cadence®

Cadence SKILL Development Reference

Product Version ICADVM20.1
October 2020

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

8
SKILL IDE Functions 165

Preface

This manual provides information on using the SKILL Development tools, which include the Lint, Profiler, Finder, Code Browser, Surveyor, and Debugger. These tools, especially the Finder, provides quick reference information for syntax and abstract statements for SKILL language functions and application programming interfaces (APIs). It also provides information on the basic SKILL language functions.

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

When you use the following SKILL development APIs, Virtuoso also checks out the **Cadence SKILL Development Environment** license (License Number 900): `getLoadLine`, `getLoadByte`, `getLoadFile`, `trace`, `step`, `next`, `stepout`, `stepend`, `meter`, `breakpt`, `tracev`, `tracep`, `watch`, `unwatch`, `profile`, `echo`, `saveContext`, `stacktrace`, `where`, and `gcsummary`.

For information on licensing, see *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 Language 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`	nmpIlUserType	nmpIl 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

Overview

Cadence SKILL Development Tools

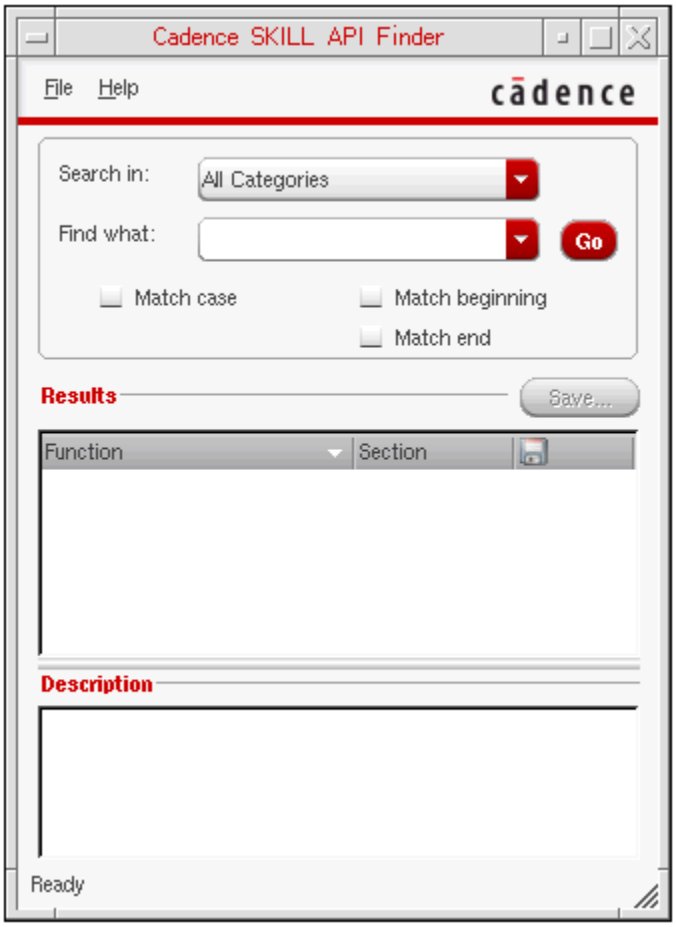
Information about the SKILL development tools is available in *Cadence SKILL IDE User Guide*.

The Walkthrough topic in this help system identifies and explains the tasks you perform when you develop SKILL programs using the SKILL development tools. Using a demonstration program, it explains the various tools available to help you measure the performance of your code and also look for possible errors and inefficiencies in your code. It includes a section on working in the non-graphical environment.

For a list of SKILL lint messages, and message groups, refer to the *Cadence SKILL IDE User Guide*.

Quick Reference Tool: Finder

Quick reference information for syntax and abstract statements for SKILL language functions and application procedural interfaces (APIs) is available using the Finder, which is accessible from the SKILL IDE window, CIW or from the UNIX command line.



For more information, see the *Cadence SKILL IDE User Guide*.

Copying and Pasting Code Examples

You can copy examples from CDSDoc windows and paste the code directly into the CIW or use the code in non-graphics SKILL mode.

To select text

- Press `Control-drag left mouse` to select a text segment of any size.

- Press `Control-double click left mouse` to select a word.
- Press `Control-triple click left mouse` to select an entire section.

2

Profiler Functions

profile

```
profile(
     s_profileField
     )
     => t
```

Description

Turns on global SKILL profiling for measuring time or memory.

The profiler is interrupt-driven. It walks the SKILL stack and records functions being executed. When `unprofile` or `profileSummary` are called, profiling is stopped. `profileSummary` prints a report of the time spent or memory allocated in the functions executed. Profiling time is cumulative, so you must call `profileReset` to reset the profiled data.

Time measurements are done with UNIX system functions that have coarse granularity at 1/60 of a second, so functions must be executed many times for the CPU times to be reasonably accurate.

SKILL Profiler is not yet supported on the Windows/Wintel platform.

Arguments

s_profileField	Can be one of the following:	
	`time`	Only time is profiled.
	`realTime`	Profiles real (elapsed) time rather than CPU time.
	`memory`	Only SKILL memory allocated is profiled.

Value Returned

`t`	Always returns `t`.

Example

```
profile( 'time)
for(i 1 10000 i+1)
profileSummary( ?file "/tmp/profile.results")
```

Reference

unprofile, profileReset, profileSummary

profileReset

```
profileReset(
     )
     => t
```

Description

Resets all SKILL profiler data.

Resets all data but keeps SKILL profiling running. Sets the accumulated CPU time and memory for all functions to zero. This is useful if you want to run the same set of profiled functions many times for different inputs so you can compare or average the results. When `profile` is first called, the profiling data is already initialized to zeros and there is no need to do an initial call to `profileReset`.

SKILL Profiler is not yet supported on the Windows/Wintel platform.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
profileSummary( ?file "myReport1" ) => t
```

Prints summary 1.

```
profileReset() => t
```

Resets the profiling timer/counter. Now run the same functions on another set of data.

```
profileSummary( ?file "myReport2" ) => t
```

Prints summary 2.

Reference

profile

profileSummary

```
profileSummary(
     [ ?file t_filename ]
     [ ?sort s_sortKey ]
     [ ?filters g_filterSpec ]
     [ ?maxFns x_maxDisplayed ]
     [ ?minSecs f_minSecs ]
     [ ?minBytes x_minBytes ]
     [ ?children g_showChildren ]
     )
     => t
```

Description

Prints a summary of profiling results, showing either the execution time or memory allocated to SKILL functions that were executed.

You select whether to profile time or memory by the argument you pass to the `profile` function. After executing the functions you are interested in, call `profileSummary` to generate a report of the CPU time spent in the functions or the amount of SKILL memory allocated in those functions. Using `profileSummary` options, you can sort and filter data to see only the functions in which you are interested. All functions are measured so you can create multiple profile summaries at the end of each session.

SKILL Profiler is not yet supported on the Windows/Wintel platform.

Arguments

`?file` *`t_filename`*
: Specifies the report file name. Defaults to `ilProf_<login>.out` in the `/tmp` directory.

`?sort` *`s_sortKey`*
: Changes the fields the profile summary is sorted by.

  `total` Seconds or bytes allocated in functions and children. This is the default.

  `inside` Seconds or bytes allocated in function only.

`?filters` *`g_filterSpec`*
: Valid values: a regular expression, a symbol, `t`, `binary`.

  A regular expression displays function names indicated by the expression. For example `^hi` displays all functions beginning with *`hi`*.

  A symbol containing a context displays all functions in that context.

  `t` displays user functions (functions which have been loaded by the user and are not read protected).

  `'binary` displays only SKILL functions implemented in C.

`?maxFns` *`x_maxDisplayed`*
: Integer indicating the maximum number of functions to be displayed in the profile summary.

  Default value: `1000`

`?minSecs` *`f_minSecs`*
: Floating-point number indicating in seconds the minimum time that a function must have spent executing before it should be displayed. This time cannot be smaller than 1/60 of a second.

  Default value: `0.0`

`?minBytes` *`x_minBytes`*
: Integer that indicates the minimum number of bytes that need to be allocated to the function before it should be displayed.

  Default value: `0`

If both `?minSecs` and `?minBytes` are specified then any function which meets the minimum requirement of either one is displayed.

`?children` *`g_showChildren`*

If `t`, then the amount of time spent in each child function and the memory allocated is printed at the bottom of the profile summary report.

Value Returned

`t`	Always returns `t`.

Example

```
profileSummary(?file "/tmp/summary.out"
?sort 'inside ?children t ?maxFns 100)
```

unprofile

```
unprofile(
     )
     => t
```

Description

Turns off SKILL profiling.

Does not reset the values. `profileSummary` also turns off profiling and then prints a report.

SKILL Profiler is not yet supported on the Windows/Wintel platform.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
unprofile( ) => t
```

3

Lint Functions

skDisableMessage

```
skDisableMessage(
     S_functionName
     S_messageName
     [ x_occurrences ]
     )
     => t
```

Description

Disables a SKILL Lint message from being reported inside a given function definition.

Often a user does not want to disable a SKILL Lint message globally but only an individual case. You can put the call to this function in the `.skinit` file in the user's home directory or the `your_install_dir/local` directory. This function can also be inserted in the file being analyzed, outside of any function definition, before the function is defined, and SKILL Lint will recognize the call. For a list of SKILL lint messages, refer to the *Cadence SKILL IDE User Guide*.

Arguments

S_functionName	Function in which the given SKILL Lint error message is not reported.
S_messageName	Name of the message to ignore when SKILL Lint analyzes the file containing the function definition.
x_occurrences	Number of times to ignore the error message inside the function. This defaults to 1 if not given.

Value Returned

`t`	Always returns `t`.

Example

```
skDisableMessage('testFunction 'Unused 1)
```

Disables the first occurrence of the SKILL Lint message `Unused` when SKILL Lint analyzes the function definition for `testFunction`.

Reference

sklint, skIgnoreMessage, skUnignoreMessage

skDisableMessageBlock

```
skDisableMessageBlock(
     l_rules
     g_exp1 ...
     )
     => g_result
```

Description

Disables one or a list of SKILL Lint messages from being reported by rules within the body of the function. You might want to turn off certain rules temporarily, and not globally, inside a particular block of code. SKILL Lint will recognize a call to this function if inserted inside the block of code being analyzed.

Note: This function does not work for summary or statistic messages generated by SKILL Lint (for example, `message:External`). In these cases, use the `sklint ?depends` and `?ignores` keyed arguments.

Arguments

l_rules	One or a list of rule(s) to ignore for all the code within the body of this function (that is, *g_exp1* ...).
g_exp1	Expressions that compose the function body.

Value Returned

g_result	The result of the last expression evaluated. It can be ignored.

Example

To disable the single SKILL Lint message for MEMBER1 from reporting (the parentheses around the message name can be omitted):

```
(procedure (aFunc a b)
     (skDisableMessageBlock MEMBER1
          (member a b)
          (geOpen)
     )
     (member 1 a)
)
```

To disable the SKILL Lint messages for MEMBER1 and geOpen from reporting:

```
(procedure (aFunc a b)
     (skDisableMessageBlock (MEMBER1 geOpen)
          (member a b)
          (geOpen)
     )
     (member 1 a)
)
```

Reference

skEnableMessageBlock

skEnableMessageBlock

```
skEnableMessageBlock(
     l_rules
     g_exp1 ...
     )
     => g_result
```

Description

Re-enables reporting of one or a list of SKILL Lint message(s), which has/have been globally turned off within the body of the this function only.

Often a user wants to turn back on certain rule(s), which has/have been turned off globally, temporarily inside a particular block of code only rather than globally. You can insert the call to this function inside that particular block of code being analyzed and SKILL Lint will recognize the call.

Arguments

l_rules	One or a list of rule(s) to ignore for all the code within the body of this function that is, *g_exp1* ...).
g_exp1...	Expressions that compose the function body.

Value Returned

g_result	The result of the last expression evaluated. It can be ignored.

Example

To disable the single SKILL Lint message for MEMBER1 from reporting (the parentheses around the message name can be omitted):

```
(procedure (aFunc a b)
     (skEnableMessageBlock MEMBER1
          (member a b)
          (geOpen)
     )
     (member 1 a)
)
```

To re-enable reporting of the SKILL Lint messages for MEMBER1 and `geOpen` temporarily inside a particular block of code:

```
(procedure (aFunc a b)
     (skEnableMessageBlock (MEMBER1 geOpen)
          (member a b)
          (geOpen)
     )
     (member 1 a)
)
```

Reference

skDisableMessageBlock

skIgnoreMessage

```
skIgnoreMessage(
     g_ignoreList
     )
     => t
```

Description

Turns the reporting of specified SKILL Lint messages off. For a message to appear, both the message and its group have to be unignored.

Message groups or individual messages can be ignored. For a list of SKILL lint messages, and message groups, refer to the *Cadence SKILL IDE User Guide*.

When a message is ignored, reporting is turned off until a call to unignore that same message is made. If the message group `hint` was turned off, on all subsequent runs of SKILL Lint all messages in the group `hint` would not be printed. These messages would also not affect the final IQ score. You can put calls to `skIgnoreMessage` in `.skinit`, the SKILL Lint startup file, in either the user's home directory or under

`your_install_dir/local`

This startup file executes whenever you run SKILL Lint.

Arguments

g_ignoreList	String or list of messages that SKILL Lint will no longer output.

Value Returned

`t`	Always returns `t`.

Example

```
skIgnoreMessage('(hint))
skIgnoreMessage("unused vars")
```

Turns off reporting of all hint and unused variable messages.

```
skUnignoreMessage('(suggestion))
```

Turns on reporting of performance suggestion messages.

Reference

sklint, skDisableMessage, skIgnoreMessage

sklint

```
sklint(
     [ ?file tl_inputFileName ]
     [ ?context t_contextName ]
     [ ?outputFile t_outputFileName ]
     [ ?ignoreGroups l_ignoreGroups ]
     [ ?globals l_globals ]
     [ ?depends l_depends ]
     [ ?rulesFile t_rulesFile ]
     [ ?ignores l_ignoresMessageList ]
     [ ?checkNlambda g_checkNlambda ]
     [ ?noPrintLog g_noPrintLog ]
     [ ?useGlobalIgnores g_useGlobalIgnores ]
     [ ?useGlobalRulesFileList g_useGlobalRulesFileList ]
     [ ?useDisableMessages g_useDisableMessages ]
     [ ?checkCdsFuncs g_checkCdsFuncs ]
     [ ?checkPvtFuncs g_checkPvtFuncs ]
     [ ?checkPubFuncs g_checkPubFuncs ]
     [ ?prefixes l_prefixList ]
     [ ?checkCdsPrefixes g_checkCdsPrefixes ]
     [ ?checkFuncPrefixes g_checkFuncPrefixes ]
     [ ?tabulate g_tabulate ]
     [ ?skPath t_skPath ]
     [ ?codeVersion t_release ]
     )
     => t / nil
```

Description

Checks a SKILL file or context and reports potential errors and ways to clean up your code.

SKILL Lint checks a SKILL file or context and reports potential errors and ways to clean up your code. In particular SKILL Lint is useful for helping programmers find unused local variables, global variables that should be locals, functions that have been passed the wrong number of arguments, and hints about how to improve the efficiency of the user's SKILL code.

SKILL Lint is usually run over a file. If a context is specified and the file is `startup.il` or is not specified, all the files ending with `.il` or `.ile` in the directory `your_install_dir/pvt/etc/context/t_contextName` are checked. By default, the SKILL Lint output prints to the Command Interpreter Window but can be printed to an output log file as well or instead. SKILL Lint prints messages about the user's code starting with the file and function name to which the message pertains. For a list of SKILL lint messages, refer to *Cadence SKILL IDE User Guide*.

