SKILL Language User Guide

Product Version 06.10
March 2003

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

Before You Start

This preface contains the following sections:

About This Manual

The SKILL language user guide

- Introduces the SKILL language to new users
- Leads users to understand advanced topics
- Encourages sound SKILL programming methods
- Introduces the Cadence® SKILL++ Language, the second generation extension language for the CAD tools from Cadence

Companion Reference Manual

Companion information for this user guide is the online manual, *SKILL Language Functions Reference*

Audience

This manual is intended for the following users:

- People learning to program
- Tool users who want to know some of the basics of SKILL
- Programmers beginning to program in SKILL.
- CAD developers who have experience programming in SKILL, both internal users and customers
- CAD integrators

About the SKILL Language

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

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

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

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

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

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

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

Quick Look at All the Chapters

Chapter 1, "Getting Started," introduces you to the SKILL programming language. This chapter shows you how to enter simple SKILL expressions and understand the system output. It introduces the list, a data structure that is central to SKILL. It presents basic file input/output functions and shows several ways to control program flow. It introduces the basic steps to creating your own SKILL procedures.

Chapter 2, "Language Characteristics," explains the basic structure and syntax of the SKILL language. It presents details of data structures introduced in chapter one, uniting basic data information in one place.

Chapter 3, "Creating Functions in SKILL," fills you in on more of the details about constructs for defining a function, identifying data types for function arguments, defining parameters, and defining local and global variables.

Chapter 4, "Data Structures," presents an in-depth view of data structures, such as symbols, property lists, defstructs, arrays, strings, and association tables. It describes what they are and how to use them.

Chapter 5, "Arithmetic and Logical Expressions," presents operators and predefined SKILL arithmetic and logical functions. It shows you how to create expressions using the operators.

It explains how SKILL performs a sequence of function evaluations and deals with function overloading.

Chapter 6, “Control Structures,” introduces more branching, iteration, and selection functions. It demonstrates how to declare local variables using *prog*. It details how to group statements where only a single statement would otherwise be allowed.

Chapter 7, “I/O and File Handling,” describes the SKILL file system interface. It presents functions that interact with ports, create formatted or unformatted output, scan input strings or characters, and manage files and directories.

Chapter 8, “Advanced List Operations,” presents an in-depth view of lists, including a conceptual overview. It discusses more advanced SKILL list functions and how to use them. As you become more familiar with SKILL, you will probably be using and building lists more frequently.

Chapter 9, “Advanced Topics,” contains concepts of interest to advanced users. It discusses SKILL architecture and implementation, evaluation, function objects, the *lambda* function designator, differences in C and SKILL macros, error handling, talking to the SKILL top level, and memory management (garbage collection). It also describes dynamic and lexical scoping.

Chapter 10, “Delivering Products,” describes contexts, which are binary representations of the internal state of the interpreter. This chapter provides information the developer needs to decide when it is better to use contexts as opposed to straight or encrypted SKILL code.

Chapter 11, “Writing Style,” shows ways to make your SKILL programs understandable, reusable, extensible, efficient, and easy to maintain.

Chapter 12, “Optimizing SKILL,” shows you what to do if code is not running as fast as it should. It presents the when, what, and how of optimizing code. It helps you focus your efforts to improve your code.

Chapter 13, “About SKILL++ and SKILL,” introduces the Cadence-supplied Scheme called SKILL++, which is the second generation extension language for the CAD tools from Cadence.

Chapter 14, “Using SKILL++,” focuses in detail on the key areas in which SKILL++ semantics differ from SKILL semantics.

Chapter 15, “Using SKILL and SKILL++ Together,” discusses the pragmatics of developing SKILL++ programs. It identifies several more factors to consider in creating applications which involve tightly integrated SKILL and SKILL++ components, such as, selecting a language, partitioning an application, cross calling between SKILL and SKILL++, and debugging a SKILL++ program.

Chapter 16, “SKILL++ Object System,” describes a system that allows for object-oriented interfaces based on classes and generic functions composed of methods specialized on those classes. A class can inherit attributes and functionality from another class known as its superclass. SKILL++ class hierarchies result from this single inheritance relationship.

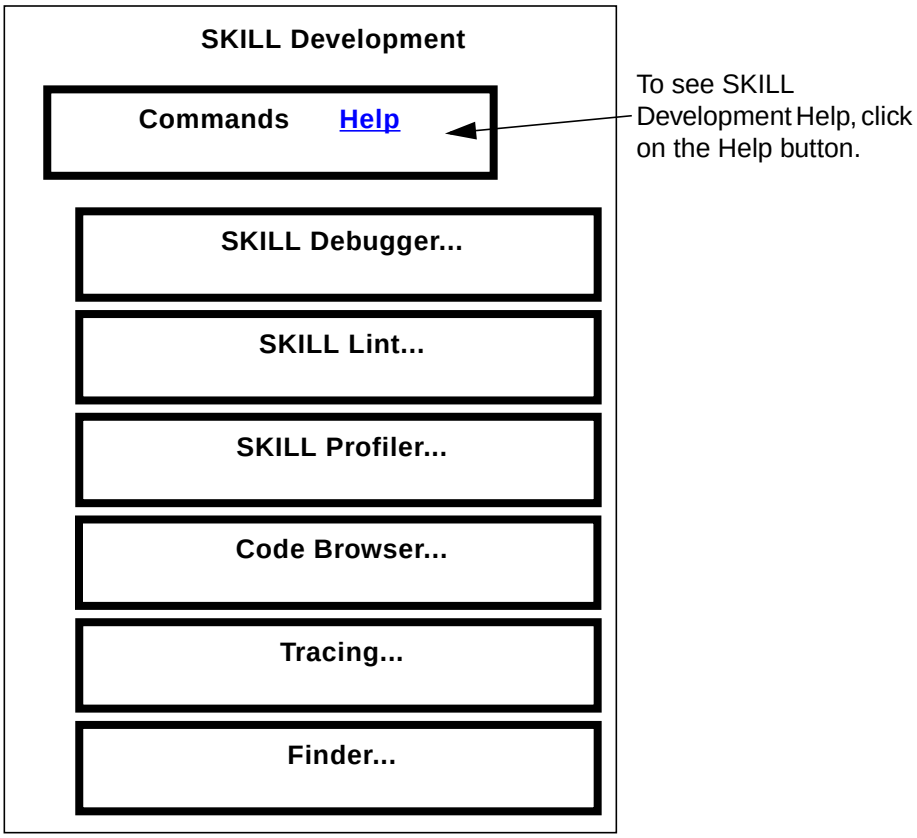
Chapter 17, “Programming Examples,” presents a variety of samples of SKILL code.

What’s New

The following changes to this document have been made for this release.

- Naming Conventions on page 62 describes Cadence naming conventions for functions, variables, and their arguments has been updated.
- The table of scaling factors that can be added on at the end of a decimal number in Scaling Factors on page 73 has been updated.

SKILL Development Help



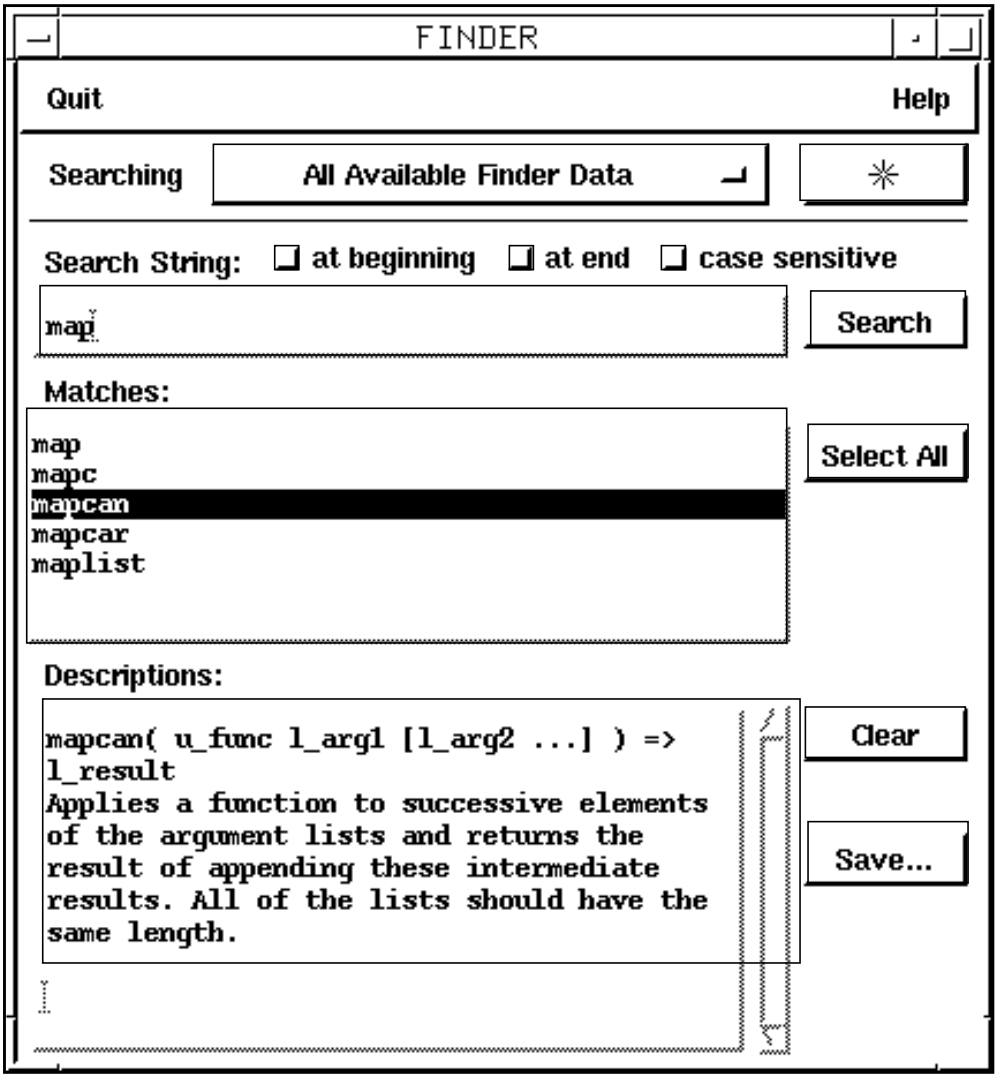
Information about the SKILL Development Toolbox is available in SKILL Development Help, which you access by clicking the *Help* button on the toolbox. Use this source for toolbox command reference information.

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

For a list of SKILL lint messages, and message groups, refer to *SKILL Development Help*.

Quick Reference Tool - Finder

Quick reference information for syntax and abstract statements for SKILL language functions and application procedural interfaces (APIs) is available using the Finder, a new tool accessible from the SKILL Development Toolbox or from UNIX.



For more information refer to "Finder" in *SKILL Development Help*.

Copying and Pasting Code Examples

You can copy examples from windows and paste the code directly into the Command Interprete wINDOW (CIW) or use the code in nongraphics SKILL mode.

