol "tS")
```

The first argument of `rexMatchp` must be a string and the second must be a string or symbol.

assert

```
assert(
     g_expression
     )
     => nil
```

Description

Enables you to insert assertions into the SKILL code, either at the top-level or within a function. It evaluates the expression (`g_expression`) and returns `nil` if the expression value is non-nil. Otherwise, throws an error and returns the unevaluated expression.

Arguments

g_expression	A generic expression.

Value Returned

nil	Assertion is successful.
g_expression	Assertion failed.

Example

```
assert(1 == 1)
=> nil
```

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 `nil` is both an atom and a list.

Arguments

g_arg	Any SKILL object.

Value Returned

`t`	If *g_arg* is an atom.
`nil`	If *g_arg* is not an atom.

Example

```
atom( 'hello )  => t

x = '(a b c)
atom( x )       => nil

atom( nil )     => t
```

bcdp

```
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

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

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
```

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: `boundp()` does not check the current language mode. If single argument is specified, SKILL semantics are used, whereas if two arguments are specified, SKILL++ semantics are used.

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

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

boundp( 'x )
=> t

boundp( 'y )
=> nil


y = 'x                ; Bind y to the constant x.
boundp( y )
```

```
=> t                    ; Returns t because y evaluates to x,
                        ; which is bound.
```

describe

```
describe(
     [ s_symbol ]
     )
     => t
```

Description

Prints information about the symbol `s_symbol`. If the symbol has a function definition, information on the argument list and other available details will be printed. If the symbol has a variable definition, information about its value and function will be printed. If the function is called without any arguments, the help message will be printed.

Arguments

s_symbol	SKILL symbol to print information for.

Value Returned

`t`	Information about the symbol is printed.

Example

```
describe('append)
Symbol append has a function definition.
Its argument list is (g_general g_general "gg")
This is a built-in function.
=> t
```

fdoc

```
fdoc(
     s_function
     )
     => t_doc / nil
```

Description

Returns the documentation string for the function bound to the symbol *`s_function`*. SKILL switch `saveInlineDoc` must be set to save and retrieve the doc string.

Arguments

`s_function`	A symbol for the SKILL function name.

Value Returned

`t_doc`	Documentation string if available.
`nil`	Inline documentation is not available.

Example

```
sstatus(saveInlineDoc t) ;; enable inline documentation in compile time
defun(myFun (a b)
"documentation for myFun: return sum a and b"
a + b
)
fdoc('myFun)
=> "documentation for myFun: return sum a and b"=
```

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

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.
```

getMuffleWarnings

```
getMuffleWarnings(
     )
     => l_list
```

Description

Returns a list of warnings that were called and suppressed by the preceding muffleWarnings command.

Arguments

None.

Value Returned

l_list	List of warnings or `nil`, if no warnings were called.

Example

```
muffleWarnings(
    warn("A first warning 1 level")
    warn("A second warning 1 level")
    muffleWarnings(
        warn("A first warning 2 level")
        warn("A second warning 2 level")
        2+2
    )
    => 4
    getMuffleWarnings()
    => ("A first warning 2 level" "A second warning 2 level")
    1+2
)
=> 3
getMuffleWarnings() => ("A first warning 1 level" "A second warning 1 level")
```

Results pertain to the preceding `muffleWarnings` command.

getSkillVersion

```
getSkillVersion(
     [g_printSubVersion]
     )
     => t_version
```

Description

Returns the major version if the argument is left `blank`; otherwise, returns the current subversion (or tarkit version) of SKILL that is running in the build

Arguments

g_printSubVersion

(Optional) Specify a flag to print the current subversion (or tarkit version) of SKILL running in the build

Value Returned

t_version	If the argument flag is left `blank`, returns the major version of SKILL running in the build. If the argument flag is specified, returns the current subversion (tarkit version) of SKILL running in the build

Example

```
getSkillVersion()
=> "SKILL04.20"

getSkillVersion(t)
"@(#)$CDS:   il skillSrc33.12-d009 08/31/11 14:50 fwinteg sjfdl803 $"
```

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

get_string

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

get_pname

getVersion

```
getVersion(
     [ g_opt ]
     )
     => t_[sub]version
```

Description

Returns the version number of the Cadence software you are currently using. If you specify the optional argument *g_opt,* as *t* (or a non-nil value), the subversion number of the Cadence software currently used is returned. By default, the full version number, including the hotfix version, of the Cadence software currently used is returned.

Use the SKILL system structure to determine the bitType (32bit / 64bit) of the current Virtuoso session:

```
system.LP64
=> nil ;; 32bit
=> t  ;; 64bit
```

or

```
system.system.ILP32
=> nil ;; 64bit
=> t  ;; 32bit
```

These `system.??` properties are initialized at startup.

Arguments

g_opt	Optional argument.
	If the optional argument, *g_opt*, is specified as *t* (or a non-nil value), the subversion number of the Cadence software currently used is returned. By default, the full version number, including the hotfix version, of the Cadence software currently used is returned.

Value Returned

t_[sub]version	String identifying the version/subversion of the program you are running.

Example

```
getVersion() => "@(#)$CDS: virtuoso version 6.1.6 07/24/2012 11:02 (cic612sun) $"
getVersion(nil) => "@(#)$CDS: virtuoso version 6.1.6 07/24/2012 11:02 (cic612sun) $"
getVersion( 'subVer ) => "sub-version IC6.1.6.DEL.410"
getVersion(t) => "sub-version  IC6.1.6.DEL.410 "
getVersion("subversion") => "sub-version  IC6.1.6.DEL.410 "
```

Reference

dbGetVersion

getWarn

```
getWarn(
     )
     => t_warning
```

Description

Returns the buffered warning if it has not already been printed.

Arguments

None.

Value Returned

t_warning	The warning message that would have been printed if it had not been intercepted by the call to `getWarn`.

Example

```
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
```

The `testWarn` function intercepts the last warning message and stores it in a global variable if `t` is passed in, and lets the system print all the warnings if `nil` is given as an argument. Use of the getWarn( ) function makes it possible to throw away a warning message, if desired.

```
testWarn( ?getLastWarn t)
=> nil
*WARNING* This is warning 1
*WARNING* This is warning 2
```

Returns `nil.` The system prints the first two warnings and the third is intercepted and stored in global variable *thrownAwayWarn*.

```
testWarn( ?getLastWarn nil)
=> nil
*WARNING* This is warning 1
*WARNING* This is warning 2
*WARNING* This is warning 3
```

Returns `nil`. The system prints all the queued warnings.

The return value may be interleaved with the warning message output. The following example shows how the output can appear in the CIW.

```
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
```

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
```

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.

isVarImported

```
isVarImported(
     s_var
     )
     => t / nil
```

Description

Checks if the specified variable was imported into SKILL++ or not.

Arguments

s_var	The variable to be checked.

Value Returned

`t`	The specified variable `s_var` was imported into SKILL++.
`nil`	Returns `nil`, if the given variable is not imported.

Example

```
isVarImported('myvar)
=> nil
```

makeSymbol

```
makeSymbol(
     S_createSymbol
     [ t_namespaceArg ]
     )
     => s_result
```

Description

Creates a symbol corresponding to the specified symbol or character string. In IC6.1.6 and later releases, optionally specify the namespace name (*t_namespace*) in which you want to create the symbol.

Note: The function `gensym()` also creates symbols. However, the symbol names are determined internally (and are therefore unique) whereas in the case of `makeSymbol()` the symbol name depends upon the string passed as a parameter to the function.

Arguments

S_createSymbol	Specifies the value for which a corresponding symbol needs to be created.
t_namespaceArg	(Optional) Specifies the name of the namespace in which you want to create the symbol.

Value Returned

s_result	Returns a symbol corresponding to the specifed string value.

Example 1

The following example creates a symbol corresponding to the specified string value, `myString`.

```
makeSymbol("myString")
=> myString
```

Example 2

The following example usesd an increment counter (`count`) to create unique symbols (`myString1, myString2, and so on`)

```
count=0
makeSymbol(strcat("myString" sprintf(nil "%L" ++count)))
```

Example 3

The following example creates a symbol, `myString,` in the namespace, `newNamespace`.

```
makeNamespace("newNamespace")
makeSymbol("myString" "newNamespace")
=> newNamespace:::myString
```

measureTime

```
measureTime(
     g_expression ...
     )
     => l_result
```

Description

Measures the time needed to evaluate an expression and returns a list with performance data (*n_utime*, *n_stime*, *n_clockTime*, and *x_pageFaults*) for the executed expressions. This is a syntax form.

- *n_utime*: The amount of user CPU time, in seconds, spent on the execution of expressions (counted with `getrusage()`).
- *n_stime*: The amount of system CPU time, in seconds, spent on the execution of expressions (counted with `getrusage()`).
- *n_clockTime*: The clock time used on execution of the expressions (in seconds) (counted with `gettimeofday()`). This function assumes that the executed expressions do not alter `getimeofday()` result.
- *x_pageFaults*: The number of page faults that occurred during the execution of the expressions (counted with `gettimeofday()`).

Arguments

g_expression	Expression(s) to be evaluated and timed.

Value Returned

l_result	Returns a list with performance data (*n_utime*, *n_stime*, *n_clockTime*, and *x_pageFaults*) for the executed expressions.

Example

```
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`.

```
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`.

```
tab = makeTable("testTable" 'unbound 5000)
=> table:testTable
```

```
result = measureTime(for(i 0 10000 tab[i] = t))
=> (0.003 0.0 0.002537012 0)
```

muffleWarnings

```
muffleWarnings(
     g_expr1 ...
     )
     => g_general
```

Description

Returns the result of the last expression evaluated. If the last expression evaluated calls the `warn` function (either SKILL `warn()` or C-level `ilWarn*`), the related message is not printed out.

To get the list of muffled warning messages, use the getMuffleWarnings function immediately after a `muffleWarnings` command.

Arguments

g_expr1	Expression(s) to be evaluated.

Value Returned

g_general	Result of the last expression evaluated.

Example

```
muffleWarnings(
    warn("A warning")
    warn("A second warning")
    1+2
)
=> 3

getMuffleWarnings()
=> ("A warning" "A second warning")
```

Result indicates 3 as the value evaluated from the last expression. The muffled warning messages are listed as a result of the `getMuffleWarnings` function.

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.

restoreFloat

```
restoreFloat(
     t_string
     )
     => f_number
```

Description

Restores a floating point number (`f_number`) from its serialized string (`t_string`) representation.

Note: `t_string` should be created by `saveFloat()`.

Arguments

t_string	A serialized float created by `ilSaveFloat()`.