Arguments

`?file` *`tl_fileFileName`*

The name of the file to be processed, or a list of file names. Each file is read and processed in turn.This option defaults to `startup.il`.

`?context` *`t_context`*

The name of the context, or an absolute-pathed context name, being processed. SKILL Lint looks under the `install_dir/`*`pvt/etc/context/t_contextName`* directory for all files ending with `.il` and `.ile` unless a file other than `startup.il` is given. If a file other than `startup.il` is given along with a context, that file is assumed to belong in that context and global variable package prefixes for that context are used if possible.

`?outputfile` *`t_outputFile`*

The name of the reporting log file. Defaults to *`context-Name`*`.log`.

`?ignoreGroups` *`l_ignoreGroups`*

The list of rule groups that should not be carried out.

`?globals` *`l_globals`*

The list of allowed globals not covered by the standard global list and the prefix list. This allows handling of obscure globals cases.

`?depends` *`l_depends`*

The list of contexts on which the code under analysis depends. This is used for loading external definitions files.

`?rulesFile` *`t_rulesFile`*

The name of an additional rules file to be read prior to processing the code. For information on how to write a `skLint` rules file, see the *Cadence SKILL IDE User Guide*.

`?ignores` *`l_ignoresMessageList`*

The list of message IDs to ignore. These messages are neither printed by SKILL Lint nor counted in the summary report at the end of the run. Message groups as well as individual messages can be ignored. For example, all messages about improving the efficiency of SKILL code can be turned off by passing in the `list '(hint suggestion)`. For a list of SKILL lint messages, and message groups, refer to the *Cadence SKILL IDE User Guide*.

`?checkNlambda` *`g_checkNlambda`*

Specifies whether to check the arguments to nlambda functions. This option should only be used by highly experienced users, as it usually leads to results that are difficult to interpret. This option defaults to `nil`.

`?noPrintLog` *`g_noPrintLog`*

Controls whether printing to the screen/`ciw` should take place. Even if switched off, printing of start and stop messages will take place. This option defaults to `nil`.

`?useGlobalIgnores` *`g_useGlobalIgnores`*

Controls whether to ignore those message IDs listed in the global variable *`skGlobalIgnores`*. This option is useful when the list of messages to ignore is constant and is held in a global list somewhere. This option defaults to `nil`.

`?useGlobalRulesFileList` *`g_useGlobalRulesFileList`*

Specifies whether to use the rules file listed in the global variable *`skGlobalRulesFiles`*. This option defaults to `nil`.

`?useDisableMessages` *`g_useDisableMessages`*

Controls whether to turn on or off disable messages to allow integrators to override message suppression put in the code. This option defaults to `t`.

`?checkCdsFuncs` *`g_checkCdsFuncs`*

Specifies whether to check both Cadence private and public functions (that is, force setting both *`checkPvtFuncs`* and *`checkPubFuncs`* to `t`). This option defaults to `nil`.

`?checkPvtFuncs` *`g_checkPvtFuncs`*

Controls whether to check Cadence private functions. This option defaults to `nil`.

`?checkPubFuncs` *`g_checkPubFuncs`*

Specifies whether to check Cadence public functions. This option defaults to `nil`.

`?prefixes` *`l_prefixes`*

The list of symbols whose print names are matched with variable names. The list may consist of functions and global variables not covered by the standard global list and the prefix list. This allows for obscure cases of globals to be handled.

`?checkCdsPrefixes` *`g_checkCdsPrefixes`*

Specifies whether the prefix checking is for Cadence public function/variables start with a lower-case character. If this argument is not set to `t` (that is, by default), the checking is for customers' function/variables prefixes start with an upper-case character. This option is for Cadence internal use only. This option defaults to `nil`.

`?checkFuncPrefixes` *`g_checkFuncPrefixes`*

Controls whether function prefixes should also be checked. If this argument is not set to `t` (that is, by default), only customers' global variable prefixes are checked. This option defaults to `nil`.

`?tabulate` *`g_tabulate`*

Controls whether to tabulate all the functions being called. This option defaults to `nil`.

`?skPath` *`t_skPath`*

The user-specified SKILL path to the file to be processed. If the option is specified, SKILL Lint will only search this path. Otherwise, the home directory will be searched first by default.

`?codeVersion` *`t_release`*

The release version of code being checked (for example, *500* for IC5.0.0). If this argument is specified all automatically generated function change messages (from `cdsFuncs.cxt`) that are equal to or before the release specified (through this argument) will be filtered out (that is, will not be reported). By default, all automatically generated function change messages (from `cdsFuncs.cxt`) will be reported.

This argument is useful when the user wants to restrict reporting of function change messages which occurred after the release for which the code being checked was written. When users check the code in IC500 they will not be interesting in seeing the information about the change in IC445, since that was before they wrote the code (or perhaps before it was migrated).

Specifying this argument will filter out both function changed and function deleted messages.

Value Returned

`t`	If SKILL Lint passed 100%.
`nil`	If SKILL Lint failed. SKILL Lint fails if there are any error or warning messages.

Example

```
sklint(?file "~/testfns.il")
```

Runs SKILL Lint over the `testfns.il` file and prints the output to the CIW.

```
sklint(?context "dbRead")
```

Runs SKILL Lint over all files loaded by `your_install_dir/pvt/etc/context/dbRead/startup.il`.

```
sklint(?file "~/testfns.il" ?outputFile "~/testfns.lint" ?noPrintLog t)
```

Runs SKILL Lint over the `testfns.il` file and prints the output to the `testfns.lint` file but not to the CIW.

```
sklint(?file "~/testfns.il" ?prefixes '(tfns) ?ignores '(hint suggestion))
```

Runs SKILL Lint over the `testfns.il` file and treats all global variables that start with the prefix `tfns` as acceptable global variables. In the above example, SKILL Lint does not print any hints or suggestions for how to make the user's SKILL code more efficient.

Runs SKILL Lint to check prefixes:

```
sklint( ?file "file.il" ?prefixes '(Pre MIX) )
```

Checks non-Cadence variable prefixes only.

```
sklint( ?file "file.il" ?checkFuncPrefixes t ?prefixes '(Pre MIX) )
```

Checks both non-Cadence function and variable prefixes.

```
sklint( ?file "file.il" ?checkCdsPrefixes t ?prefixes '(le ge) )
```

Checks Cadence variable prefixes only.

```
sklint( ?file "file.il" ?checkCdsPrefixes t ?checkFuncPrefixes t ?prefixes '(le ge)
)
```

Checks both Cadence function and variable prefixes.

Runs SKILL Lint on code that contain macros:

```
sklint ?file "dep.il file.il" ?depends '("dep.il")
```

where `dep.il` should contain macro definition(s) only, while `file.il` contains the code that call the macro(s) as defined in `dep.il`. The order of the files specifying in the ?*`file`* option is important that the file(s) contain the macro definition(s) have to be specified first. (that is, `dep.il` must be listed before `file.il`).

Note: Use this command for linting code where the macro definition(s) are contained in different file(s).

```
sklint ?file "file2.il" ?depends '("file2.il")
```

where `file2.il` contains both the macro definition(s) and the code that call the macro(s). The macro definition(s) code have to be placed on top of the code that call the macro(s) inside `file2.il`.

Note: Use this command for linting code where the macro definition(s) and the code that call the macro(s) are contained in the same file.

Reference

skDisableMessage, skIgnoreMessage

skUnignoreMessage

```
skUnignoreMessage(
     g_ignoreList
     )
     => t
```

Description

Turns the reporting of specified SKILL Lint messages on. For a message to appear, both the message and its group have to be unignored.

Message groups or individual messages can be unignored. For a list of SKILL lint messages, and message groups, refer to the *Cadence SKILL IDE User Guide*.

When a message is ignored, reporting is turned off until a call to unignore that same message is made. If the message group `hint` was turned off, on all subsequent runs of SKILL Lint all messages in the group `hint` would not be printed. You can put calls to `skUnignoreMessage` in `.skinit`, the SKILL Lint startup file, in either the user's home directory or under the `your_install_dir/local` directory. This startup file executes whenever you run SKILL Lint.

Arguments

g_ignoreList	String or list of messages that SKILL Lint will again output.

Value Returned

`t`	Always returns `t`.

Example

```
skIgnoreMessage('(hint))
skIgnoreMessage("unused vars")
```

Turns off reporting of all hint and unused variable messages.

```
skUnignoreMessage('(suggestion))
```

Turns on reporting of performance suggestion messages.

Reference

skDisableMessage, skIgnoreMessage, sklint

4

Context Functions

callInitProc

```
callInitProc(
     t_contextName
     )
     => t
```

Description

Calls all the initialization functions associated with a context.

Takes the same argument as `loadContext` (but without the `.cxt` extension) and causes all the initialization functions associated with the given context to be called. This function need not be used if the loading of the context is happening through the autoload mechanism. Use this function only when calling `loadContext` manually.

Arguments

t_contextName	Name of the context.

Value Returned

`t`	Returns `t` when initialization functions have been successfully called.

Example

```
loadContext("myContext.cxt") => t
callInitProc("myContext")    => t
```

All functions defined through `defInitProc` and `defUserInitProc` are called.

Reference

loadContext, defInitProc, `defUserInitProc`

callUserAutoInitProc

```
callUserAutoInitProc(
     t_contextName
     )
     => t / init_function_result
```

Description

Calls the `autoinit` function for the given context name.

If there is no `autoinit` function for the given context name, the function does not do anything and returns `t`.

Arguments

t_contextName	Name of the context.

Value Returned

g_result	Returns the result of the `autoinit` function.
`t`	`t` otherwise.

Example

```
callUserAutoInitProc("fake")    => t
```

checkContextBit

```
checkContextBit(
     t_contextPath
     )
     => t_type
```

Description

Checks and returns the context type of the specified context file.

Arguments

t_contextPath	Full path to the SKILL context file.

Value Returned

t_type	Returns a string (either "32bit" or "64bit") representing the context type. If the specified file is not a context file, the function displays an error message.

Example

```
checkContextBit("./myContext.cxt")
=> "32bit"
```

defCapDepends

```
defCapDepends(
     s_context
     l_dependsList
     )
     => t
```

Description

Specifies which contexts depend on which other contexts.

This can be specified in the `.skinit` file, which must reside in either the user's home directory or *`your_install_dir`*`/local` directory.

We recommend that developers put the `defCapDepends` function call in the beginning of the `startup.il` file because if SKILL Lint sees this call while analyzing the context it will determine the dependent contexts. When SKILL Lint is run on the context *`s_context`*, it loads all the dependent contexts from which it will be able to effectively type check the function calls made by the context being analyzed.

Arguments

`s_context`	Context that depends on the contexts specified by *`l_dependsList`*.
`l_dependsList`	List of contexts upon which *`s_context`* depends. In other words, *`l_dependsList`* should contain all the definitions for all the functions called in *`s_context`* that are not defined in *`s_context`*.

Value Returned

`t`	Always returns `t`.

Example

```
defCapDepends('myContext '(skillCore hiBase))
```

Tells SKILL Lint that `myContext` depends on the contexts `skillCore` and `hiBase`.

defCapPrefixes

```
defCapPrefixes(
     s_context
     l_prefixList
     )
     => t
```

Description

Specifies which prefixes are acceptable for a context's global variables.

All global variables that do not start with the stated prefixes are reported as unrecognized global variables. The call to `defCapPrefixes` can be specified in the `.skinit` file, which must reside in either the user's home directory or the *`your_install_dir`*`/local` directory.

We recommend that developers put the `defCapPrefixes` function call in the beginning of the `startup.il` file because if SKILL Lint sees this call while analyzing the context, it can determine the acceptable prefixes for the context being analyzed. If you want SKILL Lint to recognize the package prefixes when analyzing a file that contains `defCapPrefixes`, be sure to pass the *`s_context`* to SKILL Lint as the `?context`.

Arguments

`s_context`	Context for which the package prefixes should apply.
`l_prefixList`	List of acceptable package prefixes for *`s_context`*. The first letter following the prefix must be a capital. If the prefixes have been specified for a context, many of the unrecognized global variables will probably be variables that the user forgot to declare as locals.

Value Returned

`t`	Always returns `t`.

Example

```
defCapPrefixes('myContext '(my))
```

Tells SKILL Lint to expect any global variables in `myContext` to start with `my` or `_my` and to report all other global variables. Thus `myGlobalVariable` is an example of a legal global variable inside `myContext`.

defInitProc

```
defInitProc(
     t_contextName
     s_procName
     )
     => t
```

Description

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

When a context is autoloaded, it is given a chance to perform initialization before control returns to top level. It is during such an initialization that session-dependent objects like ports can be regenerated. This function permits a predefined function *s_procName* to be called whenever the context *t_contextName* is loaded.

Arguments

t_contextName	Name of context.
s_procName	Predefined function to be called when *t_contextName* is loaded.

Value Returned

t	Always returns t when set up. The function is not called at this point, but is called when the context *t_contextName* is autoloaded.

Example

```
defInitProc("myContext" 'myInit)=> t
```

Reference

defCapPrefixes, defCapDepends, setContext, saveContext

isContextLoaded

```
isContextLoaded(
     t_cxt
     )
     => t / nil
```

Description

Returns `t` if a context file with the given base name has been loaded into the current session.

Returns `nil` otherwise.

Arguments

t_cxt	Base name of the context file you want load status on.

Value Returned

`t`	The given context has already been loaded into the current environment.
`nil`	The given context has not been loaded yet. Registers a new top-level error-handler. This error-handler is called after stacktrace, when an error occurs. If an error-handler already exists, the function displays a warning message and registers a new top-level error-handler. This error-handler is called after stacktrace, when an error occurs. If an error-handler already exists, the function displays a warning message.

Example

```
isContextLoaded( "skillCore" ) => t
```

Registers a new top-level error-handler. This error-handler is called after stacktrace, when an error occurs. If an error-handler already exists, the function displays a warning message.

```
isContextLoaded( "hiBase" ) => nil
```

Reference

loadContext

Registers a new top-level error-handler. This error-handler is called after stacktrace, when an error occurs. If an error-handler already exists, the function displays a warning message.

loadContext

```
loadContext(
     t_contextFileName
     [ g_ignore64bitSubpath ]
     )
     => t / nil / error
```

Description

Loads a context file into the current session.

This function uses the SKILL path to find *t_contextFileName*, if you do not supply the full path. Additionally, if the optional argument is specified the function does not add /64bit subpath to the context file path.

Prerequisites

t_contextFileName must have been created using the function `saveContext`.

Arguments

t_contextFileName	Name of the context file you want to load.
g_ignore64bitSubpath	Specifies whether /64bit subpath should be added to the context file path or not. When set to `t` (64bit only), it does not add /64bit to the context file path and the context is loaded from the current directory.

Value Returned

`t`	The context was successfully loaded.
`nil`	Context has already been loaded.
`error`	Signals an error if: ■ The system failed to open a file ■ Virtual memory is exhausted ■ The version of context is incompatible with current software. This condition usually requires you to regenerate the context file

Example

```
loadContext( "geView.cxt" )
```

Loads "`64bit/geView.cxt`" from the SKILL path.

```
loadContext( "geView.cxt" t)
```

Loads "`geView.cxt`" from the SKILL path.

