SKILL Development Help

Product Version 06.01
November 2001

Contents

1
Cadence SKILL Language Development Help 9
Overview 10
Close Toolbox for Automatic License Check in/out 11

2
SKILL Debugger 13
Overview 14
SKILL Debugger Commands 15
Dump 15
Stacktrace 15
Where 15
Step 15
Next 15
Step Out 15
Exit Debug Toplevel 15
Continue 16
Tracing 16
Set Breakpoints 16
Debug Status 16
Clear 16
Automatic Stacktrace (Levels) 16
Enter New Debug Toplevel on Error 16
Terminate Debugging and Quit Debugger 16

3
SKILL Lint 17
Overview 17
Form Options 18
Input File 18

Context Name 18
Package Prefixes 18
Output 19
Output File 20
Check For 20
Customize Messages Form 22
Message Groups 22
Messages 24
SKILL Lint PASS/FAIL and IQ Algorithms 31
SKILL Function 32

4
SKILL Profiler 37
Overview 38
Using the SKILL Profiler 38
File Menu 39
Open Browser 39
Save As 39
Search 40
Close Window 41
Profile Menu 41
Start Profiling Time 41
Start Profiling Memory 42
Stop Profiling 42
Reset Profiling 42
Options Menu 43
Setup 43
Filters 44
Fixed Menu 46
SKILL Functions 46

5
Code Browser 47
Overview 48
Function To Expand 48

Code Browser Window 49
Commands Menu 49
    Expand Function 49
    Find Function 50
    Delete All 50
    Close 50
Misc Menu 51
    Defaults 51
    Filters 51
Pop-up Functions Menu 51
    View 51
    Expand Functions 52
    Expand Deep 52
    Unexpand 52
    Delete 52

6
Tracing 53

Overview 53
Tracing Form 54
    Tracing 54
    Type 54
    Trace Symbols 55
SKILL Functions 55

7
Finder 57

Overview 58
    Your database may vary 58
    You can add your own functions 58
    Starting up 58
Searching 58
    Simple Strings 58
    Combinations 59
    Categories 59

- Starting a search . . . 60
- Stopping a search . . . 60

Selecting from Matches . . . 60

Saving Descriptions . . . 61

- Clear . . . 61
- Save... . . . 61

More on Saving . . . 62

Cadence Data . . . 62

Customer Data . . . 63

Data Format . . . 63

Troubleshooting . . . 64

- Too Many Matches . . . 64
- Save File Is Not Writable . . . 64
- No files found . . . 64
- Descriptions Window Full . . . 64
- Font Size Unsatisfactory . . . 65

Starting in UNIX . . . 65

- Standard Mode . . . 65
- Test Modes . . . 65

8

Walkthrough . . . 67

Introduction . . . 67

- Tasks for the Program . . . 68
- Tasks for You . . . 68
- The Test Program . . . 69

Load the Program . . . 70

Run the Program . . . 71

Resolve the First Error . . . 71

Fix the Functional Error . . . 73

Run Lint . . . 75

Measure myFunction1 . . . 77

Check Performance . . . 81

Rerun the Profiler . . . 83

Using the Non-Graphical SKILL Debugger . . . 85

9
Command Line Interface 87
Command Line: Profiler 87
ilProf 87
ilProfFile 87
Command Line: Test Coverage 88
ilTCov 88
ilTCovDir 88
ilTCovReportsOnly 88
TCov Report Files 89
ilTCovSummary 89
contextName.tcovSum 89
fileName.tcov 89
contextName.d 89

10
Writing SKILL Lint Rules 91
Overview 91
Rule Structures - SK_RULE Macro 92
sl_functions 92
g_test 92
g_statement 92
SK_RULE Example 92
Rule Access Macros 93
SK_ARGS() 93
SK_CUR_FILENAME() 93
SK_NTH_ARG(n) 93
SK_FUNCTION() 94
SK_FORM([n]) 94
Rule Reporting Macros 94
Advanced Rule Macros 95
SK_CHANGED_IN(t_release) 95
SK_CHECK_STRINGFORM(t_stringForm) 96
SK_RULE(SK_CONTROL ...) 96

- SK_CHECK_FORM(l_form) 97
- SK_PUSH_FORM(l_form) SK_POP_FORM() 97
- SK_PUSH_VAR(s_var) 98
- SK_POP_VAR(s_var [dont_check]) 98
- SK_USE_VAR(s_var) 98
- SK_ALIAS(s_function s_alias) 98

Storing Rule Definitions 99
- /cds/tools/local/sklint/rules 99
- /cds/tools/sklint/rules 99

Examples 99
- Adding a New Required Argument 99
- Replacing One Function With Another 100
- Promoting Standard Format Messages 101
- Making the Code Look Nicer 101

11

Set Breakpoints Form 103

Overview 103

Function Names 104

Breakpoints 104
- Set 104
- Clear 104

Where 104
- Entry 104
- Exit 104

Condition 104

Clear All Breakpoints 105

SKILL Functions 105

1 Cadence SKILL Language Development Help

This document discusses

- Overview on page 10
- Close Toolbox for Automatic License Check in/out on page 11

The SKILL Development Toolbox tools include

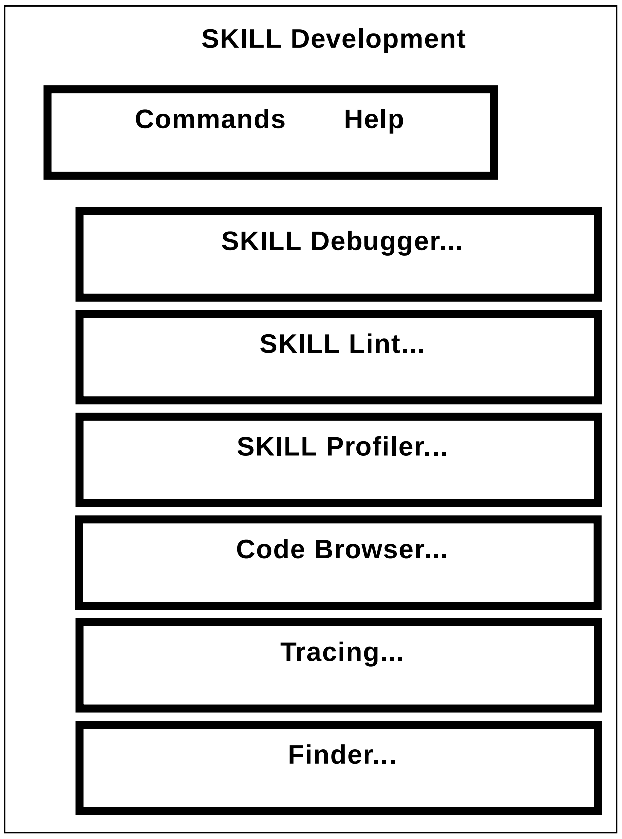
- SKILL Debugger on page 13
- SKILL Lint on page 17
- SKILL Profiler on page 37
- Code Browser on page 47
- Tracing on page 53
- Finder on page 57

See also

- Walkthrough on page 67
- Command Line Interface on page 87
- Writing SKILL Lint Rules on page 91

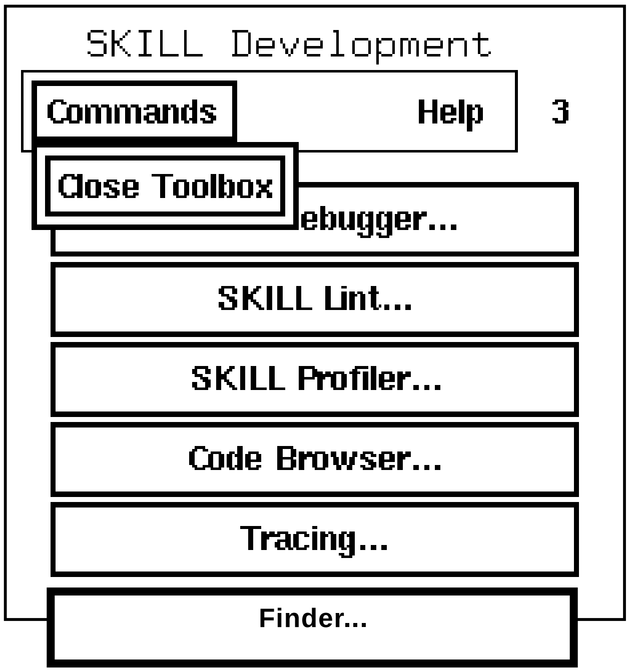
Overview

The Cadence® SKILL language development toolbox provides software tools that reduce the time it takes to develop SKILL code and that improve the efficiency and quality of the code.



Close Toolbox for Automatic License Check in/out

You do not need to check SKILL Development in and out manually. The `skillDev` license is checked out when you open the toolbox and is checked back in when you close the toolbox.



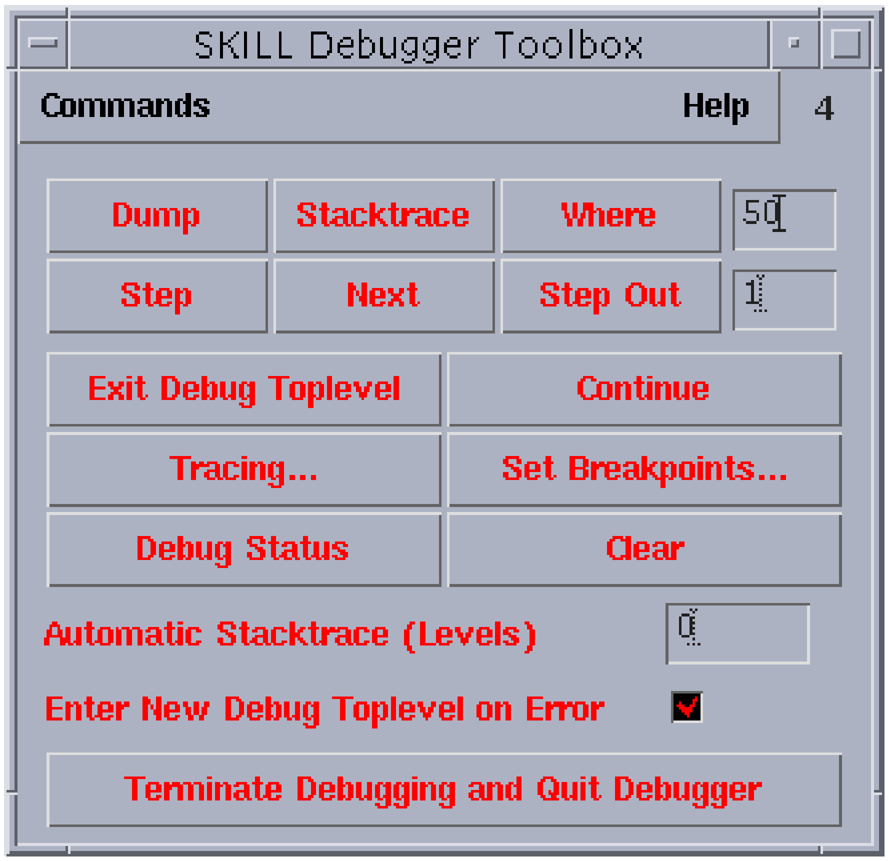
2

SKILL Debugger

This document discusses

- Overview on page 14
- SKILL Debugger Commands on page 15
    - Dump on page 15
    - Stacktrace on page 15
    - Where on page 15
    - Step on page 15
    - Next on page 15
    - Step Out on page 15
    - Exit Debug Toplevel on page 15
    - Continue on page 16
    - Tracing on page 16
    - Set Breakpoints on page 16
    - Debug Status on page 16
    - Clear on page 16
    - Automatic Stacktrace (Levels) on page 16
    - Enter New Debug Toplevel on Error on page 16
    - Terminate Debugging and Quit Debugger on page 16

Overview



To debug your SKILL code, you first bring up the SKILL Debugger Toolbox. The SKILL Debugger program is automatically installed when the toolbox is brought up. After the installation, when you run your code and an error occurs, you enter the debugger. You also load and enter the debugger, regardless of whether it is installed, when you reach a breakpoint.

When you run the code and an error occurs, use the *Dump*, *Stacktrace*, and *Where* commands to display the SKILL stack and local variables. Output appears in the CIW.

To set breakpoints, click on *Set Breakpoints* to bring up the Set Breakpoints form, enter the function names, and click on *OK* or *Apply*. Then run your program. When you reach a breakpoint, use the *Step*, *Next*, and *Step Out* commands to step through the code.

To quit the SKILL Debugger, click *Exit Debug Toplevel* in the SKILL Debugger toolbox. Each time you select *Exit Debug Toplevel*, SKILL exits the most recently entered (nested) debugger session.

SKILL Debugger Commands

Dump

Prints to the CIW the current values of all the local variables on the stack, up to the maximum specified by the number to the right of the *Where* button.

Stacktrace

Prints to the CIW all the functions on the stack, and their arguments, up to the depth specified by the number to the right of the *Where* button.

Where

Prints to the CIW all the functions and local variables on the stack, up to the depth specified by the number to the right of the *Where* button.

Step

Steps into functions from the break handler. The number of steps is specified by the number to the right of the *Step Out* button.

Next

Does not step into functions, but allows execution to proceed from the break handler until the stack returns to its current depth. This function repeats the number of times specified by the number to the right of the *Step Out* button.

Step Out

Allows execution to proceed until the evaluator returns from the current function. This function repeats the number of times specified by the number to the right of the *Step Out* button.

Exit Debug Toplevel

Exits the current SKILL Debugger toplevel.

Continue

Continues execution from a breakpoint.

Tracing

Brings up the Tracing Form on page 54.

Set Breakpoints

Brings up the Set Breakpoints Form on page 103.

Debug Status

Prints the functions, variables, and properties being traced and prints those functions that have breakpoints set or are being counted.

Clear

Clears all tracing and breakpoints.

Automatic Stacktrace (Levels)

Sets the number of functions on the stack to print every time an error occurs. This is useful if the SKILL Debugger is not installed or if the error occurs within an *errset* that prevents the SKILL Debugger from being entered.

Enter New Debug Toplevel on Error

Click the checkbox to enter the Debugger every time a SKILL error occurs.

Terminate Debugging and Quit Debugger

Uninstalls the Debugger and closes the Debugger Toolbox.

3 SKILL Lint

This document discusses

- Overview on page 17
- Form Options on page 18
    - Input File on page 18
    - Context Name on page 18
    - Package Prefixes on page 18
    - Output on page 19
    - Output File on page 20
    - Check For on page 20
- Customize Messages Form on page 22
    - Message Groups on page 22
    - Messages on page 24
- SKILL Lint PASS/FAIL and IQ Algorithms on page 31
- SKILL Function on page 32

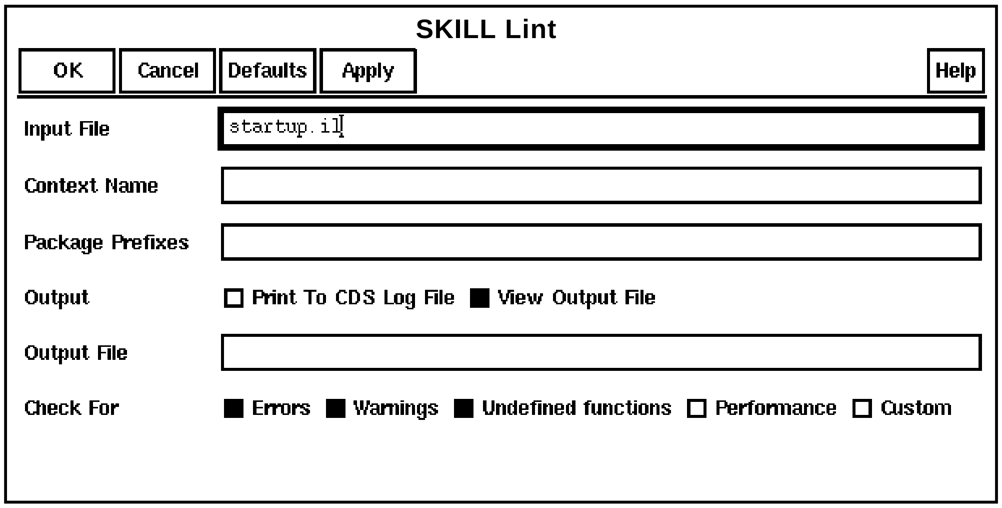
See also

- Writing SKILL Lint Rules on page 91
- Walkthrough on page 67

Overview

Examines Cadence® SKILL language code for possible errors and inefficiencies. The program is useful for detecting errors not found during normal testing. In addition, the program helps you spot unused variables and global variables that are not declared as locals. You can optionally write your own rules. See Writing SKILL Lint Rules on page 91.

To run SKILL Lint, enter the file name or context that you want analyzed and click on *OK* or *Apply*. By default, the output is displayed in the window that appears when SKILL Lint completes.



Input File	startup.il
Context Name	
Package Prefixes	
Output	☐ Print To CDS Log File ■ View Output File
Output File	
Check For	■ Errors ■ Warnings ■ Undefined functions ☐ Performance ☐ Custom

Form Options

Input File

The SKILL file to analyze. If you specify the input file, you do not need to specify the context name.

Context Name

The context to analyze. If you don't specify a file, SKILL Lint looks under the install_dir/ `tools/dfII/pvt/etc/context/ContextName` directory and analyzes the files in that directory. You may also give a directory path for the context.

Package Prefixes

The list of acceptable package prefixes for functions and global variables. SKILL Lint notes any variables that don't have the prefix, such as `tr`, that you entered. See Checking Function and Global Variable Prefixes on page 30. This helps you find a variable that you meant to declare as a local because prefixes are not normally used on local variables. This would also flag whether your program uses a global from someone else's program.

Output

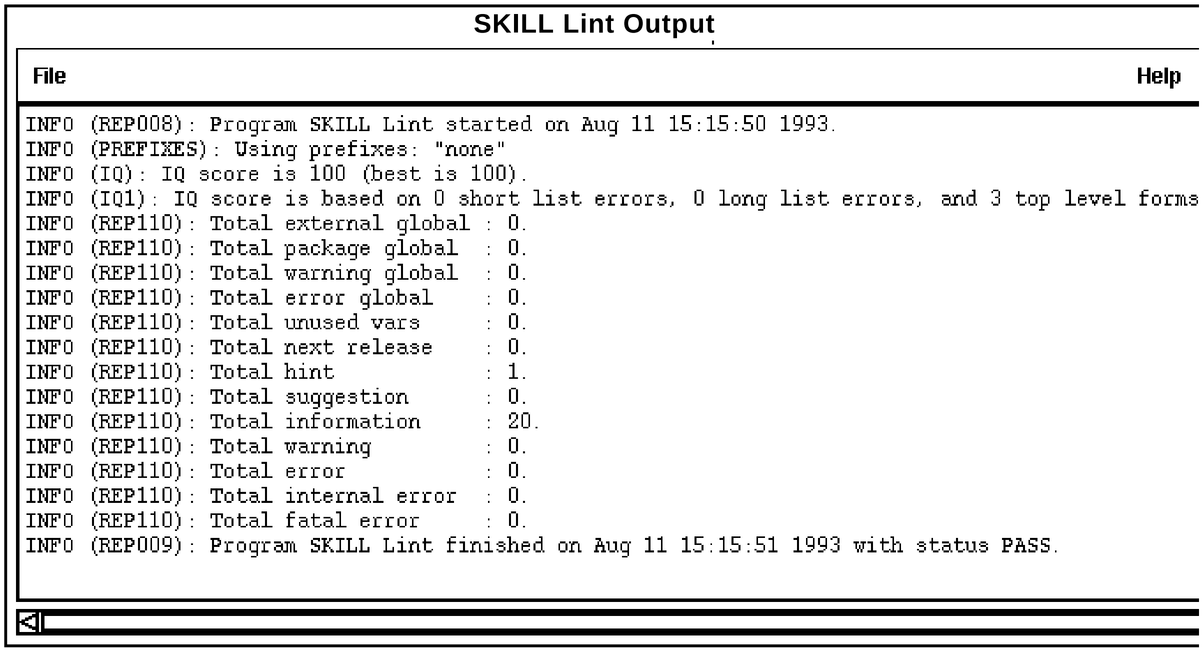
Determines where to print the output.