To select text

- Press `Control`-drag left mouse to select a text segment of any size.
- Press `Control`-double click left mouse to select a word.
- Press `Control`-triple click left mouse to select an entire section.

Other Sources of Information

For more information about Cadence SKILL language and other related products, you can consult the sources listed below.

Product Installation

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

Other SKILL Development Documentation

The following are SKILL development-related documents.

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

Related SKILL API Documentation

Cadence tools have their own application procedural interface functions. You can access the API manuals directly using the CDSDoc library SKILL menu.

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

Cadence User Interface SKILL Functions Reference contains APIs for management of windows and forms.

Software Installation and License Management Reference in the *Cadence License Manager* contains SKILL licensing functions.

Document Conventions

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

Syntax Conventions

This section describes the typographic and syntax conventions used in this manual.

`text`	Indicates text you must type exactly as it is presented.
`z_argument`	Indicates text that you must replace with an appropriate argument. The prefix (in this case, *`z_`*) indicates the data type the argument can accept. Do not type the data type or underscore.
`[ ]`	Denotes optional arguments. When used with vertical bars, they enclose a list of choices from which you can choose one.
`{ }`	Used with vertical bars and encloses a list of choices from which you must choose one.
`|`	Separates a choice of options; separates the possible values that can be returned by a Cadence® SKILL language function.
…	Indicates that you can repeat the previous argument.
=>	Precedes the values returned by a Cadence SKILL language function.
text	Indicates names of manuals, menu commands, form buttons, and form fields.

Important

The language requires many characters not included in the preceding list. You must type these characters exactly as they are shown in the syntax.

Data Types

The Cadence SKILL language supports several data types to identify the type of value you can assign to an argument. Data types are identified by a single letter followed by an underscore; for example, *t* is the data type in *t_viewNames*. Data types and the underscore are used as identifiers only: they are not to be typed.

The table below lists all data types supported by SKILL.

Data Types by Type

Prefix	Internal Name	Data Type
a	array	array
b	ddUserType	Boolean
c	opfcontext	OPF context
d	dbobject	Cadence database object (CDBA)
e	envobj	environment
f	flonum	floating-point number
F	opffile	OPF file ID
g	general	any data type
G	gdmSpecIlUserType	gdm spec
h	hdbobject	hierarchical database configuration object
l	list	linked list
m	nmpIlUserType	nmpIl user type
M	cdsEvalObject	—
n	number	integer or floating-point number
o	userType	user-defined type (other)
p	port	I/O port

Data Types by Type, *continued*

Prefix	Internal Name	Data Type
q	gdmspecListIlUserType	gdm spec list
r	defstruct	defstruct
R	rodObj	relative object design (ROD) object
s	symbol	symbol
S	stringSymbol	symbol or character string
t	string	character string (text)
u	function	function object, either the name of a function (symbol) or a lambda function body (list)
U	funobj	function object
v	hdbpath	—
w	wtype	window type
x	integer	integer number
y	binary	binary function
&	pointer	pointer type

1

Getting Started

Overview information:

Overview

The Cadence® SKILL language is a high-level, interactive programming language based on the popular artificial intelligence language, Lisp. However, the SKILL language supports a more conventional C-like syntax. This support allows a novice user to quickly learn to use the system, while expert programmers can access the full power of the Lisp language. At the simplest level, SKILL is as easy to use as a calculator. At the most sophisticated level, SKILL is a powerful programming language whose applications are virtually unlimited.

SKILL brings to the command line a functional interface to the underlying subsystems. SKILL lets you quickly and easily customize existing CAD applications and helps you develop new applications. SKILL has functions to access each Cadence tool using an application programming interface.

This document describes functions that are common to all of the Cadence tools used in either a graphic or nongraphic environment. Once you master these basic functions, you need to learn only a few new functions to access any tool in the Cadence environment.

Relationship to Lisp

This section summarizes the relationship of SKILL to Lisp. Anyone that is new to either language or to programming in general should start with, A Quick Look at the Cadence SKILL Language on page 33.

Programming Notation

In SKILL, function calls can be written in either of the following notations.

- Algebraic notation used by most programming languages, that is, `func(` *`arg1 arg2 ...`* `)`.
- Prefix notation used by the Lisp programming language, that is, (`func` *`arg1 arg2`* ...).

For comparison, here is a SKILL program written first in algebraic notation, then the same program, also implemented in SKILL, using a Lisp style of programming.

```
procedure( fibonacci(n)
      if( (n == 1 || n == 2) then
            1
      else fibonacci(n-1) + fibonacci(n-2)
      )
)
```

Here is the same program implemented in SKILL using a Lisp style of programming.

```
(defun fibonacci (n)
      (cond
            ((or (equal n 1) (equal n 2)) 1)
            (t (plus (fibonacci (difference n 1))
                     (fibonacci (difference n 2))))
      )
)
```

Data Manipulation

Because programs in SKILL are represented as lists, just as they are in Lisp, they can be manipulated like data. You can dynamically create, modify, or selectively evaluate function definitions and expressions. This ability to manipulate data is one of the primary reasons why Lisp is the language of choice for artificial intelligence applications. Because it takes full advantage of the “program is data” concept of Lisp, SKILL can be used to write flexible and powerful applications.

Many SKILL list manipulation functions are available. These functions operate, in most cases, similar to functions of the same name in Lisp, and the Franz Lisp dialect in particular.

SKILL supports a special notation for list construction from templates. This notation is borrowed from Common Lisp and allows selective evaluation within a quoted form. Selective evaluation eliminates the long sequences of calls to *list* and *append*. See Backquote, Comma, and Comma-At on page 69.

Characters

Unlike many other programming languages, including Common Lisp, SKILL does not have a separate character data type. Characters are instead represented by single character symbols. The character “A,” for example, is the symbol “A.” Unprintable characters can be referred to using the escape sequences.

A Quick Look at the Cadence SKILL Language

This section presents a quick look at various aspects of the SKILL programming language. These same subjects are discussed in greater detail in succeeding chapters.

This section introduces new terms and takes a general look at the Cadence Framework environment. It explores SKILL data, function calls, variables and operators, then tells you how to solve some common problems. Although each application defines the details of its SKILL interface to that application, this document most often refers to the Cadence Design Framework II environment when giving examples.

SKILL is the command language of the Cadence environment. Whenever you use forms, menus, and bindkeys, the Cadence software triggers SKILL functions to complete your task. For example, in most cases SKILL functions can

- Open a design window
- Zoom in by a factor of 2
- Place an instance or a rectangle in a design

Other SKILL functions compute or retrieve data from the Cadence Framework environment or from designs. For example, SKILL functions can retrieve the bounding box of the current window or retrieve a list of all the shapes on a layer.

You can enter SKILL functions directly on a command line to bypass the graphic user interface.

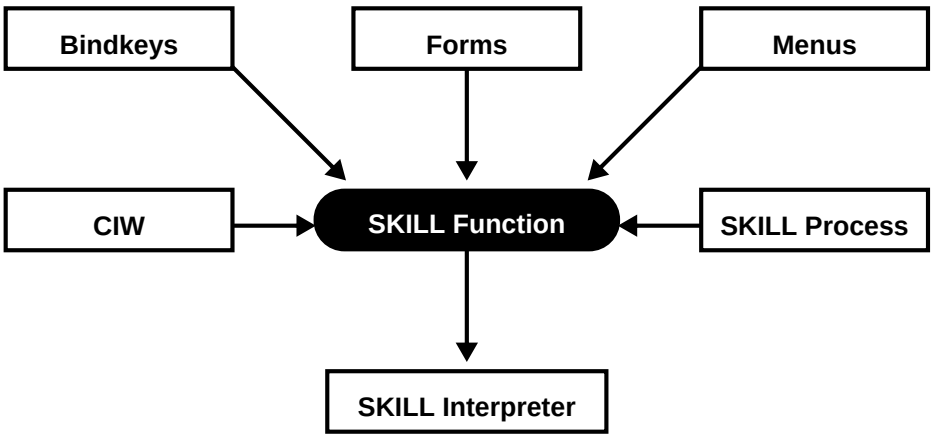
Terms and Definitions

output	The destination of SKILL output can be an xterm screen, a design window, a file, or in many cases, the Command Interpreter Window (CIW).
CIW	The start-up working window for many Cadence applications, which contains an input line, an output area, a menu banner with commands to launch various tools, and prompts for general information. The output area of the CIW is the destination of many of the examples used in this manual.
SKILL interpreter	The SKILL interpreter executes SKILL programs within the Cadence environment. The interpreter translates a SKILL program's text source code into internal data structures, which it actively consults during the execution of the program.
compiler	A compiler translates source code into a target representation, which might be machine instructions or an intermediate instruction set.
evaluation	The process whereby the SKILL interpreter determines the value of a SKILL expression.
SKILL expression	An invocation of a SKILL function, often by means of an operator supplying required parameters.

SKILL function	A named, parameterizable body of one or more SKILL expressions. You can invoke any SKILL function from the command input line available in the application by using its name and providing appropriate parameters.
SKILL procedure	This term is used interchangeably with SKILL function.

Invoking a SKILL Function

There are many ways to submit a SKILL function to the SKILL interpreter for evaluation. In many applications, whenever you use forms, menus, and bindkeys, the Cadence software triggers corresponding SKILL functions to complete your task. Normally, you do not need to be aware of SKILL functions or any syntax issues.



Bindkeys	A bindkey associates a SKILL function with a keyboard event. When you cause the keyboard event, the Cadence software sends the SKILL function to the SKILL interpreter for evaluation.
Forms	Some functions require you to provide data by filling out fields in a pop-up form.
Menus	When you choose an item in a menu, the system sends an associated SKILL function to the SKILL interpreter for evaluation.
CIW	You can directly enter a SKILL function into the CIW for immediate evaluation.

SKILL Process	You can launch a separate UNIX process that can submit SKILL functions directly to the SKILL interpreter.

You can submit a collection of SKILL functions for evaluation by loading a SKILL source code file.

What Is a Return Value?

All SKILL functions compute a data value known as the return value of the function. You can

- Assign the return value to a SKILL variable
- Pass the return value to another SKILL function

Any type of data can be a return value. SKILL supports many data types, including integers, text strings, and lists.

Simplest SKILL Data

The simplest SKILL expression is a data item. SKILL data is case sensitive. You can enter data in many familiar ways, including the following.

Sample SKILL Data Items

Data Type	Syntax Example
integer	5
floating point	5.3
text string	"Mary had a little lamb"

How Do You Call a Function?

Function names are case sensitive. To call a function, state its name and arguments in a pair of parentheses.

```
strcat( "Mary" " had" " a" " little" " lamb" )
=> "Mary had a little lamb"
```

- No spaces are allowed between the function name and the left parenthesis.
- Several function calls can be on a single line. Use spaces to separate them.
- You can span multiple lines in the command line or a source code file.

```
strcat(
      "Mary" " had" " a"
      " little" " lamb" )
=> "Mary had a little lamb"
```

- When you enter several function calls on a single line, the system only displays the return result from the final function call.