Reference

`getSkillPath`, `load`, saveContext, setContext

loadTopContextForms

```
loadTopContextForms(
     t_FileName
     [ ?debugMode g_debugMode ]
     [ ?writeProtect g_writeProtect ]
     [ ?writeProtectAll g_writeProtectAll ]
     [ ?lazyComp g_lazyComp ]
     [ ?printinfix g_printinfix ]
     [ ?integermode g_integermode ]
     [ ?mergemode g_mergemode ]
     [ ?readProtect g_readProtect ]
     )
     => t / nil
```

Description

Loads top-level SKILL or Scheme forms from a file. If the `setContext` mode is set, these forms are saved in a context. After the context is loaded, these forms are evaluated at the top-level, as if these were loaded from an `.il` or `.ils` file.

Arguments

t_FileName Name of the context file you want to load.

?debugMode *g_debugMode*

Set debug mode

?writeProtect *g_writeProtect*

Set write protect

Default value: t

?writeProtectAll *g_wirteProtectAll*

Set write protect all

Default value: t

?lazyComp *g_lazyComp*

Set the lazy compilation status

?printinfix *g_printinfix*

Set the printinfix status

Default value: t

?integermode *g_integermode*

Set the integer mode

?mergemode *g_mergemode*

Set the merge mode

Default value: t

?readProtect *g_readProtect*

Set the read protect mode

Default value: t

Value Returned

t If the file is loaded successfully.

nil If the file is not loaded.

Example

```
setContext("testc.cxt")
loadTopContextForms("./testFile.il")
saveContext("testc.cxt")
```

saveContext

```
saveContext(
     t_contextFileName
     [ g_ignore64bitSubpath ]
     )
     => t / nil
```

Description

Saves the current state of the SKILL language interpreter as a binary file. This function must be used in conjunction with `setContext`.

If the optional argument is specified the function does not add /64bit subpath to the context file path.

Saves all function and variable definitions that occur, usually due to file loading, between the calls to `setContext` and `saveContext`. Those definitions can then be loaded into a future session much faster in the form of a context using the `loadContext` function.

By default all functions defined in a context are read and write protected unless the `writeProtect` system switch was turned off (by setting (sstatus `writeProtect nil`)) when the function in question was defined between the calls to `setContext` and `saveContext`.

Arguments

t_contextFileName	Name of binary file to which the current state of the interpreter is written.
g_ignore64bitSubpath	Specifies whether /64bit should be added to the context file path or not. If set to `t`, the context is saved in the current directory and /64bit is not added to the context file path.

Value Returned

`nil`	If the saving process failed due to one of the following conditions: failed to open/create a file, exhaustion of virtual memory, presence of bad objects (such as port, db handles, and so forth)
`t`	Context was successfully saved.

Example1

```
setContext( "current")           => t
load("mySkillCode.il")           => t
defInitProc("current" 'myInit)   => t
saveContext("myContext.cxt")     => t
```

Saves as "`64bit/myContext.cxt`" in the SKILL path

```
saveContext("myContext.cxt" t)     => t
```

Saves as "`myContext.cxt`" in the SKILL path

Reference

`defInitProc`, loadContext, setContext

setContext

```
setContext(
     t_contextName
     )
     => t
```

Description

Allows contexts to be saved incrementally, creating micro contexts from a session's SKILL context.

To understand this, think of the SKILL interpreter space as being linear; the function call `setContext` sets markers along the linear path. Any SKILL files loaded between a `setContext` and a `saveContext` are saved in the file named in the `saveContext` call. This function can be used more than once during a session.

Arguments

t_contextName	Name used to identify a context.

Value Returned

`t`	Always returns `t`.

Example

```
setContext( "current")           => t
load("mySkillCode.il")           => t
defInitProc("current" 'myInit)   => t
saveContext("myContext.cxt")     => t
```

Reference

`defInitProc`, loadContext, saveContext

setSaveContextVersion

```
setSaveContextVersion(
     x_newVers
     )
     => x_oldVers
```

Description

Resets the current `saveContext` version to *`x_newVers`* and returns the previous context version. If `x_newVers` has an unsupported value or the function is called between `setContext` and `saveContext`, it returns an error.

Arguments

`x_newVers`	Specifies the new context version.

Value Returned

`x_oldVers`	Returns the old context version.

Example

```
setSaveContextVersion(getCompatContextVersion())
601
setSaveContextVersion(0)
*Error* setSaveContextVersion: unsupported context version - 0
```

getCurSaveContextVersion

```
getCurSaveContextVersion(
     )
     => x_curVers
```

Description

Returns the current `saveContext` version (the version which the new context will have.) The possible return values are, 601 for compatible contexts and 602 for native contexts (for IC 6.1.6/CAT 33.00)

Arguments

None

Value Returned

x_curVers	Returns the current `saveContext` version.

Example

```
setSaveContextVersion(getNativeContextVersion())
601
getCurSaveContextVersion()
602
```

getNativeContextVersion

```
getNativeContextVersion(
     )
     => x_nativeVers
```

Description

Returns the native context version (for IC 6.1.6/CAT 33.00, the native context version is 602).

Arguments

None.

Value Returned

x_nativeVers	Returns the native context version.

Example

```
getNativeContextVersion()
602
```

getCompatContextVersion

```
getCompatContextVersion(
     )
     => x_compatVers
```

Description

Returns the compatible context version (for IC 6.1.6/CAT 33.00, the compatible context version is 601).

Note: Do not use the 601 or 602 context version values directly in SKILL functions. Use `getCompatContextVersion`, `getNativeContextVersion`, or `getCurSaveContextVersion` instead to retrieve the values of context versions.

Arguments

None.

Value Returned

x_compatVers	Returns the compatible context version.

Example

```
getCompatContextVersion()
601
```

5

Debug Functions

break

```
break(
     )
     => none
```

Description

Forces entry to the break handler if inserted directly into a SKILL function. The default break handler is the debugger.

This function is useful if you want to stop at a particular place in a function.

Arguments

None.

Value Returned

None.

Reference

breakpt, cont, continue, unbreakpt

breakpt

```
breakpt(
     [ u_function
     [ break_condition ] ]
     )
     where break_condition can be either
     (break_tag
     g_condition
     )
     or
     (
     (break_tag
     g_condition)...
     )
     => g_result
```

Description

Sets breakpoints on one or more functions or function objects.

The SKILL debugger is the default break handler and is entered automatically when a breakpoint is encountered. The functions `breakpt` and `unbreakpt` set and clear breakpoints on the given functions or function objects.

Another way to enter the break handler is to insert the `break` function directly into a SKILL function at the point desired. Once you are in the break handler, you can examine the state of the program. If the function was loaded under `debugMode` (see `installDebugger`), you can use single stepping functions such as `step`, `next`, `stepout`, or `continue`.

If *`break_condition`* is not specified, the breakpoint is called *unconditional*. The behavior of an unconditional breakpoint is as follows: if the function is read-protected or not yet defined under `debugMode`, the breakpoint is assumed to be set at the *call* point. Otherwise, it is assumed to be set at the *entry* point of the function.

Arguments

u_function...	List of functions or function objects.
break_tag	Valid values: `call`Breaks at the calling point after evaluating all arguments in the caller's context. `entry`Breaks at the entry point after binding all formal parameters in the callee's context. `exit`Breaks at the exit point in the callee's context. `return`Breaks at the returning point in the caller's context.
g_condition	Condition expression to be evaluated in the associated break context.

Value Returned

g_result	List of functions or function objects on which the breakpoints have been set.

Example

```
installDebugger( )                ; Make sure debug mode is on.
breakpt( times )                  ; Sets a breakpoint on times.
times(2 plus( 2 3 ))              ; Invokes the break handler.
cont( )                           ; Continue executing.
unbreakpt( times )                ; Clear the breakpoint.
```

This example sets a breakpoint, enters the break handler, continues, and finally clears the breakpoint.

```
(breakpt myFun hisFun)
(breakpt myFun1 (entry n > 5))
(breakpt myFun2 ((entry n > 5) (exit result != 0)) hisFun)
```

This example sets a conditional breakpoint.

The following example sets breakpoint on a function object.

```
(installDebugger)                ; Make sure debug mode is on
=> t

(defun test (x) (printf "test : x == %L\n" x))

=> test

(putd 'test1 (getd 'test))     ;test and test1 are the same functions
=> funobj:test
```

```
(funcall 'breakpt (getd 'test))     ;set breakpoint on the funobject
(funobj:test)
(test1 8)
<<< Break >>> on entering test
Entering new debug toplevel due to breakpoint:
Debug 2> cont
test : x == 8
=> t
```

Reference

break, cont, continue, installDebugger, next, step, stepout, unbreakpt

breakptMethod

```
breakptMethod(
     [ S_name ]
     [ l_specializer [ @before | @after | @around ] ]
     [ break_condition ])
     where break_condition can be either
     (break_tag
     g_condition
     )
     or
     (
     (break_tag
     g_condition)...
     )
     => t / nil
```

Description

Sets breakpoint on the specified method's `defmethod` declaration.

If *`break_condition`* is not specified, the breakpoint is called *unconditional*. The behavior of an unconditional breakpoint is as follows: if the function is read-protected or not yet defined under `debugMode`, the breakpoint is assumed to be set at the *call* point. Otherwise, it is assumed to be set at the *entry* point of the function.

Arguments

S_name	Specifies the name of the method on which the breakpoint needs to be set
l_specializer	Specifies a list of specializers of the method to set breakpoint on. It can include `@before`, `@after`, and `@around` qualifiers
break_tag	Valid values:
	`call`Breaks at the calling point after evaluating all arguments in the caller's context
	`entry`Breaks at the entry point after binding all formal parameters in the callee's context
	`exit`Breaks at the exit point in the callee's context
	`return`Breaks at the returning point in the caller's context
g_condition	Specifies the condition expression to be evaluated in the associated break context

Value Returned

`t`	List of breakpoints set on the specified method
`nil`	If the specified method is not defined

Example

```
breakptMethod()
; when specified without arguments, lists all the breakpoints

breakptMethod(S_name)
; sets breakpoint on "S_name" function, behaves the same way as breakpt()

breakptMethod(S_name nil)
; sets breakpoint on the "generic" method of S_name.

breakptMethod(S_name @before (classA classB))
; sets breakpoint on "@before" method specialized on (classA classB) of "S_name"

breakptMethod(S_name @after (classA t))
; sets breakpoint on "@after" method specialized on (classA t) of "S_name"
```

clear

```
clear(
     )
     => t
```

Description

Clears all tracing and breakpoints.

This function undoes the effects of `tracef`, `tracep`, `tracev`, and `breakpt`.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
clear( ) => t
```

Untraces all functions and variables and clears all breakpoints.

Reference

`breakpt`, `tracef`, `tracep`, `tracev`

cont, continue

```
cont(
     )
     => no return value
     continue(
     )
     => no return value
```

Description

Continues execution from a breakpoint. `cont` and `continue` are identical.

Prerequisites

These functions work only within the break handler (which defaults to the SKILL debugger).

Arguments

None.

Value Returned

None.

Example

```
break( ) => puts you in the break handler
cont( ) => exits the break handler
```

Reference

break, breakpt

count

```
count(
     [ { s_function ... | t } ]
     )
     => g_result / t
```

Description

Counts the number of times a function has been called. This is an `nlambda` function. Returns the functions marked for counting.

Measures function call frequency and also serves as a valuable debugging aid. Both `count` and `uncount` accept a list of functions, or `t` for counting/uncounting all functions. To examine the number of times a function has been counted, call the `uncount` function. A list containing the number of times each function was called, along with the function name, is returned in the form of a list of `(number functionName)` pairs, such as,

```
((20 plus) (10 times) (5 greaterp))
```

The sublists are sorted by their first elements, using `sortcar`, so the most frequently executed functions always appear at the head of the list.

Arguments

s_function ...| t	Turns on counting for the functions given, or all functions if `t`.

Value Returned

g_result	Returns the functions marked for counting.
t	If no arguments are given and all functions are being counted.

Example

```
count( plus greaterp )          ; Counts plus and greaterp
setq( x plus( 2 3 ))            ; Use the functions being counted.
uncount( plus greaterp )
=> ((1 plus) (0 greaterp))      ; Number of times each
                                ; function was called.
```

Reference

uncount

debugQuit

```
debugQuit(
     )
     => nil
```

Description

Exits one level of the SKILL debugger.

Arguments

None.

Value Returned

`nil`	Always returns `nil`.

Example

```
installDebugger()         ; Installs the SKILL debugger.
4 / 0                     ; Generates an error and enters
                          ; the SKILL debugger.
debugQuit()               ; Exits the debugger.
uninstallDebugger( )      ; No longer enters the debugger upon error.
```

Reference

installDebugger, uninstallDebugger

debugStatus

```
debugStatus(
     )
     => nil
```

Description

Prints the functions and variables being traced, functions that have breakpoints set, functions being counted, and the line breakpoints statistics.

Note: Line breakpoints statistics include the file name and the line number on which line breakpoints have been set.

Arguments

None.

Value Returned

`nil`	Always returns `nil`.

Example

```
debugStatus() => nil
```

Returns `nil` and prints the debugging status of all functions. Sample output would look like:

```
Traced functions          (mytest)
Traced variables          (nil)
Traced properties         (nil)
Breakpoints               (myFunction1)
Counted functions         nil
Line Breakpoints          (40 (("/home/deeptik/demo.il")))
```

Reference

breakpt, clear, count, tracef, tracev, tracep, unbreakpt, uncount, untrace, untracev, untracep

dump

```
dump(
     [ x_variables ]
     )
     => nil
```

Description

Prints the current value of all the local variables on the stack. SKILL++ variables are not displayed by this function. For SKILL++ use `where` to see the lexical bindings on the stack.

`dump` is usually called from within the break or error handler.

Arguments

x_variables	Number of local variables on the stack to print, starting from the top. Defaults to printing all local variables on the stack.

Value Returned

`nil`	Always returns `nil`.

Example

Suppose `/tmp/color.il` defines function `initColor`:

```
(defun initColor (object)
  (let ((colorList '(red green yellow)) color)
       (setq color (concat (get object 'color)))
       (if (memq color colorList)
           (printf "color %s initialized" (get object 'name)))
  ])
)
```

Try this file in debugger:

```
installDebugger
=> t
1> load "/tmp/color.il"
=> t
1> (putprop 'object1 "green" 'color)
=> "green"

1> breakpt(initColor (entry (null object)) concat)
=> concat(initColor)
```

```
1>  (initColor 'object1)
*** Error in routine fprintf/sprintf:
Message: *Error* fprintf/sprintf: format spec. incompatible
with data

Debug 2> dump
colorList = (red green yellow)
color = green
object = object1
nil
```

Reference

tracev, where

gcsummary

```
gcsummary(
     )
     => t
```

Description

Prints a summary of memory allocation and garbage collection statistics in the current SKILL run.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
************* SUMMARY OF MEMORY ALLOCATION *************
Maximum Process Size (i.e., voMemoryUsed) = 3589448
Total Number of Bytes Allocated by IL = 2366720
Total Number of Static Bytes          = 1605632
------------------------------------------------------------
Type       Size   Allocated     Free      Static   GC count
------------------------------------------------------------
list         12      339968    42744     1191936          9
fixnum        8       36864    36104       12288          0
flonum       16        4096     2800       20480          0
string        8       90112    75008       32768          0
symbol       28           0        0      303104          0
binary       16           0        0        8192          0
port         60        8192     7680           0          0
array        16       20480     8288        8192          0
wtype        16       16384    16224           0          0
TOTALS       --      516096   188848     1576960          9
------------------------------------------------------------
User Type (ID)             Allocated     Free    GC count
------------------------------------------------------------
hiField              (20)       8192     7504           0
hiToggleItem         (21)       8192     7900           0
hiMenu               (22)       8192     7524           0
hiMenuItem           (23)       8192     5600           0
TOTALS                --       32768    28528           0
------------------------------------------------------------
Bytes allocated for :
      arrays       = 38176
      strings      = 43912
```

```
      strings(perm)= 68708
      IL stack     = 49140
      (Internal)   = 12288
TOTAL GC COUNT 9
----- Summary of Symbol Table Statistics -----
Total Number of Symbols = 11201
Hash Buckets Occupied = 4116 out of 4499
Average chain length = 2.721331
t
```

How to Interpret the Summary Report

Column	Contains
Type	Data type names.
Size	Size of each atom representing the data type in bytes.
Allocated	Total number of bytes allocated in the pool for the data type.
Free	Number of bytes that are free and available for allocation.
Static	Memory allocated in static pools that are not subject to GC. This memory is usually generated when contexts are built. When variables are write protected, their contents are shifted to static pools.
GC Count	Number of GC cycles triggered because the pool for this data type was exhausted.