Print To CDS Log File

Prints the SKILL Lint output to the `CDS.log` file and the CIW.

View Output File

Brings up a window containing the SKILL Lint output. If you turn on both *Print To CDS Log File* and *View Output File*, the output file is displayed after the output is printed to the `CDS.log` file. When you are done viewing the output, choose *Close Window* from the File menu.



```
INFO (REP008): Program SKILL Lint started on Aug 11 15:15:50 1993.
INFO (PREFIXES): Using prefixes: "none"
INFO (IQ): IQ score is 100 (best is 100).
INFO (IQ1): IQ score is based on 0 short list errors, 0 long list errors, and 3 top level forms
INFO (REP110): Total external global : 0.
INFO (REP110): Total package global  : 0.
INFO (REP110): Total warning global  : 0.
INFO (REP110): Total error global    : 0.
INFO (REP110): Total unused vars     : 0.
INFO (REP110): Total next release    : 0.
INFO (REP110): Total hint            : 1.
INFO (REP110): Total suggestion      : 0.
INFO (REP110): Total information     : 20.
INFO (REP110): Total warning         : 0.
INFO (REP110): Total error           : 0.
INFO (REP110): Total internal error  : 0.
INFO (REP110): Total fatal error     : 0.
INFO (REP009): Program SKILL Lint finished on Aug 11 15:15:51 1993 with status PASS.
```

Output File

The file to contain the SKILL Lint output. If you do not enter a name here, a temporary view file like the one above, rather than a permanent file, is created.

errors:	Indicates the number of errors
general warnings:	Indicates the number of general warnings.
top level forms:	Indicates the number of expressions in the input file.
IQ score	= 100 - [25*(number of short list errors) + 20*(number of long list errors) / (number of top level forms)] See SKILL Lint PASS/FAIL and IQ Algorithms on page 31.

Syntax of an output line:

- *Message Group Name* usually abbreviated and capitalized.
- (*Built-in Message Name*) in parentheses and capitalized.
- Message description.

Check For

Turns on or off different groupings of SKILL Lint messages.

Errors

Enables the messages that cause a SKILL error if the code is executed, such as error, error global or fatal error.

Warnings

Enables the messages that are potential errors and areas where you should clean up your code.

Undefined functions

Lists all the functions that cannot be executed in the executable from which you ran SKILL Lint.

Performance

Enables the messages that give hints or suggestions about potential performance problems in your SKILL code.

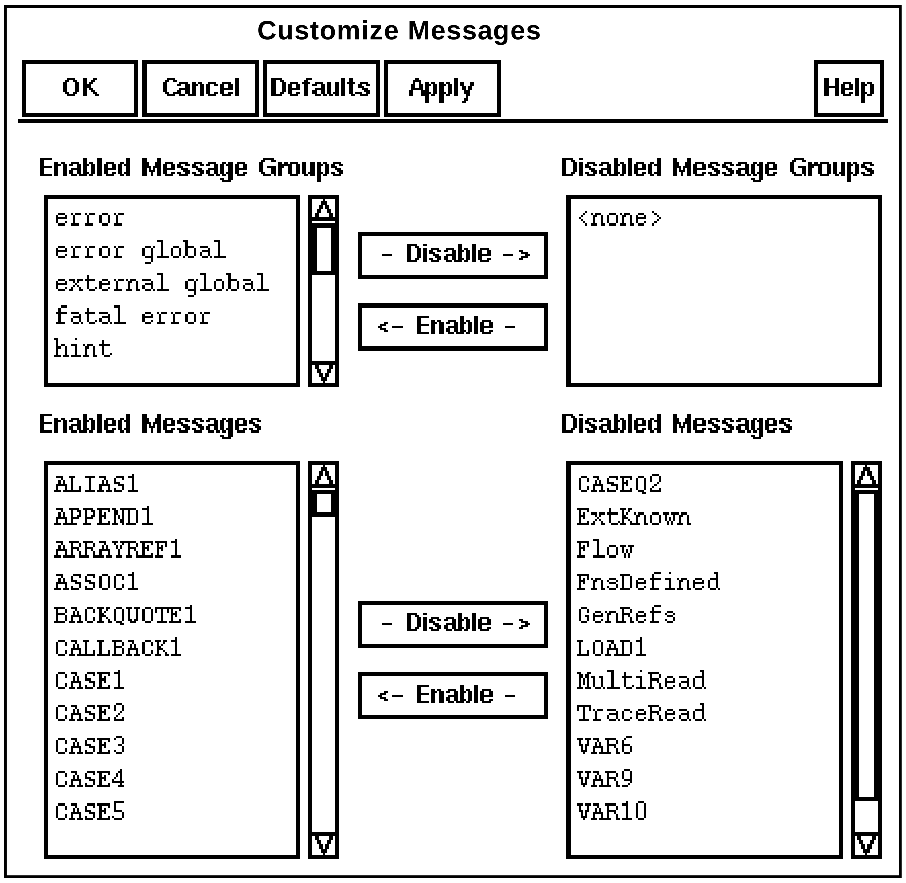
Custom

Allows you to customize the error reporting to a higher granularity. In general, you do not need to use this option. When you select Custom, the SKILL Lint form is redrawn with a *Customize Messages To Check For* button to select another form at the bottom.

When you select *Customize Messages To Check For*, the Customize Messages form appears. The form appears with the current *Check For* selections highlighted.

If you make selections, and then click *OK* or *Apply*, succeeding calls to the form reflect the messages as you last set them.

Customize Messages Form



Enabled Message Groups

```
error
error global
external global
fatal error
hint
```

Disabled Message Groups

```
<none>
```

Enabled Messages

```
ALIAS1
APPEND1
ARRAYREF1
ASSOC1
BACKQUOTE1
CALLBACK1
CASE1
CASE2
CASE3
CASE4
CASE5
```

Disabled Messages

```
CASEQ2
ExtKnown
Flow
FnsDefined
GenRefs
LOAD1
MultiRead
TraceRead
VAR6
VAR9
VAR10
```

Message Groups refers to different classes of messages reported. If a message group is disabled, no messages in that group are reported.

Messages refers to SKILL Lint messages that you can turn on or off individually.

To make selections, click on items in any of the list boxes.

To move messages between the enabled and disabled lists, use the arrow buttons.

Message Groups

The Message Group Name is listed as indicated in bold below in the Customized Message Form. The message group priority appears as the first field on an output report line.

Priority	**Message Group Name**

ERROR	**error** is the group of messages that are considered errors.
ERR GLOB	**error global** is the list of variables used as both globals and locals.
EXT GLOB	**external global** is the list of variables defined externally as globals.
Fatal Error	**fatal error** is the group of messages that prevent SKILL Lint from proceeding with analysis.
HINT	**hint** is the group of messages that tell you how to make your code more efficient.
INFO	**information** is all general information messages.
Internal Error	**internal** is the group of messages about failures of the reporting mechanism.
NEXT RELEASE	**next release** is a group of messages to flag SKILL code that will not work in the next release.
PACK GLOB	**package global** is the list of global variables that begin with the package prefix.
SUGGEST	**suggestion** is the group of messages that indicate possible ways you can increase the performance of your code.
UNUSED VAR	**unused vars** is the list of local variables that do not appear to be referenced.
WARN	**warning** is the group of messages that are potential errors.
WARN GLOB	**warning global** is the list of global variables that do not begin with a package prefix.

Messages

The Built-in Message Name appears in parentheses in the output report line. The Message Group Name appears as its associated message group priority name in the first field of the output report line. Only the SKILL core messages are listed in this table.

Built-in Message Name	**Message Group**	**Message Description**
ALIAS1	error	Both arguments to `alias` must be symbols.
APPEND1	suggestion	Consider using `cons` rather than `append`.
ARRAYREF1	error	First argument to `arrayref` must evaluate to an array.
ASSOC1	suggestion	Consider using `assq` rather than `assoc`.
BACKQUOTE1	suggestion	Possibly replace this backquote with a quote.
CASE1	warning	`case` can never be reached (after default `t`).
CASE2	warning	Symbol `t` used in `case` or `caseq` list.
CASE3	error	Duplicate value in `case` or `caseq`.
CASE5	hint	`case` can be replaced with `caseq`.
CASE6	warning	Quoted value in `case` or `caseq` (quote not required).
CASEQ1	error	You must use `case` rather than `caseq`.
CHK1	error	Type template string must be last argument.
CHK2	error	Redundant statement.
CHK3	error	Bad argument (must be a symbol).
CHK4	error	Redundant argument template.
CHK6	error	Macros cannot have `@key`, `@rest`, or `@optional`.
CHK7	error	Nlambda 1st argument must be a symbol.
CHK8	error	Entry after `@rest` not allowed.
CHK9	error	`@rest,` or `@key, ` or `@optional ` not followed by an argument.
CHK10	error	Argument duplicated.

Built-in Message Name	Message Group	Message Description
CHK11	error	Nlambda 2nd argument should be a list.
CHK12	error	Nlambda maximum of two arguments.
CHK13	error	`@optional` and `@key` cannot appear in the same argument list.
CHK14	error	Bad argument, should be a list of length 2.
CHK15	error	Bad argument, should be a list.
CHKARGS1	error	Function requires at least n arguments. See Checking the Number of Function Arguments on page 30.
CHKARGS2	error	Function takes at most n arguments. See Checking the Number of Function Arguments on page 30.
CHKARGS3	error	Key argument repeated.
CHKARGS4	error	Unknown key argument.
CHKARGS5	error	No argument following key.
CHKFORM1	error	Number of arguments mismatch.
CHKFORM2	error	Bad statement.
DBGET1	error	Second argument to ~> must be symbol or string.
DEADCODE1	warning	Unreachable code.
DECLARE1	error	Arguments to `declare` must be calls to `arrayref`, (e.g. `a[10]`).
DECODE1	error	You must use `case` or `caseq` rather than `decode`.
DEF1	error	Extra argument passed to `def`.
DEF2	error	Last argument to `def` is bad.
DEF3	hint	`nlambda`, `macro`, or `alias` should not be referenced before it is defined.
DEF4	hint	`nlambda`, `macro`, or `alias` might be referenced before it is defined.

Built-in Message Name	**Message Group**	**Message Description**
DEF5	hint	Recursive call to an `nlambda` function or macro is inefficient, call `declareNLambda` first.
DEF6	error	Definition for function `def` cannot have more than 255 required or optional arguments.
DEFSTRUCT1	error	Arguments to `defstruct` must all be symbols.
EQUAL1	hint	You can replace `== nil` with `!`.
EQUAL2	hint	You can replace `== 1` with `onep`.
EQUAL3	hint	You can replace == 0 with `zerop`.
EVALSTRING1	suggestion	Consider using `stringToFunction` when `evalstring` is called multiple times with the same string.
ExtHead	information	Known/Unknown External functions called.
ExtKnown	information	Functions called that are defined outside of analyzed code.
External	information	Functions called that are not defined.
FOR1	error	First argument to `for` must be a symbol.
Flow	information	Reports the call flow for the code analyzed.
GET1	error	Second argument to -> must be a symbol.
GET2	error	Autoload symbol is no longer used, replace `get` with `isCallable`.
GETD1	error	`getd` no longer returns a list, use the function `isCallable`.
GO1	error	go must have exactly one argument, a symbol.
GO2	error	go must be called from within a `prog` containing a label.
IF4	error	`then` and `else` required in `if` construct.
IF5	error	`else` without corresponding `then`.
IF6	hint	Remove the `then nil` part and convert to `unless`.

Built-in Message Name	Message Group	Message Description
IF7	hint	Remove the `else nil` part, and part convert to a `when`.
IF10	hint	Invert the test and replace with `unless`, as no `else` part.
IQ	information	IQ score (best is 100).
IQ1	information	IQ score is based on messages * priority.
LABEL1	warning	Label not used within scope.
LABEL2	error	More than one declaration of label within scope.
LAMBDA1	error	Bad use of `lambda`.
LET1	error	Incorrect `let` variable definition.
LET2	hint	`let` statement has no local variables, so can be removed.
LET3	hint	Variable repeated in local variable list for `let`.
LET4	warning	Variable used before available in `let` assignment.
LET5	error	`let` statements will not accept more that 255 local variables in the next release.
LOAD1	warning	Can't evaluate to an `include/load` file.
LOOP1	error	First argument must be a symbol or list of symbols.
LoadFile	information	Loading file.
MEMBER1	suggestion	Consider use of `memq` rather than `member`.
MultiRead	information	Attempt to read file more than once.
NEQUAL1	hint	You may be able to replace with !=.
NEQUAL2	hint	You can replace with !=.
NTH1	hint	Can replace call to `nth` with call to `car`, `cadr`, and so on.
NoRead	error	Cannot read file.

Built-in Message Name	Message Group	Message Description
PREFIXES	information	Using package prefixes.See Checking Function and Global Variable Prefixes on page 30.
PREFIX1	warning	Prefixes must be all lower case or all upper case. See Checking Function and Global Variable Prefixes on page 30.
PRINTF1	error	Incorrect number of format elements.
PRINTF2	error	Format argument is not a string.
PROG1	error	Bad action statement.
PROG2	hint	`prog` construct may be removed.
PROG4	hint	Variable repeated in local variable list of `prog`.
PROG5	hint	`prog` may be replaced with `progn`.
PROG6	hint	Will need a `nil` at end if `prog` removed.
PROGN1	hint	`progn` with only one statement can be removed.
PUTPROP1	information	The autoload symbol is no longer used for functions in contexts.
REMOVE1	suggestion	Consider using `remq` rather than `remove`.
REP	SKILL lint run message	Short for report. REP is not based on the content of the program.
RETURN1	warning	Not within a `prog: return`.
RETURN2	hint	Replace `return(nil)` with `return()`.
SETQ1	error	First argument should be a symbol.
SETQ2	suggestion	Possible variable initialized to `nil`.
SETQ3	suggestion	Assignment to loop variable.
SKFATAL	fatalerror	Error found from which SKILL Lint can’t proceed.
STATUS1	error	Second argument must be `t` or `nil`.
STATUS2	error	Unknown status flag.

Built-in Message Name	Message Group	Message Description
STATUS3	warning	Internal (s)status flag, don’t use.
STRCMP1	hint	Inefficient use of `strcmp`. Change to `equal`.
STRICT	information	Applying strict checking of global variable prefixes. See Checking Function and Global Variable Prefixes on page 30.
STRLEN1	hint	Inefficient use of `strlen`. Change to `equal ""`.
TraceChecks	information	Applying SKILL Lint checks.
TraceForm	information	Form being read by SKILL Lint.
TraceRead	information	This message is given for each file that is analyzed.
Unused	unused vars	Variable does not appear to be referenced.
VAR	information	Variable used or set in function/file.
VAR0	information	Variable used or set in function/file.
VAR1	error	Attempt to assign a value to `t`.
VAR4	information	Variables used as both global and local.
VAR5	information	Unrecognized global variables.
VAR6	information	Acceptable global variables.
VAR7	error global	Variable used as both a local and global.
VAR8	warning global	Global variable does not begin with package prefix.
VAR9	package global	Global variable begins with package prefix.
VAR12	warning	Argument does not appear to be referenced.
VAR13	information	Internal global variable does not appear to be referenced.
VAR14	information	Package global variable does not appear to be referenced.
VAR15	error	Variable cannot begin with keyword symbol (?).

Built-in Message Name	Message Group	Message Description
VAR15	error	Variable cannot begin with keyword symbol (?) in the next release.
VAR16	warning	Variable declaration hides a previous declaration.
WHEN1	hint	Invert test and convert to `when/unless`.

Checking the Number of Function Arguments

SKILL Lint checks that the number of arguments passed to a function matches that expected by the function. To do this it uses the previously known definition of a function, either from a previous run of SKILL Lint or a previous declaration of the procedure.

If a procedure is unknown at the time it is used, then SKILL Lint delays checking the number of arguments to the call until the procedure definition has been found.

If a procedure is used in a file before it is defined in the same file and the number of arguments to the procedure changes, it may be necessary to run SKILL Lint twice to get accurate results because the first run will use the previous declaration of the procedure.

Checking Function and Global Variable Prefixes

Functions and global variables used in SKILL code are expected to be prefixed with a suitable string. You enter these strings on the SKILL Lint form.

By default, strict checking is only applied to the customer's global variables, while functions and Cadence's prefixes are to be checked by specification, see SKILL Function on page 32.

The naming policy for functions and global variables is:

- The naming policy for function and global variable prefixes is identical:
- Cadence official (i.e. documented/supported) SKILL functions and global variables must start with a lower-case character, while three characters and all lower-case are preferred.
- Customer (and undocumented/unsupported) SKILL functions and global variables must start with an upper-case character.
- Functions or global variables must start with the prefix, or the prefix plus an optional lower-case character (one of 'i', 'v', 'c', 'b', 'e', 'f', 'm') followed immediately by an upper-case character or a '_'.

This strict checking can be switched off by disabling the *STRICT* message. In that case, the system only checks that global variables begin with a specified prefix.

SKILL Lint PASS/FAIL and IQ Algorithms

Behind all the reporting that SKILL Lint does is a system called the *standard reporting mechanism*, which

- Allows any program to report messages in a consistent manner to the screen and log files.
- Allows messages to be switched off.
- Prints a summary at the end.
- Gives a simple way of changing messages to a different language.

One part of that system is the ability to register different message classes, such as `information`, `warning` and `error`. With each class, you can indicate whether generating a message of that class should cause an overall fail.

In SKILL Lint, the following classes cause a failure, and hence status `FAIL`:

- `error global`
- `error`
- `fatal error`
- `warning`

A case fails if it has a `warning`. If the `warning` has not been printed because the message has been switched off, that does not stop it from appearing in the summary scores and the status.

A case may have an IQ score of `0`, but if there is nothing to cause a real failure, the overall status can still be pass.

The IQ score is something that is specific to SKILL Lint. It is based on the number of each class of message printed, multiplied by a factor for each different class.

- Most classes score zero.
- The following classes score 1:
  - `warning`

- ❑ `error`
- ❑ `error global`
- ❑ `warning global`
- ❑ `unused var`
- ❑ `authorization`

■ `Fatal error` scores 100.

The final score is the lower of the following two values.

- Value One: The figures are totalled up, divided by the number of top level forms (the number of `lineread` statements performed by SKILL Lint in parsing the files) and multiplied by 20. This figure is subtracted from 100 to give the score. The minimum score is zero.
- Value Two: There is a class called `shortListErrors`, which consists just of the number of `error` class messages. This is multiplied by 25 if SKILL Lint is run on a single file, or by 10 if sklint is run across multiple files. The result is again subtracted from 100.