Value Returned

f_number	The restored floating point number.

Example

```
str = saveFloat(1.4106)
=> "float:3ff6a09e667f3bcd@3ff691d14e3bcd36"
restoreFloat(str) == 1.4106
```

saveFloat

```
saveFloat(
     f_number
     )
     => t_string
```

Description

Serializes the given floating point number (`f_number`) to string (`t_string`).

Arguments

f_number	The floating point number that needs to be serialized.

Value Returned

t_string	The string representation of `f_number`.

Example

```
str = saveFloat(1.4106)
=> "float:3ff6a09e667f3bcd@3ff691d14e3bcd36"
```

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
```

setPrompts

```
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 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.

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

Name	Meaning	Default
`autoReload`	If `t`, the debugger will try to auto-reload a file that is not loaded under `debugMode` when the user tries to single-step into the code defined by that file. Note: This might not work correctly for SKILL++ functions defined using assignment.	`nil`
`classAuxAutoLoad`	If `t`, the SKILL++ code that accesses classes located in SKILL context files auto-loads the context (if this context has not already been loaded).	`nil`
`debugMode`	If `t`, provides more information for debugging SKILL programs. Allows you to redefine write-protected SKILL functions.	`nil`
`echoInput`	If `t`, each user input in CIW is repeated in the output port.	`nil`
`errsetTrace`	If `t`, prints errors and stacktrace information that is normally suppressed by `errset`.	`nil`
`forceWarnings`	If `t`, all warnings are flushed immediately, even if `getWarn()` is used or the warning is stored in a temporary buffer to be printed later in the CIW.	`nil`
`fullPrecision`	If `t`, unformatted print functions (`print`, `println`, `printlev`) print floating point numbers in full precision (usually 16 digits); otherwise, the default is about 7 digits of precision.	`nil`
`floatPrecisionChars`	Rounds off the value to the specified number of digits. For example, if set to `10`, the value has 10-digit accuracy. If `fullPrecision` is set to `t`, it is also considered and the value then has 17-digit accuracy. If `fullPrecision` is set to `nil`, the value still has 10-digit accuracy.	
`integermode`	If `t`, the parser translates all arithmetic operators into calls to functions that operate only on `fixnums`. This results in small execution time savings, particularly for compute-intensive tasks whose inner loops are dominated by integer arithmetic calculations.	`nil`

`keepSrcInfo`	If `t`, the source information (file/line information) is added to `funobject` during compilation. It is an auxiliary switch used by the V-code compiler.	`nil`
`lazyComp`	If `t`, tells V-code compiler to generate code (compile function) when it is called for the first time. That is, not to compile function after it is entered by the user (or loaded from a file) until it is called. If set to `nil`, enables the eager mode, where each function is compiled immediately after it is entered.	`t`
`mergemode`	If `t`, 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.	`t`
`multilineString`	If `t`, allows SKILL strings inside double quotes to be spanned on several lines.	`t`
`saveInlineDoc`	If `t`, when a SKILL function has inline documentation, allows the documentation string to be stored in the function symbol property.	`t`
`optimizeTailCall`	If `t`, enables the tail call recursion, which prevents runtime stack overflow when a function is called recursively. This works only in Scheme mode (`toplevel 'ils`).	
`printinfix`	Printing of arithmetic expressions and function calls in `infix` notation is turned off (on) if the second argument is `nil (t)`.	`t`
`printPretty`	If `t`, causes `printself` method to be called when printing standard objects. If set to `nil`, then `printself` method is not called and standard objects are printed as `stdobj@0x12345678`.	`t`
`pprintresult`	If `t`, causes SKILL I/O APIs to print lists with alignment and indentation (for example, print 5 elements per line)	`t`

keepNLInString	When set to nil, newline characters in strings are replaced with spaces. When set to t, the newline characters are retained as they are. This option is applicable only if status(multilineString) is t and the parsed string is inside a SKILL expression.	nil
writeProtect	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.	nil
savePcreData	When set to t, all pcre compiled objects are saved to context. After the context is loaded these objects are restored. **Note:** savePcreData can cause context incompatibility with previous versions of SKILL.	nil
stacktraceDump	Prints the local variables when an error occurs if sstatus(stacktrace t) is set. Toggle on/off with t / nil.	nil
stacktrace	Prints stack frames every time an error occurs. Toggle on or off with t or nil, or set the number of frames to display.	0
sourceTracing	If t, the debugger will try to print 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 to get its source line numbers. The source forms printed are truncated to fit on one line.	nil

`traceArgs`	If not set to `nil`, the system will save the evaluated arguments of function calls, which can then be displayed in the stacktrace. 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 expensive to retain the evaluated arguments. **Note:** Turning on this switch could slow down the execution speed significantly.	`nil`
`traceTEnable`	Allows the use of `t` as an argument to the `trace`, `tracev`, and `tracep` functions.	
`tracelength`	Limits the trace output width. If it exceeds the specifed width, ellipsis (...) are printed.	
`tracelevel`	Limits the depth of nested trace printing.	
`profCount`	If `t`, the SKILL Profiler provides the number of times a SKILL function is called (as an additional column in the table view of the profiler's result).	`nil`
`verboseLoad`	If `t`, prints the complete path of the loaded file in the CIW in debug mode.	`nil`
`verboseNamespace`	If `t`, enables the printing of warnings related to SKILL namespaces.	`nil`
`showStepResult`	If `t`, prints the expression evaluation results performed by the step command in CIW. If the SKILL IDE is also running, a new assistant window is displayed, which also displays the expression evaluation results.	`nil`
`optimizeNestedLet`	If `t`, instructs the SKILL compiler to parse the code for `let` constructions (defining local variables and local functions) and expand/remove them by moving their local variables to the top-level function's local variables section. **Note:** This variable works only for Scheme functions (for example, `.ils/.scm` files).	`nil`
`traceIndent`	If `t`, prints the trace with many '|||||', as in the earlier trace style. To print the trace in the new '|[%level]' construction, use the default value (`nil`).	`nil`

`debugMacro`	If `t`, the IL compiler sets lineNumber on the expanded macro code to lineNumber of the original form.	`nil`
`stackTraceFormat`	Controls the stacktrace output format. It can have one of three values: ■ `fullStack` prints the complete set of SKILL stack frames. ■ `onlyCall` suppresses the printing of non-function frames in the output. ■ `onlyTop` suppresses the printing of non-function frames except for the topmost function frame.	`fullStac`
`scopedMacros`	Identifies the scope of the macros and controls how the macros are expanded when processing the Scheme function body. If this switch is set to `t`, the scope is checked before expanding `MACRO/ALIAS: FUN(ARGS)`. If the function is found in local scope (in Scheme environment), the `MACRO/ALIAS` is not expanded to prevent it from calling a `GLOBAL` function.	`nil`

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 1

```
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.

```
sstatus( stacktrace 6 )         => 6
```

Prints the first six stack frames every time an error occurs.

Example 2

```
defun factorial (n) (if (n== 0) 1 (n*factorial(n-1]
=>factorial

(trace factorial);value of the traceIndent variable is nil, which is the default
value
=>(factorial)

(factorial 10)
|[1]factorial(10)
|[2]factorial(9)
|[3]factorial(8)
|[4]factorial(7)
|[5]factorial(6)
|[6]factorial(5)
|[7]factorial(4)
|[8]factorial(3)
|[9]factorial(2)
|[10]factorial(1)
|[11]factorial(0)
|[11]factorial --> 1
|[10]factorial --> 1
|[9]factorial --> 2
|[8]factorial --> 6
|[7]factorial --> 24
|[6]factorial --> 120
|[5]factorial --> 720
|[4]factorial --> 5040
|[3]factorial --> 40320
|[2]factorial --> 362880
|[1]factorial --> 3628800
3628800

(sstatus traceIndent t)
```

```
t
(factorial 10)
|factorial(10)
||factorial(9)
|||factorial(8)
||||factorial(7)
|||||factorial(6)
||||||factorial(5)
|||||||factorial(4)
||||||||factorial(3)
|||||||||factorial(2)
||||||||||factorial(1)
|||||||||||factorial(0)
|||||||||||factorial --> 1
||||||||||factorial --> 1
|||||||||factorial --> 2
||||||||factorial --> 6
|||||||factorial --> 24
||||||factorial --> 120
|||||factorial --> 720
||||factorial --> 5040
|||factorial --> 40320
||factorial --> 362880
|factorial --> 3628800
3628800
```

status

```
status(
     s_name
     )
     => g_switchValue
```

Description

Returns the value of the internal system variable named. This `nlambda` function also works in SKILL++ mode.

See the `sstatus` function for a list of the Internal System Variables.

Arguments

s_name	Name of internal system variable.

Value Returned

g_switchValue	Status of the internal system variable, usually either `t` or `nil`.

Example

```
status( debugMode ) => t
```

Checks the status of `debugMode` and returns `t` if `debugMode` is on.

The `status` function gets a switch. The `sstatus` function sets a switch.

```
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
```

theEnvironment

```
theEnvironment(
    [ u_funobj ]
    )
    => e_environment / nil
```

Description

(SKILL++ mode only) 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 = let( ( x )
    x = 3
    theEnvironment()
    ) ; let
=> envobj:0x1e0060
```

Returns the environment that the `let` expression establishes. The value of `Z` is an environment in which `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 = let( (( x theEnvironment()))
    x
    )
=> envobj:0x2fc018
eq( schemeTopLevelEnv() Z ) => t
```

Uses `theEnvironment` to illustrate that the variable initialization expressions in a `let` expression refer to the enclosing environment.

```
V = letrec( (( x theEnvironment()))
    x
    )
=> envobj:0x33506c
eq( schemeTopLevelEnv() V ) => nil
eq( V~>x V ) => t
```

Uses `theEnvironment` to illustrate that the variable initialization expressions in a `letrec` expression refers to the `letrec`'s environment.

```
W = let( (( r 3 ) ( y 4 ))
        let( (( z 5 ) ( v 6 ))
            theEnvironment()
            ) ; let
        ) ; let
=> envobj:0x456030c
W~>r => 3
W~>z => 5
W~>?? => ((z(5) (v 6)) ((r 3) y(4)))
```

Returns the environment that the nested `let` expressions establish. Notice that assigning it to the top-level variable W makes it persistent.