Reference

`gc`, `profile`, profileSummary, `needNCells`

getAllLoadedFiles

```
getAllLoadedFiles(
     [ t_path ]
     )
     => l_files / nil
```

Description

Returns a list of all files loaded since debug mode was turned on.

Arguments

t_path	Path to a SKILL file.

Value Returned

l_files	List of files.
nil	If no files have been loaded.

Example

```
getAllLoadedFiles()
     => ("loop6.il" "demo.il"
     )
```

Returns a list of files loaded since debug mode was turned on.

```
getAllLoadedFiles("demo.il")
=> ("/home/user1/demo.il" "/home/user1/loop6.il"
)
```

Returns a list of files, with their full paths, loaded since debug mode was turned on.

Reference

debugQuit, installDebugger

getCallingFunction

```
getCallingFunction(
     [ tx_nestingLevel ]
     )
     => s_functionName  / nil
```

Description

Returns the name of the calling function or procedure at the specified level in the call stack.

Arguments

tx_nestingLevel	Optional argument that indicates the nesting level of the procedure of function name to be returned.

Value Returned

s_functionName	The name of the calling function or procedure.
`nil`	If the specified level exceeds the call stack level, or the function at the specified level is unnamed, `nil` is returned.

Example

The following returns the name of the function one level up in the nest of calls. This is the same as `getCallingFunction(1)`

```
getCallingFunction()
```

The following returns the name of the current function.

```
getCallingFunction(0)
```

The following returns the name of the function one level up in the nest.

```
getCallingFunction(1)
```

The following returns the name of the function *n* levels up in the nest, or the top level function if *n-1* is the top, or `nil`.

```
getCallingFunction(n)
```

The following example returns the names of the functions in a list.

```
(defun Child () (list (getCallingFunction 3)
                      (getCallingFunction 2)
                      (getCallingFunction 1)
                      (getCallingFunction)))
(defun Parent()      (Child))
(defun GrandParent()(Parent))
(defun GreatGrandParent() (GrandParent))

(GreatGrandParent)
=> (GreatGrandParent GrandParent Parent Child)
```

getFunctions

```
getFunctions(
     t_fileName
     )
     => l_functions / nil
```

Description

Returns functions defined in a file loaded after debug mode is turned on.

Returns the functions that were defined the last time that the file was loaded. Only the file name should be used and not its full path. If no extension is given, `.il` is assumed.

Prerequisites

Turn on debug mode before loading the file.

Arguments

t_fileName	File name loaded after debug mode was turned on.

Value Returned

l_functions	A list of functions.
`nil`	If no functions were defined in that file or if the file was not loaded after debug mode was turned on.

Example

```
getFunctions( "testfns" )
```

Returns the functions defined in `testfns.il`.

Reference

`getAllLoadedFiles`, `installDebugger`

getGFbyClass

```
getGFbyClass(
     s_ClassName
     [ g_nonExistent ]
     [ g_clearGFcache ]
     )
     => l_methods
```

Description

Returns a list of generic functions specializing on a given class.

Arguments

s_ClassName	Name of the class for which you want to view the list of specializing functions.
g_nonExistent	If set to `t`, lists all the generic functions specializing only on non-defined classes.
g_clearGFcache	Specifies whether to clear the method dispatch cache. The default is `nil`.

Value Returned

l_methods	A list of generic functions.

Example

```
getGFbyClass('systemObject)
```

returns

```
(printObject)
```

ilAddTopLevelErrorHandler

```
ilAddTopLevelErrorHandler(
     [ s_handler ]
     )
     => t
```

Description

Registers a new top-level error-handler. This error-handler is called after stacktrace, when an error occurs. If an error-handler already exists, the function displays a warning message.

Arguments

s_handler	A Skill function that accepts 0 or 1 argument. When an error is raised, the error message string is passed to the handler.

Value Returned

t	Returns t, if the error-handler is successfully registered, otherwise, throws an error.

Example

To define an error handler and then register it:

```
defun(myhandler(x)
printf("TOP LEVEL ERROR HANDLER : %L" x) );
ilAddTopLevelErrorHandler('myHandler)
```

ilDebugCountLevels

```
ilDebugCountLevels(
     )
     => x_levels
```

Description

Returns the number of top-level debug and error frames present in the SKILL stack, if SKILL is in breakpoint or error top level. Otherwise, returns 1.

Arguments

None.

Value Returned

x_levels	Number of top-level error and debug frames present in the SKILL stack.

Example

To return the number of top-level error frames debugged in the SKILL stack:

```
ilDebugCountLevels();
```

ilGetGFbyClass

```
ilGetGFbyClass(
     s_ClassName
     [ g_nonExistent ]
     [ g_clearGFcache ]
     )
     => l_methods
```

Description

Returns a list of generic functions specializing on a given class.

Note: An alias to this function with the name getGFbyClass exists.

Arguments

s_ClassName	Name of the class for which you want to view the list of specializing functions.
g_nonExistent	If set to `t`, lists all the generic functions specializing only on non-defined classes.
g_clearGFcache	Specifies whether to clear the method dispatch cache. The default is `nil`.

Value Returned

l_methods	A list of generic functions.

Example

```
ilGetGFbyClass('systemObject)
```

returns

```
(printObject)
```

ilGetIdeSessionWindow

```
ilGetIdeSessionWindow(
     [ g_force ]
     )
     => w_IDE / nil
```

Description

Returns the SKILL IDE session window (*w_IDE*), or `nil` if the IDE session window does not exist. If the optional argument *g_force* is set and is not `nil`, the function creates an IDE window and displays it on the screen.

Arguments

g_force	If set and is not `nil`, the function creates an IDE window and displays it on the screen.

Value Returned

w_IDE	Returns the SKILL IDE session window.
`nil`	Returns `nil` if the IDE session window does not exist.

Example

To return the SKILL IDE session window:

```
ilGetIdeSessionWindow()
=> nil
```

```
ilGetIdeSessionWindow(t)
=> swindow:1
```

ilGetTCovFiles

```
ilGetTCovFiles(
     )
     => l_tCovedFiles / nil
```

Description

Returns the list of files processed when you run an application in test coverage mode using the command line option `-ilTCov <fileList>`.

Arguments

None.

Value Returned

l_tCovedFiles	Returns the list of files processed during SKILL test coverage using the command line option `-ilTCov <fileList>`.
`nil`	Returns `nil` if the test coverage mode is not active.

Example

If Virtuoso is run in test coverage mode with files `file1.il` and `file2.ils` by using the command line option `-ilTCov <fileList>`, you can use the `ilGetTCovFiles` function to retrieve the list of files as shown below:

```
virtuoso -ilTCov 'file1.il file2.ils'
ilGetTCovFiles()
=>  ("file1.il file2.ils")
```

ilMergeTCovData

```
ilMergeTCovData(
     l_tCovDirs
     t_resultDir
     )
     => t
```

Description

Merges `tCov` files from several directories and stores them in a single directory.

Arguments

`l_tCovDirs`	List of directories with `tCov` data. Each element in the list should be a string.
`t_resultDir`	The name of resulting `tCov` directory where the merged data is stored.

Value Returned

`t`	Always returns `t`.

Example

To merge the `tCov` data in `./tCov` and `./tCov2` and store the merged versions in `./merged_tCovData`

```
ilMergeTCovData('("./tCov"
"./tCov2") "./merged_tCovData")
=> t
```

ilRemoveMethod

```
ilRemoveMethod(
     s_genFunction
     g_className
     [ g_method ]
     )
     => t / nil
```

Description

Removes a given method from a generic function.

Note: An alias to this function with the name removeMethod exists.

Arguments

s_genFunction	Name of the generic function from which the method needs to be removed.
g_className	Name of the class or a list of classes to which the generic function belongs.
g_method	Specifies the method qualifier. It can have one of the following values: `'@before`, `'@after`, and `'@around`. If this value is not provided or is specified as `nil`, then the primary method is removed.

Value Returned

`t`	If the method is successfully removed.
`nil`	If the method is not removed.

Example

To remove the object `myClass` from the function `myFunction`:

```
ilRemoveMethod(‘myFunction ‘myClass)
ilRemoveMethod('myFunB '(classX classY) '@after)
```

ilRemoveTopLevelErrorHandler

```
ilRemoveTopLevelErrorHandler(
      )
     => t
```

Description

Unregisters the top-level error-handler, previously registered by the `ilAddTopLevelErrorHandler` function.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

To unregister the top-level error-handler:

```
ilRemoveTopLevelErrorHandler()
```

ilSlotBoundp

```
ilSlotBoundp(
     obj
     t_slotName
     )
     => t / nil
```

Description

Checks if a named slot is bound (has been assigned a value) to an instance.

Arguments

obj	An instance of some class.
t_slotName	Slot name.

Value Returned

`t`	If the slot is bound.
`nil`	If the slot is unbound.

Note: It throws an error if `obj` or `t_slotName` is invalid.

Example

```
myObject->slotX = 20
ilSlotBoundp(myObject "slotX")=> t
```

ilToolBox

```
ilToolBox(
     )
     => t / nil
```

Description

Brings up the SKILL Development toolbox.

SKILL Development is a toolbox for debugging SKILL programs using a form-based graphical user interface.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
ilToolBox () => t
```

inNext

```
inNext(
     )
     => t / nil
```

Description

Returns `t` if the function is called in the expression that is executed in the debugged code on the `next()` SKILL function.

Arguments

None.

Value Returned

`t`	If the current debugged expression is executed on `next()`.
`nil`	If the current debugged expression is not executed on `next()`.

Example

```
installDebugger()
Loading skillDev.cxt
t
1> defun( callInNext () printf("inNext() => %N\n" inNext())
)
callInNext
1> defun( test (x)
 x = list(x x)
 callInNext()
 x = list(x x)
)
test
1> breakpt test
(test)
1> test(4)
<<< Break >>> on entering test
Entering new debug toplevel due to breakpoint:
Type (help "debug") for a list of commands or debugQuit to exit the toplevel.
```

```
Debug 2> next
stopped before evaluating (x = list(x x))
Debug 2> next
stopped before evaluating callInNext()
Debug 2> next
inNext() => t
stopped before evaluating (x = list(x x))
Debug 2> callInNext()
inNext() => nil
t
Debug 2> cont
((4 4)
    (4 4)
)
```

inStepOut

```
inStepOut(
     )
     => t / nil
```

Description

Returns `t` if the function is called in the `stepout()` SKILL function.

Arguments

None.

Value Returned

`t`	If called in `stepout()`.
`nil`	If not called in `stepout()`.

Example

```
installDebugger()
Loading skillDev.cxt
t
1> breakpt test
(test)
1> defun( callInStepOut () printf("inStepOut() => %N\n" inStepOut()))
callInStepOut
1> defun( test (x)
x = list(x x)
callInStepOut()
)
function test redefined
test
1> test(4)
<<< Break >>> on entering test
Entering new debug toplevel due to breakpoint:
Debug 2> callInStepOut()
inStepOut() => nil
t
Debug 2> next
stopped before evaluating (x = list(x x))
Debug 2> callInStepOut()
inStepOut() => nil
t
Debug 2> next
stopped before evaluating callInStepOut()
Debug 2> stepout
```

```
inStepOut() => t
t
```

installDebugger

```
installDebugger(
     )
     => t / nil
```

Description

Installs the SKILL debugger as the error handler so that the debugger is entered automatically upon error. Turns on debug mode.

`installDebugger` also turns on debug mode and allows all functions, including those that are write protected, to be redefined. Debug mode stores cross-referencing information about functions and files as well as more information for stacktraces. Debug mode also changes the prompt to display the number of nested debuggers plus one. You might find it desirable to put the `installDebugger` function in your initialization file while you are developing your code.

Arguments

None.

Value Returned

`t`	If the debugger is successfully installed.
`nil`	Otherwise.

Example

```
installDebugger()        ; Installs the SKILL debugger.
4 / 0                    ; Generates an error and enters
                         ; the SKILL debugger.
stacktrace               ; Displays the SKILL stack.
debugQuit()              ; Exits the debugger.
```

Reference

uninstallDebugger, debugQuit, stacktrace

listAlias

```
listAlias(
     [ s_aliasName ]
     )
     => s_functionName / l_propertyList / nil
```

Description

Prints a (property) list of all current aliases and associated function symbols, or the function symbol that the given alias defines to.

Arguments

s_aliasName	Symbol name of the alias

Value Returned

s_functionName	Function symbol that the given alias name defines to.
l_propertyList	Property list of all current aliases and associated function symbols (This is printed when no argument is specified).
nil	The given alias name is not an alias for any function, or there's no alias defines for any function at all.

Example

```
;; Defines 'lf' and 'e' as the aliases of the listFunctions()}
;; and edit() functions, respectively
alias(lf listFunctions) => lf
alias(e edit) => e

;; Prints the name of the function that 'lf' aliases to
listAlias('lf) => listFunctions

;; Prints A property list of all current aliases and associated
;; function symbols
listAlias() => (e edit lf listFunctions)

;; The given alias name is not an alias for any function
listAlias('bogus) => nil

;; Remove 'lf' and 'e' as aliases
unalias(lf) => (lf)
unalias(e)  => (e)

;; There's no alias defines for any function anymore
listAlias() => nil
```

listFunctions

```
listFunctions(
     t_pattern
     [ g_listAllFuncs ]
     )
     => l_functions / nil
```

Description

Returns all public function names that contain the given substring or match the given regular expression. If the second (optional) argument is specified as t (or a non-nil value), `listFunctions` would look at the SKILL Virtual Machine, rather than the cdsFinder database, and return all `isCallable` Cadence public functions and user-defined SKILL functions that contain the given substring or match the given regular expression.

By default, the source of the returned values is the SKILL directory in the SKILL Finder database, which is located in the doc hierarchy under *your_install_dir*/doc/finder/SKILL. `listFunctions` will also look in the *your_install_dir*/local/finder/SKILL directory, as well as directories specified in `CDS_FINDER_PATH` (refer to the "*Environment Variable for Additional Finder Data Directories"* section of *Cadence SKILL IDE User Guide* for usage of `CDS_FINDER_PATH`), for any personal functions you may have created and placed there in the appropriate SKILL Finder format, which is described in the toolbox help files of *Cadence SKILL IDE User Guide*.

The returned function names can be used for passing a list of function names that match a given pattern to another function.

Arguments

t_pattern	Pattern to search for.
g_listAllFuncs	If specified as `t` (or a non-nil value), all `isCallable` Cadence public and user-defined function names that contain the given substring or match the given regular expression are returned. Default is `nil`.

Value Returned

l_functions	All public function names that match the given *t_pattern*.
`nil`	If no functions are found that match the pattern.

Example

```
apply( 'tracef listFunctions( "hi" ))
```

Calls the `tracef` function with all the functions that contain the substring `hi`.