There is no cost to the IQ or pass/fail for undefined functions with respect to the score.

SKILL Function

```
sklint(
     [?file tl_file]
     [?context t_contextName]
     [?outputFile t_outputFileName]
     [?ignoreGroups l_ignoreGroups]
     [?globals l_globals]
     [?depends l_depends]
     [?rulesFile t_rulesFile]
     [?ignores l_ignores]
     [?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
```

Arguments

file	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	The name of the context, or an absolute-pathed context name, being processed.
outputFile	The name of the reporting log file. Defaults to `<contextName>.log`.
ignoreGroups	The list of rule groups that should not be carried out.
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	The list of contexts on which the code under analysis depends. This is used for loading external definitions files.
rulesFile	The name of an additional rules file to be read prior to processing the code.
ignores	The list of message IDs to ignore.
checkNlambda	Specifies whether to check the arguments to nlambda functions. This option should only be used by very experienced users, as it generally leads to results that are difficult to interpret. This option defaults to nil.
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	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	Specifies whether to use the rules file listed in the global variable *skGlobalRulesFiles*. This option defaults to nil.
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	Specifies whether to check both Cadence private and public functions (i.e. force setting both *checkPvtFuncs* and *checkPubFuncs* to t). This option defaults to nil.
checkPvtFuncs	Controls whether to check Cadence private functions. This option defaults to nil.
checkPubFuncs	Specifies whether to check Cadence public functions. This option defaults to nil.
prefixes	The list of allowed 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	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 (i.e. by default), the checking is for customers' function/variables prefixes - start with an upper-case character. Note that this option is for Cadence internal use only. This option defaults to nil.
checkFuncPrefixes	Controls whether function prefixes should also be checked. If this argument is not set to t (i.e. by default), only customers' global variable prefixes are checked. This option defaults to nil.
tabulate	Controls whether to tabulate all the functions being called. This option defaults to nil.
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 `"."` will be searched first by default.
codeVersion	The release version of code being checked (e.g. "447" for IC4.4.7). 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 (i.e. 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 IC447 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 only filter out function changed messages. Function deleted messages will always be reported.

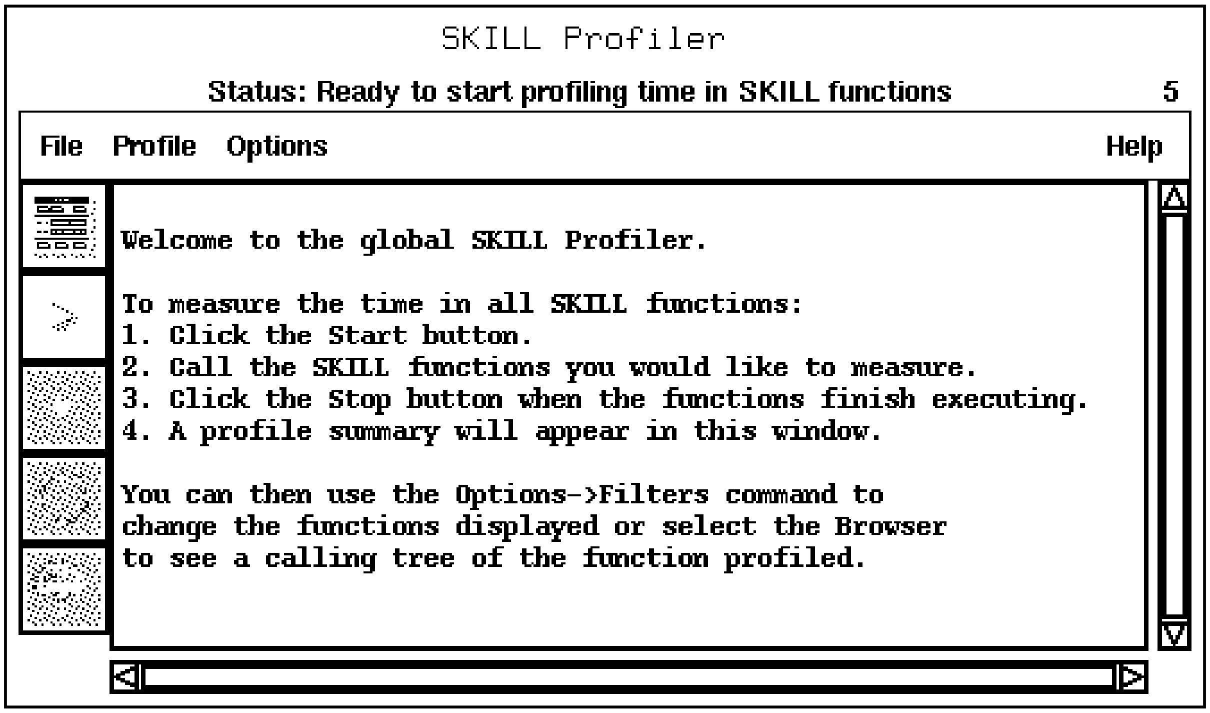
4 SKILL Profiler

This document discusses

- Overview on page 38
- Using the SKILL Profiler on page 38
- File Menu on page 39
    - Open Browser on page 39
    - Save As on page 39
    - Search on page 40
    - Close Window on page 41
- Profile Menu on page 41
    - Start Profiling Time on page 41
    - Start Profiling Memory on page 42
    - Stop Profiling on page 42
    - Reset Profiling on page 42
- Options Menu on page 43
    - Setup on page 43
    - Filters on page 44
- Fixed Menu on page 46
- SKILL Functions on page 46

Overview

The SKILL Profiler tells you where your Cadence® SKILL language programs are taking the most time and allocating the most memory.



```
Welcome to the global SKILL Profiler.

To measure the time in all SKILL functions:
1. Click the Start button.
2. Call the SKILL functions you would like to measure.
3. Click the Stop button when the functions finish executing.
4. A profile summary will appear in this window.

You can then use the Options->Filters command to
change the functions displayed or select the Browser
to see a calling tree of the function profiled.
```

The SKILL Profiler

- Measures the time spent in each function that executes longer than 1/60th of a second.
- Shows how much SKILL memory is allocated in each function.
- Measures performance without having to modify function definitions.
- Displays a function call tree graph of all functions executed and the time or memory spent in those functions.
- Allows you to filter functions so you can see only those functions in which you are interested.

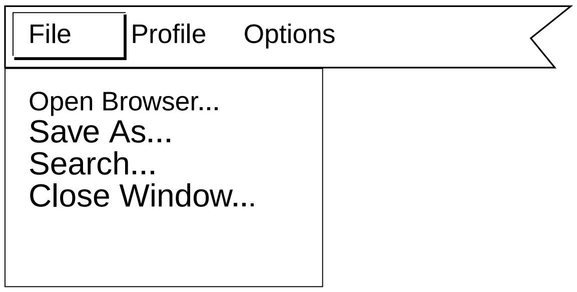
Using the SKILL Profiler

To run the SKILL Profiler

1. Click the *Start Profiling* icon.
2. Execute the SKILL functions you want to measure.
3. Click the *Stop Profiling* icon.

   A profile summary appears in the window. It lists the functions and the CPU time spent in them. The *gc* function represents the time spent in garbage collection. If the amount of time in *gc* is high, you should profile the SKILL memory usage by selecting *Memory allocated in SKILL functions* in the Setup form and rerunning the SKILL Profiler.

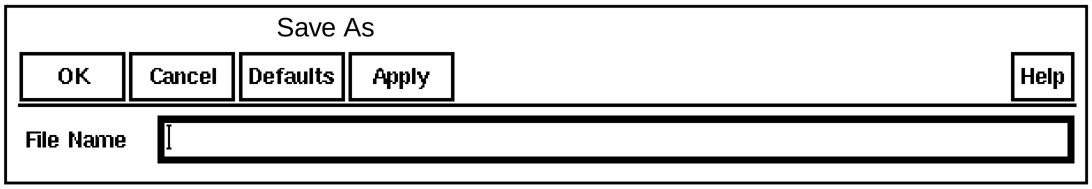
File Menu



Open Browser

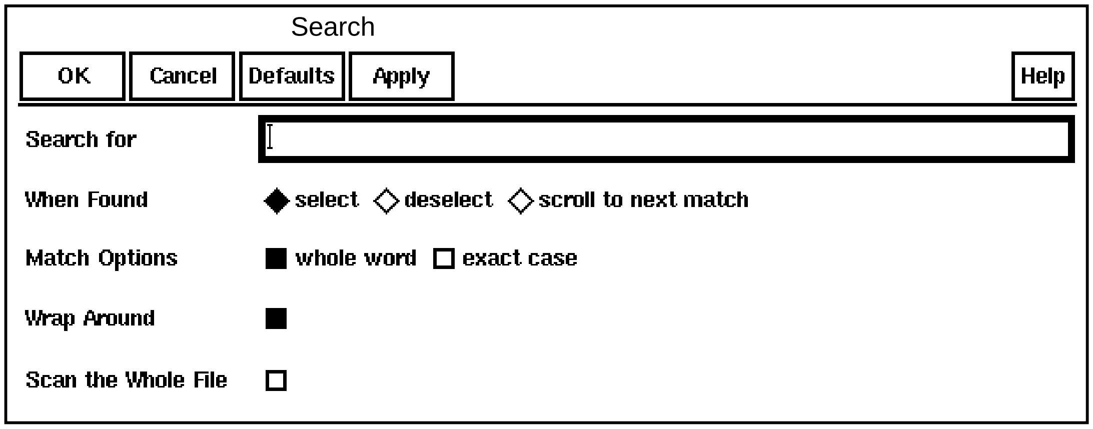
Opens the Code Browser at the top level if it has not been brought up already. The Code Browser lets you examine the calling tree of a function’s children and how much time was spent in each function.

Save As



The *Save As* form allows you to save your results for later reference.

Search



Search for

When Found [✓ select] deselect scroll to next match

Match Options [✓ whole word] exact case

Wrap Around [✓]

Scan the Whole File

Search for

The text string you want to search for.

When Found

select

Adds the string to the selection list.

deselect

Removes the string from the selection list.

scroll to next match

Moves the cursor to the next match without selecting.

Match Options

whole word

Requires that the string be a separate word (surrounded by spaces).

exact case

Requires that the capitalization of the string exactly match the entry in *Search for*.

Wrap Around

Continues the search at the beginning of the file once the end of the file is reached.

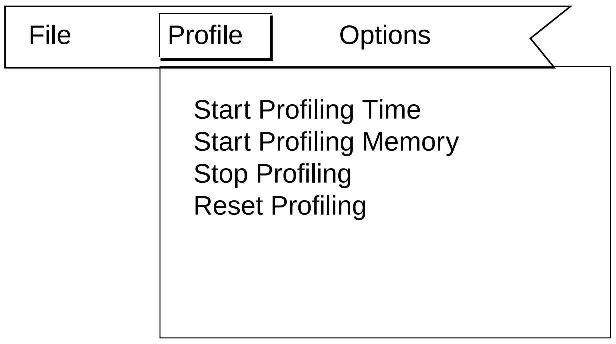
Scan the Whole File

Selects at once all occurrences of the string in the file.

Close Window

Exits the SKILL Profiler.

Profile Menu



Start Profiling Time

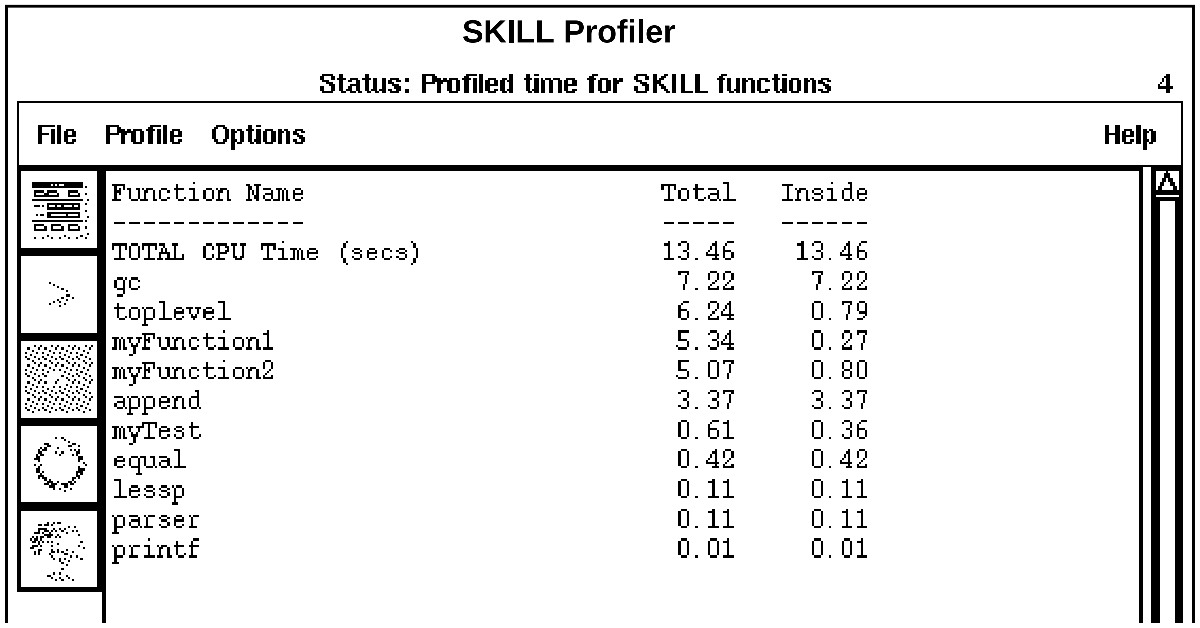
Starts measuring the time spent in all SKILL functions executed after this command is selected.

Start Profiling Memory

Starts measuring the memory allocated in all SKILL functions executed after this command is selected.

Stop Profiling

Turns SKILL profiling off and displays the summary in the SKILL Profiler window.

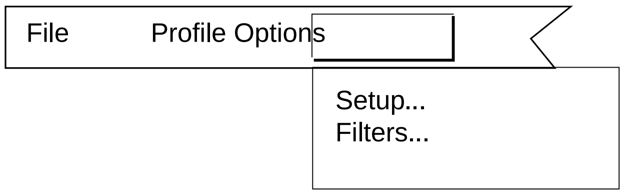


```
Function Name                        Total   Inside
-------------                        -----   ------
TOTAL CPU Time (secs)                13.46    13.46
gc                                    7.22     7.22
toplevel                              6.24     0.79
myFunction1                           5.34     0.27
myFunction2                           5.07     0.80
append                                3.37     3.37
myTest                                0.61     0.36
equal                                 0.42     0.42
lessp                                 0.11     0.11
parser                                0.11     0.11
printf                                0.01     0.01
```

Reset Profiling

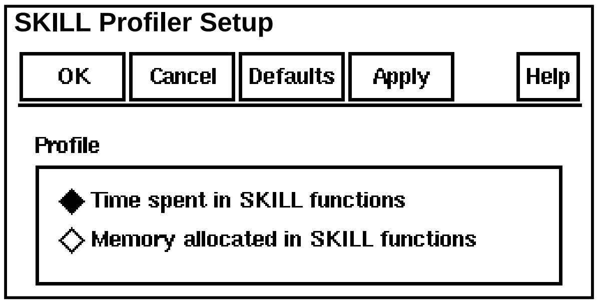
Sets the time taken in each function back to zero or the memory allocated in each function back to zero.

Options Menu



Setup

Brings up the SKILL Profiler Setup form.



Profile

◆ Time spent in SKILL functions
◇ Memory allocated in SKILL functions

Select whether you want time or memory profiled.

Time spent in SKILL functions

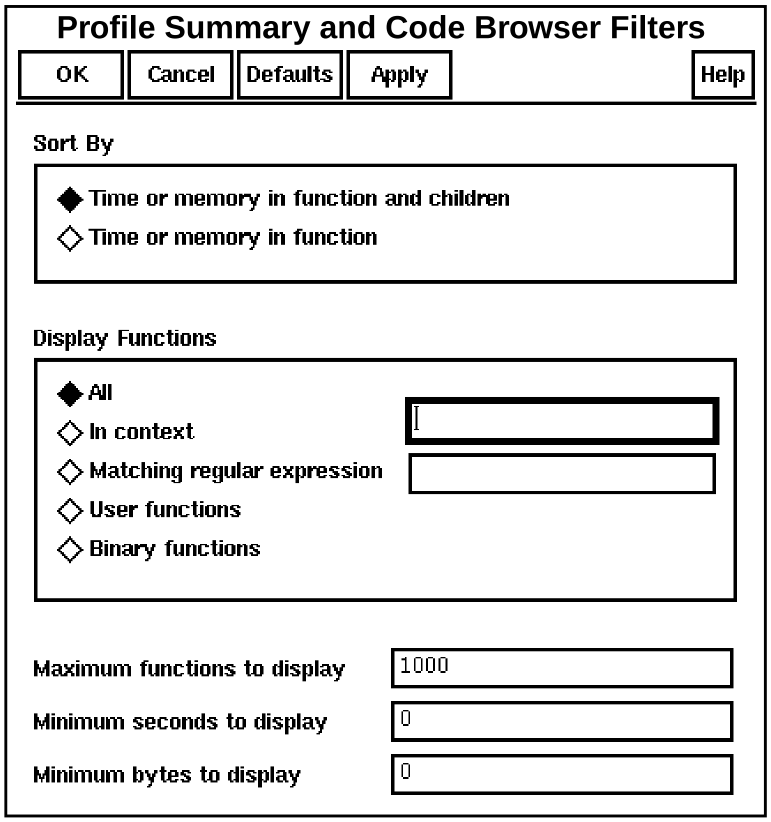
Measures time spent in SKILL functions when the profiler is started.

Memory allocated in SKILL functions

Measures memory allocated by SKILL functions when the profiler is started.

Filters

Allows you to choose which data you want to display in the SKILL Profiler and Code Browser windows. This form can also be brought up from the Code Browser Window using the *Misc - Filters* command.



Sort By

[✓ Time or memory in function and children]
Time or memory in function

Display Functions

[✓ All]
In context
Matching regular expression
User functions
Binary functions

Maximum functions to display	1000
Minimum seconds to display	0
Minimum bytes to display	0

Sort By

Time or memory in function and children

Sorts functions by the time or memory inside the function and its children.

Time or memory in function

Sorts functions by the time or memory inside the function.

Display Functions

All

Displays all functions in the profiler window and expands all children in the Code Browser window.

In context

Displays only those functions in the context given.

Matching regular expression

Displays only functions matching the regular expression given.

User functions

Displays functions that are not read protected.

Binary functions

Displays only those functions that are implemented in C and not SKILL.

Maximum functions to display

Limits the maximum functions to be displayed.

Minimum seconds to display

Displays only profiled functions that consumed more time than entered in the text entry field.

Minimum bytes to display

Displays only profiled functions that consumed more memory than entered in the text entry field.

Fixed Menu

The fixed menu icons in the left column are accelerators for several of the text commands in the menus.