Operators Are SKILL Functions

SKILL provides many operators. Each operator corresponds to a SKILL function. Here are some examples of useful operators:

Sample SKILL Operators

Operators in Descending Precedence	Underlying Function	Operation
**	expt	arithmetic
* /	times quotient	arithmetic
+ –	plus difference	arithmetic
++s, s++	preincrement, postincrement	arithmetic
== !=	equal nequal	tests for equality tests for inequality
=	setq	assignment

The following example shows several function calls using operators on a single line. The calls are separated by spaces. The system only displays the return result from the final function call.

```
x = 5 y = 6 x+y
=> 11
```

Using Variables

You do not need to declare variables in SKILL. SKILL creates a variable the first time it encounters the variable in a session. Variable names can contain

- Alphanumeric characters
- Underscores (_)
- Question marks

The first character of a variable cannot be a digit. Use the assignment operator to store a value in a variable. You enter the variable name to retrieve its value. The `type` SKILL function returns the data type of the variable's current value.

```
lineCount = 4            => 4
lineCount                => 4
type( lineCount )        => fixnum
lineCount = "abc"        => "abc"
lineCount                => "abc"
type( lineCount )        => string
```

Alternative Ways to Enter a Function

In addition to calling a function by stating its name and arguments in a pair of parentheses, as shown below, you can use two other syntax forms to invoke SKILL functions.

```
strcat( "Mary" " had" " a" " little" " lamb" )
```

- You can place the left parenthesis to the left of the function name (Lisp syntax).

  ```
  ( strcat "Mary" " had" " a" " little" " lamb" )
  => "Mary had a little lamb"
  ```

- You can omit the outermost levels of parenthesis if the SKILL function is the first element at your SKILL prompt, that is, at the top level.

  ```
  strcat "Mary" " had" " a" " little" " lamb"
  => "Mary had a little lamb"
  ```

You can use all three syntax forms together. In programming, it is best to be consistent. Cadence recommends the first style noted above.

Solving Some Common Problems

Here are three of the most common SKILL problems.

System Doesn't Respond

If you type in a SKILL function and press Return but nothing happens, you most likely have one of these problems.

- Unbalanced parentheses

- Unbalanced string quotes
- The wrong log file filter set

You might have entered more left parentheses than right parentheses. The following steps trigger a system response in most cases.

1. Type a closing right bracket ( ]) character. This character closes all outstanding right parentheses.
2. If you still don't get a response, type a double quote (") character followed by a right bracket ( ]) character.

In most cases, the system then responds.

Inappropriate Space Characters

Do not put any space between the function name and the left parenthesis. Notice that the following error messages do not identify the extra space as the cause of the problem.

- Trying to use the *strcat* function to concatenate several strings.

```
strcat ( "Mary" " had" " a" " little" " lamb")
Message: *Error* eval: not a function - "Mary"
```

- Trying to make an assignment to a variable.

```
greeting = strcat ( "happy" " birthday" )
Message: *Error* eval: unbound variable - strcat
```

Data Type Mismatches

An error occurs when you pass inappropriate data to a SKILL function. The error message includes a type template that indicates the expected type of the offending argument.

```
strcat( "Mary had a" 5 )
Message: *Error* strcat: argument #2 should be either
a string or a symbol (type template = "S") - 5
```

Here are the characters used in type templates for some common data types.

Some Common Data Types

Data Type	Character in Type Template
integer number	x
floating point number	f

Some Common Data Types

Data Type	Character in Type Template
symbol or character string	S
character string (text)	t
any data type (general)	g

For a complete list of data types supported by SKILL, see Data Types on page 71.

What Is a SKILL List?

A SKILL list is an ordered collection of SKILL data objects. The list data structure is central to SKILL and is used in many ways.

The elements of a list can be of any data type, including variables and other lists. A list can contain any number of objects (or be empty). The empty list can be represented either by empty parentheses “( )” or the special atom `nil`. The list must be enclosed in parentheses. Lists can contain other lists to form arbitrarily complex data structures. Here are some examples:

Sample Lists

List	Explanation
(1 2 3)	A list containing the integer constants 1, 2, and 3
(1)	A list containing the single element 1
( )	An empty list (same as the special atom *nil*)
(1 (2 3) 4)	A list containing another list as its second element

SKILL displays a list with parentheses surrounding the members of the list. The following example stores a list in the variable `shapeTypeList`, then retrieves the variable’s value.

```
shapeTypeList = '( "rect" "polygon" "rect" "line" )
shapeTypeList => ( "rect" "polygon" "rect" "line" )
```

SKILL provides an extensive set of functions for creating and manipulating lists. Many SKILL functions return lists. SKILL can use multiple lines to display lists. SKILL stores the appropriate integer value in the `_itemsperline` global variable.

Building Lists

There are several main ways to build a list.

- Specify all the elements of the list literally with the single quote (`'`) operator.
- Specify all the elements as evaluated arguments to the `list` function.
- Add an element to an existing list with the `cons` function.
- Merge two lists with the `append` function.

Both the `cons` and `append` functions allocate a new list and return it to you. You should store the return result in a variable. Otherwise, you cannot refer to the list later.

The following functions allow you to construct new lists and work with existing lists in different ways.

Making a list from given elements

The single quote (`'`) operator builds a list using the arguments exactly as they are presented. The values of `a` and `b` are irrelevant. The `list` function fetches the values of variables for inclusion in a list.

```
'( 1 2 3 )                  => ( 1 2 3 )
a = 1                       => 1
b = 2                       => 2
list( a b 3 )               => ( 1 2 3 )
```

Adding an element to the front of a list (cons)

You should store the return result from *cons* in a variable. Otherwise, you cannot refer to the list later. Commonly, you store the result back into the variable containing the target list.

```
result = '( 2 3 )            => ( 2 3 )
result = cons( 1 result )    => ( 1 2 3 )
```

Merging two lists (append)

You should store the return result from `append` in a variable. Otherwise, you cannot refer to the list later.

```
oneList = '( 4 5 6 )            => ( 4 5 6 )
aList = '( 1 2 3 )              => ( 1 2 3 )
bList = append( oneList aList)  => ( 4 5 6 1 2 3 )
```

Common Questions and Answers

People often feel that `nil`, and the `cons` and `append` functions are difficult to understand. Let's look at some typical questions.

Typical Questions

Question	Answer
What's the difference between *nil* and '*(nil)*?	*nil* is a list containing nothing. Its length is 0. '*(nil)* builds a list containing a single element. The length is 1.
How can I add an element to the end of a list?	Use the *list* function to build a list containing the individual elements. Use the *append* function to merge it to the first list. More efficient ways to add an element to the end of a list are discussed in a later chapter.
Can I reverse the order of the arguments to the *cons* function? Will the results be the same?	You might think that simply reversing the elements to the *cons* function will put the element on the end of the list. However, this is not the case.
What's the difference between *cons* and *append*?	The *cons* function requires only that its second argument be a list. The length of the resulting list is one more than the length of the original list. The *append* function requires that both its arguments be lists. The length of the resulting list is the sum of the lengths of the two argument lists. *append* is slower than *cons* for large lists.

Accessing Lists

Lists are stored internally as a series of branching decision points. Think of the left branch as pointing to the first element and the right branch as pointing to the rest of the list (further branch decision points). The *car* function follows the left branch and the *cdr* function follows the right branch.

Retrieving the first element of a list (car*)*

car returns the first element of a list. *car* was a machine language instruction on the first machine to run Lisp. *car* stands for *contents of the address register*.

```
numbers = '( 1 2 3 )          => ( 1 2 3 )
car( numbers )                => 1
```

Retrieving the tail of the list (cdr*)*

cdr returns a list minus the first element. *cdr* was a machine language instruction on the first machine to run Lisp. *cdr* stands for *contents of the decrement register*.

```
numbers = '( 1 2 3 )          => ( 1 2 3 )
cdr( numbers )                => ( 2 3 )
```

Retrieving an element given an index (nth*)*

nth assumes a zero-based index. *nth(0 numbers)* is the same as *car(numbers)*.

```
numbers = '( 1 2 3 )          => ( 1 2 3 )
nth( 1 numbers )              => 2
```

Determining if a data object is in a list (member*)*

The *member* function cannot search all levels in a hierarchical list. It only looks at the top-level elements. Internally the *member* function follows right branches until it locates a branch point whose left branch dead ends in the element.

```
numbers = '( 1 2 3 )          => ( 1 2 3 )
member( 4 numbers )           => nil
member( 2 numbers )           => ( 2 3 )
```

Counting the elements in a list (length*)*

length determines the length of a list, array, or association table.

```
numbers = '( 1 2 3 )          => ( 1 2 3 )
length( numbers )             => 3
```

Modifying Lists

The following functions operate on variables without changing their value or creating new variables.

Coordinates

An xy coordinate is represented by a two-element list. The colon (:) binary operator builds a coordinate from an x value and a y value.

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

The functions *xCoord* and *yCoord* access the x coordinate and the y coordinate.

```
xCoord( aCoordinate ) => 300
yCoord( aCoordinate ) => 400
```

- You can use the single quote (') operator or *list* function to build a coordinate list.
- You can use the *car* function to access the x coordinate and *car*(*cdr* (...)) to access the y coordinate.

Bounding Boxes

A bounding box is represented by a list of the lower-left and upper-right coordinates. Use the *list* function to build a bounding box that contains

- Coordinates specified with the binary operator (:).

  ```
  bBox = list( 300:400 500:450 )
  ```

- Coordinates specified by variables.

  ```
  lowerLeft        = 300:400
  upperRight       = 500:450
  bBox             = list( lowerLeft upperRight )
  ```

You can use the single quote (') operator to build the bounding box if the coordinates are specified by literal lists.

```
bBox = '(( 300 400 ) ( 500 450 ))
```

Bounding boxes provide a good example of working with the *car* and *cdr* functions. Use any combination of four *a*'s (each *a* executes another *car*) or *d*'s (each *d* executes another *cdr*).

Using *car* and *cdr* with Bounding Boxes

Functions	Meaning	Example	Expression
car	car(...)	lower left corner	ll = car(bBox)
cadr	car(cdr(...))	upper right corner	ur = cadr(bBox)
caar	car(car(...))	x-coord of lower left corner	llx = caar(bBox)
cadar	car(cdr(car(...)))	y-coord of lower left corner	lly = cadar(bBox)

Using *car* and *cdr* with Bounding Boxes

Functions	Meaning	Example	Expression
caadr	car(car(cdr(...)))	x-coord of upper right corner	urx = caadr(bBox)
cadadr	car(cdr(car(cdr(...]	y-coord of upper right corner	ury = cadadr(bBox)

File Input/Output

This section introduces how to

- Display values using default formats and application-specific formats
- Write UNIX text files
- Read UNIX text files

Displaying Data

Display data using

- The *print* and *println* functions
- The *printf* function

The *print* and *println* Functions

The SKILL interpreter has a default display format for each kind of data. The *print* and *println* functions use this format to display data.