```
Q = letrec(
    ( ;; begin locals
        ( X 6 )
        ( self
            lambda( ( )
                theEnvironment()
                ) ; lambda
            ) ; self
        ) ;;; end of locals
    self
    ) ; letrec
=> funobj:0x1e38b8
Q() => envobj:0x1e00e4
theEnvironment( Q ) => envobj:0x1e00e4 ;in debug mode only
```

Returns a function object which, in turn, returns its local environment.

unbindVar

```
unbindVar(
     s_varName
     )
     => t
```

Description

Resets a SKILL or Scheme variable so that its value becomes unbound..

Arguments

s_varName	The name of a variable.

Value Returned

`t`	The variable is not bound anymore.

Example

```
myVar = 42
unbindVar(myVar)
boundp('myVar)
=> nil ; this variable is not bound anymore
```

13

Function and Program Structure

addDefstructClass

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

Description

Creates a class for the `defstruct`.

By default, an instance of a `defstruct` does not have a class. You cannot use `Instance` 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

```
defstruct( card rank suit ) => t

x = _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

Instance

alias

```
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

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

Aliases `path` to the `getSkillPath` function.

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

Aliases `e` to the `edit` function.

apply

```
apply(
     slu_func
     [g_arg ...]
     l_args
     )
     => g_result
```

Description

Applies the given function to the given argument list.

`apply` takes two or more arguments. The first argument must be the name of a function, or a function object, or a list containing a `lambda`/`nlambda`/`macro` expression. The remainder of the arguments are used to construct the list of arguments passed to the function specified by the first argument; the `g_arg` arguments are individual arguments, which are prepended to the `l_args` argument to create a combined list of arguments.

Note: The last argument to apply must always be a list.

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.
g_arg	Optional arguments that are prepended to `l_args` to create a combined list of arguments.
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
apply('plus list(1 2 3 4))    ; adds 1, 2, 3 and 4
=> 10
apply('plus 1 2 list(3 4))    ; adds 1, 2, 3 and 4.
=> 10
```

Reference

,

argc

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

Description

Returns the number of arguments passed to a SKILL script. Used to enhance the SKILL script environment. This function works only for scripting with SKILL standalone executable (skill).

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
```

An example using a SKILL executable:

```
$ skill -V
    @(#)$CDS: skill version 07.02 09/19/2007 09:08 (cat61lnx) $
$ cat /tmp/foo.il
    (printf "argc is %d, argv[0] is %s, argv is %L\n" (argc) (argv 0) (argv))
$ skill /tmp/foo.il -someArg -someArg2
    argc is 2, argv[0] is /tmp/foo.il, argv is ("-someArg" "-someArg2")
```

Reference

argv

argv

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

Description

Returns the arguments passed to a SKILL script. Used to enhance the SKILL script environment. This function works only for scripting with SKILL standalone executable (skill).

Arguments

x_int	Optional argument; it must be a positive integer.

Value Returned

g_result	The return value depends on the arguments passed.

.

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

An example using a SKILL executable:

```
$ skill -V
    @(#)$CDS: skill version 07.02 09/19/2007 09:08 (cat61lnx) $
$ cat /tmp/foo.il
```

```
    (printf "argc is %d, argv[0] is %s, argv is %L\n" (argc) (argv 0) (argv))
$ skill /tmp/foo.il -someArg -someArg2
    argc is 2, argv[0] is /tmp/foo.il, argv is ("-someArg" "-someArg2")
```

Reference

begin

```
SKILL mode
     begin(
     g_exp1
     [ g_exp2 ...
     g_expN ]
     )
     => g_result
     SKILL++ mode
     begin(
          def1
          [ def2 ...
               defN ]
     )
     => g_result
```

Description

In the 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. Whereas, in the 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.

Arguments

g_exp1, g_exp2, g_expN

Arbitrary expressions.

Value Returned

g_result Value of the last expression, *g_expN*.

Example 1

The following example describes the begin function in the SKILL mode.

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

Example 2

The following example describes the begin function in the SKILL++ mode.

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

Reference

progn

clearExitProcs

```
clearExitProcs(
     [ g_tcovItem ]
     )
     => t
```

Description

Removes all registered exit procedures. When the optional argument *g_tcovItem* is set to `t`, it removes all exit procedures except those needed for the `ilTCov` reports.

Arguments

g_tcovItem	Optional argument, which when set to `t` does not clear the `tCov` exit hook.

Value Returned

`t`	Always returns `t`.

Example

```
clearExitProcs( )=> t
```

declareLambda

```
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 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(
     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_name1 ... s_nameN`* as nprocedures (`nlambdas`) to be defined later. This is much like C's "extern" declarations. Because the calling sequence for `nlambdas` is 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_name1	One or more function names.

Value Returned

s_nameN	The last name in the arguments.

Example

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

Reference

declareSQNLambda

```
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

```
declareSQNLambda( step next stepout ) => nil
```

defdynamic

```
defdynamic(
     s_varName
     g_Value
     [ t_docString ]
     )
     => g_value
```

Description

This syntax form sets the dynamic variable `s_varName` to `g_value.` In SKILL, this function works as a `defvar`. In Scheme, `g_value` is evaluated in the current lexical scope.

Arguments

s_varName	Name of the dynamic variable.
g_Value	New value of the dynamic variable.
t_docString	A documentation string (currently ignored).

Value Returned

g_value	Value of the dynamic variable.

Example

```
kx
=> *Error* eval: unbound variable - kx
(inScheme x = 0 (defdynamic kx x "test") kx)
=> *Error* eval: unbound variable - kx
kx
=> 0
```

defglobalfun

```
defglobalfun(
     s_funcName
     ( l_formalArglist )
     g_expr1 ...
     )
     => s_funcName
```

Description

Defines a global function with the name and formal argument list you specify.

Note: The functions that you define using `defglobalfun` are defined within a lexical scope, but are globally accessible.

For `defglobalfun` there must be white space between *`s_funcName`* and the open parenthesis. Expressions within the 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.

Example

Define two global functions, `test_set` and `test_get` using `defglobalfun` and that reference a lexical variable *`secret_val:`*

```
toplevel 'ils
ILS-<2> (let ((secret_val 1))
(defglobalfun test_set (x) secret_val  = x)
(globalProc test_get() secret_val)
)
```

```
ILS-<2> test_get()
=> 1
ILS-<2> test_set(2)
=> 2

ILS-<2> test_get()
=> 2
```

define

```
define(
     s_var
     g_expression
     )
     => s_var

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

Description

(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 or within the body of following syntax forms: `define` (another call to `define`), `lambda`, `let`, `letrec`, `defun`, 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

  ```
  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

let, letrec, letseq

define_syntax

```
define_syntax(
     s_name
     g_expander ...
     )
     => s_name
```

Description

Creates a syntax rule using the `syntax_rule` expander form.

Arguments

s_name	Name of the syntax rule.
g_expander	Expander form, which can be of the form (`syntax_rules` (...) ...).

Value Returned

s_name	Returns a lambda expression, which evaluates to a single argument procedure that performs the specified syntactic transformation.

Example

The following example defines a syntax rule “cut” for parsing input form data.

```
toplevel('ils)
(define_syntax cut_internal
(syntax_rules (X XXX)
((_ (slot_name ...) (proc arg ...))
(lambda (slot_name ...) ((begin proc) arg ...))) ((_ (slot_name ...)
(proc arg ...) XXX) (lambda (slot_name ... @rest rest_slot) (apply
proc arg ... rest_slot))) ((_ (slot_name ...) (position ...) X se ...)
(cut_internal (slot_name ... x) (position ... x) se ...)) ((_
(slot_name ...) (position ...) nse se ...) (cut_internal (slot_name
...) (position ... nse) se ...))))

(define_syntax cut
(syntax_rules ()
((cut slots_or_exprs ...)
(cut_internal () () slots_or_exprs ...))))

((cut times 2 X) 3) => 6
((cut times 2 XXX) 2 3 4) => 48
((cut list 1 X 3 XXX) 2 4 5 6) => (1 2 3 4 5 6)
```

defmacro

```
defmacro(
     s_macroName
     ( l_formalArglist )
     g_expr1 ...
     )
     => 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 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_expr1	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
```

defsetf

```
defsetf(
     s_accessFn
     s_updateFn
     )
     => setf_<s_accessFn>
```

Description

`defsetf` is a macro that allows you to extend generalized variables. It creates a `setf_*()` macro which is used in `setf` to update a value which can be accessed by `s_accessFn()`.

Arguments

s_accessFn	An access function name.
s_updateFn	A function which replaces the value which is accessed by *s_accessFn*.

Value Returned

setf_<s_accessFn>

A macro which is used in `setf`.

Example

```
defun(CAR (x) car(x))
CAR
defmacro(SETCAR (x new)
`(car rplaca(,x ,new))
)
SETCAR
test_ls = '(1 2 3 4 5 6)
(1 2 3 4 5
6
)
expandMacro('defsetf(CAR SETCAR))

defmacro(setf_CAR
(newval \@rest more)
constar('SETCAR
   append(more
    list(newval)
   )
   )
```

```
)
    defsetf(CAR SETCAR)
setf_CAR
    setf(CAR(test_ls) 10)
10
assert(test_ls == '(10 2 3 4 5 6))
nil
```

defun

```
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 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`, `@key`, and `@aux`. See procedure for a detailed description of these argument list parameters.

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
=> 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

let

defUserInitProc

```
defUserInitProc(
     t_contextName
     u_func
     [ autoInit ]
     )
     => ( 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.
u_func	Function to be called when context file is loaded.
[autoInit]	

Value Returned

((*t_contextName s_procName*))

Always returns an association list when set up. The function is not called at this point, but is called when the *t_contextName* context is loaded.

Example

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

Reference

defInitProc, callInitProc

destructuringBind

```
destructuringBind(
     l_lambdaList
     l_expression
     [ g_body ]
     )
     => g_result
```

Description

Enables you to bind variables in a lambda-list to the values of these variables. The list of values is obtained by evaluating the `l_expression`. The `destructuringBind` macro then evaluates the `g_body` form.

Note: `destructuringBind` does not check the correctness of `l_lambdaList`.

Arguments

l_lambdaList	A lambda list.
l_expression	An expression that is evaluated and its result is assigned or bound to the variables in the lambda list.
g_body	A sequence of one or more expressions.

Value Returned

g_result	Result of evaluation.

Example

```
(destructuringBind (a b @optional (c 1)) '(1 2)
        printf("a=%L b=%L c=%L\n" a b c))
=> a=1 b=2 c=1
```

dynamic

```
dynamic(
     s_varName
     )
     => g_value / error
```

Description

This syntax form returns the value of the dynamic variable *s_varName*. If *s_varName* is not a dynamic variable, it returns an error.

Arguments

s_varName	Name of the dynamic variable.

Value Returned

g_value	Value of the dynamic variable *s_varName*.

Example