Set up Profiling, same as *Options - Setup*

Start Profiling, depending on the Setup options, same as *Profile - Start Profiling Time* or *Start Profiling Memory*

Stop Profiling, same as *Profile - Stop Profiling*

Reset Profiling, same as *Profile - Reset Profiling*

Browse Profiled Results, same as *File - Open Browser*

SKILL Functions

```
profile( s_function ...) => t
unprofile( s_function ... | t) => t
profileSummary(
      [?file t_filename]
      [?allp g_listAll])
      => t
profileReset() => t
```

5

Code Browser

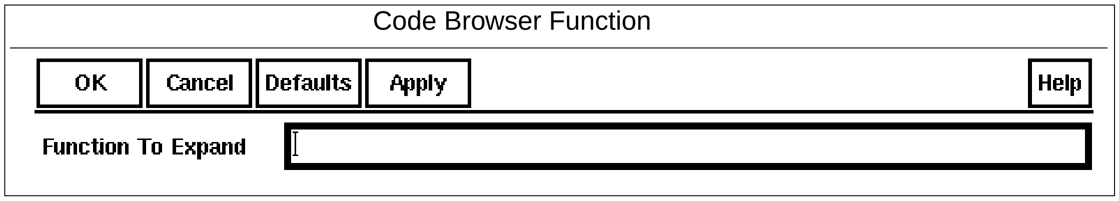
This document discusses

- Overview on page 48
  - Function To Expand on page 48
- Code Browser Window on page 49
- Commands Menu on page 49
  - Expand Function on page 49
  - Find Function on page 50
  - Delete All on page 50
  - Close on page 50
- Misc Menu on page 51
  - Defaults on page 51
  - Filters on page 51
- Pop-up Functions Menu on page 51
  - View on page 51
  - Expand Functions on page 52
  - Expand Deep on page 52
  - Unexpand on page 52
  - Delete on page 52

Overview

The Code Browser displays the calling tree of user-defined functions. The calling tree shows the child functions called by the parent functions.

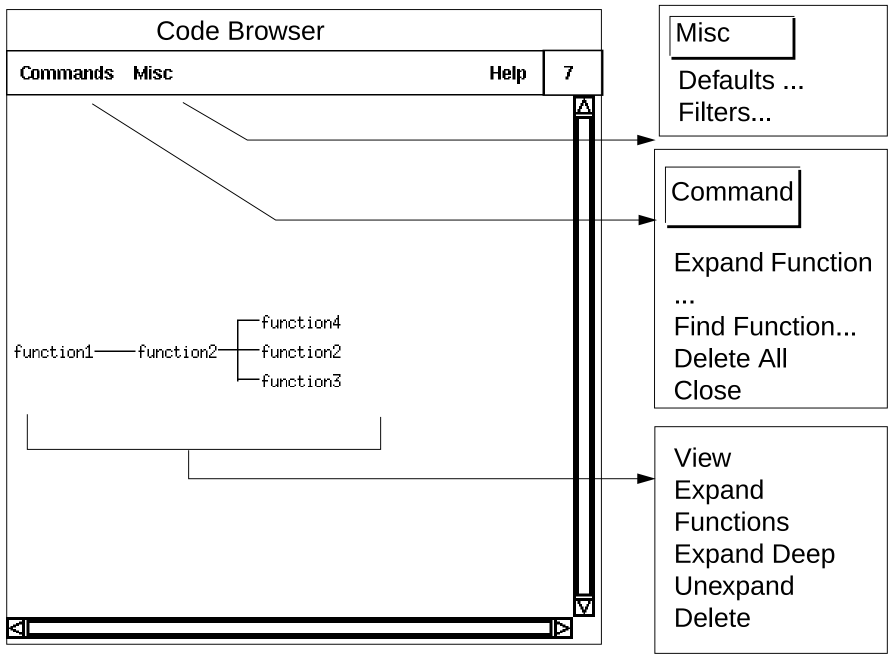
- You can expand the entire tree or one node at a time.
- You can also view the function definition of any user-defined function.



Function To Expand

The function whose calling structure you want to see. When you enter the function name and click on *OK* or *Apply*, the Code Browser window appears.

Code Browser Window



The Code Browser window shows the tree structure of the code. Initially the first level of the hierarchy is shown. When you click on a function name, the next level of the hierarchy appears to the right.

To see the Functions menu, move the pointer to a function name and hold down the middle mouse button. The menu is the same for all functions.

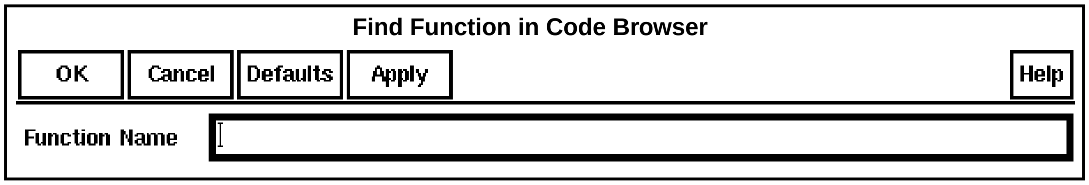
Commands Menu

Expand Function

Brings up the Code Browser Function form that lets you add functions to expand.

Find Function

Brings up a form that asks for the name of a function.



1. Type in the name and click *OK* or *Apply*.

   The Code Browser call graph is searched for the first instance of the function. If found, the function is highlighted and left-justified in the Code Browser window.

2. If you click *OK* or *Apply* again for the same function, the next instance of that function is searched for.

Delete All

Deletes all functions from the Code Browser.

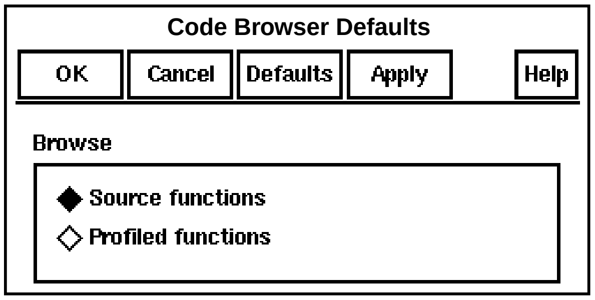
Close

Closes the Code Browser window.

Misc Menu

Defaults

Displays the Code Browser Defaults form that allows you to choose whether to browse the source code function calling tree or the profiled function calling tree.



Browse
[✓ Source functions]
Profiled functions

Source functions

Expands the call tree of functions based on their source code definitions.

Profiled functions

Expands the call tree of functions that were profiled using the SKILL Profiler and displays the time spent in the functions.

Filters

Brings up a form that allows you to filter which functions are displayed. See the Profile Summary and Code Browser Filters form under *SKILL Development - Profiler*.

Pop-up Functions Menu

View

Brings up a window that displays the source code selected in pretty printed form.

Expand Functions

Displays the children functions of the node selected. Uses the Profile Summary and Code Browser Filters form to determine which functions are expanded.

Expand Deep

Displays all user functions recursively until the entire calling tree is expanded.

Unexpand

Deletes all functions called by the one selected from the Code Browser window.

Delete

Deletes the function selected from the Code Browser window.

6

Tracing

This document discusses

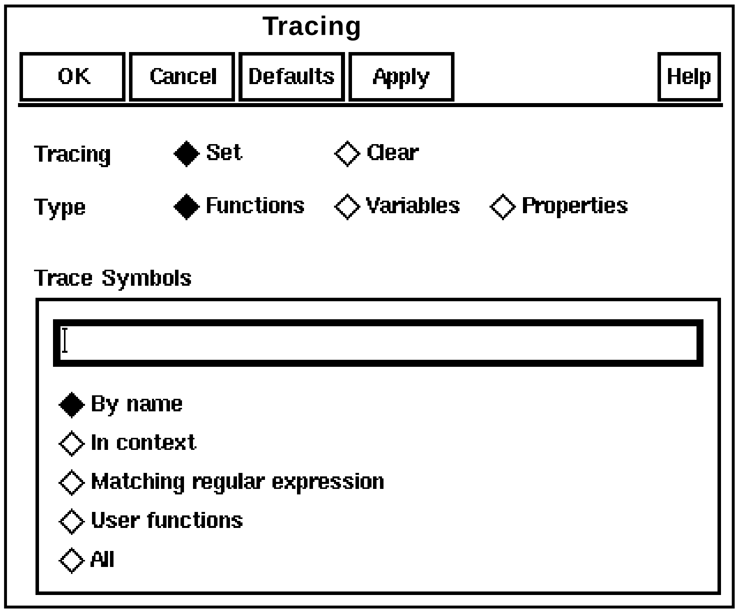
- Overview on page 53
- Tracing Form on page 54
  - Tracing on page 54
  - Type on page 54
  - Trace Symbols on page 55
- SKILL Functions on page 55

Overview

You can trace SKILL function calls as well as property and variable assignments

- To trace functions, variables, or properties, select *Set* in the *Tracing* field.
- To untrace functions, select *Clear* in the *Tracing* field.
- To turn tracing on for *Functions*, *Variables*, or *Properties*, select the ones you want traced in the *Type* field.
- To trace individual symbols, type them in the *Trace Symbols* text entry field and click *Apply*.
- To trace all functions, variables, or properties, click *All*, then *Apply*.

Tracing Form



Tracing [✓ Set] Clear

Type [✓ Functions] Variables Properties

Trace Symbols

[✓ By name]
In context
Matching regular expression
User functions
All

Tracing

Set

Turns tracing on for the *Trace Symbols* selected.

Clear

Turns tracing off for the *Trace Symbols* selected.

Type

Functions

Makes *Trace Symbols* apply to functions.

Variables

Makes *Trace Symbols* apply to variables.

Properties

Makes *Trace Symbols* apply to properties.

Trace Symbols

By name

Traces or untraces the names in the text entry field.

In context

Traces or untraces all functions in the context entered in the text entry field.

Matching regular expression

Traces or untraces all the functions matching the regular expression in the text entry field.

User functions

Traces or untraces all the functions that are not read protected.

All

Traces or untraces all functions.

SKILL Functions

```
tracef( [ s_function | t_fileName ] ... | t)
     => g_result
untrace( s_function | t_fileName ... | t) => g_result
tracev( s_variable ... | t) => g_result
untracev( s_variable ... | t) => g_result
tracep( s_variable ... | t) => g_result
untracep( s_variable ... | t) => g_result
```

7 Finder

This document discusses

- Overview on page 58
- Searching on page 58
  - Simple Strings on page 58
  - Combinations on page 59
  - Categories on page 59
  - Starting a search on page 60
  - Stopping a search on page 60
- Selecting from Matches on page 60
- Saving Descriptions on page 61
- More on Saving on page 62
- Cadence Data on page 62
- Customer Data on page 63
- Data Format on page 63
- Troubleshooting on page 64
- Starting in UNIX on page 65
  - Standard Mode on page 65
  - Test Modes on page 65

Overview

The Finder is a quick reference tool that displays the abstracts and syntax statements for language functions and APIs.

Your database may vary

The database will vary according to the products loaded on your system. Each separate product loads its own language information in the Cadence hierarchy that the Finder reads.

You can add your own functions

You can add your own functions locally for quick reference because any information that is properly formatted and located can be displayed.

Starting up

You can start up the Finder from the SKILL Development toolbox or from a UNIX command line.

Searching

You don't need complicated syntax to find information because you are searching a restricted database.

Simple Strings

Some examples:

Locate non-case sensitive matches anywhere in a word (default).

Search For: **`string`**

Matches: `buildString`, `evalstring`, `stringp`, ...

Search for matches at the beginning of a word only.

Search For: **`string` [x]** *at beginning*

Matches: `stringp`, `stringToFunction`, ...

Search for case sensitive matches at the end of a word only.

Search For: **`string`** **[x]** *at end* **[x]** *case sensitive*

Matches: `evalstring`, `get_string`, `loadstring`, ...

Combinations

The **`.*`** syntax lets you find most any combination of strings.

Some examples using the default settings:

Search For: **`hi.*view`**

Matches: `hiEnableTailViewfile`, `hiGetViewBBox`, ...

Search For: **`asi.*list`**

Matches: `asiAddDesignVarList`, `asiDisplayNetlistOption`, ...

Search For: **`ipc.*process`**

Matches: `ipcBatchProcess`. `ipcBeginProcess`, ...

Some equivalents:

[x] *at beginning* works the same as `^string`

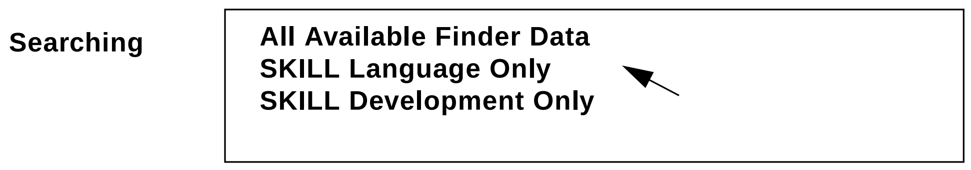
[x] *at end* works the same as `string$`.

Categories

➤ Use the default *All Available Finder Data*.

The Finder searches the Cadence database and the optional Customer database and identifies the functions it finds by category in the Searching pull-down list. The *database will vary* according to the products loaded on your system.

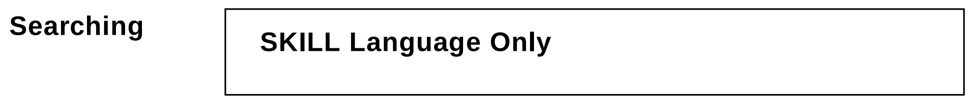
➤ Select a category from the Searching pull-down list.



Searching

All Available Finder Data
SKILL Language Only
SKILL Development Only

All searches are then confined to the functions in that category.



Searching

SKILL Language Only

➤ To search for ALL the items in the category, type `.*` in the *Search for* window.

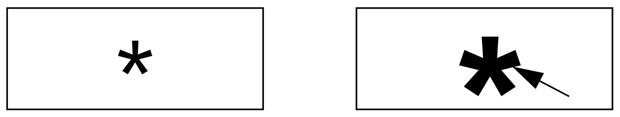
Starting a search

➤ After you enter a search string, click *Search* or type carriage return to find matches to a string.

The search is limited to 500 matches.

Stopping a search

The *asterisk* button changes to bold during the time that a search is being processed.

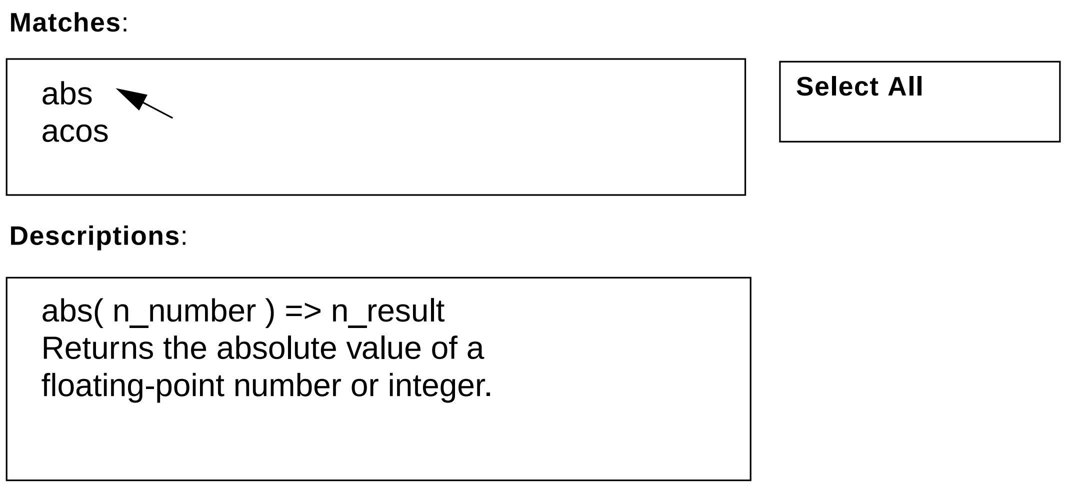


➤ Click on the *bold asterisk* button to stop a search in progress.

Selecting from Matches

The items in the *Matches* window are the database entries that match the search string. Scroll bars appear if the matches are too long or too wide for the window.

➤ Click on a function name to get its expanded description.



Matches:

abs
acos

Descriptions:

abs(n_number) => n_result
Returns the absolute value of a
floating-point number or integer.

➤ Click on *Select All* to get expanded descriptions for *all* matches.

Expanded descriptions are displayed for *all* matches in the same order they appear in the *Matches* window.

Saving Descriptions

You can edit the contents of the *Descriptions* window before you save the information.

Clear

Clears the contents of the window only, not previous saves to the log file.

Save...

Opens a save-to-file form that displays all files with the *.sav* extension in the specified directory.

OK

Appends the contents of the *Descriptions* window to the `finder.sav` file in your home directory. If this file does not exist, it is created.

Cancel

Does not save any information and no modifications are made to the file name or to the saved default.

More on Saving

You can save the contents of the *Descriptions* window to a file other than the default file.

➤ Traverse the hierarchy.

- ❑ To descend the hierarchy, double click on a directory.
- ❑ To ascend the hierarchy, double click on a **..** directory.
- ❑ To update the files displayed in the *Directories* and *Files* windows , single click on a file or directory and click *Filter*. This has the same effect as the first two bullets above.

➤ Change the file name using the `.sav` extension for any new file names.

- ❑ To append the contents of the *Descriptions* window to the specified file, click *OK* or double click on another file name in the *Files* window.
- ❑ You can edit the file name in the *Append descriptions to* window directly or type in a new name that has a `.sav` extension and click *OK*. The contents of the *Descriptions* window is saved in the new file. On the next save, click *Filter* to update the *Files* window.

Cadence Data

Cadence-supplied information is located in this hierarchy:

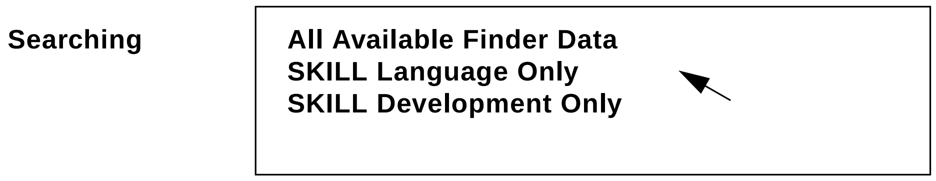
`<install>/doc/finder/<language>/<functionArea>/*.fnd`

`<install>` is the name of the Cadence installation directory.

Note: To find <*install*> use the function `cdsGetInstPath ()`.

`<language>` is the language type, such as `SKILL`.

<functionArea> is a descriptive subdirectory name for the product information, such as `SKILL_Language` or `SKILL_Development`. The convention of naming, capitalization, and underscoring to separate words is reflected in the Searching pull-down list.



.fnd is the *required* extension for the database files, such as `chap1.fnd`. All files must contain information in the appropriate data format.

Customer Data

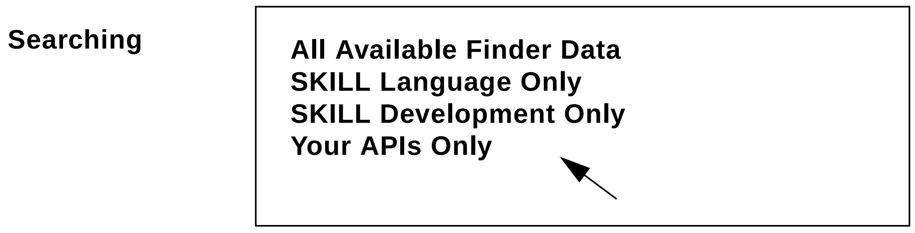
You can add your own internal functions to the database. Customer-supplied information can be placed in this hierarchy:

```
<install>/local/finder/<language>/<functionArea>/*.fnd
```

where the directories and files are analogous to those in the Cadence Data description. The program looks in this directory at start up. The directory names found are reflected in the *Searching* pull-down list.