Sample Display Formats

Data Type	Example of the Default Format
integer	5
floating point	1.3
text string	"Mary learned SKILL"
variable	bBox
list	(1 2 3)

The *print* and *println* functions display a single data value. *println* is the same as *print* followed by a newline character.

```
for( i 1 3 print( "hello" ))     ;Prints hello three times.
"hello""hello""hello"
for( i 1 3 println( "hello" ))   ;Prints hello three times.
"hello"
"hello"
"hello"
```

The *printf* Function

The *printf* function writes formatted output. This example displays a line in a report.

```
printf(
      "\n%-15s %-15s %-10d %-10d %-10d %-10d"
      layerName purpose
      rectCount labelCount lineCount miscCount
      )
```

The first argument is a conversion control string containing directives.

```
%[-][width][.precision]conversion_code
      [-]                 = left justify
      [width]             = minimum number of character positions
      [.precision]        = number of characters to be printed
      conversion_code
            d - decimal(integer)
            f - floating point
            s - string or symbol
            c - character
            n - numeric
            L - list (Ignores width and precision fields.)
            P - point list (Ignores width and precision fields.)
            B - Bounding box list (Ignores width and precision.)
```

The %L directive specifies the default format. This directive is a convenient way to intersperse application-specific formats with default formats. The *printf* function returns *t*. For more information on directives, see Formatted Output on page 165.

```
aList = '(1 2 3)
printf( "\nThis is a list: %L" aList ) => t
This is a list: (1 2 3)
aList = nil
printf( "\nThis is a list: %L" aList ) => t
This is a list: nil
```

If the conversion control directive is inappropriate for the data item, *printf* displays an error.

```
printf( "%d %d" 5 nil )
Message: *Error* fprintf/sprintf:
      format spec. incompatible with data - nil
```

Writing Data to a File

To write text data to a file

1. Use the *outfile* function to obtain an output port on a file.
2. Use an optional output port parameter to the *print* and *println* functions and/or use a required port parameter to the *fprintf* function.
3. Close the output port with the *close* function.

Both *print* and *println* accept an optional second argument, which should be an output port associated with the target file. Use the *outfile* function to obtain an output port for a file. Once you are finished writing data to the file, use the *close* function to release the port. The following code

```
myPort = outfile( "/tmp/myFile" )
for( i 1 3
      println( list( "Number:" i) myPort )
      )
close( myPort )
```

writes this data to the file `/tmp/myFile`.

```
("Number:" 1)
("Number:" 2)
("Number:" 3)
```

Notice how SKILL displays a port:

```
myPort = outfile( "/tmp/myFile" )
port:"/tmp/myFile"
```

Use a full path with the *outfile* function. Keep in mind that *outfile* returns *nil* if you don't have write access to the file or if it can't be created in the directory specified in the path. The *print* and *println* functions display an error if the port argument is *nil*. Notice that the type template uses a *p* character to indicate a port is expected.

```
println( "Hello" nil )
Message: *Error* println: argument #2 should be an I/O port
      (type template = "gp") - nil
```

Unlike the *print* and *println* functions, the *printf* function does *not* accept an optional port argument. Use the *fprintf* function to write formatted data to a file. Its first argument should be an output port associated with the file. The following code

```
myPort = outfile( "/tmp/myFile" )
for( i 1 3
      fprintf( myPort "Number: %d\n" i )
      )
close( myPort )
```

writes this data to the file `/tmp/myFile`.

```
Number: 1
Number: 2
Number: 3
```

Reading Data from a File

To read a text file

1. Use the *infile* function to obtain an input port.
2. Use the *gets* function to read the file a line at a time and/or use the *fscanf* function to convert text fields upon input.
3. Close the input port with the *close* function.

Use the *infile* function to obtain an input port on a file. The *gets* function reads the next line from the file. This example prints every line in the `~/.cshrc` file.

```
inPort = infile( "~/.cshrc" )
when( inPort
      while( gets( nextLine inPort )
            println( nextLine )
      )
      close( inPort )
)
```

The *fscanf* function reads data from a file according to format directives. This example prints every word in `~/.cshrc`.

```
inPort = infile( "~/.cshrc" )
when( inPort
      while( fscanf( inPort "%s" word )
            println( word )
      )
      close( inPort )
)
```

The *gets* function reads the next line from the file. The arguments of *gets* are the variable that will receive the next line and the input port. *gets* returns the text string or *nil* when the end of file is reached.

The *fscanf* function reads data from a file according to conversion control directives. The arguments of *fscanf* are

- Input port
- Conversion control string
- Variable(s) that will receive the matching data values

fscanf returns the number of data items matched.Format directives commonly found include the following.

Some Common Format Directives

Format Specification	Data Type	Scans Input Port
%d	integer	for next integer
%f	floating point	for next floating point
%s	text string	for next text string
%e	floating point	for next floating point
%g	floating point	for next floating point

For common output format specifications, see Formatted Output on page 165.

Flow of Control

This section introduces you to

- Relational Operators: <, <=, >, >=, ==, !=
- Logical Operators: !, &&, ||
- Branching: if, when, unless
- Multi-way Branching: case
- Iteration: for, foreach

Iteration refers to repeatedly executing a collection of SKILL expressions by changing - usually incrementing or decrementing - the value of one or more loop variables.

Relational Operators

Use the following operators to compare data values. SKILL generates an error if the data types are inappropriate. These operators all return *t* or *nil*.

Sample Relational Operators

Operator	Arguments	Function	Example	Return Value
<	numeric	lessp	3 < 5 3 < 2	t nil
<=	numeric	leqp	3 <= 4	t
>	numeric	greaterp	5 > 3	t
>=	numeric	geqp	4 >=3	t
==	numeric string list	equal	3.0 == 3 "abc" == "ABc"	t nil
!=	numeric string list	nequal	"abc" != "ABc"	t

It is helpful to know the function name because error messages mention the function (*greaterp* below) instead of the operator (>).

```
1 > "abc"
Message: *Error* greaterp: can’t handle (1 > "abc")
```

Logical Operators

SKILL considers *nil* as FALSE and any other value as TRUE. The and (&&) and or (||) operators only evaluate their second argument if they need to determine the return result.

Sample Logical Operators

Operator	Arguments	Function	Example	Return Value
&&	general	and	3 && 5 5 && 3 t && nil nil && t	5 3 nil nil

Sample Logical Operators

Operator	Arguments	Function	Example	Return Value
||	general	or	3 || 5 5 || 3 t || nil nil || t	3 5 t t

The && and || operators return the value last computed. Consequently, both && and || operators can be used to avoid cumbersome *if* or *when* expressions.

Using &&

When SKILL creates a variable, it gives the variable a value of *unbound* to indicate that the variable has not been initialized yet. Use the *boundp* function to determine whether a variable is *bound*. The *boundp* function

- Returns *t* if the variable is bound to a value.
- Returns *nil* if it is not bound to a value.

Suppose you want to return the value of a variable *trMessages*. If *trMessages* is *unbound*, retrieving the value causes an error. Instead, use the expression

```
boundp( 'trMessages ) && trMessages
```

Using ||

Suppose you have a default name, such as *noName*. Suppose you have a variable, such as *userName*. To use the default name if *userName* is *nil*, use the following expression

```
userName || "noName"
```

The if Function

Use the *if* function to selectively evaluate two groups of one or more expressions. The condition in an *if* expression evaluates to *nil* or non-*nil*. The return value of the *if* expression is the value last computed.

```
if( shapeType == "rect"
      then
            println( "Shape is a rectangle" )
            ++rectCount
      else
            println( "Shape is not a rectangle" )
```

```
            ++miscCount
      )
```

SKILL does most of its error checking during execution. Error messages involving *if* expressions can be obscure. Be sure to

- Be aware of the placement of the parentheses: *if(… then … else …)*.
- Avoid white space immediately after the *if* keyword.
- Use *then* and *else* when appropriate to your logic.

Consider the error message when you accidentally put white space after the *if* keyword.

```
shapeType = "rect"
if ( shapeType == "rect"
      then
            println( "Shape is a rectangle" )
            ++rectCount
      else
            println( "Shape is not a rectangle" )
            ++miscCount
      )

Message: *Error* if: too few arguments (at least 2 expected, 1 given)...
```

Consider the error message when you accidentally drop the *then* keyword, but include an *else* keyword, and the condition returns *nil*.

```
shapeType = "label"
if( shapeType == "rect"
            println( "Shape is a rectangle" )
            ++rectCount
      else
            println( "Shape is not a rectangle" )
            ++miscCount
      )

Message: *Error* if: too many arguments ...
```

The when and unless Functions

Use the *when* function whenever you have only *then* expressions.

```
when( shapeType == "rect"
            println( "Shape is a rectangle" )
            ++rectCount
      ) ; when

when( shapeType == "ellipse"
      println( "Shape is an ellipse" )
      ++ellipseCount
      ) ; when
```

Use the *unless* function to avoid negating a condition. Some users find this less confusing.

```
unless( shapeType == "rect" || shapeType == "line"
      println( "Shape is miscellaneous" )
      ++miscCount
      ) ; unless
```

The *when* and *unless* functions both return the last value evaluated within their body or *nil*.

The case Function

The *case* function offers branching based on a numeric or string value.

```
case( shapeType
      ( "rect"
            ++rectCount
            println( "Shape is a rectangle" )
            )
      ( "line"
            ++lineCount
            println( "Shape is a line" )
            )
      ( "label"
            ++labelCount
            println( "Shape is a label" )
            )
      ( t
            ++miscCount
            println( "Shape is miscellaneous" )
            )
      ) ; case
```

The optional value *t* acts as a catch-all and should be handled last. The *case* function returns the value of the last expression evaluated. In this example:

- The value of the variable *shapeType* is compared against the values *rect*, *line*, and *label.* If SKILL finds a match, the several expressions in that arm are evaluated.
- If no match is found, the final arm is evaluated.
- When an arm's target value is actually a list, SKILL searches the list for the candidate value. If SKILL finds the candidate value, all the expressions in the arm are evaluated.

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

The for Function

The index in a *for* expression is saved before the *for* loop and is restored to its saved value after the *for* loop is exited. SKILL does most of its error checking during execution. Error messages involving *for* expressions can be obscure. Be sure to

- Be aware of the placement of the parentheses: *for(…)*.
- Avoid white space immediately after the *for* keyword.

The example below adds the integers from one to five to an intermediate *sum*. *i* is a variable used as a counter for the loop and as the value to add to *sum*. Counting begins with one and ends with the completion of the fifth loop. *i* increases by one for each iteration through the loop.

```
sum = 0
for( i 1 5
      sum = sum + i
      println( sum )
      )
=> t
```

SKILL prints the value of *sum* with a carriage return for each pass through the loop:

```
1
3
6
10
15
```

The *for* function always returns *t*.