```
kx
=> *Error* toplevel: undefined variable - kx
(inScheme x = 9 (setf (dynamic kx) x))
9=>
kx
=>9
```

dynamicLet

```
dynamicLet(
      (
      [(s_var1 g_init1)
      (s_var2 g_init2)
      ...]
      )
      => g_result
```

Description

Evaluates the init forms `(`*`g_init1`*`,` *`g_init2`*`, ...)` in the current lexical environment, and then binds the variables `(`*`s_var1`*`,` *`s_var2`*`, ...)` in parallel. The variables are bound as SKILL dynamic variables for the duration of the body forms.

In SKILL, this syntax form is compiled as a `let()`.

Arguments

`s_varName`	Name of a dynamicaly scoped local variable.

Value Returned

`g_result`	The result of the last executed expression in the body.

Example

```
(dynamicLet ((X 21))
   (let ((a 100)
         b)
      (dynamicLet ((X a)
              (Y (progn b=12
                        13)))
(printf "(inSkill X) == %L\n" (inSkill X))
(printf "a == %L\n" a)
       (assert (inSkill X) == a)
       (assert (inSkill X) == 100)
       (assert (inSkill Y) == 13)
       (assert a == 100)
       (assert b == 12)))
     (assert (inSkill X) == 21))
(inSkill X) == 100
a == 100
nil
```

```
(inScheme (dynamicLet ((X 21))
  (let ((a 100)
         b)
        (dynamicLet ((X a)
                (Y (progn b=12
                          13)))
(printf "(inSkill X) == %L\n" (inSkill X))
(printf "a == %L\n" a)
        (assert (inSkill X) == a)
        (assert (inSkill X) == 100)
        (assert (inSkill Y) == 13)
        (assert a == 100)
        (assert b == 12)))
  (assert (inSkill X) == 21))
)
```

```
(inSkill X) == 100
a == 100
nil
```

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))
        then 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

, 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. 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`.

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)..."))
```

When working in the CIW, to ensure that the `errset.errset` variable is not modified internally in the Virtuoso design environment, do not separate `errset` and `errset.errset`. For example, use this construct:

```
errset(sqrt('x)), errset.errset
=> ("sqrt" 0 t nil    ("*Error* sqrt: cannot handle sqrt(x)"))
```

Reference

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	Symbol to determine the language mode to use. The valid values are: ■ `'ils`, which indicates that the given string is evaluated in SKILL++ mode. ■ `'il`, which indicates that the given string is evaluated in SKILL code. This is the default mode.

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

error, evalstring

eval

```
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

```
eval( 'plus( 2 3 ) )    => 5
```

Evaluates the expression `plus(2 3)`.

```
x = 5                       => 5
eval( 'x )                  => 5
```

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

```
eval( list( 'max 2 1 ) ) => 2
```

`Evaluates the expression max(2 1).`

Reference

evalstring,

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

t_string	String containing the SKILL expression to be evaluated.
s_langMode	Symbol to determine the language mode to use. The valid values are: ■ `'ils`, which indicates that the given string is evaluated in SKILL++ mode. ■ `'il`, which indicates that the given string is evaluated in SKILL code. This is the default mode.

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.

expandMacro

```
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

```
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

, defmacro

fboundp

```
fboundp(
     s_functionName
     )
     => g_definition / nil
```

Description

Returns the function binding, if defined, for a specified function name.

The function examines only the current function binding 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

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.
`nil`	No function definition exists for the specified function name.

Example

```
fboundp( 'xyz ) => nil ;assuming there is no function named xyz
fboundp( 'defstruct ) => funobj:0x261108 ;a non-nil result
fboundp( 'cadr ) => lambda:cadr
fboundp( 1 ) => nil
```

flet

```
flet(
     l_bindings
     [g_body]
     )
     => g_result
```

Description

Enables you to define local functions with LET semantics.

The names of functions defined by `flet` retain their local definitions only within the body of `flet`. Also, the function definition bindings are visible only in the body of `flet`. This helps in defining a local version of function which in turn calls the global version of the function with the same name but with different arguments.

Note: `flet` can only be used in Scheme mode.

Arguments

l_bindings	A list of variables or a list of the form *(s_variable g_value)*.
g_body	A sequence of one or more expressions.

Value Returned

g_result	Result of evaluation.

Example

```
(flet ((foo (x) (list x)))(foo 1))

=> (1)
```

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                         ;Define a procedure
sum3(1 2 3)
=> 6
```

getd

```
getd(
     s_functionName
     )
     => g_definition / nil
```

Description

Returns the function binding for a function name.

Note: In Scheme mode, function bindings are treated as regular value bindings. Therefore, `getd()` returns any value bound to the symbol.

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.

getFnWriteProtect

```
getFnWriteProtect(
     s_name
     )
     => t / nil
```

Description

Checks if the given function is write-protected.

The value is `t` if *`s_name`* is write-protected; `nil` otherwise.

Arguments

s_name	Name of the function.

Value Returned

`t`	The function is write protected.
`nil`	The function is not write protected.
	Signals an error if the function is not defined.

Example

```
getFnWriteProtect( 'strlen ) => t
```

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

getVarWriteProtect

```
getVarWriteProtect(
     s_name
     )
     => t / nil
```

Description

(SKILL mode only) 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

,

globalProc

```
globalProc(
     s_funcName(
     l_formalArglist
     )
     g_expr1 ...
     )
     => s_funcName
```

Description

Defines a global function using a formal argument list.

Note: The functions that you define using `globalProc` are defined within a lexical scope, but are globally accessible.

The body of `globalProc` is a list of expressions to be evaluated one after another when *`s_funcName`* is called. There must be no white space between `globalProc` and the open parenthesis that follows, nor between *`s_funcName`* and the open parenthesis of *`l_formalArglist`*. However, for `defglobalfun` there must be white space between *`s_funcName`* and the open parenthesis. This is the only difference between the two functions.

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.

Example

Define two global functions, `test_set` and `test_get` using and `globalProc` that reference a lexical variable *secret_val:*

```
toplevel 'ils
ILS-<2> (let ((secret_val 1))
(defglobalfun test_set (x) secret_val  = x)
(globalProc test_get() secret_val)
)
ILS-<2> test_get()
=> 1
ILS-<2> test_set(2)
=> 2

ILS-<2> test_get()
=> 2
```

isCallable

```
isCallable(
     s_function
     )
     => t / nil
```

Description

Checks if a function is defined or is autoloadable from a context.

Arguments

s_function	Name of a function.

Value Returned

t	The specified function is defined or is autoloadable.
nil	The specified function is not defined or is not autoloadable.

Example

```
isCallable( 'car) => t
procedure( myFunction( x ) x+1)
isCallable('myFunction) => t
```

Reference

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

labels

```
labels(
     l_bindings
     [ g_body ]
     )
     => g_result
```

Description

Enables you to define local functions with LET semantics.

`labels` is similar to the `flet` function except that in `labels`, the scope of name bindings for the functions defined by `labels` encompasses the function body as well as the function definitions themselves.

Note: `labels` can only be used in Scheme mode.

Arguments

l_bindings	A list of variables or a list of the form *(s_variable g_value)*.
g_body	A sequence of one or more expressions.

Value Returned

g_result	Result of evaluation.

Example

```
(labels ((sum (x)
         (if (plusp x)
              x + (sum (sub1 x))
              0)))
         (sum 10))

=> 55
```

lambda

```
lambda(
     ( s_formalArgument )
     g_expr1 ...
     )
     => 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

```
(lambda( (x y) x + y ) 5 6)
=> 11
```

let

```
SKILL mode
     let(
     l_bindings
     g_expr1 ...
     )
     => g_result
     SKILL++ mode
     let(
     [ s_var ]
          (
               ( s_var1 s_initExp1 )
               ( s_var2 s_initExp2 )
          ...
          )
     body
     )
     => g_result
```

Description

In the SKILL mode, this function provides a faster alternative to `prog` for binding local variables only. This is a `syntax` form. In the SKILL++ mode, this function declares a lexical scope. 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.

The SKILL mode argument *`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.

Whereas, the functions, `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 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

l_bindings	(SKILL mode) Local variable bindings, can either be bound to a value or `nil` (the default).
g_expr1	(SKILL mode) Any number of expressions.
s_var	(SKILL++ mode) When the optional *s_var* is given, this becomes a `named let`. A `named let` is just like an ordinary `let` except that *s_var* is bound within the body to a function whose formal arguments are the bound variables and whose body is *body*.
s_var1	(SKILL++ mode) Name of local variable. The variables are bound to fresh locations holding the result of evaluating the corresponding *initExp*.
s_initExp	(SKILL++ mode) Expression evaluated for the initial value. The *initExps* are evaluated in the current environment (in some unspecified order).
body	(SKILL++ mode) 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_result	The result of the last expression evaluated.

Example 1

The following example describes the use of the `let` function in the SKILL mode.

```
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 )                     ; Call function test.
=> "return string"              ; z is returned because 6 == 6.
```

Example 2

The following example describes the use of the `let` function in the SKILL++ mode.

```
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 ) ) )
        foo( 5 )
        ) ; let
    ) ; let
=> 10                                                 ;not 15

let( ((x 2) (y 3))
    define( f(z) x*z+y)
    f(5)
    )
=> 13
```

Reference

letrec, letseq

letrec

```
letrec(
        (
            ( s_var1 s_initExp1 )
            ( s_var2 s_initExp2 )
        ...
        )
    body
    )
    => g_result
```

Description

(SKILL++ mode) 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 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

```
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.

letseq

```
letseq(
        (
            ( s_var1 initExp1 )
            ( s_var2 initExp2 )
            ...
        )
    body
    )
    => g_result
```

Description

A `letseq` expression can be used in both SKILL and SKILL++ `modes`. 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_result	Value of the last expression of *body*.

Example

```
letseq( ( ( x 1 ) ( y x+1 ) )
    y
    ) ; letseq
=> 2
```

The code above is a more convenient equivalent to the code below in which you control the sequence explicitly by the nesting.

```
let( ( ( x 1 ) )
    let( ( ( y x+1 ) )
        y
        )
    )
```

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 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. 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

```
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

nlambda

```
nlambda(
      (
      s_formalArgument
      )
      g_expr1 ...
      )
      => u_result
```

Description

(SKILL mode only) 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 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

nprocedure

```
nprocedure(
     s_funcName(
     s_formalArgument
     )
     g_expr1 ...
     )
     => s_funcName
```

Description

(SKILL mode only) 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 argument list to its single formal argument. `lambda` functions, on the other hand, evaluate each argument in the 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

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 your code appear more like other LISP dialects.

The last argument in *l_formalArglist* can be a string denoting type-checking characters, specified using the argument type template. For more information about specifying the argument type template, see Type Checking in Cadence SKILL Language User Guide.

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`, `@key`, and `@aux`.

@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.