For example:

```
<install>/local/finder/SKILL/Your_APIs/your.fnd
```



Data Format

The program expects the following three-string format for each unique entry in the `*.fnd` text files:

```
    ("functionName"
    "syntax string"
    "Abstract information.")
```

For example:

```
("abs"
"abs( n_number ) => n_result"
"Returns the absolute value of a floating-point number or integer.")
```

Identical functions can be stored together:

```
("sh, shell"
"sh( [t_command] ) => t/nil
shell( [t_command] ) => t/nil"
"Starts the UNIX Bourne shell sh as a child process to execute a command
string.")
```

Troubleshooting

Too Many Matches

`More than` **`500`** `matches have been found. Please use a more restrictive search string.`

Change the search string to limit the number of matches.

Save File Is Not Writable

`<`*`filename`*`> is not a writable file. Please enter a new file name.`

This message appears if any aspect of specifying the file name results in an error. Click *OK*. The error dialog disappears, leaving the file name entry dialog on screen so you can enter another name.

No files found

Look in the Cadence database directory to see if any files were loaded at installation. See Test Modes.

`<`*`install`*`>/doc/finder/<`*`language`*`>/<`*`functionArea`*`>/*.fnd`

Descriptions Window Full

```
WARNING: The display has reached its maximum capacity. Please save
(if desired) and clear the window.
```

If the number of characters in the *Descriptions* window exceeds one Megabyte, the current expansion operation aborts and the error message above appears. To clear the window, *Save* if desired, then click *Clear*.

Font Size Unsatisfactory

To change the font size, adjust the *`Finder*fiTextEntryFont`* variable in your .`Xdefaults` file. If you make the font larger or smaller, the *Finder* window is drawn proportionately larger or smaller. The default is

```
Finder*fiTextEntryFont: -*-courier-bold-r-*-*-12-*
```

Starting in UNIX

You can start the Finder from a UNIX command line.

Standard Mode

➤ To start the Finder from a UNIX command line in standard mode:

```
cdsFinder
```

Test Modes

➤ To turn on test mode, start the Finder with a `-t` option:

```
cdsFinder -t
```

Information is written to `/tmp/finder.tst` file as well as standard output. The Finder reports directories it finds in the database and the number of entries found per file.

➤ To check for duplicate instances of each name in the specified directory's data files:

```
cdsFinder -t checkdir
```

For example:

```
cdsFinder -t doc/finder/SKILL/SKILL_Language
```

8

Walkthrough

This document discusses

- Introduction on page 67
  - Tasks for the Program on page 68
  - Tasks for You on page 68
  - The Test Program on page 69
- Load the Program on page 70
- Run the Program on page 71
- Resolve the First Error on page 71
- Fix the Functional Error on page 73
- Run Lint on page 75
- Measure myFunction1 on page 77
- Check Performance on page 81
- Rerun the Profiler on page 83
- Using the Non-Graphical SKILL Debugger on page 85

Introduction

You need a Cadence® SKILL Development (`skillDev`) license to use the SKILL Development environment.

This walkthrough is designed

- to demonstrate how to use the SKILL Development environment
- to introduce the debugging tools

The program used in this walkthrough, `demo.il`, demonstrates how a simple program can be

- fixed quickly
- sped up substantially

Note: You can copy and paste examples from these windows.

- Press *Control-drag left mouse* to select a 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.

Tasks for the Program

Here's what the program should do (but doesn't until you fix it). When you type `myFunction1()` in the CIW, it should

- Print a starting message.
- Loop from 1 to 10000.
- Print an ending message.
- Return the numbers from 1 to 999.

What you learn from this exercise can be applied to developing and debugging much larger programs for CAD-specific tasks, such as writing a database traversal program to count fanout.

Tasks for You

For this walkthrough you'll perform the following tasks:

- Run the program.
- Fix the SKILL error using the SKILL Debugger.
- Fix the functionality error where the starting message is printed out after the 10th object instead of the 1st by using breakpoints and single stepping.
- Run SKILL Lint over the file and follow the suggestions.
- Profile the time and memory used to run `myFunction1` and analyze the results.
- Run SKILL Lint over the file, checking for performance suggestions.

- Fix the file to improve performance significantly.
- Measure the time of `myFunction1` again to confirm performance speedup.

The Test Program

This is the `demo.il` program that you will use in this walkthrough.

```
/* demo.il - This file is used for a walkthrough of the
 * SKILL Development Environment.
 */
/*******************************************************
* myFunction1 - This function must
* Count from 1 to 10000.
* Return a list of numbers from 1 to 1000 in any order.
********************************************************/
(procedure myFunction1()
      let((x y z myList)
            for( i 1 10000
             myList = myFunction2(i)
            )
            myList
      )
)
/*******************************************************
* myFunction2 - This function must
*       Print a starting message on the 1st object.
*       Print an ending message at the 1000th object.
*       Return a list of numbers less than 1000 in any order.
********************************************************/
(procedure myFunction2(object myList)
      if(myTest(object)
            then printf("Starting with object %d...\n" object)
      )
      if(object == 1000
            then printf("Ending with object %d...\n" object)
      )
      if(object < 1000
            then append(myList ncons(object ))
            else myList
      )
)
/*******************************************************
* myTest - This function must
* return t if object equals one.
********************************************************/
(procedure myTest(object)
      if(object == 10
            then t
            else nil
      )
)
```

Load the Program

1. Select *Open -> SKILL Development*.

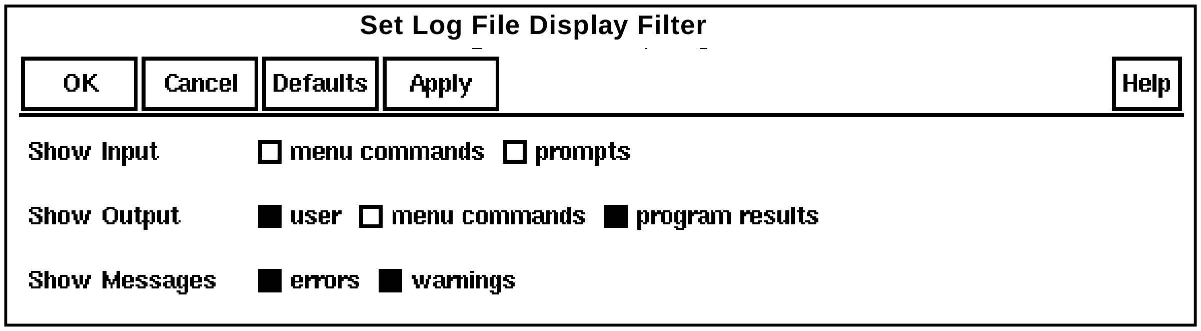
   The system indicates that it is “`Loading skillDev.cxt`” and the SKILL Development Toolbox appears.

2. Select *SKILL Debugger*.

   The SKILL Debugger Toolbox appears.

3. Check *Utilities -> Log Filter* from the CIW.

   To focus on debugging the file and not get distracting output, use these settings.



Show Input ☐ menu commands ☐ prompts

Show Output ■ user ☐ menu commands ■ program results

Show Messages ■ errors ■ warnings

4. Make a copy of the `demo.il` file in your `/tmp` directory, type the following in the CIW.

```
csh(strcat("cp " prependInstallPath("samples/skill/demo.il")
" /tmp/demo.il"))
```

   ❑ Be sure to include a space after `cp` and before `/tmp`. If successful, this function returns `t`.

   ❑ Be sure you now have a copy of the file `/tmp/demo.il` and that you have write permission.

5. To assure write permission, type the following in the CIW:

```
csh("chmod a+w /tmp/demo.il")
```

6. To load the file, type the following in the CIW.

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

   Be sure that the debugger is installed.

7. Check the prompt at the bottom of the CIW.

   The prompt should be `1>`. This means the debugger is installed but not on the stack. If the prompt is greater than 1, such as `Debug 2>`, then you are in the SKILL Debugger.

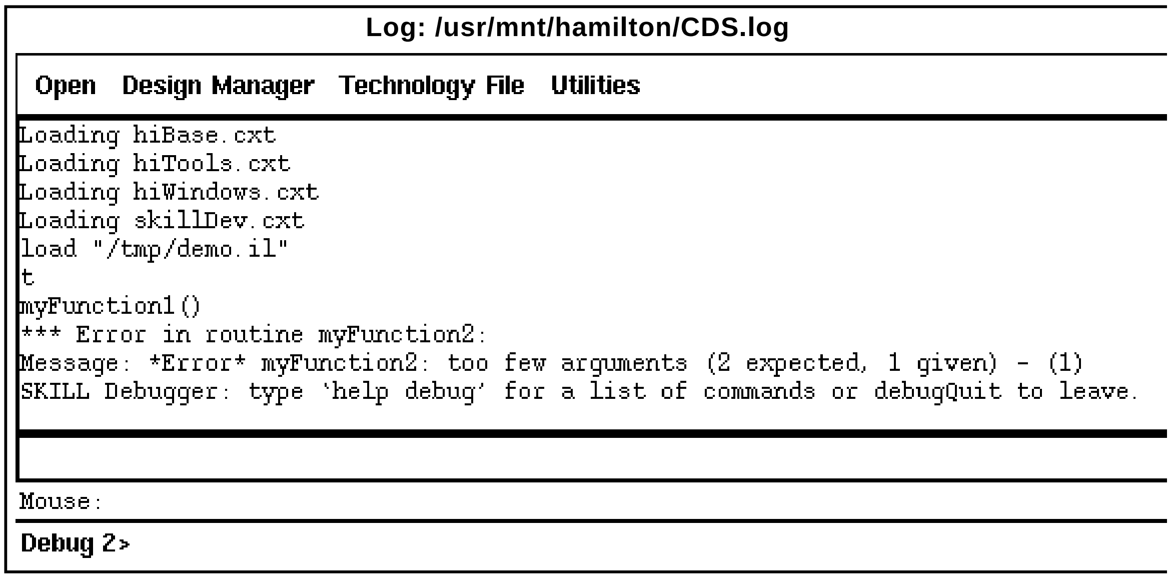
8. Select *Quit Debugger* in the SKILL Development Toolbox until you get to `1>` before continuing.

Run the Program

➤ In the CIW, type

`myFunction1()`

The system displays the following error message and the debug level changes to `Debug 2>`.



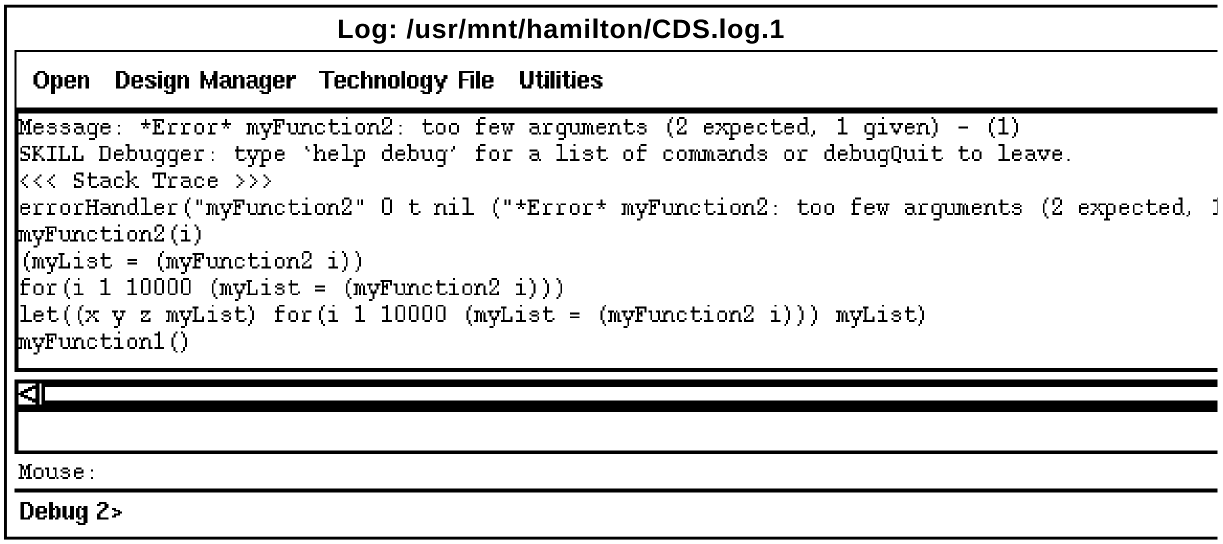
```
Loading hiBase.cxt
Loading hiTools.cxt
Loading hiWindows.cxt
Loading skillDev.cxt
load "/tmp/demo.il"
t
myFunction1()
*** Error in routine myFunction2:
Message: *Error* myFunction2: too few arguments (2 expected, 1 given) - (1)
SKILL Debugger: type 'help debug' for a list of commands or debugQuit to leave.
```

Resolve the First Error

The following error message indicates that `myFunction2` is expecting two arguments. However, only one argument is passed to `myFunction2` from some unknown function.

```
Message: *Error* myFunction2: too few arguments (2 expected, 1 given) - (1)
```

1. To find which function called `myFunction2` with the wrong number of arguments, select *Stacktrace* on the SKILL Debugger Toolbox.



```
Message: *Error* myFunction2: too few arguments (2 expected, 1 given) - (1)
SKILL Debugger: type 'help debug' for a list of commands or debugQuit to leave.
<<< Stack Trace >>>
errorHandler("myFunction2" 0 t nil ("*Error* myFunction2: too few arguments (2 expected, 1
myFunction2(i)
(myList = (myFunction2 i))
for(i 1 10000 (myList = (myFunction2 i)))
let((x y z myList) for(i 1 10000 (myList = (myFunction2 i))) myList)
myFunction1()
```

```
Debug 2>
```

By examining the stack printed in the CIW, you can see that the `errorHandler` was invoked by `myFunction2`. In turn, `myFunction2` was called by `myFunction1` inside a `for` loop.

Now edit `myFunction1` and see if you can determine what the proper second argument to `myFunction2` should be.

2. Select *Quit Debugger* to get to level one 1> again in the CIW.

3. To edit the file in an xterm window and have the file load back in when you quit the editor, type the following.

   ```
   edit( "/tmp/demo.il" t)
   ```

   The second argument `t` ensures file reloading when you quit the editor.

4. Examine the `myFunction2` function definition.

   Notice that it expects `myList` as a second argument.

5. In `myFunction1` on line 12, change

   ```
   myList = myFunction2(i)
   ```

   to

   ```
   myList = myFunction2(i myList)
   ```

6. Exit the editor by typing :wq if you are in vi to exit.

The file is automatically reloaded and you get a message verifying this and stating which functions have been redefined.

```
function myFunction1 redefined
function myFunction2 redefined
function myTest redefined
```

7. In the CIW, type

   ```
   myFunction1()
   ```

   This time the program displays the following.

   ```
   Starting with object 10 ..
   Ending with object 1000 ..
   (1 2 3 ... 999)
   ```

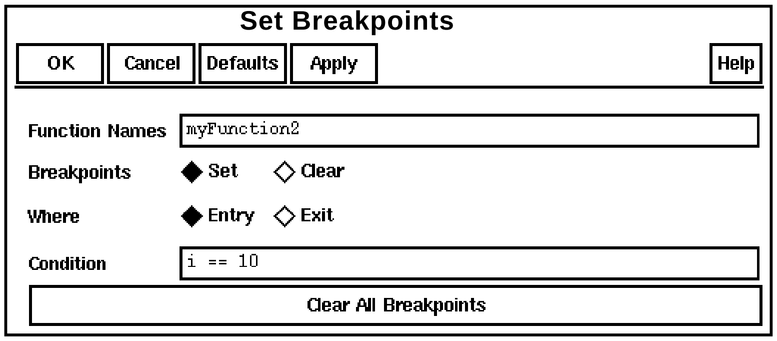
Fix the Functional Error

Notice that the message is printed for object 10. This violates the specification.

You need to change the code so that the first message is printed out for the first object instead of the 10th.

You could use the tracing form to trace all functions starting with the characters “`my`”, but that would generate over a 1000 lines of trace output. Instead, set a conditional breakpoint on `myFunction2` for when i equals 10.

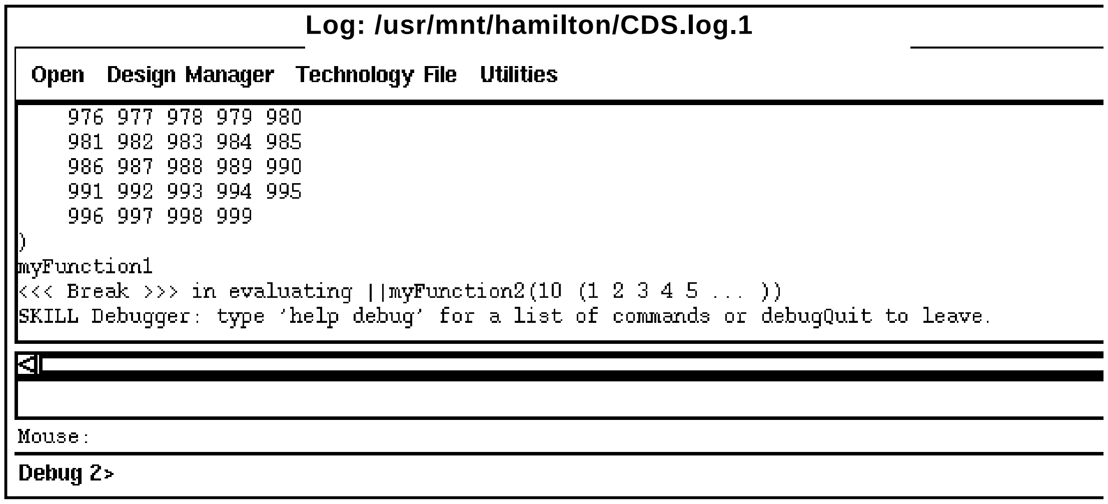
1. Select *Set Breakpoints* in the SKILL Debugger Toolbox.
2. Type `myFunction2` in the Function Names field of the Set Breakpoints form.
3. Type `i == 10` in the Condition field.