The foreach Function

The *foreach* function is very useful for performing operations on each element in a list. Use the *foreach* function to evaluate one or more expressions for each element of a list of values.

```
rectCount = lineCount = polygonCount = 0
shapeTypeList = '( "rect" "polygon" "rect" "line" )
foreach( shapeType shapeTypeList
      case( shapeType
            ( "rect"            ++rectCount )
            ( "line"            ++lineCount )
            ( "polygon"         ++polygonCount )
            ( t                 ++miscCount )
            ) ;case
      ) ; foreach
=> ( "rect" "polygon" "rect" "line" )
```

When evaluating a *foreach* expression, SKILL determines the list of values and repeatedly assigns successive elements to the index variable, evaluating each expression in the *foreach* body. The *foreach* expression returns the list of values over which it iterates.

In the example:

- The variable *shapeType* is the index variable. Before entering the *foreach* loop, SKILL saves the current value of *shapeType*. SKILL restores the saved value after completing the *foreach* loop.
- The variable *shapeTypeList* contains the list of values. SKILL successively assigns the values in *shapeTypeList* to *shapeType*, evaluating the body of the *foreach* loop once for each separate value.
- The body of the *foreach* loop is a single case expression.
- The return value of the *foreach* loop is the list contained in variable *shapeTypeList*.

If you have executed the example above, you can examine the effect of the iterations by typing the name of the counter:

```
rectCount        => 2
lineCount        => 1
polygonCount     => 1
```

Developing a SKILL Function

Developing a SKILL function includes the following tasks.

- Grouping several SKILL statements into a single SKILL statement
- Declaring a SKILL function with the *procedure* function
- Defining function parameters
- Maintaining your source code
- Loading your SKILL source code
- Redefining a SKILL function

Grouping SKILL Statements

Sometimes it is convenient to group several SKILL statements into a single SKILL statement. Use braces { and } to group a collection of SKILL statements into a single SKILL statement. The return value of the single statement is the return value of the last SKILL statement in the group. You can assign this return value to a variable.

This example computes the pixel height of *bBox* and assigns it to the *bBoxHeight* variable:

```
bBoxHeight = {
      bBox = list( 100:150 250:400)
      ll = car( bBox )
      ur = cadr( bBox )
      lly = yCoord( ll )
      ury = yCoord( ur )
      ury - lly }
```

- The *ll* and *ur* variables hold the lower-left and upper-right coordinates of the bounding box.
- The *xCoord* and *yCoord* functions return the *x* and *y* coordinate of a point.
- The *ury - lly* expression computes the height. This last statement in the group determines the return value of the group.
- The return value is assigned to the *bBoxHeight* variable.

You can declare the variables *ll, ur, ury*, and *lly* to be local variables. Use the *prog* or *let* functions to define a collection of local variables for a group of several statements. However, defining local variables is not recommended for novices.

Declaring a SKILL Function

To refer to the group of statements by name, use the *procedure* declaration to associate a name with the group. The group of statements and the name make up a SKILL function.

- The name is known as the function name.
- The group of statements is the function body.
- To execute the group of statements, mention the function name followed immediately by ( ).

The *ComputeBBoxHeight* function example below computes the pixel height of *bBox*.

```
procedure( ComputeBBoxHeight( )
      bBox = list( 100:150 250:400)
      ll       = car( bBox )
      ur       = cadr( bBox )
      lly      = yCoord( ll )
      ury      = yCoord( ur )
      ury - lly
      ) ; procedure

bBoxHeight = ComputeBBoxHeight()
```

Defining Function Parameters

To make your function more versatile, you can identify certain variables in the function body as formal parameters.

When you invoke your function, you supply a parameter value for each formal parameter.

In the following example, the *bBox* is the parameter.

```
procedure( ComputeBBoxHeight( bBox )
      ll       = car( bBox )
      ur       = cadr( bBox )
      lly      = yCoord( ll )
      ury      = yCoord( ur )
      ury - lly
      ) ; procedure
```

To execute your function, you must provide a value for the parameter.

```
bBox = list( 100:150 250:400)
bBoxHeight = ComputeBBoxHeight( bBox )
```

Selecting Prefixes for Your Functions

With only a few exceptions, the SKILL functions in this manual do not use a prefix identifier. Many examples in this manual use a “tr” prefix to indicate they are created for training purposes. If you look in other SKILL manuals, you will notice that functions for tools are usually grouped with identifiable, unique prefixes.

For example, functions used for technology file administration are all prefixed with “tc”. These prefixes vary across Cadence tools, but all use lowercase letters. It is recommended that you establish a unique prefix using uppercase letters for your own functions.

Maintaining SKILL Source Code

The Cadence environment makes it easy to invoke an editor of your choice. Set the SKILL variable *editor* to a UNIX command line able to launch your editor.

```
editor = "xterm -e vi"
```

The *ed* function invokes an editor of your choice. If you optionally use the *edl* function, the system loads your file when you quit the editor.

```
ed( "myFile.il")
```

Alternatively, you can use an editor independent of the Cadence environment.

Loading Your SKILL Source Code

The *load* function

- Evaluates each expression in a source code file]
- Is typically used to define a collection of functions
- Returns *t* if all expressions evaluate without errors
- Aborts if there are any errors, any expression following the offending expression is not evaluated

Giving a Relative Path

When you pass a relative path to the *load* function, the system resolves it in terms of a list of directories called the SKILL path. You usually establish the SKILL path in your initialization file by using the *setSkillPath* or *getSkillPath* functions.

- The *setSkillPath* function sets the path to a list of directories.
- The *getSkillPath* function returns a list of directories in search order.

Entering a Function

Sometimes you need to define a function without saving the source code file. Using the mouse in your editor, select and paste the function into the command input line.

Setting the SKILL Path

Use the *setSkillPath* function in conjunction with the *prependInstallPath* and *getSkillPath* functions to set the SKILL path.

```
trSamplesPath = list(
            prependInstallPath( "etc/context" )
            prependInstallPath( "local" )
            prependInstallPath( "samples/local" )
            )
```

Use the *prependInstallPath* function to make a path relative to the installation directory. The function prepends *`your_install_dir`* `/tools/dfII` to the path. Assuming your installation path is `/cds/9401` `trSamplesPath` is now:

```
("/cds/9401/tools.sun4/dfII/etc/context"
 "/cds/9401/tools.sun4/dfII/local"
 "/cds/9401/tools.sun4/dfII/samples/local")
```

Assuming your SKILL path is ("." "~"), you can set a new SKILL path using *setSkillPath*.

```
setSkillPath( append( trSamplesPath getSkillPath() ) )
```

The return value of *setSkillPath* indicates a path that could not be located, not the actual SKILL path.

```
("/cds/9401/tools.sun4/dfII/samples/local")
```

The actual SKILL path is

```
("/cds/9401/tools.sun4/dfII/etc/context"
 "/cds/9401/tools.sun4/dfII/local"
 "/cds/9401/tools.sun4/dfII/samples/local" "." "~")
```

For more information on the SKILL path, see Directory Paths on page 152.

Redefining a SKILL Function

Users should be safeguarded against inadvertently redefining functions. Yet, while developing SKILL code, you often need to redefine functions.

The SKILL interpreter has an internal switch called *writeProtect* to prevent the virtual memory definition of a function from being altered during a session.

By default *writeProtect* is set to *nil*. SKILL functions defined while *writeProtect* is *t* cannot be redefined during the same session. Typically, you set the *writeProtect* switch in your initialization file.

```
sstatus( writeProtect t )        ;sets it to t
sstatus( writeProtect nil )      ;sets it to nil
```

This example tries to redefine *trReciprocal* to prevent division by 0.

```
sstatus( writeProtect t ) => t
procedure( trReciprocal( x ) 1.0 / x ) => trReciprocal
procedure( trReciprocal( x ) when( x != 0.0 1.0 / x ))
*Error* def: function name is write protected
      and cannot be redefined - trReciprocal
```

2

Language Characteristics

Overview information:

Overview

This chapter explains the basic structure and syntax of the Cadence® SKILL language. The best way to learn SKILL, of course, is by using it to accomplish a real task. Before you begin using SKILL, you should study this chapter.

Programming experience is helpful for those who want to program extensively in SKILL. References to the C programming language in this text make it easier for C programmers to learn SKILL. This text does not require you to be an experienced programmer.

Experienced C programmers must remember that SKILL is not C even though the syntax appears familiar.

Naming Conventions

This section describes Cadence naming conventions for functions, variables, and their arguments.

To avoid conflict with Cadence-supplied functions and variables, customers should begin their function and variable names with uppercase letters.

Names of Functions

If you look in SKILL API reference manuals, you will notice that functions for tools are usually grouped with identifiable, unique prefixes. These prefixes vary across Cadence tools, but all use lowercase letters.

The recommended naming scheme is to

- Use casing to separate code that is developed within Cadence from that developed outside.
- Use a group prefix to separate code developed within Cadence.

Cadence internal developers should name functions with

- An optional initial underscore (_) to indicate private functions. Cadence customers should not use private functions. See Cadence-Private Functions on page 63.

- Up to three lowercase characters (occasionally more for clarity) that signify the code package.
- An *optional* further lowercase character as shown in the table below.

Name Type	Character	Meaning
Bit Field Constant	b	Bit-field constant
Constants	c	Enumerated constant
Errors	e	Name of a structure describing an error
Internal Functions	i	An internal function, these functions should not be called directly by application programs
Functions	f	Occasionally used as a function indicator.
Macros	m	Rarely used macro indicator.
Global Variables	v	Public global variable

- The name itself starting with an uppercase character.

Cadence-Private Functions

Functions beginning with an underscore are considered Cadence-private, internal functions and are not supported.

Cadence-private functions are undocumented, unsupported functions that are used internally by Cadence engineering. These Cadence-private functions are subject to change at any time, without notice, because they are not intended for public use.

Names of Variables

Cadence customers can avoid naming conflicts with Cadence-supplied variables by beginning the first letter of their variable names with uppercase letters.

You should not set the following Cadence internal variables without a full understanding of potential consequences.

Variable	Meaning
`stdin`	Standard input port.
`stdout`	Standard output port.
`piport`	Standard input port from which user input is read.
`poport`	Standard output port, which is the default for print statements.
`ptport`	Standard output port for tracing results.
`errport`	Standard output port for error messages.
`printlength`	Controls the number of elements in a list that are printed.
`printlevel`	Controls the depth of elements in a list which are printed.
`tracelength`	Controls the number of elements in a list that are printed.
`tracelevel`	Controls the depth of elements in a list that are printed during tracing.
`pprintresult`	Controls the pretty printing of the results of values returned by the top level.
`editor`	Controls the default text editor.
`gcdisable`	Toggles enabling of garbage collection. Use cautiously. Internal variable used in memory management for debugging purposes only.
`_gcprint`	Toggles the printing of garbage collection messages. Internal variable used in memory management for debugging purposes only.

You might see internal variable names when you are using the debugging tools, especially if you are dumping the stack.

Function Calls

SKILL is a functional programming language, which means that computation is both expressed and performed as a series of function calls.