```
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 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.

```
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 *`ycoord`*, those values are 0. Since no value is specified for *`color`*, *`color`*'s default value is `nil`.

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 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
                                     ; 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
=> 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
```

@aux Option

The `@aux` option provides a way to declare auxiliary variables that are local to the function body. After all other parameter specifiers (such as `@rest`, `@optional`, and `@key`) have been evaluated, the symbols following the `@aux` keyword are processed from left to right.

The `@aux` option is supported in SKILL++.

Reference

defun， let

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. 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

prog

```
prog(
     l_localVariables
     [
     [ s_label ]
     g_expr1
     ] ...
     )
     => 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 labeled unless the surrounding statement is itself a `prog`.
g_expr1	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

```
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, 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

, 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

, 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

let, prog1, prog2

putd

```
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

```
putd( 'mySqrt getd( 'sqrt ))
=> lambda:sqrt
```

Assigns the function `mySqrt` the same definition as `sqrt`.

```
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

setf_dynamic

```
setf_dynamic(
     g_value
     s_name
     )
     => g_value
```

Description

Evaluates *g_value* in the current lexical scope and updates the SKILL variable named *s_name*.

Arguments

g_value	New value of dynamic variable *s_name*.
s_name	Name of the dynamic variable.

Value Returned

g_value	New value of the dynamic variable.

Example

```
kx
=>0
(inScheme x = 9 (setf (dynamic kx) x))
=>9
kx
=>9
```

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.

```
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

,

setVarWriteProtect

```
setVarWriteProtect(
     s_name
     )
     => t / nil
```

Description

(SKILL mode only) 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
```

unalias

```
unalias(
     s_aliasName1 ...
     )
     => l_result
```

Description

Undefines the aliases specified in an argument list and returns a list containing the aliases undefined by the call. This is `nlambda` function also works in SKILL++ mode.

Caution

Use alias for interactive command entry only and never in programs.

Arguments

s_aliasName1	Symbol name of the alias.

Value Returned

l_result	List of the aliases removed.

Example

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

Aliases `path` to the `getSkillPath` function.

```
unalias path => (path)
```

Removes `path` as an alias.

unwindProtect

```
unwindProtect(
     [ g_protectedForm ]
     [ g_cleanupForm ]
     )
     => g_result
```

Description

Evaluates the *g_protectedForm* expression and then executes the SKILL *g_cleanupForm* expression. Even if the evaluation of *g_protectedForm* is interrupted or encounters an error, the *g_cleanupForm* expression is still executed. You can therefore use *g_cleanupForm* to close open file handles, reset variables, and restore the state to a known value.

If an error occurs within *g_protectedForm*, the program would normally stop after executing *g_cleanupForm*. To force continued execution despite the error, wrap `unwindProtect` with a suitable function to catch errors, such as `errset`. Even if the error is caught using `errset`, *g_cleanupForm* will still be executed.

To include more than a single expression, group expressions by using functions such as `progn`, `let`, or `prog`.

Arguments

g_protectedForm	Name of the function to be evaluated.
g_cleanupForm	Any valid SKILL expression.

Value Returned

g_result	Result of the expression evaluated.

Examples

Example 1

```
unwindProtect(undefFun() printf("cleanup form called here\n"))
```

The outputs are as follows:

```
*Error* eval: undefined function - undefFun
cleanup form called here
```

Example 2

```
unwindProtect(
    {
        printf("first statement\n")
        1/0
        printf("second statement\n")
    }
    printf("cleanup form called here\n")
)
```

The outputs are as follows:

```
first statement
*Error* quotient: Attempted to divide by zero
cleanup form called here
```

Example 3

```
errset(
    unwindProtect(
        {
            printf("first statement\n")
            1/0
            printf("second statement\n")
        }
        printf("cleanup form called here\n")
    )
)
printf("program continued after error\n")
```

The outputs are as follows:

```
first statement
cleanup form called here
program continued after error
```

warn

```
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

```
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

14

Environment Functions

cdsGetInstPath

```
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 Virtuoso and non-Virtuoso applications.

Note: Starting from version SKILL06.50 and beyond, the following calls are equivalent:

```
cdsGetInstPath("tools/[subDirPath]")
cdsGetToolsPath("[subDirPath]")
```

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

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

Reference

cdsGetToolsPath， getSkillPath

cdsGetToolsPath

```
cdsGetToolsPath(
     [ t_subDirPath ]
     )
     => t_cdsToolsPath
```

Description

Returns the absolute path of the Cadence installation `tools` directory as a string after resolving the `tools` directory appropriately. This function is provided for multiple platform support mainly to simplify access to a common Cadence installation hierarchy for all Unix platforms.

Note: SKILL code from version 6.5 and beyond should use `cdsGetToolsPath("[`*`subDirPath`*`]")` instead of `cdsGetInstPath("tools/[`*`subDirPath`*`]")`.

Arguments

`t_subDirPath`	The optional argument *`t_subDirPath`* is appended to the end of the Cadence installation `tools` directory path with a directory separator if necessary.

Value Returned

`t_cdsToolsPath`	Returns the absolute path of the Cadence installation `tools` directory as a string.

Example

```
cdsGetToolsPath() => "/cds/06.01/latest.il/tools"
cdsGetToolsPath("") => "/cds/06.01/latest.il/tools/"
cdsGetToolsPath("bin") => "/cds/06.01/latest.il/tools/bin"
```

Reference

getSkillPath

cdsPlat

```
cdsPlat()
     => t_plat
```

Description

Returns the platform for the Cadence software that is currently running; one of the following strings: `sun4v`, `sol86`, `hppa`, `ibmrs`, `wint`, `lnx86`, or `lni64`.

Arguments

None.

Value Returned

t_plat	The platform upon which the Cadence software is running. One of the following strings:
	`"sun4v"`
	`"sol86"`
	`"hppa"`
	`"ibmrs"`
	`"wint"`
	`"lnx86"`
	`"lni64"`

Example

```
system("uname")
-> SunOS
   0
cdsPlat()
-> "sun4v"
```

changeWorkingDir

```
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`.

```
changeWorkingDir( "/usr5/design/cpu") => t
changeWorkingDir( "test")
```

Signals an error about a non-existent directory.

Reference

cputime

```
cputime(
      )
      => x_result
```

Description

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

Arguments

None.

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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on 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

createDirHier

```
createDirHier(
     t_pathName
     )
     => t / nil
```

Description

Creates all directories specified in the given SKILL path that do not already exist

The permissions associated with new directories are subject to the file creation mask on systems supporting that concept. If the directory with the specified name already exists, `nil` is returned. The directory names in the given SKILL path can be specified with either absolute or relative; the SKILL path is used in the latter case.

Note: A path that is anchored to the current directory, for example, ./, ../, or ../../.., etc., is not considered as a relative path.

Arguments

t_pathName	Specifies a (hierarchical) SKILL path consisting of all the directories that need to be created

Value Returned

`t`	Returns `t` if all the directories specified in the given SKILL path are created
`nil`	Returns `nil` if a directory with the same name already exists or an incorrect SKILL path is specified 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

```
createDirHier(“./dir1/dir2”); creates the directories /dir1/dir2/ as specified in
the given SKILL path
```

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

deleteDir

```
deleteDir(
     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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

deleteFile

```
deleteFile(
     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. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

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 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 closeMyDataBase 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`.

getCurrentTime

```
getCurrentTime(
    )
    => t_timeString
```

Description

Returns a string representation of the current time.

Arguments

None.

Value Returned

t_timeString	Current time in the form of a string. The format of the string is `month day hour:minute:second year`.

Example

```
getCurrentTime( )=> "Jan 26 18:15:18 1994"
```

This format is also used by the `compareTime` function.

getInstallPath

```
getInstallPath(
    )
    => l_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

l_string	Returns the installation path as a list of a single string.

Example

```
getInstallPath() => ("/usr5/cds/5.0")
```

Reference

getSkillPath

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

getShellEnvVar

```
getShellEnvVar(
     t_UnixShellVariableName
     )
     => t_value / nil
```

Description

Returns the value of a UNIX environment variable, if it has been set. This function expands the environment variable name specified in the argument.

Arguments

`t_UnixShellVariableName`

Name of the UNIX shell environment variable.

Value Returned

t_value	Value of named UNIX environment variable.
nil	No environment variable with the given name has been set.

Example 1

```
getShellEnvVar("SHELL") => "/bin/csh"
```

Returns the current value of the `SHELL` environment variable.

Example 2

```
setShellEnvVar("ITER" "1")  => t
setShellEnvVar("EDITOR_COPY_$ITER" "$EDITOR")  => t
getShellEnvVar("EDITOR_COPY_$ITER")  => "gedit"
unsetShellEnvVar("EDITOR_COPY_$ITER")  => t
getShellEnvVar("EDITOR_COPY_$ITER")  => nil
```

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 *Cadence 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")
```

getTempDir

```
getTempDir(
      )
      => t_TempDir
```

Description

Returns the system temp directory as a string.

Arguments

None.

Value Returned

t_TempDir	The name of your current temp directory.

Example

```
getTempDir() => "/tmp"
```

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`.

Note: Ensure that the logged-in user has execute permissions for the directory.

Arguments

None.

Value Returned

t_currentDir	The name of your current working directory.

Example

```
getWorkingDir() => "~/project/cpu/layout"
```

Reference

isDir

```
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`. A path that is anchored to the current directory, for example, `./`, `../`, or `../../..`, and so on, 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

```
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`.

```
isDir("test")   => nil
```

Result if `test` does not exist.

Reference

getSkillPath

prependInstallPath

```
prependInstallPath(
     S_name
     )
     => t_string
```

Description

Prepends the Cadence DFII 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

, getSkillPath,

setShellEnvVar

```
setShellEnvVar(
     t_varName_or_nameValuePair
     [ t_varValue ]
     )
     => t / nil
```

Description

Sets or updates the value of the UNIX environment variable. This function expands the environment variable name specified in the argument.

Arguments

t_varName or *nameValuePair*	
	Environment variable name or assignment expression (<name>=<value>)
t_varValue	Value of the environment variable

Value Returned

t	If the shell environment variable was set.
nil	If the shell environment variable was not set.

Example 1

```
setShellEnvVar("PWD=/tmp")    => t
```

Sets the parent working directory to the `/tmp` directory.

```
getShellEnvVar("PWD")         => "/tmp"
```

Gets the parent working directory.