Function Names: myFunction2
Breakpoints: [✓ Set] Clear
Where: [✓ Entry] Exit
Condition: i == 10

4. Click *OK*.

5. Type `myFunction1()` in the CIW.

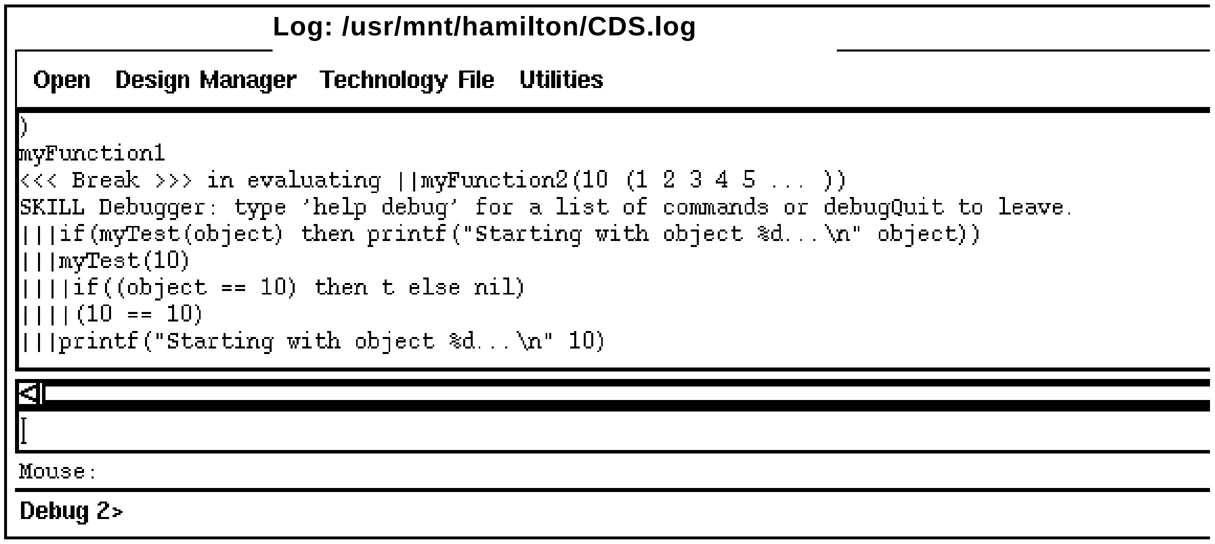

Log: /usr/mnt/hamilton/CDS.log.1
Open Design Manager Technology File Utilities

```
    976 977 978 979 980
    981 982 983 984 985
    986 987 988 989 990
    991 992 993 994 995
    996 997 998 999
)
myFunction1
<<< Break >>> in evaluating ||myFunction2(10 (1 2 3 4 5 ... ))
SKILL Debugger: type 'help debug' for a list of commands or debugQuit to leave.
```

```
Mouse:
Debug 2>
```

A breakpoint has been reached with `i`'s value 10. You can now single step inside `myFunction2` to see why the message is being printed out for when object equals 10 instead of 1.

6. Click *Step* in the SKILL Debugger five times.


Log: /usr/mnt/hamilton/CDS.log
Open Design Manager Technology File Utilities

```
)
myFunction1
<<< Break >>> in evaluating ||myFunction2(10 (1 2 3 4 5 ... ))
SKILL Debugger: type 'help debug' for a list of commands or debugQuit to leave.
|||if(myTest(object) then printf("Starting with object %d...\n" object))
|||myTest(10)
||||if((object == 10) then t else nil)
||||(10 == 10)
|||printf("Starting with object %d...\n" 10)
```

```
Mouse:
Debug 2>
```

From the single stepping above, you can see that `myFunction2` called `myTest` with a value of 10. The `myTest` function returns `t` if the value passed in is equal to 10 but should return `t` if the value is equal to 1.

7. Select *Quit Debugger* in the SKILL Debugger Toolbox.

8. Select *Clear* in the SKILL Debugger Toolbox to clear the breakpoint.
9. In the CIW, type

   ```
   edit( "/tmp/demo.il" t)
   ```

   The system brings up an xterm window with `demo.il` open for editing.
10. On line 40 in `myTest`, change

    ```
    if(object == 10
    ```

    to

    ```
    if(object == 1
    ```
11. Type :wq to exit the editor.

    The file is automatically reloaded. You get a message verifying this and stating which functions have been redefined.

    ```
    function myFunction1 redefined
    function myFunction2 redefined
    function myTest redefined
    ```
12. In the CIW, type

    ```
    myFunction1()
    ```

    This time the program successfully executes:

    ```
    Starting with object 1 ..
    Ending with object 1000 ..
    (1 ... 999)
    ```
13. Close the SKILL Debugger Toolbox.

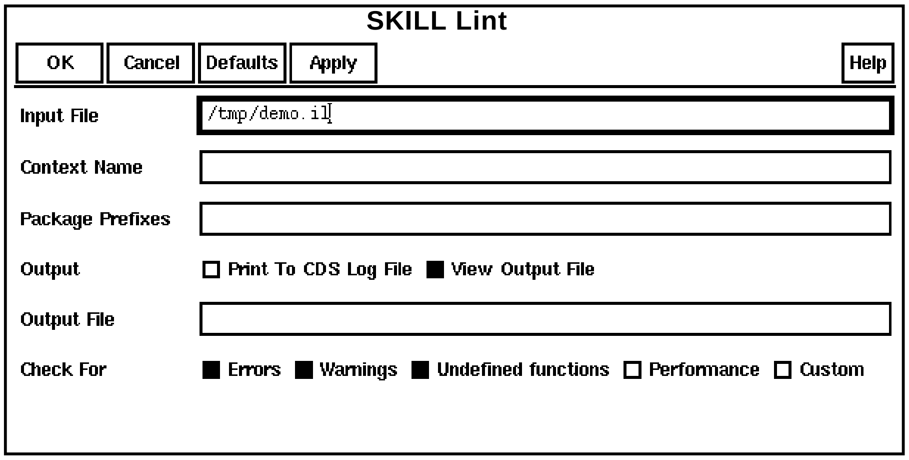
    Congratulations, you now have a working program.

Run Lint

First run SKILL Lint using the default settings.

1. Select *SKILL Lint* from the SKILL Development Toolbox.

The SKILL Lint form appears.

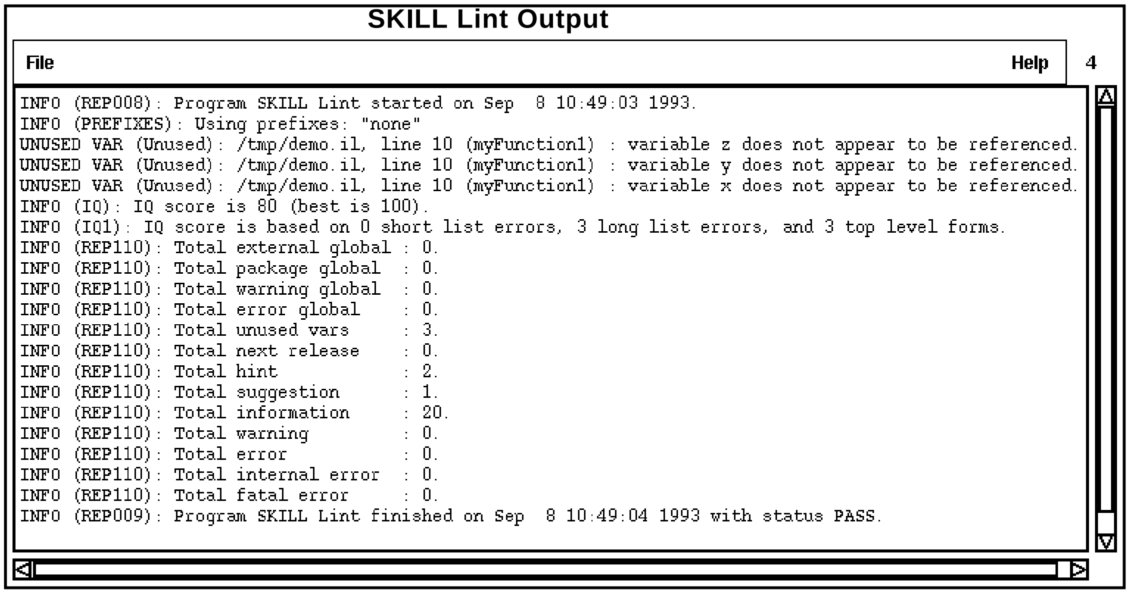


Input File	/tmp/demo.il
Context Name	
Package Prefixes	
Output	☐ Print To CDS Log File ■ View Output File
Output File	
Check For	■ Errors ■ Warnings ■ Undefined functions ☐ Performance ☐ Custom

2. Type `/tmp/demo.il` as the Input File name.

3. Click *OK*.

SKILL Lint processes the `demo.il` program then displays the results in the SKILL Lint Output form.



```
INFO (REP008): Program SKILL Lint started on Sep  8 10:49:03 1993.
INFO (PREFIXES): Using prefixes: "none"
UNUSED VAR (Unused): /tmp/demo.il, line 10 (myFunction1) : variable z does not appear to be referenced.
UNUSED VAR (Unused): /tmp/demo.il, line 10 (myFunction1) : variable y does not appear to be referenced.
UNUSED VAR (Unused): /tmp/demo.il, line 10 (myFunction1) : variable x does not appear to be referenced.
INFO (IQ): IQ score is 80 (best is 100).
INFO (IQ1): IQ score is based on 0 short list errors, 3 long list errors, and 3 top level forms.
INFO (REP110): Total external global : 0.
INFO (REP110): Total package global  : 0.
INFO (REP110): Total warning global  : 0.
INFO (REP110): Total error global    : 0.
INFO (REP110): Total unused vars     : 3.
INFO (REP110): Total next release    : 0.
INFO (REP110): Total hint            : 2.
INFO (REP110): Total suggestion      : 1.
INFO (REP110): Total information     : 20.
INFO (REP110): Total warning         : 0.
INFO (REP110): Total error           : 0.
INFO (REP110): Total internal error  : 0.
INFO (REP110): Total fatal error     : 0.
INFO (REP009): Program SKILL Lint finished on Sep  8 10:49:04 1993 with status PASS.
```

Make changes according to the suggestions in the output. SKILL Lint gives this code an IQ score of 80 out of a possible 100 points:

```
INFO (IQ): IQ score is 80 (best is 100).
```

To improve the score, you must clean up the unused variables indicated with:

```
UNUSED VAR (Unused): /tmp/demo.il, line 10 (myFunction1) :
variable z does not appear to be referenced.
UNUSED VAR (Unused): /tmp/demo.il, line 10 (myFunction1) :
variable y does not appear to be referenced.
UNUSED VAR (Unused): /tmp/demo.il, line 10 (myFunction1) :
variable x does not appear to be referenced.
```

4. Edit the program. Type

   ```
   edit( "/tmp/demo.il" t)
   ```

5. In `myFunction1`, change line 10 from

   ```
   let((x y z myList)
   ```

   to

   ```
   let(( myList)
   ```

   The current code never uses the `x`, `y`, and `z` variables.

6. Type :`wq` to exit the vi editor.

   The file is automatically reloaded.

7. Run SKILL Lint again following steps 1, 2, and 3.

   Notice that SKILL Lint displays the results in the SKILL Output window which includes the line:

   ```
   INFO (IQ): IQ score is 100 (best is 100).
   ```

   SKILL Lint gives this code an IQ score of 100 points:

8. Close the SKILL Lint output window.

9. Run `myFunction1()` again to be sure its operation has not been affected by the change.

   The results should be unchanged.

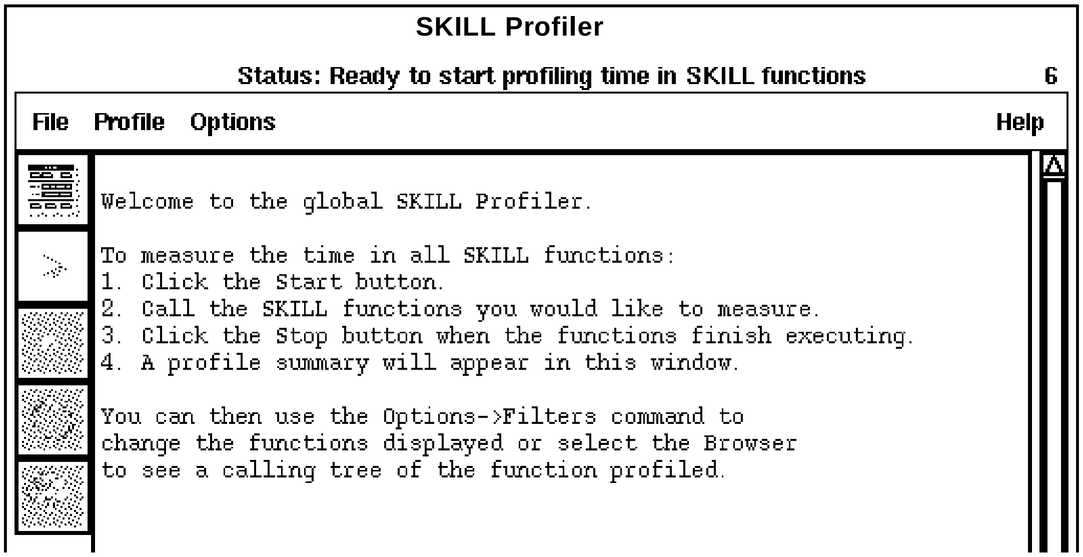
   The following sections will show you how to analyze and fix the performance of the functions in `demo.il` to achieve a significant speedup.

Measure myFunction1

You need to measure the time taken and memory used to run `myFunction1` and analyze the results to gain some insight into why this program takes so long.

1. Select *SKILL Profiler* from the SKILL Development Toolbox.

The SKILL Profiler window appears. Notice that the three lower icons for stop, reset, and browse are not available until you profile a function.



Status: Ready to start profiling time in SKILL functions 6

```
Welcome to the global SKILL Profiler.

To measure the time in all SKILL functions:
1. Click the Start button.
2. Call the SKILL functions you would like to measure.
3. Click the Stop button when the functions finish executing.
4. A profile summary will appear in this window.

You can then use the Options->Filters command to
change the functions displayed or select the Browser
to see a calling tree of the function profiled.
```

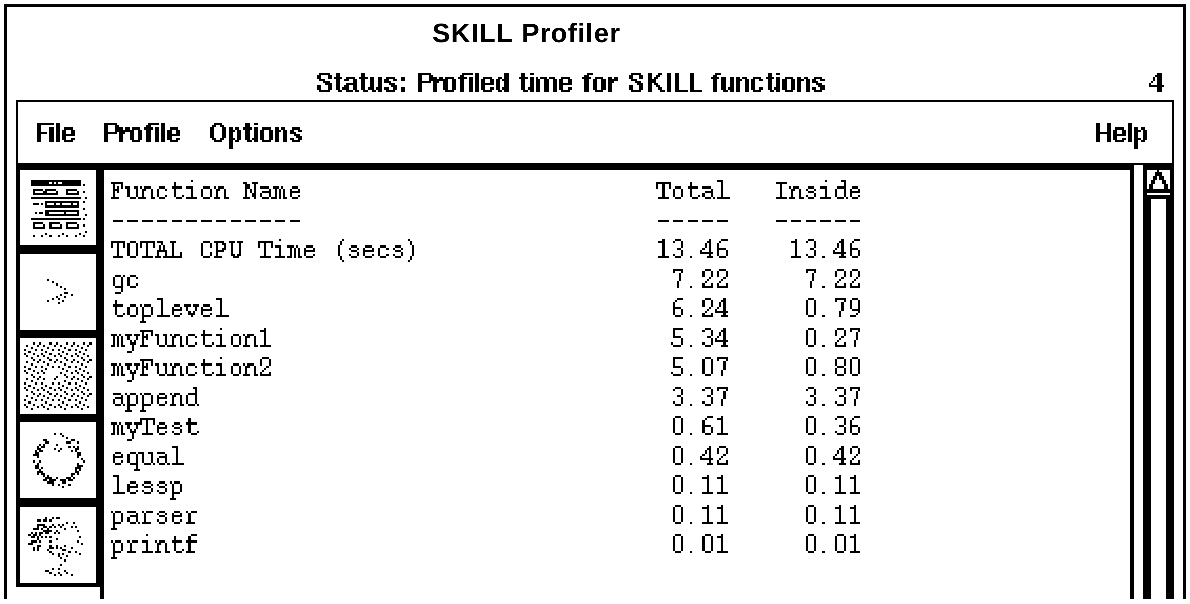
2. In the SKILL Profiler window, select the *Start Profiling* icon to begin profiling.

3. Type `myFunction1()` in the CIW followed by a carriage return.

As soon as you start, only the stop button is available.

4. When `myFunction1` is finished, select the *Stop Profiling* icon to finish profiling.

The results of the profile are printed in the SKILL Profiler window.



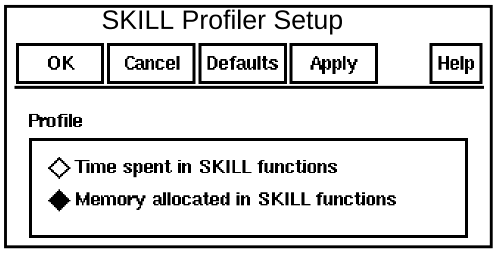
```
Function Name                    Total   Inside
-------------                    -----   ------
TOTAL CPU Time (secs)            13.46   13.46
gc                                7.22    7.22
toplevel                          6.24    0.79
myFunction1                       5.34    0.27
myFunction2                       5.07    0.80
append                            3.37    3.37
myTest                            0.61    0.36
equal                             0.42    0.42
lessp                             0.11    0.11
parser                            0.11    0.11
printf                            0.01    0.01
```

Notice that gc is taking the most time. gc stands for garbage collection and is the time spent by SKILL collecting memory which is no longer being used. You can use the SKILL Profiler to track down large memory producers. By reducing the memory used, you can reduce the time spent in garbage collection.

5. For later reference, select *File -> Save As* and fill in the form that appears with a file name, such as `/tmp/prof.out`.

6. Select the *Set up Profiling* icon to bring up the Setup form.

7. In the Setup form, select *Memory allocated in SKILL functions*. Click *OK*.



Profile

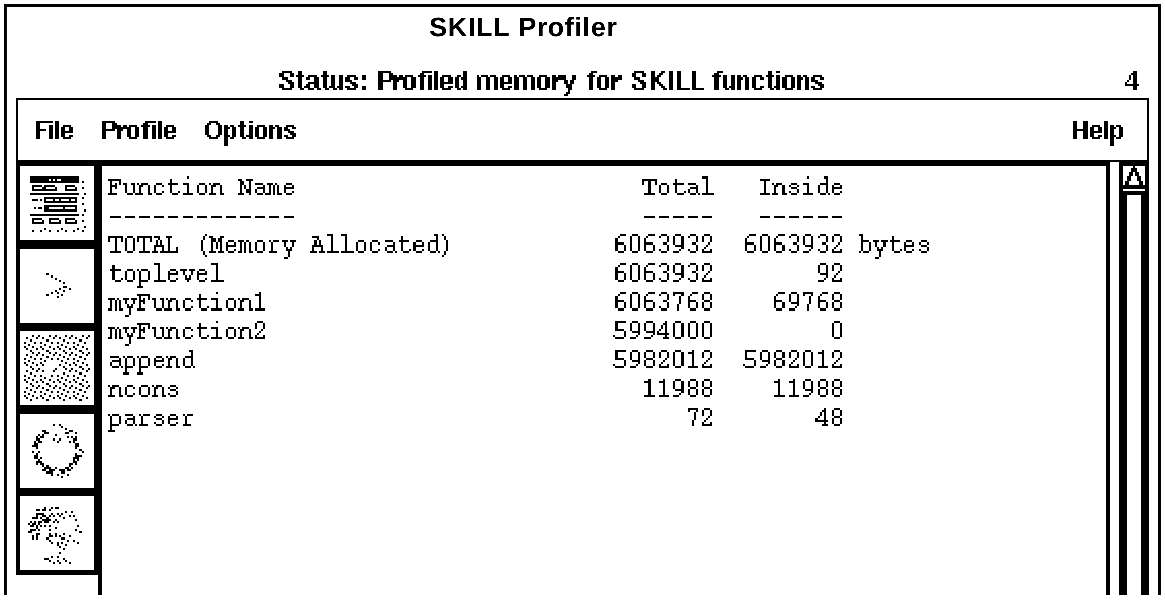
◇ Time spent in SKILL functions
◆ Memory allocated in SKILL functions

8. Select the *Start Profiling* icon in the SKILL Profiler window.

9. Type `myFunction1()` in the CIW.

10. Select the *Stop Profiling* icon in the SKILL Profiler window.

The profiler summary report is displayed.



```
Function Name                        Total   Inside
-------------                        -----   ------
TOTAL (Memory Allocated)           6063932  6063932 bytes
toplevel                           6063932       92
myFunction1                        6063768    69768
myFunction2                        5994000        0
append                             5982012  5982012
ncons                                11988    11988
parser                                  72       48
```

The profile summary indicates that over 6 Megabytes of SKILL memory have been used by our small example.