- Every operator in SKILL corresponds to a predefined function. The operator "+," for example, corresponds to the function *plus*.
- You can add two numbers together either by calling the function, for example, *plus(x y)*, or by using the *infix* expression *x+y*.

In SKILL, even control statements are implemented by calls to special functions; the special functions then evaluate their arguments in a specific order.

Function calls can be written in either of the following notations.

- Algebraic notation used by most programming languages, that is, `func(` *`arg1 arg2`* ... `)`.
- Prefix notation used by the Lisp programming language, that is; (`func` *`arg1 arg2`* ...).

Remember that there must be no white space between the function name and the left parenthesis in the algebraic notation.

The functional programming concepts implemented in SKILL are reflected in the basic syntax of the language. SKILL programs are written as sequences of nested function calls. To utilize SKILL fully, you must first have a good grasp of the underlying concepts of functional programming.

SKILL Syntax

This section describes SKILL syntax, which includes the use of special characters, comments, spaces, parentheses, and other notation.

Special Characters

Certain characters are special in SKILL. These include the *infix* operators such as less than (<), colon (:), and assignment (=). The table below lists these special characters and their meaning in SKILL.

Note: All non-alphanumeric characters (other than '_' and '?') must be preceded ("escaped") by a backslash ('\') when you use them in the name of a symbol.

Special Characters in SKILL

Character	Name	Meaning
\	backslash	Escape for special characters
( )	parentheses	Grouping of list elements, function calls
[ ]	brackets	Array index, super right bracket
{ }	braces	Grouping of expressions using *progn*
'	single quote	Quoting the expression to prevent its evaluation
”	double quote	String delimiter
,	comma	Optional delimiter between list elements; also used within the scope of a backquoted expression to force the evaluation of the expression
;	semicolon	Line-style comment character
:	colon	Bit field delimiter, range operator
.	period	getq operator
+, –, *, /	arithmetic	For arithmetic operators; the /* and */ combinations are also used as comment delimiters
!,^,&,|	logical	For logical operators
<,>,=	relational	For relational and assignment operators; < and > are also used in the specification of bit fields
#	pound sign	Signals special parsing if it appears in the first column
@	“at” sign	If first character, implies reserved word; also used with comma to force evaluation and list splicing in the context of a backquoted expression
?	question mark	If first character, implies keyword parameter
‘	backquote	Quoting the expression prevents its evaluation, with support for the comma (,) and comma-at (,@) operators to allow evaluation within backquoted forms
%	percent sign	Used as a scaling character for numbers
$	–	Reserved for future use

Comments

SKILL permits two different styles of comments. One style is block-oriented, where comments are delimited by /* and */. For example:

```
/* This is a block of (C style) comments
comment line 2
comment line 3 etc.
*/
```

The other style is line-oriented where the semicolon (;) indicates that the rest of the input line is a comment. For example:

```
x = 1                           ; comment following a statement
; comment line 1
; comment line 2 and so forth
```

For simplicity, line-oriented comments are recommended. Block-oriented comments cannot be nested because the first */ encountered terminates the whole comment.

White Space

White space sometimes takes on semantic significance and a few syntactic restrictions must therefore be observed. See Solving Some Common Problems on page 38.

Write function calls so the name of a function is immediately followed by a left parenthesis; no white space is allowed between the function name and the parenthesis. For example:

`f(a b c)` and `g()` are legal function calls, but
`f (a b c)` and `g ()` are illegal.

The unary minus operator must immediately precede the expression it applies to. No white space is allowed between the operator and its operand. For example:

`-1`, `-a`, and `-(a*b)` are legal constructs, but
`- 1`, `- a`, and `- (a*b)` are illegal.

The binary minus (subtract) operator should either be surrounded by white space on both sides or there should be no white space on either side. To avoid ambiguity, one or the other method should be used consistently. For example:

`a - b` and `a-b` are legal constructs for binary minus, but `a -b` is illegal.

Parentheses

There is a subtle point about SKILL syntax that C programmers, in particular, must be very careful to note.

Parentheses in C

In C, the relational expression given to a conditional statement such as *if, while*, and *switch* must be enclosed by an outer set of parentheses for purely syntactical reasons, even if that expression consists of only a single Boolean variable. In C, an *if* statement might look like:

```
if (done) i=0; else i=1;
```

Parentheses in SKILL

In SKILL, however, parentheses are used for calling functions, delimiting multiple expressions, and controlling the order of evaluation. You can write function calls in prefix notation

```
(fn2 arg1 arg2) or (fn0)
```

as well as in the more conventional algebraic form:

```
fn2(arg1 arg2) or fn0()
```

The use of syntactically redundant parentheses causes variables, constants, or expressions to be interpreted as the names of functions that need to be further evaluated. Therefore,

- Never enclose a constant or a variable in parentheses by itself. For example, (1), (x).
- For arithmetic expressions involving *infix* operators, you can use as many parentheses as necessary to force a particular order of evaluation, but never put a pair of parentheses immediately outside another pair of parentheses, for example, *((a + b))*; the expression delimited by the inner pair of parentheses would be interpreted as the name of a function.

For example, because *if* evaluates its first argument as a logical expression, a variable containing the logical condition to be tested should be written without any surrounding parentheses; the variable by itself is the logical expression. This is written in SKILL as:

```
if( done then i = 0 else i = 1)
```

Super Right Bracket

When you are entering deeply nested expressions, it often becomes tedious to match up each left parenthesis with a right parenthesis at the end of the expression. The right bracket] can be used as a super right parenthesis to close off all open parentheses that are still

pending. It is a convenient shorthand notation for interactive input, but it is not recommended for use in programs. For example:

```
f1( f2( f3( f4( x ) ) ) )
```

can also be written as

```
f1( f2( f3( f4( x ]
```

Backquote, Comma, and Comma-At

SKILL supports a special notation for list construction from templates. This notation allows selective evaluation within a quoted form. This selective evaluation eliminates long sequences of calls to *list* and *append*.

In absence of commas and the comma-at (,@) construction, backquote functions in exactly the same way as single quote. However, if a comma appears inside a backquoted form, the expression that immediately follows the comma is evaluated, and the result of evaluation replaces the original form.

Commas are still acceptable as argument list separators in all contexts except within the scope of a backquote. This means that the backquote comma syntax does not have any implications for SKILL code created before the *backquote comma* facility was implemented. For example:

```
y = 1
'(x y z)         => (x y z)
'(x ,y z)         => (x 1 z)
```

The comma-at construction causes evaluation just as the comma does, but the results of evaluation must be a list, and the elements of the list, rather than the list itself, replace the original form. For example:

```
x = 1
y = '(a b c)
'(,x ,y z)       => (1 (a b c) z)
'(,x ,@y z)       => (1 a b c z)
```

Here's an example of a simple macro implemented with backquote:

```
defmacro(myWhen (@rest body)
••••••'if( ,car( body) progn( ,@cdr(body))))
```

The expression

```
a = 2
b = 7
myWhen( eq( a b ) printf( "The same\n" ) list( a b ))
```

expands to

```
if( eq( a b ) progn( printf( "The same\n" ) list( a b )))
```

Line Continuation

SKILL places no restrictions on how many characters can be placed on an input line, even though SKILL does impose an 8191 character limit on the strings being input. The parser reads as many lines as needed from the input until it has read in a complete form (that is, expression). If there are parentheses that have not yet been closed or binary *infix* operators whose right sides have not yet been given, the parser treats carriage returns (that is, newlines) just like spaces.

Because SKILL reads its input on a form-by-form basis, it is rarely necessary to “continue” an input line. There might be times, however, when you want to break up a long line for aesthetic reasons. In that case, you can tell the parser to ignore a carriage return in the input line simply by preceding it immediately with a backslash (\).

```
string = "This is \
a test."
=> "This is a test."
```

Length of Input Lists

The SKILL parser imposes a limit of 6000 on the number of elements that can be in a list being read in. Internally and on output, there is no limit to how many elements a list can contain.

Data Characteristics

This section describes the following basic data characteristics:

- Data Types
- Numbers
- Strings
- Atoms
- Escape Sequences
- Symbols
- Characters

These other SKILL data characteristics are discussed in the chapters indicated:

- Lists - refer to Advanced List Operations on page 177

- Property Lists, Defstructs, and Arrays - refer to Data Structures on page 93
- Type Predicates - refer to Arithmetic and Logical Expressions on page 123

Data Types

SKILL supports several data types, including integer and floating-point numbers, character strings, arrays, and a highly flexible linked list structure for representing aggregates of data.

For symbolic computation, SKILL has data types for dealing with symbols and functions.

For input/output, SKILL has a data type for representing I/O ports. The table below lists all the data types supported by SKILL with their internal names and single-character mnemonic abbreviations.

Data Types Supported by SKILL

Data Type	Internal Name	Single Character Mnemonic
array	array	a
Cadence database object	dbobject	d
floating-point number	flonum	f
any data type	general	g
linked list	list	l
integer or floating point number		n
user-defined type		o
I/O port	port	p
defstruct		r
relative object design (ROD) object	rodObj	R
symbol	symbol	s
symbol or character string		S
character string (text)	string	t
function object		u
window type		w
integer number	fixnum	x

Data Types Supported by SKILL

Data Type	Internal Name	Single Character Mnemonic
binary function	binary	y

Numbers

SKILL supports the following numeric data types:

- Integers
- Floating-point
- Scaling factors

Integers

Unless they are preceded by one of the prefixes listed in the table below, integers are interpreted as decimal numbers. Binary numbers are prefixed with “0b,” octal numbers with a leading “0,” and hexadecimal numbers with “0x.” Integers (*fixnum*’s) in SKILL are stored internally as 32-bit wide numbers.

Prefixes for Binary/Octal/Hexadecimal Integers

Radix	Prefix	Examples [value in decimal]
binary	0b or 0B	0b0011 [3] 0b0010 [2]
octal	0	077 [63] 011 [9]
hexadecimal	0x or 0X	0x3f [63] 0xff [255]

Floating-Point Numbers

You use the same syntax for floating-point numbers in SKILL as you do in most programming languages. You can have an integer followed by a decimal point, a fraction, and an optionally signed exponent preceded by “e” or “E”. Either the integer or the fraction must be present to avoid ambiguity. The following examples illustrate correct syntax:

```
10.0, 10., 0.5, .5, 1e10, 1e–10, 1.1e10
```

Scaling Factors

SKILL provides a set of scaling factors that can be added on at the end of a decimal number (integer or floating point) to achieve the desired scaling.

- Scaling factors must appear immediately after the numbers they affect; spaces are not allowed between a number and its scaling factor.
- Only the first nonnumeric character that appears after a number is significant; other characters following the scaling factor are ignored.
- If the number being scaled is an integer, SKILL tries to keep it an integer; the scaling factor must be representable as an integer (that is, the scaling factor is an integral multiplier and the result does not exceed the maximum value that can be represented as an integer). Otherwise a floating-point number is returned. The scaling factors are listed in the following table.