```
apply( 'tracef listFunctions( "x" t ))
```

Calls the `tracef` function with all `isCallable` Cadence public and user-defined functions that contain the substring "*x*"

```
listFunctions("^hi")
```

Lists all the functions that begin with “hi”

```
listFunctions("^db.*Copy")
```

Lists all the functions that begin with db and have “Copy” in their names.

Reference

`listVariables`, rexCompile, rexMatchp

listVariables

```
listVariables(
     t_pattern
     )
     => l_variables
```

Description

Returns all variable names that match the given substring or regular expression as part or all of their print name.

They can be used for passing a list of variable names that match a given pattern to a function.

Arguments

t_pattern	Pattern to search for.

Value Returned

l_variables	All variable names that match the given *t_pattern*.

Example

```
apply( 'tracev listVariables( "myVars" ))
```

Traces the variables that match the pattern `myVars`.

Reference

`listFunctions`, rexCompile

memoryAllocated

```
memoryAllocated(
     )
     => f_megabytesAllocated
```

Description

Returns the amount of memory allocated by a process. The returned value is an approximation in megabytes and might not include the memory that has been allocated, but the amount that is unused.

The returned value is intended to be larger than the value returned from a previous invocation of this function, in case, a large amount of memory has been allocated.

Arguments

None.

Arguments

f_megabytesAllocated

Amount of memory that the process has allocated (an approximation in megabytes.)

Example

```
memoryAllocated() => 1.743386
```

next

```
next(
     [ x_steps ]
     )
     => none
```

Description

Allows execution to proceed until the next expression. This function only works if executed from within a break handler and if the code you want to step through was loaded under `debugMode`. See `installDebugger`.

`next` reenters the break handler after completing *`x_step`* expressions, as long as the program has not finished.

You cannot execute the `next` function inside functions that are read protected.

Arguments

`x_steps`	Number of SKILL expressions to execute at or above the current stack depth.

Value Returned

None.

Example

Suppose `/tmp/color.il` defines function `initColor`:

```
(defun initColor (object)
  (let ((colorList '(red green yellow)) color)
       (setq color (concat (get object 'color)))
       (if (memq color colorList)
          (printf "color %s initialized" (get object 'name)))
  )
)
```

Try this file in debugger:

```
installDebugger
=> t
(sstatus sourceTracing t)             ; Turns on sourceTracing
                                      ; to get line numbers.
=> t
```

```
1> load "/tmp/color.il"
=> t

1> (putprop 'object1 "green" 'color)
=> "green"

1> breakpt(initColor (entry (null object)) concat)
=> concat(initColor)

1>  (initColor 'object1)
<<< Break >>> on calling concat with args ("green")
at line 3 in file /tmp/color.il

Debug 2> next                          ; Proceeds to 'if' form
stopped at line 4 in file /tmp/color.il
before evaluating if(memq(color colorList) printf("color %s initialized"...

Debug 2> step                          ; Steps into 'if' form
stopped at line 4 in file /tmp/color.il
before evaluating memq(color colorList)

Debug 2> next                           ; Proceeds to 'printf' form
stopped at line 5 in file /tmp/color.il
before evaluating printf("color %s initialized" get(object 'name))
```

Reference

breakpt, installDebugger, step

pp

```
pp(
     s_functionName
     [ p_outputPort ]
     )
     => nil
```

Description

Pretty prints the definition of a function. The function must not be read-protected. This is an `nlambda` function.

Each function definition is printed in a manner that allows it to be read back into SKILL. `pp` does not evaluate its first argument but does evaluate the second argument, if given.

Arguments

`s_functionName`	Name of the function to be pretty printed.
`p_outputPort`	Output port to print to. Default is `poport`.

Value Returned

`nil`	Pretty prints the function.

Example

```
procedure(fac(n) if(n <= 1 1 n*fac(n-1)))=> fac
pp fac
procedure(fac(n)
    if((n <= 1) 1
        (n * fac(n - 1))
    )
)
=> nil
```

Defines the factorial function `fac` then pretty prints it to `poport`.

Reference

profile, `pprint`

printFunctions

```
printFunctions(
     t_pattern
     [ p_outport ]
     [ g_listAllFuncs ]
     )
     => t
```

Description

Prints all function names that contain the given substring or match the given regular expression.

These functions are useful for finding functions that contain the same substring or finding an individual function when you know only part of the name.

By default, the source of the returned values is the SKILL directory in the SKILL Finder database, which is located in the doc hierarchy under *your_install_dir*/doc/finder/SKILL. printFunctions will also look in the *your_install_dir*/local/finder/SKILL directory, as well as directories specified in CDS_FINDER_PATH (refer to the *Environment Variable for Additional Finder Data Directories Help* section of *Cadence SKILL IDE User Guide* for usage of CDS_FINDER_PATH), for any personal functions you may have created and placed there in the appropriate SKILL Finder format, which is described in the toolbox help files of *Cadence SKILL IDE User Guide.*

If the third (optional) argument is specified to t, printFunctions would look at the SKILL Virtual Machine, rather than the cdsFinder database, and returns all isCallable Cadence public functions and user-defined SKILL functions that contain the given substring or match the given regular expression.

Arguments

t_pattern	Pattern to search for.
p_outport	Optional output port. Default is `poport`.
g_listAllFuncs	If specified to `t`, all `isCallable` Cadence public and user-defined function names that contain the given substring or match the given regular expression are returned. Default is `nil`.

Value Returned

`t`	Always returns `t` (after printing all the function names that contain the given substring or match the given regular expression).

Example

```
printFunctions( "installDebug" ) => t
```

Returns `t` and prints all the function names that contain the substring `installDebug`.

```
printFunctions( "x" nil t)
```

Returns `t` and prints all `isCallable` Cadence public and user-defined function names that contain the substring *x*.

```
p = outfile( "outfile" ) => port:"outfile"
printFunctions( "x" p t )
```

Returns `t` and writes all `isCallable` Cadence public and user-defined function names that contain the substring *x* to outfile.

Reference

`listFunctions`, `printVariables`, rexCompile

printObject

```
printObject(
     g_object
     [ p_outputPort ]
     )
     => g_result
```

Description

A generic function that writes a description of an object to an output port.

If you define a method for this generic function, it should call one or more of the SKILL print functions. See the *Cadence SKILL Language User Guide* for a discussion of generic functions.

Arguments

g_object	Object whose print representation you want.
p_outputPort	Specified output port.

Value Returned

g_result	Returns the return value of the method that was called.

Example

```
P = makeInstance( 'Point ?name "P" ?x 3 ?y 4 )
    => stdobj:0x1d003c
ILS-<2> defmethod( printObject ((p Point)
    @optional ( port poport ))
    fprintf( poport "%s @ %n:%n\n" p->name p->x p->y )
    nil
    )
t
ILS-<2> printObject( P )
P @ 3:4
nil
```

Prints `P @ 3:4`, the location of `P`.

```
ILS-<2> defstruct( card rank suit )
t
ILS-<2> mycard = make_card( ?rank 2 ?suit "spades" )
array[4]:2304000
ILS-<2> addDefstructClass( card )
funobj:0x1c98f8
```

```
ILS-<2> printObject( mycard )
Loading skillDev.cxt
Structure of type card:
  rank: 2
  suit: "spades"
t
```

Reference

printf, print, println, fprintf, defmethod

printstruct

```
printstruct(
     g_object
     )
     => t
```

Description

Prints the contents of an association table or defstruct in a tabular format.

For debugging purposes, the `printstruct` function prints the contents of a structure in a readable form. It recursively prints nested structures.

Arguments

g_object	Defstruct or association table to be printed.

Value Returned

t	Prints contents of the defstruct or association table.

Example

```
defstruct(myStruct slot1 slot2) => t
```

```
struct = make_myStruct(?slot1 "one" ?slot2 'two)=> array[4]:3872800
printstruct(struct)
```

```
Structure of type myStruct:
slot1: "one"
slot2: two => t
```

Reference

`defstruct, defstructp, makeTable`

printVariables

```
printVariables(
     t_pattern
     [ p_outport ]
     )
     => t
```

Description

Prints all variable names that contain the given substring or match the given regular expression, along with their values.

This function is useful for finding variables that contain the same substring or finding an individual variable when you know only part of the name. The `printVariables` function also prints the value of each variable it finds.

Arguments

t_pattern	Pattern to search for.
p_outport	Optional output port. The default is `poport`.

Value Returned

`t`	Always returns `t` and prints the value of each variable it finds.

Example

```
printVariables( "stack" )
                 _stacktrace 0
=> t
```

Prints all the variables with their values that contain the substring `stack` and returns `t`. The underscore (_) at the beginning of `_stacktrace` indicates that it is an internal system variable.

Reference

`listVariables`, `printFunctions`

removeMethod

```
removeMethod(
     s_genFunction
     g_className
     [ g_role ]
     )
     => t / nil
```

Description

Removes a given method from a generic function.

Note: For compatibility with previous releases, an alias to this function with the name, `ilRemoveMethod` exists.

Arguments

s_genFunction	Name of the generic function from which the method needs to be removed.
g_className	Name of the class or a list of classes to which the generic function belongs.
g_role	Specifies the method qualifier. It can have one of the following values: '@before, '@after, '@around, or nil. If this value is not provided or is specified as `nil`, then the primary method is removed.

Value Returned

`t`	If the method is successfully removed.
`nil`	If the method is not removed.

Example

To remove the object `myClass` from the function `myFunction`:

```
removeMethod(‘myFunction ‘myClass)
removeMethod('myFunB '(classX classY) '@after)
```

resume

```
resume(
     [ g_result ]
     )
```

Description

Exits the interactive top-level loop started with the most recently invoked `toplevel` function and returns its argument to the caller of `toplevel`. Do not use this function programmatically; use it only as an interactive command.

The `resume` function itself does not return. It returns value of the `toplevel` function.

- To start a top-level interactive loop in SKILL++ mode, type

  ```
  toplevel( 'ils )
  ```

- To start a top-level interactive loop in SKILL mode, type

  ```
  toplevel( 'il )
  ```

Arguments

g_result	Optional value to be returned as the result from the previous `toplevel` calls.

Value Returned

Returns the return value of the `toplevel` function. The `resume` function itself does not return.

Example

Following is a transcript of a brief session, including prompts.

```
> R = toplevel( 'ils )
ILS-<2> resume( 1 )
1
> R
1
```

Reference

`toplevel`, `errset`

skillDebugger

```
skillDebugger(
     )
     => nil
```

Description

Activates the SKILL Debugger. Usually invoked by a break or error handler.

You do not normally call it; instead it is invoked by the break or error handler. The SKILL debugger is the default break handler, and can also be used as the current error handler by calling the `installDebugger` function.

When you enter the debugger, the prompt changes to `debug #>` where `#` is a number identifying the number of nested debuggers plus one. Once in the SKILL debugger, you can examine the stack and local variables with functions such as `stacktrace`, `dump`, and `where`. You can also execute any SKILL function normally available because the debugger calls the SKILL top level. To quit the function, use `debugQuit`.

If the SKILL debugger is entered from a breakpoint, the following functions can be used to resume execution: `step`, `next`, `stepout`, and `continue`. If an error occurs and an `errset` is on the stack, the SKILL Debugger will not be invoked. To debug errors in this case, set `_stacktrace` to an integer value greater than zero or set breakpoints before the error occurs.

Arguments

None.

Value Returned

`nil`	Always returns `nil`.

Example

```
skillDebugger()       ; Calls the debugger.
debugQuit()           ; Exits the debugger.
alias q debugQuit     ; Alias used for faster typing.
```

Reference

debugQuit, dump, cont, continue, installDebugger, next, stacktrace, step, stepend, stepout, where, uninstallDebugger

skillDevStatus

```
skillDevStatus(
     )
     => t / nil
```

Description

Returns the current status of the SkillDev license.

Arguments

None.

Value Returned

`t`	SkillDev license is checked out.
`nil`	SkillDev license is not checked out.

Example

```
skillDevStatus()
=> nil
```

stacktrace

```
stacktrace(
     [ g_unevaluated ]
     [ x_depth ]
     [ x_skip ]
     [ p_port ]
     )
     => x_result
```

Description

Prints the functions on the stack and their arguments to the depth specified, or to the bottom of the stack.

Observes the following rules:

- When debug mode is on `stacktrace`, it prints the evaluated function arguments by default if the status switch `traceArgs` has been set to `t`.
- When debug mode is off, `stacktrace` always prints the unevaluated arguments.
- If the status switch `stacktrace` is set (using the sstatus function) to an integer, it prints that number of stack frames automatically whenever an error occurs.
- If there are no functions on the stack, that is, you are at the top, then `stacktrace` does not print anything and returns 0.

Note: `stacktrace` has a more flexible interface for user convenience. Thus if the first argument is a number it will interpret it to be *x_depth*, otherwise if it is non-`nil` it will take it to be *g_unevaluated*.

Prerequisites

This function is usually used inside the break or error handler.

Arguments

g_unevaluated	If `t`, always prints the unevaluated function parameters.
x_depth	Number of stack levels to print, defaults to all.
x_skip	`stacktrace` skips over the number of function calls specified by *x_skip*. This argument defaults to 1.
p_port	Port for the `stacktrace` output, defaults to the error port.

Value Returned

x_result	The number of stack frames printed.

Example

```
stacktrace()
```

Prints all the functions on the stack.

```
stacktrace( 5 )
```

Prints the top five functions on the stack.

```
stacktrace( t 5 3 ptport )
```

Prints the five functions on the stack that come after the first three to the trace port.

```
sstatus( stacktrace 6 )
```

Prints the first six stack frames every time an error occurs.

Reference

`breakpt`, `cont, continue`, `dump`, `installDebugger`, `next`, sstatus, `step`, `stepend`, `stepout`, `unbreakpt`, `uninstallDebugger`, `where`

step

```
step(
     [ x_steps ]
     )
```

Description

Steps into functions and executes a given number of SKILL functions. This function only works if executed from within a break handler and if the code you want to step through was loaded under `debugMode`. See installDebugger.

The number of steps defaults to 1 if there is no argument given. After completing *x_steps*, `step` re-enters the break handler before executing its next function, as long as the program has not finished. You cannot step inside functions that are read protected.

Arguments

x_steps	Number of SKILL commands to execute.

Value Returned

None.

Example

Suppose `/tmp/color.il` defines function `initColor`:

```
(defun initColor (object)
  (let ((colorList '(red green yellow)) color)
       (setq color (concat (get object 'color)))
       (if (memq color colorList)
           (printf "color %s initialized" (get object 'name)))
  )
)
```

Try this file in debugger:

```
installDebugger
=> t


1> (sstatus sourceTracing t)
                     ; Turns on sourceTracing to get line numbers
=> t
```

```
1>  load "/tmp/color.il"
=> t


1> (putprop 'object1 "green" 'color)
=> "green"

1> breakpt(initColor (entry (null object)) concat)
=> concat(initColor)

1> (initColor 'object1)
<<< Break >>> on calling concat with args ("green")
at line 3 in file /tmp/color.il

Debug 2> next
stopped at line 4 in file /tmp/color.il
before evaluating if(memq(color colorList) printf("color %s initialized"...