It also shows that almost all the memory was allocated inside `append`.

The `append` function takes two lists and copies both of them to form a third list. This copying consumes lots of memory.

A much more efficient alternative would be to use a function such as *cons* which adds an element to the beginning of the list and does not require copying the entire list.

Before changing any code, run SKILL Lint to locate potential performance problems in the `demo.il` file.

Check Performance

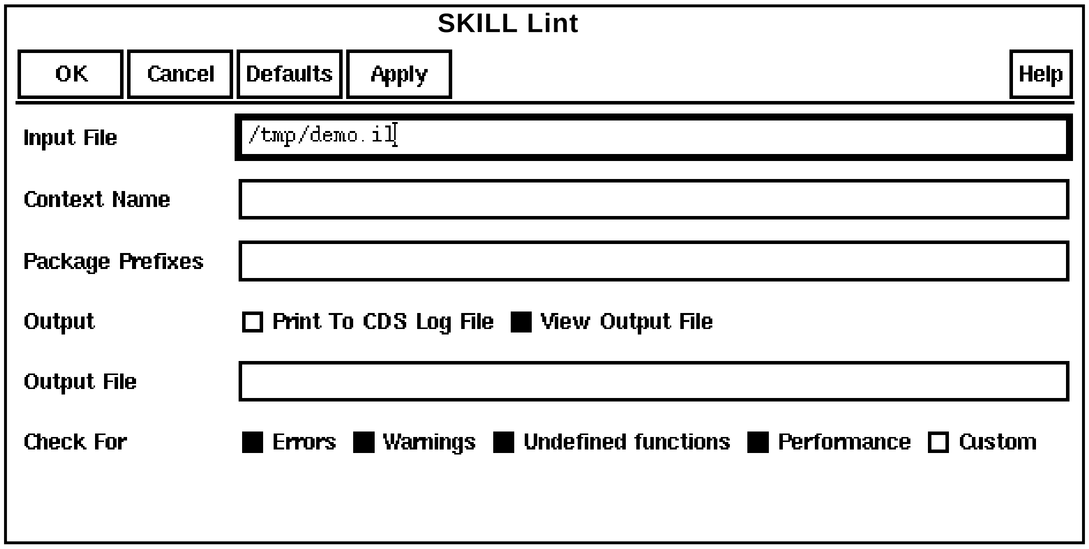
Now let SKILL Lint identify the performance problems of `demo.il`.

1. Select *SKILL Lint* from the SKILL Development Toolbox.

   The SKILL Lint form appears.

   a. Type `/tmp/demo.il` as the Input File name.

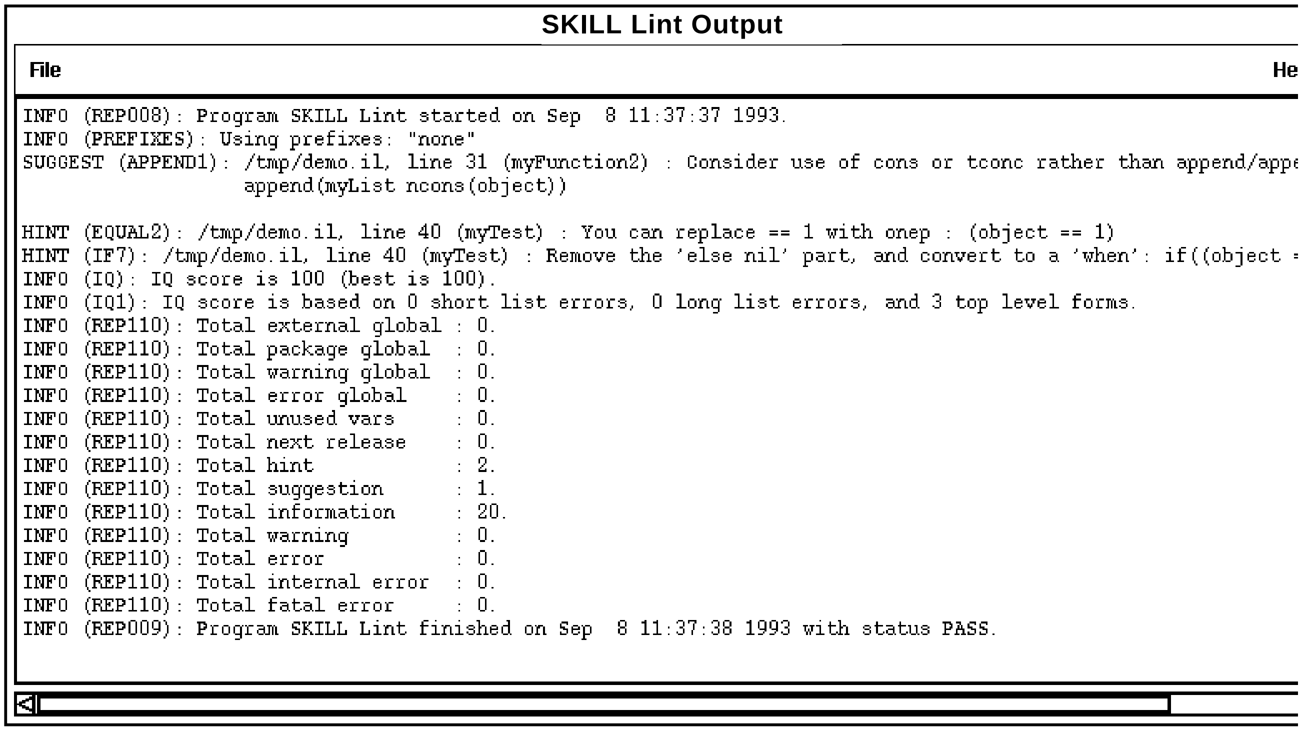
   b. Select the *Check For - Performance* toggle button.



Input File	/tmp/demo.il
Context Name	
Package Prefixes	
Output	☐ Print To CDS Log File ■ View Output File
Output File	
Check For	■ Errors ■ Warnings ■ Undefined functions ■ Performance ☐ Custom

   c. Click *OK*.

SKILL Lint processes the `demo.il` program, then displays the results in the SKILL Lint Output form.



```
INFO (REP008): Program SKILL Lint started on Sep  8 11:37:37 1993.
INFO (PREFIXES): Using prefixes: "none"
SUGGEST (APPEND1): /tmp/demo.il, line 31 (myFunction2) : Consider use of cons or tconc rather than append/appe
                   append(myList ncons(object))

HINT (EQUAL2): /tmp/demo.il, line 40 (myTest) : You can replace == 1 with onep : (object == 1)
HINT (IF7): /tmp/demo.il, line 40 (myTest) : Remove the 'else nil' part, and convert to a 'when': if((object =
INFO (IQ): IQ score is 100 (best is 100).
INFO (IQ1): IQ score is based on 0 short list errors, 0 long list errors, and 3 top level forms.
INFO (REP110): Total external global : 0.
INFO (REP110): Total package global  : 0.
INFO (REP110): Total warning global  : 0.
INFO (REP110): Total error global    : 0.
INFO (REP110): Total unused vars     : 0.
INFO (REP110): Total next release    : 0.
INFO (REP110): Total hint            : 2.
INFO (REP110): Total suggestion      : 1.
INFO (REP110): Total information     : 20.
INFO (REP110): Total warning         : 0.
INFO (REP110): Total error           : 0.
INFO (REP110): Total internal error  : 0.
INFO (REP110): Total fatal error     : 0.
INFO (REP009): Program SKILL Lint finished on Sep  8 11:37:38 1993 with status PASS.
```

Act on the following hints and suggestions:

```
SUGGEST (APPEND1): /tmp/demo.il, line 31 (myFunction2) : Consider
use of cons or tconc rather than append/append1: append(myList
ncons(object))
```

```
HINT (EQUAL3): /tmp/demo.il, line 40 (myTest) : You can replace
== 1 with onep : (object == 1)
```

2. Edit `demo.il`.

a. To use less memory and for faster execution, change line 31

```
then append(myList ncons(object))
```

to

```
then cons(object myList)
```

Note that the program is not concerned about the order of the sequence. Counting up or counting down are equally acceptable.

b. For faster execution, change line 40

```
object == 1
```

to

```
onep(object)
```

Notice that the `if/then/else` clause can be removed because functions return the last expression executed. Finally, you can replace the function `myTest` with `onep` on line 24.

The original code is the following:

```
if(myTest(object)
      then printf("Starting with object %d...\n" object)
)
if(object == 1000
      then printf("End processing with object %d...\n" object)
)
if(object < 1000
      then append(myList ncons(object ))
      else myList
)
```

The recommended changes are:

```
if(onep(object)
      then printf("Starting with object %d...\n" object)
)
if(object == 1000
      then printf("Ending with object %d...\n" object)
)
if(object < 1000
      then cons(object myList)
      else myList
)
```

3. Close the file after your edits.
4. Close the SKILL Lint output file.

Rerun the Profiler

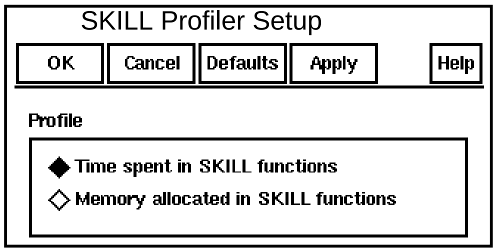
Analyze the results to see whether the recommended changes improved performance.

1. Select *SKILL Profiler* from the SKILL Development Toolbox.

   The SKILL Profiler window appears.

2. Select the *Set up Profiling* icon to bring up the Setup form.

The SKILL Profiler Setup form appears.


SKILL Profiler Setup

Profile

[✓ Time spent in SKILL functions]
Memory allocated in SKILL functions

3. Select *Time spent in SKILL functions*.
4. Click *OK* to close the form.
5. In the SKILL Profiler window, select the *Reset Profiling* icon to clear previous values.

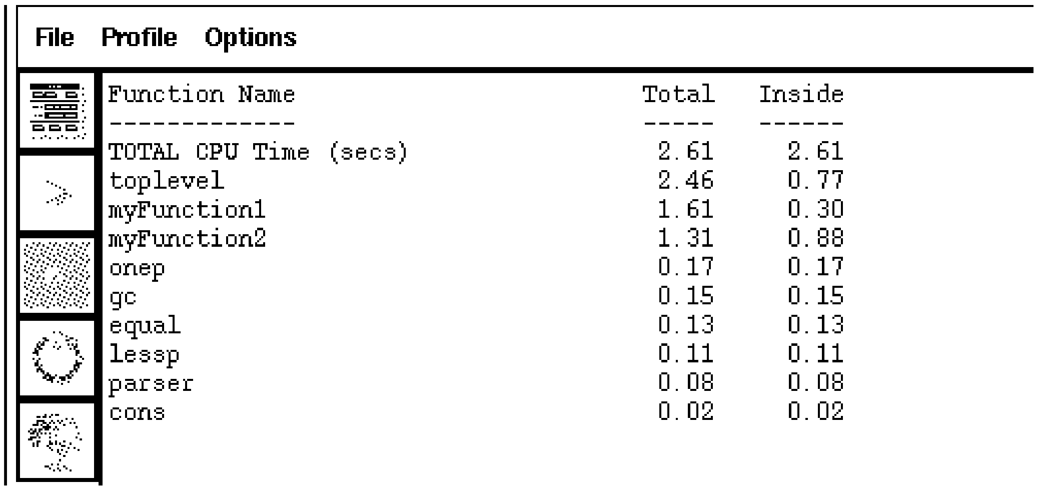
6. In the SKILL Profiler window, select the *Start Profiling* icon to begin profiling.

7. Type `myFunction1()` in the CIW followed by a carriage return.

   Notice that now the numbers are printed in reverse order.

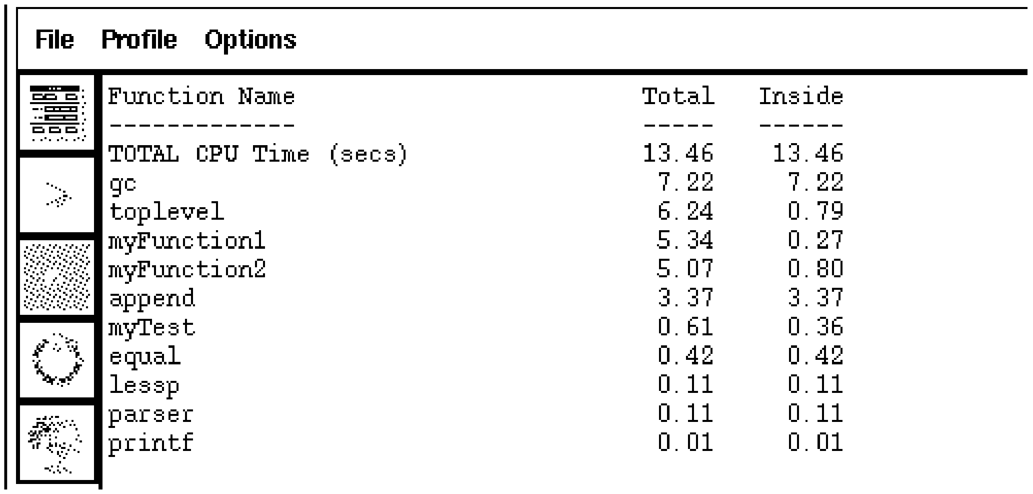
8. When `myFunction1` is finished, select the *Stop Profiling* icon.

The results of the profile are printed in the SKILL Profiler window.



```
Function Name                               Total   Inside
-------------                               -----   ------
TOTAL CPU Time (secs)                        2.61     2.61
toplevel                                     2.46     0.77
myFunction1                                  1.61     0.30
myFunction2                                  1.31     0.88
onep                                         0.17     0.17
gc                                           0.15     0.15
equal                                        0.13     0.13
lessp                                        0.11     0.11
parser                                       0.08     0.08
cons                                         0.02     0.02
```

Compare these times with the previous results below in `/tmp/prof.out` and notice a significant speedup.



```
Function Name                               Total   Inside
-------------                               -----   ------
TOTAL CPU Time (secs)                       13.46    13.46
gc                                           7.22     7.22
toplevel                                     6.24     0.79
myFunction1                                  5.34     0.27
myFunction2                                  5.07     0.80
append                                       3.37     3.37
myTest                                       0.61     0.36
equal                                        0.42     0.42
lessp                                        0.11     0.11
parser                                       0.11     0.11
printf                                       0.01     0.01
```

Using the Non-Graphical SKILL Debugger

Here is a brief example of how to use the non-graphical SKILL debugger. It is a trivial example intended to give you a sense of what a debugging session is like. This example uses traditional Lisp style syntax.

```
(defun initItem ( item )
      (let ((shapeList '(path polygon ellipse))shape)
            (setq shape (concat (get item 'shape)))
            (if (memq shape shapeList)
(printf "\nPath %s initialized" (get item 'name))
)))
(installDebugger)        ; Install the SKILL debugger.
(sstatus traceArgs t)    ; Keep evaluated arguments for
                         ; Stacktrace display.
(alias q debugQuit )     ; Alias debugQuit to a shorter name.
(putprop 'item1 "path" 'shape)
(initItem 'item1)
*** Error in routine fprintf/sprintf:
Message: *Error* fprintf/sprintf: format spec.incompatible. SKILL
Debugger: type ‘help debug' for a list of commands or debugQuit to
leave.
Debug 2> where
<<< Stack Trace >>>
errorHandler("fprintf/sprintf" 0 t nil ("*Error* fprintf.))
printf("\nPath %s initialized" nil)
if(memq(shape shapeList) printf("\nPath %s initialized"...
      shape = path
      shapeList = (path polygon ellipse)
let(((shapeList '&) shape) (shape = concat(get(item &)))
      item = item1
initItem(item1)
5
Debug 2> q
1>
```

The stack trace generated by `where` shows that `printf` expected a string but got `nil` because there was no property called `name` on `item1`.

Notice that `where` prints out the local variables and their values in each function. Other functions such as `stacktrace` and `dump` also help you examine the state of the program at the point the error occurred. Setting breakpoints and single stepping can be used for more difficult bugs.

9 Command Line Interface

This document discusses

- Command Line: Profiler on page 87
- Command Line: Test Coverage on page 88
- TCov Report Files on page 89

Command Line: Profiler

Most executables containing the Cadence® SKILL language take command line options to turn on SKILL Profiling. The SKILL Profiler can be turned on when you start an executable by passing it the following arguments.

```
executableName -ilProf [time/memory] -ilProfFile [filename]
```

Because you enter these commands at the shell level, you use the shell syntax, hence the dash options. For example:

```
cds0 -ilProf memory -ilProfFile /tmp/profMem.out
```

ilProf

Turns on SKILL Profiling for `time` by default. If `memory` is given as the argument, that is used instead.

ilProfFile

Specifies the destination file for the SKILL Profiling results. The file name should follow this argument. The default is `ilProf.out` in the directory from which the executable was started.

When the executable is exited, the profile summary file is written out.

Command Line: Test Coverage

SKILL Test Coverage lets you determine which code was executed during a session. This information lets you increase the coverage of your test cases and thereby improve the quality of your SKILL code.

When you start up SKILL test coverage, you must pass the executable command line arguments telling SKILL which files or context to measure. When those contexts or files are loaded, they automatically compile the functions to include `tCov` instructions. When the SKILL session ends, report files are written out.

➤ Run SKILL Test Coverage using the following command line arguments.

```
executableName -ilTCov [context/fileNames]
-ilTCovDir directory -ilTCovReportsOnly
```

Because you enter these commands at the shell level, you use the shell syntax, hence the dash options and the single quotes around the context file names, to submit them as one argument to the option.

For example:

```
cds0 -ilTCov 'hiBase.cxt hiTools.cxt' -ilTCovDir /tmp/test
```

ilTCov

Followed by a list of contexts is the only argument required to run SKILL Test Coverage. Alternatively, you can pass in a list of SKILL files in the current directory.

ilTCovDir

Takes the directory into which all report files are written as its argument. For contexts, a subdirectory for each context is created under the directory given, and report files for that context are written into that directory. If this argument is not given, the report files are written to the same directory from which the SKILL files being measured were loaded.

ilTCovReportsOnly

Allows you to print only the summary report files and not the files that actually show the source code annotated with test coverage information. This option greatly reduces exit time during test coverage.

TCov Report Files

When the SKILL session is over and SKILL exits, the four report files for SKILL Test Coverage listed below are written out.

ilTCovSummary

The overall summary file that contains the percent of expressions executed for all contexts and functions monitored. This file is written to the directory from which the SKILL executable was started.

contextName.tcovSum

A summary report for the given file or context showing what percentage of expressions and functions were executed. This file is placed in the directory containing the source code for the context or in a directory under the
`-ilTCovDir`, if one is given.

fileName.tcov

A `tcov` file for each source file or source file in a context showing each function definition and which expressions were executed. This file is placed in the directory containing the source code or in a directory under the `-ilTCovDir`, if one is given.

contextName.d

A temporary file used to collect data across multiple runs. This file is placed in the directory containing the source code or in a directory under the
`-ilTCovDir`, if one is given.