Scaling Factors

Character	Name	Multiplier	Examples
Y	Yotta	10^{24}	10Y [10e+25]
Z	Zetta	10^{21}	10Z [10e+22]
E	Exa	10^{18}	10E [10e+19]
P	Peta	10^{15}	10P [10e+16]
T	Tera	10^{12}	10T [10e+13]
G	Giga	10^{9}	10G [10,000,000,000]
M	Mega	10^{6}	10M [10,000,000]
k or K	kilo	10^{3}	10k [10,000]
%	percent	10^{-2}	5% [0.05]
m	milli	10^{-3}	5m [5.0e-3]
u	micro	10^{-6}	1.2u [1.2e-6]
n	nano	10^{-9}	1.2n [1.2e-9]
p	pico	10^{-12}	1.2p [1.2e-12]
f	femto	10^{-15}	1.2f [1.2e-15]
a	atto	10^{-18}	1.2a [1.2e-18]
z	zepto	10^{-21}	1.2z [1.2e-21]

Scaling Factors

Character	Name	Multiplier	Examples
y	yocto	10^{-24}	1.2y [1.2e-24]

Strings

Strings are sequences of characters, for example, “abc” or “123.” A string is marked off by double quotes, just as in the C language; the empty string is represented as “ ”. The SKILL parser limits the length of input strings to a maximum of 8191 characters. There is, however, no limit to the length of strings created during program execution. Strings of >8191 characters can be created by applications and used in SKILL if they are not given as arguments to SKILL string manipulation functions.

You specify

- Printable characters (except a double quote) as part of a string without preceding them with the backslash (\) escape character
- Unprintable characters and the double quote itself by preceding them with the backslash (\) escape character as in the C language

Atoms

An *atom* is any data object that is not an aggregate of other data objects. In other words, atom is a generic term covering data objects of all scalar data types. Built into SKILL are several special atoms that are fundamental to the language.

nil — The *nil* atom represents both a false logical condition and an empty list.

t — The symbol *t* represents a true logical condition.

Both *nil* and *t* always evaluate to themselves and must never be used as the name of a variable.

unbound — To make sure you do not use the value of an uninitialized variable, SKILL sets the value of all symbols and array elements initially to *unbound* so that such an error can be detected.

Escape Sequences

The table below lists all the escape sequences supported by SKILL. Any unprintable character can be represented by listing its ASCII code in two or three octal digits after the backslash. For example, BELL can be represented as \007 and Control-c can be represented as \003.

Escape Sequences

Character	Escape Sequence
new-line (line feed)	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
backslash	\\
double quote	\”
ASCII code ddd (octal)	\ddd

Symbols

Symbols in SKILL correspond to variables in C. In SKILL, we often use the terms “symbol” and “variable” interchangeably even though symbols in SKILL are used for other things as well. Each symbol has the following components:

- Print name, which is limited to 255 characters
- Value, which is *unbound* if a value has not been assigned
- Function binding
- Property list (The Property List of a Symbol on page 97.)

Symbol names can contain alphanumeric characters (a-z, A-Z, 0-9), the underscore (_) character, and the question mark (?). If the first character of a symbol is a digit, it must be preceded by the backslash character. Other printable characters can be used in symbol names by preceding each special character with a backslash (\). The following examples

```
Var0
Var_Name
\*name\+
```

are all legal names for symbols. The internal name for the last symbol is *name+.

You can assign values to variables using the equals sign (=) assignment operator. You do not need to declare a variable before assigning a value to it, but you cannot use an unassigned variable, that is, an *unbound* variable. Variables are untyped, which means that the same variable name can store any data type.

It is not advisable to give symbols both a value and a function binding, such as

```
myTest = 3                      ;assigns the value of 3 to myTest.
procedure( myTest( x y ) x+y )  ;declares the function myTest.
```

Characters

SKILL represents characters by symbols instead of using a separate data type. For example, the function *getc* returns the symbol representing a character. To verify this, read the characters one by one in the string “abc”.

```
myPort = instring( "abc" )=> port:"*string*"
char = getc( myPort ) => a ;;; the first character
type( char ) => symbol      ;;; is represented by the symbol a.
getc( myPort ) => b         ;;; the next character
getc( myPort ) => c         ;;; the next character
getc( myPort ) => nil       ;;; end of string
close( myPort )
```

- Use the %c format with the *printf* function to print a single character. For example

  ```
  char = 'A
  printf("Character = %c\n" char ) => t
  ```

  and prints

  ```
  Character = A
  ```

- **Use extreme care** when referring to symbols that represent certain characters. Certain characters must be escaped if they are the initial character in a symbol name. Specifically, you must escape any ASCII character other than an alphabetic character [a-z, A-Z], the underline character (_) or the question mark character (?).

- You can use a character’s ASCII octal value to represent any character other than NULL. Be aware that if the octal code you use as a symbol name defines a printable character then SKILL uses that character as a print name for the symbol. For example,

  ```
  char = '\120  char => P
  printf( "Char = %c\n" '\120 ) => t
  ```

  and prints

  ```
  Char = P
  ```

- As an alternative to ASCII octal values, you can refer to the unprintable characters listed in the Escape Sequences on page 75. For example, the formfeed character is represented by \f and the newline character is represented by \n.

The table below lists all the ASCII characters and their corresponding symbol. The only ASCII character that cannot be handled by SKILL is NULL (ASCII code 0) because the null character always terminates a string in UNIX.

Symbols for ASCII Characters

Character(s)	SKILL Symbol Name(s)
a, b, ..., z	a, b, ..., z
A, B, ..., Z	A, B, ..., Z
?, _	?, _
0, 1, ..., 9	\0, \1, ..., \9
^A, ^B, ..., ^Z, ... (octal codes 001-037)	\001, \002, ..., \032, ... (in octal)
<space>	\<space> (backslash followed by a space)
! . ; : ,	\! \. \; \: \,
( ) [ ] { }	\(\) \[\] \{ ...
“ # % & + - *	\“ \# \% ...
< = > @ / \ ^ ~	\< \= \> ...
DEL	\177

3

Creating Functions in SKILL

Overview information:

Basic Concepts

"Getting Started" on page 31 introduces you to developing a Cadence® SKILL language function. This chapter fills you in on more of the details about constructs for defining a function and defining local and global variables. Several sections in "Advanced Topics" on page 205 elaborate on topics in this chapter.

Terms and Definitions

function, procedure	In SKILL, the terms procedure and function are used interchangeably to refer to a parameterized body of code that can be executed with actual parameters bound to the formal parameters. SKILL can represent a function as both a hierarchical list and as a function object.
argument, parameter	The terms argument and parameter are used interchangeably. The actual arguments in a function call correspond to the formal arguments in the declaration of the function.
byte-code	A generic term for the machine code for a "virtual" machine.
virtual machine	A machine that is not physically built, but is emulated in software instead.
function object	The set of byte-code instructions that implement a function's algorithm. SKILL programs can treat function objects on a basic level like other data types: compare for equality, assigning to a variable, and pass to a function.
function body	The collection of SKILL expressions that define the function's algorithm.
compilation	The generation of byte-code that implements the function's algorithm.
compile time	SKILL compiles function definitions when you load source code. Top-level expressions are compiled and then executed.
run time	The time during which SKILL evaluates a function object.

Kinds of Functions

SKILL has several different kinds of functions, classified by the internal names of *lambda*, *nlambda*, and *macro*. SKILL follows different steps when evaluating these functions.

- Most of the functions you will define are *lambda* functions. SKILL executes a *lambda* function after evaluating the parameters and binding the results to the formal parameters.
- You will probably not need to define an *nlambda* function. However, several built-in SKILL functions are *nlambda* functions.

  An *nlambda* function should be declared to have a single formal argument. When evaluating an *nlambda* function, SKILL collects all the actual argument expressions unevaluated into a list and binds that list to the single formal argument. The body of the nlambda can selectively evaluate the elements of the argument list.
- It is not likely that you will write many macros. A *macro* function allows you to adapt the normal SKILL function call syntax to the needs of your application. Unlike *lambda* and *nlambda* functions, SKILL evaluates a macro at compile-time. When compiling source code, if SKILL encounters a *macro* function call, it evaluates the function call immediately and the last expression computed is compiled in the current function object.

Syntax Functions for Defining Functions

SKILL supports the following syntax functions for defining functions. You should use the *procedure* function or the *defun* function in most cases.

procedure

The *procedure* function is the most general and is easiest to use and understand. Anything that can be done with the other function definition functions can be done with a *procedure* and possibly a *quote* in the call.

The *procedure* function provides the standard method of defining functions. Its return value is the symbol with the name of the function. For example:

```
procedure( trAdd( x y )
      printf( "Adding %d and %d ... %d \n" x y x+y )
      x+y
      ) => trAdd
trAdd( 6 7 ) => 13
```

lambda

The *lambda* function defines a function without a name. Its return value is a function object that can be stored in a variable. For example:

```
trAddWithMessageFun = lambda( ( x y )
      printf( "Adding %d and %d ... %d \n" x y x+y )
      x+y
      ) => funobj:0x1814b90
```

You can subsequently pass a function object to the *apply* function together with an argument list. For example:

```
apply( trAddWithMessageFun '( 4 5 ) ) => 9
```

The use of *lambda* can render code difficult to understand. Often the function being defined is required at some other point in the program and so a procedural definition is better. However, the *lambda* structure can be useful when defining special purpose functions and for passing very small functions to functions such as *sort*. For example, to sort a list *signalList* of disembodied property list objects by a property named *strength*, do the following:

```
signalList = '(
      ( nil strength 1.5 )
      ( nil strength 0.4 )
      ( nil strength 2.5 )
      )
sort( signalList
      'lambda( ( a b ) a->strength <= b->strength )
      )
```

Refer to “Declaring a Function Object (lambda)” on page 210 for further details.

nprocedure

Do not use the *nprocedure* function in new code that you write. It is only included in the system for compatibility with prior releases.

- To allow your function to accept an indeterminate number of arguments, use the *@rest* option with the *procedure* function.
- To allow your function to receive arguments unevaluated, use the *defmacro* function.

defmacro

The *defmacro* function provides a means for you to define a *macro* function. You can use a macro to design your own customized SKILL syntax. Your macro is responsible for translating

your custom syntax at compile time into a SKILL expression to be compiled and subsequently executed.

Refer to "Macros" on page 212 for further discussion and examples.

mprocedures

The *mprocedure* function is a more primitive alternative to the *defmacro* function. The *mprocedure* function has a single argument. The entire custom syntax is passed to the *mprocedure* function unevaluated.

Do not use the *mprocedure* function in new code. It is only included in the system for compatibility with prior releases. Use the *defmacro* function instead. If you need to receive an indeterminate number of unevaluated arguments, use an @*rest* argument.

Refer to "Macros" on page 212 for further discussion and examples.

Summary of Syntax Functions

The following table summarizes each syntax function for declaring a function. You should think twice about using anything other than *procedure*.