Example 2

```
setShellEnvVar("TEST=/tmp")    => t
```

Sets the Test directory to the `/tmp` directory.

```
setShellEnvVar("TEST" "/home")    => t
```

Sets the Test directory to the home directory.

```
setShellEnvVar("TEST")  => nil
WARNING* setShellEnvVar: must have an equal sign to set a value - "TEST"
```

Returns nil, as an equal to sign is required to set the value.

```
setShellEnvVar("=/tmp")  => nil
*WARNING* setShellEnvVar: the argument should include a variable name - "=/tmp"
```

Returns nil, as the argument does not have a variable name.

```
setShellEnvVar("TEST = /tmp")  => nil
*WARNING* setShellEnvVar: must not have a space before the equal sign - "TEST = /tmp
```

Returns nil, as the argument has a space before the equal to sign.

Example 3

```
setShellEnvVar("ITER" "1")  => t
setShellEnvVar("EDITOR_COPY_$ITER" "$EDITOR")  => t
getShellEnvVar("EDITOR_COPY_$ITER")  => "gedit"
```

Reference

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. Also:

- 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 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,

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

setShellEnvVar

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

The output of the `system()` command is redirected to a UNIX terminal window

```
system( "date" )
Tue Aug 22 16:24:33 IST 2017
0
system( "daa" )
sh: daa: not found
1
```

Reference

sh, shell

unsetShellEnvVar

```
unsetShellEnvVar(
     t_envVarName
     )
     => t / nil
```

Description

Removes an environment variable from the environment of the calling process. This function expands the environment variable name specified in the argument. If the environment variable (`t_envVarName`) does not exist in the current environment, the environment is left unchanged.

Arguments

t_envVarName	A string representing the environment variable name.

Value Returned

t	The environment variable is successfully removed.
`nil`	The environment variable does not exist or there is an error condition.

Example 1

```
setShellEnvVar("test=testValue")
=> t
getShellEnvVar("test")
=> "testValue"
unsetShellEnvVar("test")
=> t
getShellEnvVar("test")
=> nil
```

Example 2

```
setShellEnvVar("ITER" "1")  => t
setShellEnvVar("EDITOR_COPY_$ITER" "$EDITOR")  => t
getShellEnvVar("EDITOR_COPY_$ITER")  => "gedit"
unsetShellEnvVar("EDITOR_COPY_$ITER")  => t
getShellEnvVar("EDITOR_COPY_$ITER")  => nil
```

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

15

Namespace Functions

makeNamespace

```
makeNamespace(
     t_name
     )
     => o_namespace / nil
```

Description

Creates a SKILL namespace with the given *t_name*. A namespace or its parts can be saved in a context and loaded with the context.

Arguments

t_name	Name for the namespace.

Value Returned

o_namespace	Returns the namespace object when successfully created.
nil	Returns nil if the namespace is not created or a namespace with the same name already exists.

Example

```
makeNamespace("METHODS")
=> ns@METHODS
```

findNamespace

```
findNamespace(
     t_name
     )
     => o_namespace / nil
```

Description

Returns the namespace object with the given name.

Arguments

t_name	Specify the name for which you want retrieve the namespace object.

Value Returned

o_namespace	Returns the namespace object.
nil	Returns nil if no namespace object exists with the given name.

Example

```
findNamespace("A")
=> ns@A
```

useNamespace

```
useNamespace(
     t_namespace
     )
     => t / nil
```

Description

Sets the given namespace for use and imports its symbols into the current namespace.

Arguments

t_namespace	Specify the name of the namespace that you want to use.

Value Returned

t	Returns t when the given namespace is successfully set for use.
nil	Returns nil if the given namespace is not set.

Example

```
useNamespace("METHODS")
 => t
```

unuseNamespace

```
unuseNamespace(
     t_namespace
     )
     => t / nil
```

Description

Unsets the given namespace.

Arguments

t_namespace	Specify the name of the namespace that you want to unset.

Value Returned

`t`	Returns t when the given namespace is successfully unset for use.
`nil`	Returns nil if the given namespace cannot be unset.

Example

```
unuseNamespace("METHODS")
 => t
```

importSymbol

```
importSymbol(
     l_symbolList
     [t_namespace]
     )
     => t / nil
```

Description

Imports symbols into the given namespace. By default, this function imports into the `IL` (or default) namespace.

Arguments

l_symbolList	Specify a list of symbols that you want to import into the default namespace
t_namespace	(Optional). Specifies the name of the namespace into which you want to import the given symbols.

Value Returned

t	Returns t if the symbols are successfully imported into the namespace (given or default).

Example

```
importSymbol('(A::level A::value))

 => t
```

findSymbol

```
findSymbol(
     t_name
     [ ?namespace t_namespace ]
     )
     => s_symbolName / nil
```

Description

Searches for a symbol that is specified as a string in the given namespace and returns its corresponding SKILL symbol.

Arguments

t_name	A string value to specify the name of the symbol you want to search for.
?namespace *t_namespace*	(Optional) The namespace in which you want to search for the symbol.

Value Returned

s_symbolName	Returns the name of the symbol.
nil	Returns `nil` if no such symbol exists in the namespace.

Example

```
> (Namespace "my")
ns@my
> 'my:::aaa
my:::aaa
> (findSymbol "aaa" ?namespace "my")
my:::aaa
> (findSymbol "bbb" ?namespace "my")
nil
```

addToExportList

```
addToExportList(
     l_symbols
     )
     => t
```

Description

Adds the specified symbols to the namespace export list. This function does not throw any errors if a symbol is already exported.

Note: You can export any symbol from your namespace.

Arguments

l_symbols	Specify the symbols that you want to add to the namespace export list.

Value Returned

`t`	Returns t when the specified symbols are successfully added to the namespace export list.

Example

```
> (addToExportList '(newNameSpace:::aaa newNameSpace:::bbb))
t
> (useNamespace "newNameSpace")
t
> (getSymbolNamespace 'aaa)
ns@newNameSpace
```

getSymbolNamespace

```
getSymbolNamespace(
     s_name
     )
     => o_namespace
```

Description

Returns the namespace where the symbol was created.

Arguments

s_name	Specifies the name of the symbol for which you want to retrieve the namespace where the symbol was created

Values Returned

o_namespace	Returns the namespace where the specified symbol was created.

Example

```
getSymbolNamespace('car)
 => ns@IL
```

removeFromExportList

```
removeFromExportList(
     l_symbolList
     )
     => t
```

Description

Removes symbols referenced in *l_symbolList* from the export list of its namespace. This function will not throw an error, if some of the symbols are not exported. If a symbol from *l_symbolList* was imported by `useNamespace` it will not removed by `unuseNamespace`.

Arguments

l_symbolList	Specifies the symbols that you want to remove from the export list of your namespace.

Value Returned

t	Returns t when the referenced symbols are successfully removed.

Example

```
> (removeFromExportList '(jane::aaa))
t
> (useNamespace "jane")
t
> (getSymbolNamespace 'aaa)
nil
```

addToNamespace

```
addToNamespace(
     t_namespaceName
     l_symbolList
     )
     => t
```

Description

Adds and imports the given list of symbol names to the export list of the namespace *t_namespaceName*.

Arguments

t_namespaceName	Specify the name of the namespace to which you want to add the given list of symbols.
l_symbolList	Specifies the symbols that you want to add to the export list of the specified namespace.

Value Returned

`t`	Returns t when the list of symbols are successfully added.

Example

```
> (addToNamespace "A" '("a" "b" "c"))
t
> (getSymbolNamespace 'a)
ns@A
```

shadow

```
shadow(
     l_symbols
     [t_namespace]
     )
     => t
```

Description

Adds symbols *s_symbol* to the shadow list of the default namespace. The symbols which are added to the shadow list are not overridden by import.

Arguments

l_symbols	Specify a list of symbols to be protected in the default namespace.
t_namespace	(Optional) Specify the namespace in which these symbols should be protected. The default value is the "`IL`" namespace.

Value Returned

`t`	Returns `t` to indicate that the symbol was added to the shadow list of the current namespace.

Example

```
aaddToExportList('(p1:::x p1:::y p1:::z))
=> t
addToExportList('(p2:::x p2:::y p2:::z))
=> t
useNamespace("p1")
=> t
useNamespace("p2")
*error* useNamespace symbol name conflict - p2::x p2::y p2::z
unuseNamespace("p1")
shadow(shadow('x  y  z))
=> t
useNamespace("p2")
=> t
```

shadowImport

```
shadowImport(
     l_symbols
     [t_namespace]
     )
     => t
```

Description

Adds symbols to the namespace shadow list.

Note: By default, all warnings related to namespaces are suppressed in the `shadowImport` function.

Arguments

l_symbols	Specify the list of symbols that you want to add to the shadow list.
t_namespace	(Optional) Specify the namespace of the shadow list to which you want to add the symbols. If you do not provide a namespace, the symbols are added to the shadow list of the default namespace, `IL`.

Value Returned

`t`	Returns t when the symbols are successfully added to the namespace shadow list.

Example

```
shadowImport('(methods::drawPolygon))

 => t
```

removeShadowImport

```
shadowImport(
     l_symbols
     [t_namespace]
     )
     => t
```

Description

Removes the specified symbols from the namespace shadow list.

Arguments

l_symbols	Specify the list of symbols that you want to remove from the shadow list.
t_namespace	(Optional) Specify the namespace of the shadow list from which you want to remove the symbols. If you do not provide a namespace, the symbols are removed from the shadow list of the default namespace, `IL`.

Value Returned

`t`	Returns t when the symbols are successfully removed from the namespace shadow list.

Example

```
removeShadowImport('drawPolygon)
 => t
```

unimportSymbol

```
unimportSymbol(
     l_symbolList
     [ t_namespace ]
     )
     => t
```

Description

Unimports symbols from the given namespace. By default, this function unimports from the `IL` (or default) namespace.

Arguments

l_symbolList	Specify a list of symbols that you want to unimport from the default namespace.
t_namespace	(Optional). Specifies the name of the namespace from which you want to unimport the given symbols.