10 Writing SKILL Lint Rules

This document discusses

■ Overview on page 91

■ Rule Structures - SK_RULE Macro on page 92

■ Rule Access Macros on page 93

■ Rule Reporting Macros on page 94

■ Advanced Rule Macros on page 95

■ Storing Rule Definitions on page 99

- ❑ /cds/tools/local/sklint/rules on page 99
- ❑ /cds/tools/sklint/rules on page 99

■ Examples on page 99

- ❑ Adding a New Required Argument on page 99
- ❑ Replacing One Function With Another on page 100
- ❑ Promoting Standard Format Messages on page 101
- ❑ Making the Code Look Nicer on page 101

Overview

Cadence® SKILL Lint has been extended to allow users to write their own rules to output SKILL Lint messages.

While there are a large number of built-in rules within SKILL Lint, there are times when specific rules are required according to a user’s own situation.

This document details how to write rules for SKILL Lint and gives some examples of the types of rules that can be written.

Rule Structures - SK_RULE Macro

The main structure of the rules is as follows:

```
SK_RULE( sl_functions g_test g_statement ...)
```

The `SK_RULE` macro is the main entry point for writing a rule. A number of macros are provided for writing rules. These macros all use capital letters and the prefix `SK_` only.

sl_functions

The first argument is the name of the function to which the rule applies. Rules in SKILL Lint always apply to a particular function. For example, there is a rule associated with the `setq` function (the assignment operator) which says that the first argument must be a symbol. The first argument to `SK_RULE` may be a single function name, or it may be a parenthesized list of function names if the same rule is to be applied to multiple functions.

g_test

The second argument is a single SKILL statement. This is known as the test statement. The rules work by applying a series of commands whenever a call to the function(s) named is found in the code under analysis. The test function is evaluated first, and the rest of the commands are carried out only if the test function evaluates to non-nil.

g_statement ...

Subsequent arguments are the rules commands, which are executed whenever a call to the named function(s) is found, providing that the test statement evaluates to non-`nil`. These commands can be any SKILL statements

While the rule command statements are being evaluated, a number of macros are available for accessing the SKILL code being checked and for reporting any problems found. These macros are all detailed in the Rule Reporting Macros section. The simplest macro is `SK_ARGS()`, which takes no arguments itself, and returns the list of arguments to the function call being tested.

SK_RULE Example

This simple rule checks for calls to the `ggTestData` function, which currently has two arguments, plus an optional third. Suppose in the next release, the third argument becomes mandatory. We then want to find any current calls with only two arguments:

```
SK_RULE( ggTestData
      length(SK_ARGS()) == 2
      warn(“Found call to ggTestData with
                  only 2 arguments.\n”)
)
```

- The first argument to `SK_RULE` specifies that the rule is to be applied to any calls to the function `ggTestData`.
- The second argument is a test that the number of arguments, as returned by the `SK_ARGS()` macro, is 2.
- The final statement, which is only carried out if the test was true, prints out a warning to the user that such a call was found.

Rule Access Macros

You can use the following macros in either the test statement or the rule commands.

SK_ARGS()

Returns the list of the arguments to the function call under test. This macro takes no arguments. The list values returned by this macro should never be destructively altered (using `rplaca` etc.) because that would produce unknown effects.

SK_CUR_FILENAME()

Returns the name of file currently being checked, within a SKILL Lint rule. For example:

```
SK_RULE( test
         t
         printf( "Current file being checked is: ’%s’\n"
                SK_CUR_FILENAME() )
)
```

SK_NTH_ARG(n)

Returns the appropriate argument in the function call. The single argument to this macro specifies an argument number. The argument number is zero-based, so that argument 1 is the second argument to the function call. The list values returned by this macro should never be destructively altered (using `rplaca` etc.) because that would produce unknown effects.

SK_FUNCTION()

Returns the name of the function call under test. This might be needed to establish the function name where the same rule is being used for several different functions. The list values returned by this macro should never be destructively altered (using `rplaca` etc.) because that would produce unknown effects.

SK_FORM([n])

Returns the entire function call under test as a list. The `SK_ARGS()` macro effectively is the same as `cdr(SK_FORM())` and the `SK_FUNCTION` macro is effectively `car(SK_FORM())`.

If an argument is given, then this macro returns the call further up the call stack. For example, if an `if` is called from within a `foreach` which is within a `let`, then `SK_FORM(2)` returns the call to `let`. Note that `SK_FORM(0)` is just `SK_FORM()`. The list values returned by this macro should never be destructively altered (using `rplaca` etc.) because that would produce unknown effects.

Rule Reporting Macros

The following macros allow the reporting of problems to the user in the same format as given by the standard messages generated by SKILL Lint.

```
SK_ERROR( type format arg ...)
SK_WARNING( type format arg ...)
SK_HINT( type format arg ...)
SK_INFO( type format arg ...)
```

These macros allow reporting of hints, warnings and errors to the user. The arguments are the identifier for the message, the format string, as used by `printf`, and the arguments for printing. For example:

```
SK_WARNING( GGTESTDATA “This function now requires 3 arguments: %L\n”
SK_FORM())
```

This will print a message of the form:

```
WARN (GGTESTDATA) myFile.il line 32 : This function now requires
3 arguments: ggTestData(abc 78.6)
```

The file name and line number are added automatically.

These macros should be used within the commands of the rule to report messages to the user when problems are found. To allow the user to control the reporting of these messages in the same way as other SKILL Lint messages, it is necessary first to register the messages with the reporting system. This is necessary only to allow the user to disable particular messages. To do this, add a call to the following macro OUTSIDE of the rule definition:

```
SK_REGISTER( type )
```

For example, add the following:

```
SK_REGISTER( GGTESTDATA )
```

outside of the call to `SK_RULE` to register the message type. Once the call to `SK_REGISTER` has been carried out, the SKILL Lint user will see the message identifier on the Message Customization form, and will thus be able to disable the reporting of this message.

The example rule shown above could thus be written as:

```
SK_RULE( ggTestData
      length(SK_ARGS()) == 2
      SK_ERROR( GGTESTDATA “This function now requires 3 arguments: %L\n”
      SK_FORM())
)
SK_REGISTER( GGTESTDATA )
```

Advanced Rule Macros

The following are some more advanced rule macros that *are not expected to be generally required*.

SK_CHANGED_IN(t_release)

This macro is used to specify the release version (e.g. "447" for IC4.4.7) that a function is changed. The `SK_CHANGED_IN()` macro must be embedded as the second argument of `SK_RULE`. For example:

```
SK_RULE( myFunc
     SK_CHANGED_IN("447")
     SK_INFO( myFunc
     . . .
)
```

`SK_CHANGED_IN()` evaluates to non-nil if the code being checked, as specified with the sklint argument ?codeVersion, is from an earlier release than the release specified through the argument of `SK_CHANGED_IN()` and the SKILL Lint rules message that describes function change (only) will not be reported. The argument must me a numeric string of the release version (e.g. "447" for IC4.4.7). If `?codeVersion` is not specified, `SK_CHANGED_IN()` will always evaluate to nil and a function change rules message will be reported.

This macro is useful when the user wants to restrict reporting of function change rule messages which occurred after the release for which the code being checked was written. When users check the code in IC447 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).

If the function changes more than once, then there should be a separate SKILL Lint rule for each change, each with a different `SK_CHANGED_IN()` macro.

Note: `SK_CHANGED_IN()` should only be used for filtering out function changed rule messages. Function deleted rule messages should always be reported.

SK_CHECK_STRINGFORM(t_stringForm)

This macro is similar to SK_CHECK_FORM() but it is used to check SKILL form in strings (e.g. callback string). This macro is added to deal with the problem that when a string form is converted to a SKILL form, the line number of the string form will be messed up and causes an incorrect line number to be reported.

An example of usage:

```
procedure( test()
     let( (c)
        c = myFunc(
             "foreach(i ’(1 2 3 4) a=i)"
        )
      c
     )
)
SK_RULE( myFunc
             t
             SK_CHECK_STRINGFORM( SK_ARGS() )
)
```

Note: The argument to SK_CHECK_STRINGFORM must be a string.

SK_RULE(SK_CONTROL ...)

The `SK_RULE` macro has an optional first argument which is the keyword `SK_CONTROL`. When this keyword is given, it means that this rule is a “controlling” rule. This means that the arguments to the function are not themselves checked by SKILL Lint. Usually, SKILL Lint will first apply checking to all the arguments of a function call and then to the call itself. However, if there is a controlling rule, then the arguments are not checked automatically. This type of rule is generally needed for `nlambda` expression (for example `nprocedures`) where only some of the arguments are evaluated.

SK_CHECK_FORM(l_form)

This macro can be used to apply checking to a statement. This is generally useful within a controlling rule. The argument is a list whose first element is the SKILL code to be checked.

For example, consider a rule to be written for the `if` function (ignoring for the moment that there are internal rules for `if`.) This function evaluates all its arguments at one time or another, except for the `then` and `else` keywords. Writing a rule for `if` would require a controlling rule, which would call this macro to check all the arguments except for the `then` and `else`. For example:

```
SK_RULE( SK_CONTROL if
      t
      foreach(map statement SK_ARGS()
      unless(memq(car(statement) ‘(then else))
            SK_CHECK_FORM(statement)
)
      )
)
```

The `SK_CONTROL` keyword means that the arguments to `if` will not be checked automatically. The test in this case is `t`, which means that the rule will be applied to all calls to `if`. The rule command is a call to `foreach`, with `map` as the first argument. Each time through the loop the statement is a new `cdr` of the arguments. We check that this is not a `then` or `else`, and if not, then call `SK_CHECK_FORM` to check the argument.

Note: The argument to `SK_CHECK_FORM` ***must be a list*** whose first element is the statement to check, not the statement itself.

It is important to call the checker on all appropriate arguments to a function, even if they are just symbols, because the checker handles trapping of variables which are unused, or are illegal globals and so forth.

There should only be a single control rule for any function.

SK_PUSH_FORM(l_form)
SK_POP_FORM()

These two macros are used to indicate an extra level of evaluation, such as is introduced by the various branches of a `cond` or `case` function call. These macros should not be needed by most user rules. They are used in very special circumstances to indicate to the dead-code spotting routines where branches occur in the code.

SK_PUSH_VAR(s_var)

Declares a new variable. For example, the rules for `let`, `setof`, etc. declare the variables in their first argument using this function. The function should be called *before* calling `SK_CHECK_FORM` on the statements in the body of the routine.

SK_POP_VAR(s_var [dont_check])

Pops a variable that was previously declared by `SK_PUSH_VAR`. Unless the second argument is `t`, the variable is checked to see whether it was used by any of the statements which were checked between the calls to `SK_PUSH_VAR` and `SK_POP_VAR`.

For example, consider a new function called `realSetOf`. Assume this function works just like `setof`, except that it removes any duplicates from the list that is returned. The rule is a control rule which pushes the variable given as the first argument, checks the rest of the arguments, and then pops the variable, checking that it was used within the loop:

```
SK_RULE( SK_CONTROL realSetOf
      t
      SK_PUSH_VAR(car(SK_ARGS()))
      map(‘SK_CHECK_FORM cdr(SK_ARGS()) )
      SK_POP_VAR(car(SK_ARGS()))
      )
```

SK_USE_VAR(s_var)

Marks the given variable as having been used. Usually a variable is marked as having been used if it is passed to a function. However, if a function has a controlling rule, and does not call `SK_CHECK_FORM` then it might wish to mark a variable as having been used. For example, the rule for `putprop` marks the first argument as having been used. The same rule ignores the third argument (the property name) and calls the checker on the second argument. If `putprop` did not have a controlling rule, then the symbol used for the property name would get marked as having been used and would probably be reported as an error global.

SK_ALIAS(s_function s_alias)

This macro can be used where one function should be checked with the same rules as another function. For example, it is fairly common to see functions replacing `printf`, which add a standard prefix to the function. For example:

```
procedure( ERROR(fmt @rest args)
      fmt = strcat(“ERROR: “ fmt)
      apply(‘printf cons(fmt args))
      )
```

It would be nice to check calls to ERROR with the same rules as are used for `printf` (mainly to check that the number of arguments matches that expected by the format string.) This can be achieved using the following call:

```
    SK_ALIAS( ERROR printf )
```

This macro, like `SK_REGISTER`, is used outside of any rule definitions.

Storing Rule Definitions

Rule definitions should be placed in files and stored in the Cadence distribution hierarchy. The files must be placed in *one* of two places.

/cds/tools/local/sklint/rules

Given a Cadence installation in `/cds`, the files should be placed in:
`/cds/tools/local/sklint/rules` and named with a `.il` extension.
These files are loaded on each run of SKILL Lint.

/cds/tools/sklint/rules

Alternatively, the files may be stored in: `/cds/tools/sklint/rules`. These files are loaded only the first time SKILL Lint is run (actually when the SKILL Development environment context is loaded.)

It is recommended that all user rules are placed in the `.../local/sklint/rules` directory, because these are not likely to be removed when a new release of Cadence code is installed. Also, while the rules are under development it is more useful to have the rules loaded on each run of the tool.

Examples

The following sample rules show how the macros are used.

Adding a New Required Argument

Suppose in the next release of the code, the `ggTestData` function is being changed so that a new third argument is required. The following rule is provided by the group which supplies `ggTestData` to trap problems which will arise at the next release:

```
SK_RULE( ggTestData
      length(SK_ARGS()) == 2
      SK_WARNING( GGTESTDATA
      strcat( "This function will require 3 arguments in the
next release: %L\n"
      "The extra argument will specify the width of the widget.\n")
SK_FORM())
)
SK_REGISTER( GGTESTDATA )
```

Replacing One Function With Another

Suppose the standard `setof` function is not a true `setof` because it doesn't remove repeated elements (it is a `bagof` function.) A replacement, called `trueSetof`, does remove repeated elements. The rule needs to handle the fact that the first argument is a loop variable. Also, `trueSetof` allows multiple statements in the body of the function call, instead of the one statement allowed by `setof`.

The following controlling function

- Declares the loop variable.
- Checks the body statements (not forgetting the one which defines the original set).
- Checks that the loop variable was used.

```
SK_RULE( realSetof
      t
      let( ((args SK_ARGS()))
      when(symbolp(car(args))
            SK_PUSH_VAR(car(args))
            )
            map('SK_CHECK_FORM cdr(args))
            when(symbolp(car(args))
            SK_POP_VAR(car(args))
            )
      )
)
```

Note that:

- This rule used `let` to declare a local variable for the args to save calling the `SK_ARGS()` macro multiple times.
- We checked the loop variable was a symbol, in case the user did something very odd, but we did not at this time report a problem if the loop variable is not a symbol.

A second rule can be defined to check that the loop variable is given as a symbol:

```
SK_RULE( realSetof
      !symbolp(car(SK_ARGS()))
      SK_ERROR( REALSETOF1 "First argument must be a symbol: %L\n"
            SK_FORM())
```

```
    )
    SK_REGISTER( REALSETOF1 )
```

Promoting Standard Format Messages

To promote standard format messages, suppose a new system has been written which provides three new function, `ggInfo`, `ggWarn` and `ggError`. The functions work similarly to `printf`, taking the same arguments, but they change the format a little and also copy the messages to various log files. A rule is needed to check that the format string matches the given number of arguments. This rule is exactly that which is applied to the `printf` function itself, so we want to alias these functions to use the same rules as for `printf`:

```
    SK_ALIAS( (ggInfo ggWarn ggError) printf )
```

Making the Code Look Nicer

To make the code look nicer, suppose we want to prevent lots of nested calls to the boolean operators, `null`, `or` and `and`. For example:

```
    !a && ((b || !c) && (!d || !b)
```

is difficult to understand and should probably be split into several statements with associated comments. The rule we want is to look at a call to one of the boolean operators and see whether there are other boolean operators within it:

```
    SK_RULE( (null and or)
          ggCountBools(SK_FORM()) > 5
          SK_HINT( BOOLS “Lots of boolean calls found : %L\n”
                SK_FORM())
          )
          SK_REGISTER( BOOLS )
```

The `ggCountBools` function might be:

```
    procedure( ggCountBools(args)
          let( ((i 0))
                foreach(arg args
                      when(listp(arg) && memq(car(arg) ‘(null or and))
                            i = i + 1 + ggCountBools(cdr(arg))
                      )
                )
                i
          )
    )
```

The only problem with this is that it tends to report problems more than one time, where booleans are nested deeply. We can improve this by looking at the function call higher in the call stack and seeing whether that is a boolean function itself. If so, then there is no point in checking the current call:

```
SK_RULE( (null and or)
      !memq(car(SK_FORM(1)) ‘(null and or)) &&
                        ggCountBools(SK_FORM()) > 5
      SK_HINT( BOOLS “Lots of boolean calls found : %L\n”
                              SK_FORM())
      )
```

11

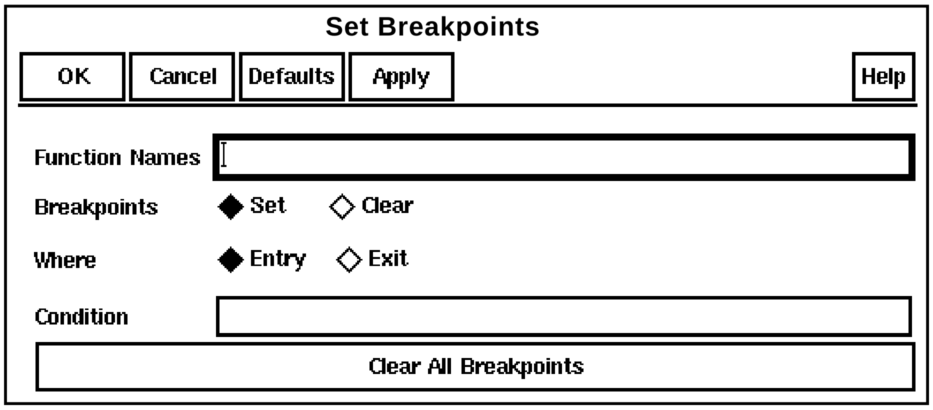
Set Breakpoints Form

This document discusses

- Overview on page 103
- Function Names on page 104
- Breakpoints on page 104
- Where on page 104
- Condition on page 104
- Clear All Breakpoints on page 105
- SKILL Functions on page 105

Overview

You bring up the Set Breakpoints Form by selecting *Set Breakpoints* on the Cadence® SKILL Debugger Toolbox.



Function Names
Breakpoints [✓ Set] Clear
Where [✓ Entry] Exit
Condition

Function Names

A list of functions on which to set or clear breakpoints. Separate each function from the others by a space.

Breakpoints

Determines whether the breakpoints are set or cleared.

Set

Sets breakpoints for the function names given.

Clear

Clears breakpoints for the function names given.

Where

Allows you to hit a breakpoint when entering or exiting a function.

Entry

Sets a breakpoint for when the function is entered.

Exit

Sets a breakpoint for when the function is exited.

Condition

Allows you to give a condition in SKILL code that must evaluate to `t` to reach a breakpoint. The condition is evaluated in the scope of the calling function. An example of a condition would be: `i==10`.

Clear All Breakpoints

Clears all breakpoints.

SKILL Functions

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

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

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

step( [x_steps])
next( [x_steps])
stepout( [x_steps])
cont() or continue()
debugStatus() => nil
clear() => t
installDebugger() => t/nil
uninstallDebugger() => t/nil
breakpt( s_function...) => g_result
unbreakpt( s_function... | t) => g_result
```