Debug 2> step
stopped at line 4 in file /tmp/color.il
before evaluating memq(color colorList)

Debug 2> next
stopped at line 5 in file /tmp/color.il
before evaluating printf("color %s initialized" get(object 'name))

Debug 2> step
stopped at line 5 in file /tmp/color.il
before evaluating get(object 'name)
```

Reference

breakpt, cont, continue, dump, next, stepend, stepout, unbreakpt, where

stepend

```
stepend(
     [ x_stepN ]
     )
```

Description

Allows execution to proceed to the end of the `nth` enclosing form and displays its result. `stepend` cannot proceed past the end of the current function. This function only works if executed from within a break handler and if the code you want to step through was loaded under `debugMode`. See installDebugger.

Comparing `step` and `stepend`:

- `step` proceeds to the "beginning" of the `(n + 1)th` possibly enclosed form and displays the next form to be evaluated.
- `stepend` proceeds to the end of the `nth` enclosing form and displays its result. `stepend` cannot proceed past the end of the current function.

Arguments

x_stepN	Number of forms to step through.

Value Returned

None.

Example

Suppose `/tmp/color.il` defines function `initColor`:

```
(defun initColor (object)
  (let ((colorList '(red green yellow)) color)
       (setq color (concat (get object 'color)))
       (if (memq color colorList)
           (printf "color %s initialized" (get object 'name)))
  )
)
```

Try this file in debugger:

```
installDebugger
=> t
```

```
1> (sstatus sourceTracing t)
                ; Turns on sourceTracing to get line numbers
=> t

1>  load "/tmp/color.il"
=> t

1> (putprop 'object1 "green" 'color)
=> "green"

1> breakpt(initColor (entry (null object)) concat)
=> concat(initColor)

1> (initColor 'object1)
<<< Break >>> on calling concat with args ("green")
at line 3 in file /tmp/color.il

Debug 2> next
stopped at line 4 in file /tmp/color.il
before evaluating if(memq(color colorList) printf("color %s initialized"...

Debug 2> step
stopped at line 4 in file /tmp/color.il
before evaluating memq(color colorList)

Debug 2> stepend
stopped at line 4 in file /tmp/color.il
after evaluating memq(color colorList)
==> (green yellow)

Debug 2> stepend
stopped at line 5 in file /tmp/color.il
after evaluating get(object 'name)
==> nil
```

Reference

step, stepout

stepout

```
stepout(
     [ x_steps ]
     )
```

Description

Allows execution to proceed until the evaluator returns from the current function.

It reenters the break handler when the current function returns to its caller.

Arguments

x_steps	Number of function call levels to return from before reentering the break handler. Defaults to 1.

Value Returned

None.

Example

Suppose `/tmp/color.il` defines function `initColor`:

```
(defun initColor (object)
  (let ((colorList '(red green yellow)) color)
       (setq color (concat (get object 'color)))
       (if (memq color colorList)
           (printf "color %s initialized" (get object 'name)))
  )
)
```

Try this file in debugger:

```
installDebugger
=> t

1> (sstatus sourceTracing t)
                     ; Turns on sourceTracing to get line numbers
=> t

1>  load "/tmp/color.il"
=> t

1> (putprop 'object1 "green" 'color)
=> "green"
```

```
=> initColor

1> (putprop 'object1 "green" 'color)
=> "green"

1> breakpt(initColor (entry (null object)) get)
=> (get initColor)

1> (initColor 'object1)
<<< Break >>> on calling get with args (object1 color)
at line 3 in file /tmp/color.il

Debug 2> stepout
<<< Break >>> on calling get with args (object1 name)
at line 5 in file /tmp/color.il
                                          ; stop at next 'get'
```

Reference

breakpt, cont, continue

toplevel

```
toplevel(
     [ s_langMode ]
     [ e_envobj ]
     )
     => g_result
```

Description

Starts an interactive top-level loop in either SKILL or SKILL++ mode.

All expressions you enter while the loop is in progress are evaluated with the specified language mode and optional environment. If you don't specify a language mode, then classic-SKILL is the default.

Note: The defining forms (such as, `define`, `defun`, `procedure`) entered at the top-level prompt are treated as "toplevel" definitions, not as local ones, even if a non-toplevel environment is supplied. (The same is true for `eval` with an explicit environment.)

Arguments

s_langMode	Must be a symbol. Valid values: `'ils` Indicates SKILL++. `'il` Indicates SKILL.
e_envobj	When the given *s_langMode* is for SKILL++, an optional environment object can be supplied, and the forms entered will be evaluated within the given (lexical) environment (except for the defining forms, like `define`, `defun`, and `procedure`, which will always add definitions to the top-level environment).

Value Returned

g_result	Argument passed to a call to `resume`.

Example

```
> R = toplevel( 'ils )
ILS-<2> resume( 1 )
1
> R
1
```

Starts an interactive loop, with prompt `ILS-<2>` and immediately returns the value 1 to the outer top level.

```
ILS-<2> R = let( ( ( x 1 ) ( y 2 ) )
toplevel( 'ils theEnvironment() )
)
ILS-<3> x
1
ILS-<3> y
2
ILS-<3> x = 4
4
ILS-<3> resume( x )
4
ILS-<2> R
4
```

Starts an interactive loop, with prompt `ILS-<3>`, in the environment established by the `let` expression. The `resume` function returns the current value of the local variable `x` to the outer top level, with prompt
`ILS-<2>`.

Reference

`resume`, errset

tracef

```
tracef(
     [ { s_function | t_fileName ... | t } ]
     )
     => g_result
```

Description

Turns on tracing of specified functions. Shows the functions called with their evaluated arguments and return values. This is an `nlambda` function.

The output port for tracing is `ptport`, which defaults to `poport`.

- If `t` is passed in, all functions are traced. However, this probably produces more information than you want and your program runs much more slowly.
- If you do not give `tracef` an argument, it returns a list of all functions being traced.
- If the argument is a string, `tracef` assumes it is a file name. `tracef` checks to see if a file was loaded after debug mode was turned on and if so, traces all functions defined in that file.
- If the symbol `debugFiles` is passed in, all functions in all files loaded since debug mode was turned on are traced.

If you want to force certain functions not to be traced even though you have turned on tracing for many or all functions, you can add the property `notrace` with a non-`nil` value on the name of the function. For example, to prevent `plus` from being traced use `putprop('plus t 'notrace )` or `plus.notrace = t`.

Arguments

s_function	Any function that you want to trace.
t_filename	Any file containing functions that you want to trace.
t	Turns tracing on for all functions.

Value Returned

g_result	Functions or files traced.

Example

```
(defun f (x) (add1 x))     ; Defines a function f.
=> f

(tracef f                  ; Turns on tracing for f.
=> (f)

f(3)
|f(3)
|f --> 4
=> 4

(tracef t                  ; Turns on tracing for all functions.
|tracef --> t
=> t

f(3)
|f(3)
||add1(3)
||add1 --> 4
|f --> 4
=> 4
```

Suppose `testfuns.il` defines functions `f1` and `f2`:

```
(defun f1 (x) x+1)
(defun f2 (y) f1(y)+1)
installDebugger            ; debug was turned on before loading a file
=> t

1> load "testfuns.il"
=> t

1> tracef("testfuns.il")
                          ; tracing for all functions in the file
("testfuns.il")
```

```
1> f2 3
|f2(3)
||f1(3)
||f1 --> 4
|f2 --> 5
5
```

Reference

tracev, untrace, untracev

tracelevlimit

```
tracelevlimit(
     [ x_depth ]
     )
     => t
```

Description

Limits the indentation level and hence the call depth while tracing functions, arrays, or variables.

Specifying `x` traces the properties till level x.

Arguments

x_depth	Indentation level to which property(s) should be traced.

Value Returned

`t`	Always returns `t`.

Example

If we define several functions, such that each function in turn calls another function, the call depth can be limited using `tracelevlimit(x_depth)`.

```
defun(func_a() var_a=1 putprop('var_aa 1 'prop_x) println("A") t)
defun(func_b() var_b=2 putprop('var_bb 1 'prop_x) println("B") func_a() t)
defun(func_c() var_c=3 putprop('var_cc 1 'prop_x) println("C") func_b() t)
defun(func_d() var_d=4 putprop('var_dd 1 'prop_x) println("D") func_c() t)
```

We can set the indentation level as 2:

```
tracelevlimit(2) => t
```

So, when we call `func_d()`, the ouput will be:

```
|func_d()
||putprop(var_dd 1 prop_x)
||putprop => 1
||println("D")
D
```

```
||println => nil
||func_c()
"C"
"B"
"A"
||func_c => t
|fun_d => t
t
```

tracelevunlimit

```
tracelevunlimit(
      )
     => t
```

Description

Turns off limiting of the the indentation level and hence the call depth while tracing functions, arrays, or variables.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
tracelevunlimit() => t
```

tracep

```
tracep(
     [ { s_property...| t } ]
     )
     => g_result
```

Description

Turns on tracing of assignments to specified properties. This is an `nlambda` function.

If `t` is passed in, all properties are traced.

Note: Passing `t` as an argument to the `tracep` function is allowed only if you set the internal system variable `traceTEnable` to `t` by typing `sstatus(traceTEnable t)` in the CIW.

If no argument is given all properties being traced are returned. `tracep` prints the evaluated value assigned to the property and its previous value. The output port for tracing is `ptport`, which defaults to `poport`.

Arguments

s_property	Name of property(s) to be traced.
`t`	Trace all properties.

Value Returned

g_result	Properties successfully marked for tracing.

Example

```
tracep myProp
putprop( 'foo 5 'myProp)
|Property myProp on foo set to 5, was nil => 5
untracep( myProp ) => ( myProp )
```

Reference

untracep

tracev

```
tracev(
     [ { s_variable ... | t } ]
     )
     => g_result
```

Description

Turns on tracing of assignments to specified variables. This is an `nlambda` function.

If `t` is passed in, all variables are traced.

Note: Passing `t` as an argument to the `tracev` function is allowed only if you set the internal system variable `traceTEnable` to `t` by typing `sstatus(traceTEnable t)` in the CIW.

If no argument is given all variables being traced are returned. `tracev` prints the unevaluated and evaluated value assigned to the variable. The output port for tracing is `ptport`, which defaults to `poport`.

Note: SKILL++ variables currently cannot be traced with `tracev`.

Arguments

s_variable	Name of the variable(s) to be traced.
`t`	Trace all variables.

Value Returned

g_result	Variables successfully marked for tracing.

Example

```
tracev x             ; Traces the variable x.
x = 5                ; Shows the old and new value of x.
untracev x           ; Clears tracing for the variable x.
tracev t             ; Traces all variable assignments.
untracev t           ; Clears all variable tracing.
```

Reference

tracef, untrace, untracev

unbreakpt

```
unbreakpt(
     [ { u_function... | t } ]
     )
     => g_result
```

Description

Clears breakpoints. This is an `nlambda` function.

The SKILL debugger is the default break handler and is entered automatically before evaluating functions or function objects with breakpoints set. The functions `breakpt` and `unbreakpt` set and clear breakpoints on the given functions and function objects. Once you are in the break handler, you can examine the state of the program and use single stepping functions such as `step`, `next`, `stepout`, or `continue`.

We recommend that you turn debug mode on before using the break handler with sstatus`( debugMode t )` to change the prompts to tell you when you enter and exit the break handler. Another way to enter the break handler is to insert the `break` function directly into a SKILL function at the point desired.

Arguments

u_function	Function or `t` to `unbreakpt` all functions.

Value Returned

g_result	List of functions or function objects whose breakpoints have been cleared.

Example

This example sets a breakpoint, enters the break handler, continues, and finally clears the breakpoint.

```
breakpt( times )              ; Sets a breakpoint on times.
sstatus( debugMode t )        ; Make sure debug mode is on.
times(2 plus( 2 3 ))          ; Invokes the break handler.
cont                          ; Continue executing.
unbreakpt( times )            ; Clear the breakpoint.
```

This example sets a breakpoint on a function object and later clears the breakpoint.

```
(installDebugger)               ; Make sure debug mode is on
=> t
(defun test (x) (printf "test : x == %L\n" x))
=> test
(putd 'test1 (getd 'test))     ;test and test1 are the same functions
=> funobj:test
(funcall 'breakpt (getd 'test))     ;set breakpoint on the funobject
(funobj:test)
(test1 8)
<<< Break >>> on entering test
Entering new debug toplevel due to breakpoint:
Debug 2> cont
test : x == 8
=> t
(funcall 'unbreakpt (getd 'test))   ;remove breakoint
(funobj:test)
(test1 8)
test : x == 8
=> t
```

Reference

break, breakpt, cont, continue, next, sstatus, step, stepend, stepout

unbreakptMethod

```
unbreakptMethod(
     [ s_name [ @before | @after ] ]
     [ l_specializer ]
     )
     => t
```

Description

Removes breakpoints set on the specified method.

Arguments

s_name	The method for which the breakpoint has to be cleared.
l_specializer	List of specializers of the specified method. It can include `@before`, `@after`, and `@around` qualifiers.

Value Returned

`t`	List of removed breakpoints.

Example

```
unbreakptMethod(S_name)
; removes the breakpoint on "S_name" function, behaves the same way as unbreakpt()

unbreakptMethod(S_name nil)
; removes the breakpoint on "generic" method of S_name.

unbreakptMethod(S_name @after (classA t))
; removes the breakpoint on "@after" method specialized on (classA t) of "S_name"

unbreakptMethod(S_name (classB t))
; removes the breakpoint on primary method specialized on (classB t) of "S_name".

unbreakptMethod()
; when specified without arguments, returns t
```

uncount

```
uncount(
     [ { s_function ... | t } ]
     )
     => g_result
```

Description

Turns off counting and returns the current count results. This is an `nlambda` function.

`count` allows you to count the number of times a function has been called. `count` and `uncount` measure function call frequency and also serve as a valuable debugging aid. Both `count` and `uncount` accept more than one function argument, or `t` for counting/uncounting all functions.

To examine the number of times a function has been counted, call the `uncount` function. A list containing the number of times each function was called, along with the function name, is returned in the form of a list of `(number functionName)` pairs, such as,

```
((20 plus) (10 times) (5 greaterp))
```

The sublists are sorted by their first elements, using `sortcar`, so the most frequently executed functions always appear at the head of the list.

Arguments

s_function	Turns off counting for the functions given.
`t`	Turns off counting for all functions.

Value Returned

g_result	List containing the number of times each function was called, along with the function name.

Example

```
count( plus greaterp )        ; Counts plus and greaterp
setq( x plus( 2 3 ))          ; Use the functions being counted.
uncount( plus greaterp)
=> ((1 plus)(0 greaterp))     ; Number of times each
                              ; function was called.
```

Reference

count

uninstallDebugger

```
uninstallDebugger(
     )
     => t / nil
```

Description

Uninstalls the SKILL debugger as the error handler. Turns off debug mode.

Restores the normal system error handler, which prints the error message and returns to the nearest toplevel. Also turns debug mode off and restores write protection on all functions.

Arguments

None.

Value Returned

`t`	If the debugger is successfully uninstalled.
`nil`	Otherwise.

Example

```
installDebugger()      ; Installs the SKILL debugger.
4 / 0                  ; Generates an error and enters
                       ; the SKILL debugger.
debugQuit()            ; Exits the debugger.
uninstallDebugger( )   ; Restores the old error handler.
```

Reference

debugQuit, installDebugger

untrace

```
untrace(
     [ { s_function | t_fileName ... | t } ]
     )
     => g_result
```

Description

Turns tracing off for all functions specified that were traced using the `tracef` function. This is an `nlambda` function.