Values Returned

`t`	Returns t, if successful

Example

```
unimportSymbol('(A::level A::value))
 => t
```

16

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 `-vector` becomes `Vector`. (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 666
- Expressions on page 667
- Functions on page 668

Lexical Structure

Scheme/SKILL++ Equivalents Table – Lexical Structure

Scheme	SKILL++	Comment
Boolean literals `#t`, `#f`	Supported.	Use `t` for `#t`, `nil` for `#f`.
Character literals `#\...`	Unsupported.	Character type not supported.
Simple numeric literals such as integers & floats	Supported.	Use 0..., 0x..., and 0b... for #o..., #x..., and #b... (octal/hex/binary integers).
String literals "..."	Same.	
Vector literals #(...)	Same.	
case-insensitive symbols	Unsupported.	Symbols in SKILL++ are always case-sensitive.
`nil` as a symbol	Unsupported.	In SKILL++, just as in SKILL, `nil` is not a symbol.
Special symbol constituent characters such as !, $, %, &, *, /, <, =, and so forth.	Unsupported.	Some of these are used for (infix) operators in SKILL++, others are illegal characters. `?` is used for keyword prefix.
' (single quote)	Same.	Shorthand for `quote`.
‘ (back quote)	Same.	Shorthand for `quasiquote` in Scheme and for `_backquote` in SKILL++.
, (comma)	Same.	Shorthand for `unquote` in Scheme and for `_comma` in SKILL++.
,@	Same.	Shorthand for `unquote-splicing` in Scheme and for `_commaAt` in SKILL++.

Expressions

Scheme/SKILL++ Equivalents Table – Expressions

Scheme	SKILL++	Comment
(improper lists), such as (d ... . d)	Unsupported.	SKILL++ lists must end with `nil`.
(procedure calls), such as (f e ...)	Same.	Can be written as `f(e ...)` in SKILL++ if `f` is a symbol (variable).
(and e ...)	Same.	
(begin e ...)	Same.	Equivalent to `progn` in SKILL++.
(case ((d ...) e ...) ... [(else e ...)])	Same.	
(cond (e ...) ... [(else e ...)])	Same.	
(define x e)	Same.	One can also use SKILL's `procedure` syntax to define functions in SKILL++.
(define (x v ...) body)		
(do ((v e [e]) ...) (e ...) e ...)	Same.	
(if e1 e2 e3)	Same.	SKILL++ allows extended `if` syntax (with `then` and `else` keywords) as in SKILL.
(lambda (x ...) body)	Same.	Improper variable list such as `(x ... . y)` can't be used as formals in SKILL++. Use SKILL style `@rest`, `@optional` instead.
(let [x] ((v e) ...) body)	Same.	
(let* ((v e) ...) body)	Supported.	Use `letseq` instead of `let*` in SKILL++.
(letrec ((v e) ...) body)	Same.	
(or e ...)	Same.	
(set! x e)	Supported.	Use `setq` or the infix `=` operator.

Functions

Scheme/SKILL++ Equivalents Table – Functions

Scheme	SKILL++	Comment
+, -, *, /	Infix only.	Equivalent to functions `plus`, `difference`, `times`, and `quotient` in SKILL++.
<, <=, >, >=	Infix only.	Equivalent to functions `lessp`, `leqp`, `greaterp`, and `geqp` in SKILL++.
=	Supported.	Used as the infix assignment operator in SKILL++. For equality, use the infix operator `==` or function `equal`.
abs	Same.	
acos	Same.	
angle	Unsupported.	
append	Same.	Takes two arguments only.
apply	Same.	
asin	Same.	
assoc	Same.	
assq	Same.	
assv	Same.	
atan	Same.	In SKILL++, `atan` takes one argument only; `atan2` takes two arguments.
boolean?	Supported.	Use `booleanp`.
car, cdr, caar, ..., cddddr	Same.	
call-with-current-continuation	Unsupported.	
call-with-input-file	Unsupported.	
call-with-output-file	Unsupported.	
ceiling	Same.	

Scheme/SKILL++ Equivalents Table – Functions

Scheme	SKILL++	Comment
char->integer	Unsupported.	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.
char-alphabetic?	Unsupported.	Character type not supported.
char-ci<=?	Unsupported.	Character type not supported.
char-ci<?	Unsupported.	Character type not supported.
char-ci=?	Unsupported.	Character type not supported.
char-ci>=?	Unsupported.	Character type not supported.
char-ci>?	Unsupported.	Character type not supported.
char-downcase	Unsupported.	Character type not supported.
char-lower-case?	Unsupported.	Character type not supported.
char-numeric?	Unsupported.	Character type not supported.
char-upcase	Unsupported.	Character type not supported.
char-upper-case?	Unsupported.	Character type not supported.
char-whitespace?	Unsupported.	Character type not supported.
char<=?	Unsupported.	Character type not supported.
char<?	Unsupported.	Character type not supported.
char=?	Unsupported.	Character type not supported.
char>=?	Unsupported.	Character type not supported.
char>?	Unsupported.	Character type not supported.
char?	Unsupported.	Character type not supported.
close-input-port	Supported.	Use `close`.
close-output-port	Supported.	Use `close`.
complex?	Unsupported.	
cons	Same.	The second argument must be a list.
cos	Same.	

Scheme/SKILL++ Equivalents Table – Functions

Scheme	SKILL++	Comment
current-input-port	Supported.	Use the `piport` global variable.
current-output-port	Supported.	Use the `poport` global variable.
denominator	Unsupported.	
display	Same.	
eof-object?	Unsupported.	SKILL++ reader returns `nil` on EOF.
eq?	Supported.	Use `eq`.
equal?	Supported.	Use `equal`.
eqv?	Supported.	Use `eqv`.
even?	Supported.	Use `evenp`.
exact->inexact	Unsupported.	
exact?	Unsupported.	
exp	Same.	
expt	Same.	
floor	Same.	Use `fix` or `floor`.
for-each	Supported.	Use `mapc`.
gcd	Unsupported.	
imag-part	Unsupported.	
inexact->exact	Unsupported.	
inexact?	Unsupported.	
input-port?	Supported.	Use `inportp`.
integer->char	Unsupported.	Character type not supported. Use `intToChar` for the same effect on symbols.
integer?	Supported.	Use `fixp` or `integerp`.
lcm	Unsupported.	
length	Same.	Works for both lists and vectors.
list	Same.	

Scheme/SKILL++ Equivalents Table – Functions

Scheme	SKILL++	Comment
list->vector	Supported.	Use `listToVector`.
list-ref	Supported.	Use `nth`.
list?	Supported.	Use `listp`.
log	Same.	
magnitude	Unsupported.	
-polar	Unsupported.	
-rectangular	Unsupported.	
-string	Unsupported.	
-vector	Supported.	Use `Vector`.
map	Supported.	Use `mapcar` instead. `map` in SKILL++ behaves differently from `map` in standard Scheme.
max	Same.	
member	Same.	
memq	Same.	
memv	Same.	
min	Same.	
modulo	Same.	`modulo` differs from `mod` in SKILL++, which is the same as `remainder`.
negative?	Supported.	Use `minusp` or `negativep`.
newline	Same.	
not	Same.	New for SKILL++. Same as ! operator.
null?	Supported.	Use `null`.
number->string	Supported.	Use `sprintf`.
number?	Supported.	Use `numberp`.
numerator	Unsupported.	
odd?	Supported.	Use `oddp`.
open-input-file	Supported.	Use `infile`.

Scheme/SKILL++ Equivalents Table – Functions

Scheme	SKILL++	Comment
open-output-file	Supported.	Use `outfile`.
output-port?	Supported.	Use `outportp`.
pair?	Supported.	Use `dtpr` or `pairp`.
peek-char	Unsupported.	
positive?	Supported.	Use `plusp`.
procedure?	Supported.	Use `procedurep`.
quotient	Same.	
rational?	Unsupported.	
rationalize	Unsupported.	
read	Supported.	Or use `lineread`. Returns `nil` on EOF.
read-char	Unsupported.	Character type not supported. Use `getc` for similar effect.
real-part	Unsupported.	
real?	Supported.	Use `floatp` or `realp`.
remainder	Same.	Use `mod` or `remainder`.
reverse	Same.	
round	Same.	
set-car!	Supported.	Use `rplaca` or `setcar`.
set-cdr!	Supported.	Use `rplacd` or `setcdr`.
sin	Same.	
sqrt	Same.	
string	Unsupported.	
string->number	Supported.	Use `readstring`.
string->symbol	Supported.	Use `concat` or `stringToSymbol`.
string-append	Supported.	Use `strcat`.
string-ci<=?	Unsupported.	

Scheme/SKILL++ Equivalents Table – Functions

Scheme	SKILL++	Comment
string-ci<?	Unsupported.	
string-ci>?	Unsupported.	
string-length	Supported.	Use `strlen`.
string-ref	Unsupported.	Use `getchar` for similar effect.
string-set!	Unsupported.	Strings in SKILL++ are immutable.
string<?	Supported.	Use `alphalessp` or `strcmp`.
string=?	Supported.	Use `alphalessp` or `strcmp`.
string>=?	Supported.	Use `alphalessp` or `strcmp`.
string>?	Supported.	Use `alphalessp` or `strcmp`.
string?	Supported.	Use `stringp`.
substring	Supported.	Argument values differ. SKILL++ uses `index` and `length`. Scheme standard uses `start` and `end` (`index`).
symbol->string	Supported.	Use `get_pname` or `symbolToString`.
symbol?	Supported.	Use `symbolp`.
tan	Same.	
truncate	Same.	
vector	Same.	
vector-length	Supported.	Use `length`.
vector->list	Supported.	Use `vectorToList`.
vector-ref	Supported.	Use `arrayref` or the `a[i]` syntax.
vector-set!	Supported.	Use `setarray` or the `a[i] = v` syntax.
vector?	Supported.	Use `arrayp` or `vectorp`.
write	Same.	
write-char	Unsupported.	
zero?	Supported.	Use `zerop`.

17

Mapping Symbols to Values

There are many objects in SKILL which map symbols to values. The `get` function tries to work for all of them. Of course this over-intelligence causes confusion in cases such as hash tables.

Reader-Writer Correspondence

The functions come in reader/writer pairs as given below. The functions (`get`, `getq`, `getqq`) in the left-hand column, read from a given object. The functions (`putprop`, `putpropq`, `putpropqq`) in the right-hand column, write to (or modify) a given object.

```
get   <--> putprop
getq  <--> putpropq
getqq <--> putpropqq
```

Using the Infix Operator

The `get` and `putprop` functions have no corresponding infix operators. The infix operators for the other four functions are as given in the following table:

			Examples	
Function	Infix Operator	LHS or RHS	Function Call	Infix Operator
`getq`	`->`	LHS	`getq(obj prop)`	`obj->prop`
`putpropq`	`-> =`	RHS	`putpropq(obj value prop)`	`obj->prop = value`
`getqq`	`.`	LHS	`getqq(obj prop)`	`obj.prop`
`putpropqq`	`.=`	RHS	`putpropqq(obj value prop)`	`obj.prop = value`

Evaluating Arguments

The functions differ about which of their arguments are taken as literals or are evaluated. The following table describes the arguments that are evaluated for each of the four functions:

Function Call Using Infix Operator	Function Call Using the Syntax	Arguments Evaluated
–	get(obj prop)	obj, prop
–	putprop(obj value prop)	obj, value, prop
obj->prop	getq(obj prop)	obj
obj->prop=value	putpropq(obj value prop)	obj, value
obj.prop	getqq(obj prop)	
obj.prop=value	putpropqq(obj value prop)	value

The following are equivalent:

```
getq(obj prop)  <--> get(obj 'prop)
getqq(obj prop) <--> get('obj 'prop) <--> getq('obj prop)

putpropq(obj value prop)  <--> putprop(obj value 'prop)
putpropqq(obj value prop) <--> putprop('obj value 'prop) <-->
putpropq('obj value prop)
```

Except for the quoting semantics, all the functions behave the same. They retrieve the value associated with a symbol in a specified object. If a string is given rather than a symbol as the property name, the effect is as if the function were called with the symbol that has the print-name.