If the argument is a string, `untrace` assumes it is a file name and checks if the file was loaded after debug mode was turned on. If it was, it untraces all functions defined in that file.

Arguments

s_function	Any function that should no longer be traced.
t_filename	Any file containing functions that should no longer be traced.
`t`	Turns off tracing for all functions that had tracing turned on.

Value Returned

g_result	List of the functions or files that were untraced.

Example

```
untrace( plus)
```

Turn off tracing for the `plus` function.

```
untrace( "testfns")
```

Turns off tracing for all functions in the `testfns` file assuming it was loaded after debug mode was turned on.

```
untrace( t )
```

Clears all tracing.

Reference

`tracef`, `tracev`, `untracev`

untracep

```
untracep(
     [ { S_property...| t } ]
     )
     => g_result
```

Description

Turns off tracing of the specified properties. This is an `nlambda` function.

Arguments

S_property	Name of the property(s) to be untraced.
`t`	Untrace all properties.

Value Returned

g_result	Properties successfully marked for untracing.

Example

```
tracep myProp
putprop( 'foo 5 'myProp)
|Property myProp on foo set to 5, was nil
=> 5
untracep( myProp ) => ( myProp )
```

Reference

tracep

untracev

```
untracev(
     [ { s_variable ... | t } ]
     )
     => g_result
```

Description

Turns off tracing for assignments to specified variables. This is an `nlambda` function.

Arguments

s_variable	Name of the variable(s) to be untraced.
t	Untrace all variables.

Value Returned

g_result	Variables successfully marked for untracing.

Example

```
tracev x             ; Traces the variable x.
x = 5                ; Shows the old and new value of x.
untracev x           ; Clears tracing for the variable x.
tracev t             ; Traces all variable assignments.
untracev t           ; Clears all variable tracing.
```

Reference

tracev, tracef, untrace

unwatch

```
unwatch(
     [ s_symbol...]
     )
     => t
```

Description

Clears watchpoints set on the specified variables.

Arguments

s_symbol	Name of the variables for which the watchpoints need to be cleared. This argument is optional.

Value Returned

`t`	Watchpoints successfully cleared.

Example

```
unwatch( x ) => t
```

watch

```
watch(
     [ s_symbol...]
     )
     => t / l_watchedVars / nil
```

Description

Sets watchpoints on the specified variables. If `watch()` is called without arguments, it returns the list of variables being watched. If no variables are being watched, it returns `nil`.

Arguments

s_symbol	The variables to be watched. This argument is optional.

Value Returned

`t`	Variables successfully marked for watching.
l_watchedVars	If called without arguments, returns the list of variables being watched.
`nil`	No variables are being watched.

Example

```
watch(x) => t
watch()=> (x)
watch() => nil
```

where

```
where(
     [ g_unevaluated ]
     [ x_depth ]
     [ x_skip ]
     [ p_port ]
     )
     => x_result
```

Description

Prints the functions on the stack and their arguments to the depth specified, or to the bottom of the stack, including the local variables and their bindings.

It is similar to `stacktrace`, but in addition to printing out the functions on the stack it also prints out the local variables and their bindings. The `where` function observes the following rules:

- When debug mode is on and the `traceArgs` status switch has been set to non-`nil`, prints the evaluated function arguments unless *`g_unevaluated`* is set to `t`.
- When debug mode is off, `where` always prints the unevaluated arguments.
- `where` skips over the number of function calls specified by *`x_skip`*.
- If there are no functions on the stack (you are at the top, for example) `where` does not print anything and returns 0.

Prerequisites

This function is usually used inside the break or error handler.

Arguments

g_unevaluated	If `t`, prints the unevaluated function parameters.
x_depth	Number of stack levels to print, defaults to all.
x_skip	Number of levels to skip, defaults to 1.
p_port	Output port. Defaults to the error port.

Value Returned

x_result	Number of stack frames printed.

Example

Suppose `/tmp/color.il` defines function `initColor`:

```
(defun initColor (object)
  (let ((colorList '(red green yellow)) color)
       (setq color (concat (get object 'color)))
       (if (memq color colorList)
           (printf "color %s initialized" (get object 'name)))
  )
)
```

Try this file in debugger:

```
installDebugger
=> t
1> (sstatus sourceTracing t)
                     ; Turns on sourceTracing to get line numbers
=> t

1>  load "/tmp/color.il"
=> t

1> (putprop 'object1 "green" 'color)
=> "green"

1> (initColor 'object1)
*** Error in routine fprintf/sprintf:
Message: *Error* fprintf/sprintf: format spec. incompatible
with data

Debug 2> where
<<< Stack Trace >>>
errorHandler("fprintf/sprintf" 0 t nil ("*Error* fprintf/sprintf: format spec.
incompatible with data" nil))
printf("color %s initialized" get(object 'name))
at line 5 in file /tmp/color.il
if(memq(color colorList) printf("color %s initialized" get(object 'name)))
```

```
let(((colorList '&) color) (color = concat(get(object &))) if(memq(color colorList)
printf("color %s initialized" get(object &))))
        colorList = (red green yellow)
        color = green
        object = object1
initColor('object1)
5
```

Reference

sstatus, stacktrace

wherels

```
whereIs(
     s_function
     )
     => t / nil
```

Description

Prints the last file loaded in debug mode in which a function was defined, as well as the starting line number.

Prerequisites

Turn on debug mode before loading the file.

Arguments

s_function	Function whose file you want to locate.

Value Returned

`t`	If the function is found.
`nil`	Otherwise.

Example

```
whereIs( myFunction ) => t
```

Returns `t` if the function was found and prints the file and starting line number of the `myFunction` function if it was loaded after debug mode was turned on. A sample output is

```
Function myFunction was loaded from file ~/myFunctions.il at line 126.
```

Reference

`getFunctions`

6

Finder Functions

startFinder

```
startFinder(
     [ ?funcName t_funcName ]
     )
     => t / nil
```

Description

Starts the SKILL documentation Finder utility. If the `t_funcName` argument is provided, the corresponding documentation of the function is displayed in the Finder. If a Finder window is already existing, it will be updated or a new window will be displayed.

Note: The Finder window remains open, unless specifically closed, even after the parent window has been closed.

Arguments

`?funcName` *`t_funcName`*	Name of the function to be searched in the Finder

Value Returned

`t`	Finder is launched successfully and documentation of `t_funcName` is returned.
`nil`	Otherwise.

Example

To start the Finder utility.

```
startFinder
=> 0
```

To display the documentation of the function in the Finder utility.

```
startFinder(?funcName "")
=> t
```

fndResetDb

```
fndResetDb(

     )
     => t
```

Description

Resets previously loaded Finder database.

Arguments

None.

Value Returned

`t`	Finder database is successfully reset.

Example

```
help('myFunction)
=>  nil
; documentation for myFunction is not yet loaded (added after finder init).
fndResetDb()
;reset database
help('myFunction)
; documentation is available now
```

7

Tabulator Functions

skTabulate

```
skTabulate(
     ?fileNames g_tabulatedFileNames
     [ ?reportFile t_reportFile ]
     [ ?showFile g_showFile ]
     [ ?dontResolveSymLink g_dontResolveSymLink ]
     [ ?customerInfo l_customerInfo ]
     [ ?printFiltered g_printFiltered ]
     [ ?ext l_ext ]
     [ ?infoFile g_infoFile ]
     [ ?defnFile g_defnFile ]
     [ ?recurse g_recurse ]
     [ ?exclude g_exclude ]
     [ ?recurseExclude g_recurseExclude ]
     [ ?sendReport g_sendReport ]
     => t
```

Description

Runs the SKILL Tabulator in batch mode.

Arguments

`?fileNames` *`g_tabulatedFileNames`*

The names of the files or directories to be tabulated. Should be a string or a list of strings.

`?reportFile` *`t_reportFile`*

The name of the generated report file. Defaults to `"./skillTab.out"`.

`?showFile` *`g_showFile`*

Specifies whether to display the generated report file. Defaults to `nil`.

`?dontResolveSymLink` *`g_dontResolveSymLink`*

Specifies whether to resolve symbolic links. Defaults to `nil`.

`?customerInfo` *`l_customerInfo`*

A disembodied property list containing customer information, in the following form:

```
'(nil customerEmail <t_email>
customerName  <t_name>
companyLocation <t_location>
companyName <t_compName>
customerPhone <t_phone>
custProjName <t_projName>
custProjInfo <t_projInfo>)
```

`?printFiltered` *`g_printFiltered`*

Specifies whether to print user defined functions. Defaults to `t`.

`?ext` *`l_ext`*

List of file extensions to be tabulated. The default options are `".il"`, `"ile"`, and `"cdsinit"`.

`?infoFile` *`g_infoFile`*

The file name for the information file, which contains the names of the files that refer each function.

`?defnFile` *`g_defnFile`*

The file name for the definition file, which contains the names of the files that define each function.

`?recurse` *`g_recurse`*

Specifies whether to recurse directories. Defaults to `t`.

`?exclude` *`g_exclude`*

List of files to be excluded from tabulation. Defaults to `nil`.

`?recurseExclude` *`g_recurseExclude`*

List of recursively excluded files. Defaults to `nil`.

`?sendReport` *`g_sendReport`*

Specifies whether to send the generated report to Cadence. Defaults to `nil`.

Value Returned

`t` Always returns `t`.

Example

```
skTabulate(
    list("./file1.il" "file2.il")
    ?reportFile "./myTabOutput.txt"
    ?showFile nil
    ?dontResolveSymLink nil
    ?customerInfo '(nil customerEmail "user@xyz.com" customerName "AN"
       companyLocation "SJ" companyName "myCompany" customerPhone "123456"
       custProjName "" custProjInfo "")
    ?printFiltered t
    ?defnFile "./myDefnFile.txt"
    ?recurseExclude nil
    ?infoFile "./myInfoFile.txt"
    ?ext '("il")
    ?sendReport t
     );
```

skTabulateSKILL

```
skTabulateSKILL(
     )
     => t
```

Description

Brings up the SKILL Tabulator UI form.

The SKILL Tabulator is part of the SKILL Surveyor program. Users should access the SKILL Tabulator through SKILL Surveyor, part of the Conversion tool box.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
skTabulateSKILL() => t
```

8

SKILL IDE Functions

ilgInvokeIDE

```
ilgInvokeIDE(
     )
     => t
```

Description

Displays the SKILL IDE main window.

Arguments

None.

Value Returned

`t`	Always returns `t`.

Example

```
ilgInvokeIDE()
=> t
```

ilgRunSKILLIDE

```
ilgRunSKILLIDE(
     [ ?fileList lt_fileList ]
     )
     => t / nil
```

Description

Launches SKILL IDE and opens the files listed in *lt_fileList*.

Arguments

`?fileList` *lt_fileList*

A list of file names (strings) that need to be opened in SKILL IDE.

Value Returned

`t`	SKILL IDE was launched successfully.
`nil`	SKILL IDE could not be launched.

Example

```
ilgRunSKILLIDE(?fileList (list "demo.il"))
=> t
```

ilgLastDir

```
ilgLastDir(
     )
     => t_dirPath
```

Description

Returns the directory path of the file currently open in SKILL IDE.

Arguments

None.

Value Returned

t_dirPath	Directory path of the file currently open in SKILL IDE.

Example

```
ilgLastDir()
=> /home/usr1
```

ilgAddRecentFiles

```
ilgAddRecentFiles(
     l_fileList
     )
     => t
```

Description

Adds the files listed in `l_fileList` to the SKILL IDE *File* menu.

Arguments

`l_fileList`	The list of files to be added to the SKILL IDE *File* menu.

Value Returned

`t`	The listed files were successfully added to the SKILL IDE *File* menu.

Example

```
ilgAddRecentFiles('("loop2.il"))
=> t
```

ilgAppendText

```
ilgAppendText(
     t_text
     [ w_tab ]
     )
     => t / nil
```

Description

Inserts text into the SKILL IDE editor window at the current cursor location.

Arguments

t_text	The text to be inserted.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	The text was successfully inserted.
`nil`	The text was not inserted.

Example

```
ilgAppendText("hello IDE")
=> t
```

ilgCopy

```
ilgCopy(
     [ w_tab ]
     )
     => t
```

Description

Copies the selected text into public text buffer. This function does not work on read-only files.

Arguments

w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	The text was successfully copied.
`nil`	The operation was not successful because the file is read-only.

Example

```
ilgCopy( )
=> t
; text copied into public text buffer
```

ilgCut

```
ilgCut(
     [ w_tab ]
     )
     => t / nil
```

Description

Cuts and pastes the selected text into public text buffer. This function does not work on read-only files.

Arguments

w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

t	The text was successfully cut.
nil	The text was not cut because the file was read-only.

Example

```
ilgCut( )
=> t
```

ilgFindIdent

```
ilgFindIdent(
     x_column
     x_row
     [ ?tab w_tab ]
     )
     => l_coord / nil
```

Description

Returns the nearest identifiers from a given opening parenthesis location '(' in the SKILL IDE editor tab.

Arguments

x_column	Column co-ordinates of the opening parenthesis '('.
x_row	Row co-ordinates of the opening parenthesis '('.
`?tab` *w_tab*	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

l_coord	List containing two identifier strings, one before the opening parenthesis and the other after it.
`nil`	Returns `nil` if *x_column* and *x_row* are not coordinates of '('.

Example

If the first row in SKILL IDE tab is: `a ( b ((c)) ) d ( "e" )`

```
ilgFindIdent(1 1) => nil
ilgFindIdent(3 1) => ("a" "b")
ilgFindIdent(4 1) => nil
ilgFindIdent(7 1) => ("b" "")
ilgFindIdent(8 1) => ("" "c")
ilgFindIdent(17 1) => ("d" "\"e\"")
```

ilgFindParenthesis

```
ilgFindParenthesis(
     x_column
     x_row
     [ ?level S_level ]
     [ ?direction S_direction ]
     [ ?tab w_tab ]
     )
     => l_coord / nil
```

Description

Returns the closest parenthesis in a file currently open in the SKILL IDE editor.

Arguments

x_column	Column co-ordinates of the specified location.
x_row	Row co-ordinates of the specified location.
?level *S_level*	Specifies whether to search the closest `outer` or `inner` parenthesis in a file. Default is `outer`.
?direction *S_direction*	Search direction (symbol). Valid values are `forward` and `backward`. Default is `forward`.
?tab *w_tab*	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

l_coord	List containing the co-ordinates of the pair of parenthesis.
`nil`	Returns `nil` if no parenthesis pair was found.

Example

If the first row in SKILL IDE tab is: `"a ( b ((c)) ) d"`

```
ilgFindParenthesis(1 1) => nil
ilgFindParenthesis(1 1 ?level 'inner) => ((3 1) (9 1))
```

```
ilgFindParenthesis(4 1 ?level 'outer) => ((3 1) (9 1))
ilgFindParenthesis(10 1 ?level 'inner ?direction 'backward) => ((3 1) (9 1))
```

ilgFoldLine

```
ilgFoldLine(
     [ x_column ]
     [ x_row ]
     )
     => t / nil
```

Description

Folds a document block in SKILL IDE at (`x_column x_row`) where (`x_column x_row`) is the location of an opening parenthesis. If this function is called without arguments, the current cursor location is used for computing (`x_column x_row`).

Arguments

x_column	Column co-ordinates of the opening parenthesis '('.
x_row	Row co-ordinates of the opening parenthesis '('.

Value Returned

`t`	Returns `t` if the document block was successfully folded.
`nil`	Returns `nil` otherwise.

Example