The following are equivalent:

```
get(obj 'prop)  <--> get(obj "prop")
getq(obj prop)  <--> getq(obj "prop")
getqq(obj prop) <--> getqq(obj "prop")
putprop(obj value 'prop)  <--> putprop(obj value "prop")
putpropq(obj value prop)  <--> putpropq(obj value "prop")
putpropqq(obj value prop) <--> putpropqq(obj value "prop")
```

Working with Lists

If the given object is a list, `get`, `getq`, `putprop`, and `putpropq` assume it is a DPL and consequently read or modify the named field of the DPL.

Working with Symbols

If the given object is a symbol, `get`, `getq`, `putprop`, and `putpropq` read or modify the symbol's property list.

Working with Hash Tables

For the cases of hash tables (returned by `Instance`) the functions arrayref and setarray can be used instead. There are also `[]` and `[]=` infix operators which obey the following equivalence:

```
hash->prop
hash['prop]
getq(hash prop)
get(hash 'prop)
get(hash "prop")
getq(hash "prop")
arrayref(hash 'prop)
```

And, the following are equivalent:

```
hash->prop = value
hash['prop] = value
putpropq(hash value prop)
putpropq(hash value "prop")
putprop(hash value 'prop)
putprop(hash value "prop")
setarray(hash 'prop value)
```

Working with SKILL++

For the cases of SKILL++ instances of `standardObject` (returned by `Table`), the functions `slotValue` and `setSlotValue` can be used in accordance to the equivalence sets specified below.

The following are equivalent:

```
self->slot
getq(self slot)
get(self 'slot)
slotValue(self 'slot)
```

And, the following are equivalent:

```
self->slot = value
putprop(self value 'slot)
putprop(self value "slot")
putpropq(self value slot)
putpropq(self value "slot")
setSlotValue(self 'slot value)
```

In addition to the uses described above, applications that embed SKILL (such as Virtuoso and AllegroPCB) extend the capabilities of the `get` and `putprop` family of functions to work intuitively on their data structures, these include hi forms, menus and widgets, dbobjects, CDF objects, waveform objects, and many other types of objects.

18

setf Helper Functions

setf_*<helper>* Functions

The following table lists all the `setf_<helper>` functions. For more information about these functions, refer to the setf_*<helper>* function. For more information about the `setf` function, see setf.

Function Name	Description
`setf_arrayref`	An expander function for setf, which returns the result of the corresponding setf operation. In the function, replace *helper* with `arrayref`.
`setf_caaar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *helper* with `caar`.
`setf_caadr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *helper* with `caadr`.
`setf_caar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *helper* with `caar`.
`setf_cadar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *helper* with `cadar`.
`setf_caddr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *helper* with `caddr`.
`setf_cadr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *helper* with `cadr`.

`setf_car`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `car`.
`set_cdaar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cdaar`.
`setf_cdadr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cdadr`.
`setf_cdar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cdar`.
`setf_cddar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cddar`.
`setf_cdddr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cdddr`.
`setf_cddr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cddr`.
`setf_cdr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `cdr`.
`setf_get`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `get`.
`setf_getSG`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `getSG`.
`setf_getSGq`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `getSGq`.
`setf_getShellEnvVar`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `getShellEnvVar`.

`setf_getd`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `getd`.
`setf_getq`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `getq`.
`setf_getqq`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `getqq`. For example, `(setf mysymbol.myprop 42)` sets `mysymbol.myprop` to value 42.
`setf_last`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `last`.
`setf_leftEdge`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `leftEdge`.
`setf_lowerLeft`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `lowerLeft`.

`setf_nth`	An expander function for `setf`, which returns the result of the corresponding `setf` operation to support `setf(nth(...) ...)` expressions. In the function, replace *`helper`* with `nth`.
	For example:
	`myList = '(1 2 3 4); A user-defined list`
	`setf(nth(2  myList) 0); Set the 2nd element (zero-based) of myList`
	`myList is now modified:`
	`(1 2 0 4)`
	`setf(nthelem(1 myList) 6); set the 1st element of myList (assuming one-based index)`
	`myList is now modified:`
	`(6 2 0 4)`

`setf_nthcdr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `nthcdr`.
`setf_nthcdr`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `nthcdr`.

`setf_nthelem`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `nthelem`.
`setf_rightEdge`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `rightEdge`.
`setf_slotValue`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `slotValue`.
`setf_topEdge`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `topEdge`.
`setf_bottomEdge`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `bottomEdge`.
`setf_upperRight`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `upperRight`.
`setf_xCoord`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `xCoord`.
`setf_yCoord`	An expander function for `setf`, which returns the result of the corresponding `setf` operation. In the function, replace *`helper`* with `yCoord`.

Note: In addition to the above helper functions, you can create your own `setf` helper functions.

19

Type Introspection Functions

Type introspection is the ability of a function to determine the type or property of an object at runtime. SKILL provides the following type introspection functions:

Function Name	Description
dtpr	Checks if an object is a non-empty list.
listp	Checks if an object is a list.
pairp	Checks if an object is a `cons` object, that is, a non-empty list.
arrayp	Checks if an object is an array.
defstructp	Checks if an object is an instance of a particular `defstruct`.
tablep	Checks if an object is an association table.
type, typep	Returns a symbol whose name denotes the type of a SKILL object. The functions `type` and `typep` are identical.
vectorp	Checks if an object is a vector. Behaves the same as `arrayp`.
integerp	Checks if an object is an integer. This function is the same as `fixp`.
otherp	Checks if an object is a user type object, such as an association table or a window.
symbolp	Checks if an object is a symbol.
symstrp	Checks if an object is either a symbol or a string.
outstringp	Checks whether the specified value is an outstring port.
pcreObjectp	Checks to see whether the given argument is a `pcreObject` or not.
stringp	Checks if an object is a string.
evenp	Checks if a number is an even integer.
oddp	Checks if an object is an odd integer.

Function Name	Description
floatp	Checks if an object is a floating-point number. Same as `realp`.
fixp	Checks if an object is an integer, that is, a fixed number.
minusp	Checks if a value is a negative number. Same as `negativep`.
plusp	Checks if the given object is equal to one.
onep	Checks if the given object is equal to one.
realp	Checks if a value is a real number. Same as `floatp`.
zerop	Checks if an object is equal to zero.
numberp	Checks if a data object is a number, that is, either an integer or floating-point number.
inportp	Checks if an object is an input port.
outportp	Checks if an object is an output port.
openportp	Checks if the given argument is a port object and it is open (for input or output), `nil` otherwise.
portp	Checks if an object is an input or output port.
bcdp	Checks if an object is a binary primitive function.
booleanp	Checks if an object is a boolean.
getFunType	Returns a symbol denoting the function type for a given function object.
isMacro	Checks if the given symbol denotes a macro.
isCallable	Checks if a function is defined or is autoloadable from a context.
boundp	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.
fboundp	Checks if the given name has a function binding.
getFnWriteProtect	Checks if the given function is write-protected.
getVarWriteProtect	Checks if a variable is write-protected.
isVarImported	Checks if the specified variable was imported into SKILL++ or not.

Function Name	Description
fdoc	Returns the documentation string for the function bound to the symbol *s_function*. SKILL switch `saveInlineDoc` must be set to save and retrieve the doc string.
procedurep	Checks if an object is a procedure, or function, object.

20

The Standalone skill Program

The standalone `skill` application offers an interactive environment for users to execute SKILL functions.

Note: This application supports only the SKILL functions documented in the current reference. It does not include Cadence Virtuoso application components or support related application-specific SKILL functions.

Syntax

The syntax for running `skill` is as follows:

```
skill [<options>] [<IL file(s)…>]
```

It is described as follows:

options	One or more of these options can be used, separated by spaces. ■ `-c`: read SKILL functions from a string. ■ `-e`: abort when a file with an error is encountered. ■ `-f`: ignore the `.ilinit` file. ■ `-i`: switch to interactive mode, instead of exiting, after a specified IL file is loaded.
IL file(s)	When `skill` is invoked with one or more IL files, the files are loaded in the order in which they are specified and the application exits after loading the last file. When `skill` is invoked without any IL file, an interactive prompt is displayed at which the use can enter SKILL functions.

When no options or IL files are specified, the skill program is started in interactive mode where an input prompt is displayed for the user to type in commands or operations.

Examples

Runs `skill` in interactive mode from the shell prompt. If the user types in `4*10`, the value returned is `40`. The `exit` command closes the program and returns the user to the shell prompt:

```
sh> skill
> 4*10
40
> exit
sh>
```

Runs `skill` by reading SKILL functions from a string:

```
skill -c "144 / 12"
=> 12
```

Runs `skill` in interactive mode after the specified IL file is loaded:

```
skill -i new.il
```

Runs `skill` in interactive mode with a prompt at which the use can enter SKILL functions:

```
skill -I new.il
```

Runs `skill` by reading SKILL functions from the specified IL files in the order in which they are specified:

```
skill new.il new1.il
```

Using skill in a Script

The `skill` application can also be used as a script similar to sh or Perl.

The first line of the script must have a command to invoke the 'skill' application. Arguments used in the script are treated as strings, as in the case of other scripting languages. The script must include the `exit()` call to terminate the script. If it is not included, `skill` treats the arguments as IL files and will attempt to load them.

A sample script follows:

```
#! /cdsHier/tools/dfII/bin/skill
printf("Hello world\n")
when( argc() == 2
    printf("Arguments: %s %s\n",argv(1) argv(2))
    printf("Types: %s %s\n",type(argv(1)),type(argv(2)))
)
exit(0)
```

Before running the script, ensure that the script file has permissions set as follows:

```
chmod + x
```