```
[-]defun(test (x y "xx")
"test function"
x + y
4    )
ilgFoldLine(6 1)
=> t
```

ilgUnfoldLine

```
ilgUnfoldLine(
     [ x_column ]
     [ x_row ]
     )
     => t / nil
```

Description

This function unfolds a document block in SKILL IDE at position (`x_column` `x_row`), if it was folded. Here (`x_column` `x_row`) is the location of an opening parenthesis. If this function is called without arguments, the current cursor location is used for computing (`x_column` `x_row`).

Arguments

x_column	Column co-ordinates of the opening parenthesis '('.
x_row	Row co-ordinates of the opening parenthesis '('.

Value Returned

`t`	Returns `t` if the document block was successfully unfolded.
`nil`	Returns `nil` otherwise.

Example

If code block at line 1 is folded:

```
[+]defun(test (x y)
)
ilgUnfoldLine(6 1)
=> t
```

ilgGetCursorLocation

```
ilgGetCursorLocation(
     [ w_tab ]
     )
     => l_location
```

Description

Returns the location of the cursor in the SKILL IDE editor window.

Arguments

w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

l_location	Cursor location in the current or specified tab.

Example

```
ilgGetCursorLocation( )
=> (9 11)
```

ilgGetEditLock

```
ilgGetEditLock(
     [ w_tab ]
     )
     => S_mode
```

Description

Returns the read-only property for a SKILL IDE editor window.

Arguments

w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

s_mode	Can be one of the following: `lock`: Cannot edit text in the tab. `partialLock`: Cannot edit the text manually, but can edit it programmatically with `ilgAppendText`, `ilgCut`, and `ilg-Paste`. `unlock`: Document is editable.

Example

```
ilgGetEditLock( )
=> unlock
```

ilgGetHighlight

```
ilgGetHighlight(
     [ ?loc l_location ]
     [ ?tab w_tab ]
     )
     => l_highlightIDs / nil
```

Description

Returns the list of highlight IDs for the text highlighted in the SKILL IDE editor window.

Arguments

`?loc l_location`	Location (`x_column x_row`) of the area highlighted in the SKILL IDE editor window.
`?tab w_tab`	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`l_highlightIDs`	A list representing the highlight IDs of the highlighted text.
`nil`	Returns if nothing is highlighted in the given area.

Example

```
ilgSetHighlight('(4 11) '(18 12) "orange")
=> (nil l1 (4 11) l2 (18 12)
color "orange" fullWidth nil)
;highlights the area between (4 11) and (18 12)
ilgGetHighlight(?loc '(4 11))
=>  ((nil l1
    (4 11) l2
    (18 12)
    color "#ffa500" fullWidth nil
    )
    (nil l1
    (4 11) l2
    (2 13)
    color "#ffa500" fullWidth nil
    )
)
```

ilgGetSelectedLocation

```
ilgGetSelectedLocation(
     [ w_tab ]
     )
     => l_location / nil
```

Description

Returns the location co-ordinates of the current selection in the SKILL IDE editor window.

Arguments

w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

l_location	Location co-ordinates of the current selection in the SKILL IDE editor window.
`nil`	Returns `nil` if nothing is selected in the SKILL IDE editor window.

Example

```
ilgGetSelectedLocation( )
=> ((1 11)
    (4 11)
    )
```

ilgGetText

```
ilgGetText(
     [ l_location_begin ]
     [ l_location_end ]
     [ w_tab ]
     )
     => t_text
```

Description

Returns the text between *l_location_begin* and *l_location_end*. If these locations are not provided, the entire text in the SKILL IDE editor window is returned.

Arguments

l_location_begin	Column co-ordinates of the cursor location.
l_location_end	Row co-ordinates of the cursor location.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

t_text	The text between *l_location_begin* and *l_location_end*. If *l_location_begin* and *l_location_end* are not provided, the entire text in the SKILL IDE editor window is returned.

Example

```
ilgGetText()
=> "hello ide!"
```

Note: If the specified locations are such that *l_location_end* precedes *l_location_begin*, the *l_location_begin* and *l_location_end* co-ordinates are swapped, so that `ilgGetText` returns the text between *l_location_begin* and *l_location_end*. For example:

```
ilgGetText ( '(1 1) '(5 1) )
=> "(pro"
```

```
ilgGetText ('(5 1) '(1 1) )
=> "(pro"
```

ilgPaste

```
ilgPaste(
     [ w_tab ]
     )
     => t / nil
```

Description

Pastes the text from the clipboard/buffer to the cursor location in the SKILL IDE editor window. This function does not work on read-only files.

Arguments

w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	Returns `t` if the text was pasted to the current cursor location.
`nil`	Returns `nil` if the document in the SKILL IDE editor window is read-only.

Example

```
ilgPaste()
=> t
```

ilgPositionInComment

```
ilgPositionInComment(
     x_column
     x_row
     [ ?tab w_tab ]
     )
     => t / nil
```

Description

Checks if the specified co-ordinates (`x_column` `x_row`) fall within a comments block in the SKILL IDE document.

Arguments

x_column	Column co-ordinates of the specified location.
x_row	Row co-ordinates of the specified location.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	Returns `t` if the specified co-ordinates (`x_column` `x_row`) fall within a comments block in the SKILL IDE document.
`nil`	Returns `nil`, otherwise.

Example

If the SKILL IDE editor window has the following text:

```
defun(test (x y "xx")
/* function test */
x + y
) ; return sum x and y
```

```
ilgPositionInComment(1 1)
=> nil
```

```
ilgPositionInComment(10 2)
=> t
```

```
ilgPositionInComment(8 4)
=> t
```

ilgRegisterSelectionCB

```
ilgRegisterSelectionCB(
     g_name
     )
     => t
```

Description

Registers a SKILL callback for SKILL IDE, which is called when some text is selected in the SKILL IDE editor window.

Arguments

g_name	The function to be registered as a callback. It can be either a symbol, function name, or a lambda function. The function being registered should accept 3 arguments: two lists (begin and end selection location) and window variable (current tab window).

Value Returned

`t`	Returns `t` if the function has been registered as a callback.

Example

To register a selection callback:

```
defun(mySelCallback (l_begin  l_end  wTab)
printf("Selected text %L - %L in tab window %L\n"
l_begin l_end wTab))
ilgRegisterSelectionCB('mySelCallback)
=> t
```

ilgSetErrorMarker

```
ilgSetErrorMarker(
     x_line
     t_description
     [ w_tab ]
     )
     => t
```

Description

Sets the error marker with the pop-up description *t_description* on line *x_line* of the SKILL IDE editor window.

Arguments

x_line	Line number on which the error marker needs to be set.
t_description	Description of the error marker.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

t	Returns t if the error marker has been set.

Example

```
ilgSetErrorMarker(4 "this line is marked")
=> t
; Sets the error marker on line 4.
```

ilgResetErrorMarker

```
ilgResetErrorMarker(
     x_line
     [ w_tab ]
     )
     => t
```

Description

Clears the error marker that was set by `ilgSetErrorMarker()` on line *x_line* in the SKILL IDE editor window.

Arguments

x_line	Line number from which the error marker needs to be cleared.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	Returns `t` if the error marker has been cleared.

Example

```
ilgResetErrorMarker(4)
; clears the error marker at line 4
```

ilgSetWarningMarker

```
ilgSetWarningMarker(
     x_line
     t_description
     [ w_tab ]
     )
     => t
```

Description

Sets the warning marker with the pop-up description *t_description* on line *x_line* of the SKILL IDE editor window.

Arguments

x_line	Line number on which the warning marker needs to be set.
t_description	Description of the warning marker.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

t	Returns t if the warning marker has been set.

Example

```
ilgSetWarningMarker(4 "this line is marked")
=> t
; Sets the warning marker on line 4.
```

ilgResetWarningMarker

```
ilgResetWarningMarker(
     x_line
     [ w_tab ]
     )
     => t
```

Description

Clears the warning marker that was set by `ilgSetWarningMarker()` on line *`x_line`* in the SKILL IDE editor window.

Arguments

`x_line`	Line number from which the warning marker needs to be cleared.
`w_tab`	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	Returns `t` if the warning marker has been cleared.

Example

```
ilgResetWarningMarker(4)
; clears the warning marker at line 4
```

ilgSetHighlight

```
ilgSetHighlight(
     l_location_begin
     l_location_end
     t_color
     [ ?fullWidth g_fullWidth ]
     [ w_tab ]
     )
     => l_highlightID / nil
```

Description

Highlights the area within *l_location_begin* and *l_location_end* in the color specified by *t_color*.

Arguments

l_location_begin	Begin location (`x_column x_row`) of the area to be highlighted.
l_location_end	End location (`x_column x_row`) of the area to be highlighted.
t_color	A string representing the color name, which can either be a predefined color (for example, Highlight[1-5]) set by `ilgSetColor` or a color value understood by QT.
`?fullWidth` *g_fullWidth*	When set to `t`, the entire line is highlighted, default value is `nil`.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

l_highlightID	A list representing the highlight operation; this value can be passed to `ilgResetHighlight()`.
`nil`	Returns `nil` if the highlight operation fails.

Example

```
ilgSetHighlight('(4 11) '(18 12) "orange")
=> (nil l1 (4 11) l2 (18 12)
color "orange" fullWidth nil)
;highlights the area between (4 11) and (18 12)
```

```
ilgSetHighlight('(5 22) '(15 24) "forest green" ?fullWidth t)
=> (nil l1 (5 22) l2 (15 24)
color "forest green" fullWidth t
)
;highlights the entire line starting at (5 22)
```

ilgResetHighlight

```
ilgResetHighlight(
     l_highlightID
     [ w_tab ]
     )
     => t
```

Description

Resets highlight in a SKILL IDE document between location *l_location_begin* and *l_location_end*.

Arguments

l_highlightID	Line number from which the error marker needs to be cleared.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

t	Returns t if the error marker has been cleared.

Example

```
ilgResetHighlight(myHighlightID_1)
=> t
```

ilgSearchText

```
ilgSearchText(
     t_text
     [ ?loc l_location ]
     [ ?direction s_direction ]
     [ w_tab ])
     => l_location / nil
```

Description

Searches the specified text in the SKILL IDE document.

Arguments

t_text	Text to be searched.
?loc *l_location*	Location co-ordinates of the search area. By default, search area is the whole document.
?direction *s_direction*	Search direction (symbol). Valid values are `forward` and `backward`. Default is `forward`.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

l_location	List of all occurrences of `t_text` in the search area.
`nil`	Returns `nil`, otherwise.

Example

To search for the word IDE in the SKILL IDE window

```
ilgSearchText("IDE")
=>
((7 1)
    (11 4)
    (1 7)
```

```
    (1 9)
)
```

ilgSelectText

```
ilgSelectText(
     l_location_begin
     l_location_end
     [ w_tab ]
     )
     => t
```

Description

Selects the text between location *l_location_begin* and *l_location_end* in the SKILL IDE editor window.

Arguments

l_location_begin	Begin location (x_column x_row) of the text to be selected.
l_location_end	End location (x_column x_row) of the text to be selected.
w_tab	The window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

t	Returns t, if the text was successfully selected.

Note: The function throws an error if the input arguments are incorrect.

Example

To select the text from (line 2 column 1) to (line 10 column 8) in the SKILL IDE window

```
ilgSelectText('(1 2)  '(8 10))
=> t
```

ilgSetColor

```
ilgSetColor(
     t_text
     t_value
     [ x_alpha ]
     )
     => t / nil
```

Description

Sets a custom color for the given SKILL IDE GUI control.

Arguments

t_text

SKILL IDE GUI control name. Can be one of the following:

`Step, Error, Cross, SelectPattern, MatchParent, MismatchParent, Keyword, KeywordBg, Comment, CommentBg, Number, NumberBg, String, StringBg, Text, TextBg, TextArea, Highlight1, Highlight2, Highlight3, Highlight4, Highlight5`

t_value

A string representing the color name.

- #RGB (each of R, G, and B is a single hex digit)
- #RRGGBB
- #RRRGGGBBB
- #RRRRGGGGBBBB
- A name from the list of colors defined in the list of SVG color keyword names provided by the World Wide Web Consortium. For example, "steelblue" or "gainsboro". These color names work on all platforms.

Note: These color names are not the same as defined by Qt.

GlobalColor enums, for example, green and Qt:green do not refer to the same color.

- Transparent - Represents the absence of a color.
- X11 only: Any valid X11 color name.

The color is invalid if name cannot be parsed.

x_alpha

A value representing the alpha color component (in range 0 - 255).

Value Returned

`t`

Returns `t`, if the color was successfully set.

`nil`

Returns `nil`, otherwise.

Example

To set the color of text in the SKILL IDE window to “steelblue”

```
ilgSetColor("Text" "steelblue")
=> t
```

ilgScrollToLocation

```
ilgScrollToLocation(
     l_location
     [ w_tab ]
     )
     => t / nil
```

Description

Scrolls to the specified location in the specified SKILL IDE editor window or tab.

Arguments

l_location	Coordinates of the location to scroll to. The list is in the format *x_column x_row*.
w_tab	ID of the SKILL IDE editor window or tab. If not specified, the current tab is considered.

Value Returned

`t`	Scrolls to the specified location.
`nil`	Returns `nil`, otherwise.

Example

To scroll to column `2` and row `30` in the current SKILL IDE window:

```
ilgScrollToLocation(`(2 30))
=> t
```

ilgSetCursorLocation

```
ilgSetCursorLocation(
     l_location
     [ w_tab ]
     )
     => t / nil
```

Description

Sets the cursor location in a SKILL IDE document.

Arguments

l_location	Location co-ordinates of the cursor.
w_tab	Window ID of the SKILL IDE editor window. Default is the current tab window.

Value Returned

`t`	Sets the cursor location successfully.
`nil`	Returns `nil`, otherwise.

Example

To set the cursor location in the SKILL IDE window

```
ilgSetCursorLocation('(9 11))
=> t
ilgSetCursorLocation('(37 4))
=> nil
```

ilgSetEditLock

```
ilgSetEditLock(
     s_mode
     [ w_tab ]
     )
     => t
```

Description

Changes the read-only property for a SKILL IDE editor window.

Arguments

S_mode	Edit lock mode. Valid values are: `lock`: Cannot edit text in the tab. `partialLock`: Cannot edit the text manually, but can edit it programmatically with `ilgAppendText`, `ilgCut`, and `ilg-Paste`. `unlock`: Document is editable. **Note:** The mode cannot be set to `unlock` or `partialLock` for a read-only file.
w_tab	Window ID of the SKILL IDE editor window. Default is the current tab window

Value Returned

`t`	Returns `t` if the operation is successful.
`nil`	Returns `nil`, otherwise.

Example

To lock the SKILL IDE editor window

```
ilgSetEditLock('lock)
=> t
```

ilgUnregisterSelectionCB

```
ilgUnregisterSelectionCB(
     [ S_name ]
     )
     => t / nil
```

Description

Unregisters a selection callback that was previously registered using `ilgRegisterSelectionCB`.

Arguments

S_name	Name of the callback function.

Value Returned

`t`	Returns `t` if the function is unregistered successfully.
`nil`	Returns `nil`, otherwise.

Example

To unregister a selection callback:

```
ilgUnregisterSelectionCB('mySelCallback)
=> t
```

To unregister all selection callbacks:

```
ilgUnregisterSelectionCB() ;call without arguments
=> t
```