Comparison of Syntax Functions

Syntax Function	Function Type	Argument Evaluation	Execution
procedure	lambda	The arguments are evaluated and bound to the corresponding formal arguments.	The expressions in the body are evaluated at run time. The last value computed is returned.
defmacro	macro	The arguments are bound unevaluated to the corresponding formal arguments.	The expressions in the body are evaluated at compile time. The last value computed is compiled.
mprocedure	macro	The entire function call is bound to the single formal argument.	The expressions in the body are evaluated at compile time. The last value computed is compiled.

Comparison of Syntax Functions

Syntax Function	Function Type	Argument Evaluation	Execution
nprocedure	nlambda	All arguments are gathered unevaluated into a list and bound to the single formal argument.	The expressions in the body are evaluated at run time. The last value computed is returned.

Defining Parameters

You can declare how parameters are to be passed to your function by adding certain at (@) options in the formal argument list. The @ options are @*rest*, @*optional*, and @*key*. You can use these options in *procedure*, *lambda*, and *defmacro* argument lists.

@rest Option

The @*rest* option allows an arbitrary number of parameters to be passed to a function in a list. The name of the parameter following @*rest* is arbitrary, although *args* is a good choice.

The following example illustrates the benefits of an @*rest* argument.

```
procedure( trTrace( fun @rest args )
      let( ( result )
            printf( "\nCalling %s passing %L" fun args )
            result = apply( fun args )
            printf( "\nReturning from %s with %L\n" fun result )
            result
            ) ; let
      ) ; procedure
```

For example, invoking the *trTrace* function passing *plus* and *1, 2, 3* returns *6*.

```
trTrace( 'plus 1 2 3 ) => 6
```

and displays the following output in the CIW.

```
Calling plus passing (1 2 3)
Returning from plus with 6
```

- The *trTrace* function calls the *fun* function and passes the arguments it received.
- The *apply* function calls a given function with the given argument list. *trTrace* passes the @*rest* argument list directly to the *apply* function.

The *trTrace* function must accept an arbitrary number of arguments. The number of arguments passed can vary from call to call.

Another benefit of @*rest* is that it puts the arguments into a single list. The *trTrace* function would be less convenient to use if the caller had to put *fun*'s arguments into a list.

@optional Option

The @*optional* option gives you another way to specify a flexible number of arguments. With @*optional*, each argument on the actual argument list is matched up with an argument on the formal argument list.

You can provide any optional parameter with a default value. Specify the default value using a default form. The default form is a two-member list. The first member of this list is the optional parameter's name. The second member is the default value.

The default value is assigned only if no value is assigned when the function is called. If the procedure does not specify a default value for an argument, *nil* is assigned.

If you place @*optional* in the argument list of a procedure definition, any parameter following it is considered optional.

The *trBuildBBox* function builds a bounding box.

```
procedure( trBuildBBox( height width @optional
      ( xCoord 0 ) ( yCoord 0 ) )
            list(
            xCoord:yCoord ;;; lower left
            xCoord+width:yCoord+height ) ;;; upper right
      ) ; procedure
```

Both *length* and *width* must be specified when this function is called. However, the coordinates of the box are declared as optional parameters. If only two parameters are specified, the optional parameters are given their default values. For *xCoord* and *yCoord*, those values are 0.

Examine the following calls to *trBuildBBox* and their return values:

```
trBuildBBox( 1 2 )       => ((0 0) (2 1))
trBuildBBox( 1 2 4 )     => ((4 0) (6 1))
trBuildBBox( 1 2 4 10)   => ((4 10) (6 11))
```

@key Option

@*optional* relies on order to determine what actual arguments are assigned to each formal argument. The @*key* option lets you specify the expected arguments in any order.

For example, examine the following generalization of the *trBuildBBox* function. Notice that within the body of the function, the syntax for referring to the parameters is the same as for ordinary parameters:

```
procedure( trBuildBBox(
      @key ( height 0 ) ( width 0 ) ( xCoord 0 ) ( yCoord 0 ) )

      list(
            xCoord:yCoord ;;; lower left
            xCoord+width:yCoord+height ) ;;; upper right
      ) ; procedure
trBuildBBox()                          => ((0 0) (0 0))
trBuildBBox( ?height 10 )              => ((0 0) (0 10))
trBuildBBox( ?width 5 ?xCoord 10 )     => ((10 0) (15 0))
```

Combining Arguments

@key and *@optional* are mutually exclusive; they cannot appear in the same argument list. Consequently, there are two standard forms that *procedure* argument lists follow:

```
procedure(functionname([var1 var2 ...]
      [@optional opt1 opt2 ...]
      [@rest r])
      .
      .
)
procedure(functionname([var1 var2 ...]
      [@key key1 key2 ...]
      [@rest r])
      .
      .
)
```

Type Checking

Unlike most conventional languages that perform type checking at compile time, SKILL performs dynamic type checking when functions are executed (not when they are defined). Each SKILL lambda or macro function can have as part of its definition an argument template that defines the types of arguments that the function expects. Type checking is not supported in *mprocedure* functions.

Type characters are discussed in “Data Characteristics” on page 70. For type checking purposes, you can use several “composite” type characters representing a union of data types. These type characters are shown in the table below.

Composite Characters for Type Checking

Character	Meaning
S	Symbol or string
n	Number (fixnum, flonum)

Composite Characters for Type Checking

Character	Meaning
u	Function – either the name of a function (symbol) or a lambda function body (list)
g	Any data type

Specifying the Argument Type Template

You specify the argument type template as a string of type characters at the end of a formal argument list. If the template is present, SKILL matches the data type of each actual argument against the template at the time the function is invoked. For example:

```
procedure( f(x y "nn") x**2 + y**2 )
```

nn specifies that *f* accepts two numerical arguments.

```
procedure( comparelength(str len "tx") strlen(str) == len)
```

tx specifies that the first argument must be a string and the second must be an integer.

Local Variables

When you write functions, you should make your variables local. Refer to the before naming any variables.

Defining Local Variables (let, prog)

Using the let Function

Use the *let* function to establish temporary values for local variables.

- Include a list of the local variables followed by one or more SKILL expressions. These variables are initialized to *nil*.
- The SKILL expressions make up the body of the *let* function, which returns the value of the last expression computed within its body.
- The local variables are known only within the *let* statement. The values of the variables are not available outside the *let* statement.

```
procedure( trGetBBoxHeight( bBox )
      let( ( ll ur lly ury )
```

```
            ll       = car( bBox )
            lly      = cadr( ll )
            ur       = cadr( bBox )
            ury      = cadr( ur )
            ury - lly
        ) ; let
    ) ; procedure
```

- The local variables are *ll, ur, lly*, and *ury*.
- They are initialized to *nil*.
- The return value is the *ury - lly*.

Using the *prog* Function

A list of local variables and your SKILL statements make up the arguments to the *prog* function.

```
prog( ( local variables ) your SKILL statements )
```

The *prog* function allows an explicit loop to be written because the *go* function is supported within the *prog*. In addition, *prog* allows you to have multiple return points through use of the *return* function. If you are not using either of these two features, *let* is much simpler and faster.

Initializing Local Variables to Non-nil Values

A *let* expression can initialize local variables to non-*nil* values. Make a two element list with the local variable and its initial value. You cannot refer to any other local variable in the initialization expression.

```
procedure( trGetBBoxHeight( bBox )
      let( ( ( ll car( bBox ) ) ( ur cadr( bBox ) ) lly ury )
            lly            = cadr( ll )
            ury            = cadr( ur )
            ury - lly
      ) ; let
) ; procedure
```

Global Variables

Besides predefined functions that you are not allowed to modify, there are several variable names reserved by various system functions. They are listed in “Naming Conventions” on page 62.

The use of global variables in SKILL, as with any language, should be kept to a minimum.

Following standard naming conventions and running SKILL Lint can reduce your exposure to problems associated with global variables.

Testing Global Variables

Applications typically initialize one or more global variables. Before an application runs for the first time, it is likely that its global variables are *unbound*. In such circumstances, retrieving the value of such a global variable causes an error.

Use the *boundp* function to check whether a variable is *unbound* before accessing its value. For example:

```
boundp( 'trItems ) && trItems
```

returns *nil* if *trItems* is *unbound* and returns the value of *trItems* otherwise.

Avoiding Name Clashes

Two applications might accidentally access and set the same global value. Use a standard naming scheme to minimize the chance of this problem occurring. SKILL Lint can flag global variables that do not obey your naming scheme. For details, refer to "Cadence SKILL Lint" in *SKILL Development Help*.

Assume that *trApplication1* and *trApplication2* are two application functions that are supposed to be totally independent. In particular, the order in which they are executed should not matter. Assume both rely on a single global variable. To observe what can happen if the two applications were accidentally coded to use the same global variable, consider the following example.

```
procedure( trApplication1()
      when( !boundp( 'sharedGlobal ) ;;; not set
            sharedGlobal = 1
            ) ; when
      ) ; procedure

procedure( trApplication2()
      when( !boundp( 'sharedGlobal ) ;;; not set
            sharedGlobal = 2
            ) ; when
      ) ; procedure
```

The order in which you run *trApplication1* and *trApplication2* determines the final value of *sharedGlobal*.

```
sharedGlobal = 'unbound
trApplication1()         => 1
sharedGlobal             => 1
trApplication2()         => nil
sharedGlobal             => 1
```

```
sharedGlobal = 'unbound
trApplication2()         => 2
sharedGlobal             => 2
trApplication1()         => nil
sharedGlobal             => 2
```

Name “clashes” can also occur between functions because programmers can be using the same function names. In this case, a subsequent function definition either overwrites a previous one, or, if *writeProtect* is set, the function definition fails with an error.

Naming Scheme

The recommended naming scheme is to

- Use casing to separate code that is developed within Cadence from that developed outside.
- Use a group prefix to separate code developed within Cadence.

All code developed by Cadence Design Systems should name global variables and functions with an optional underscore; up to three lowercase characters that signify the code package; an optional further lowercase character (one of c, i, or v) and then the name itself starting with an uppercase character. For example, *dmiPurgeVersions()* or *hnlCellOutputs*. All code developed outside Cadence should name global variables by starting them with an uppercase character, such as *AcmeGlobalForm.*

Reducing the Number of Global Variables

One other technique to reduce the number of global variables is to consolidate a collection of related globals into a disembodied property list or a symbol’s property list. That symbol becomes the only global.

This technique could even be extended to associate one symbol with an entire software module. The disadvantage of this approach is that long property lists involve an access time penalty.

Redefining Existing Functions

You often need to redefine a function that you are debugging. The procedure defining constructs allow you to redefine existing functions; however, functions that are write protected cannot be redefined.

- A function not being executed can be redefined if the write protection switch was turned off when the function was initially defined. To turn off the *writeProtect* switch, type

  ```
  sstatus( writeProtect nil )
  ```

- When building contexts, *writeProtect* is always set to *t*.

Aside from debugging, the ability to have multiple definitions for the same function is useful sometimes. For example, within the Open Simulation System (OSS) “default” netlisting functions can be overridden by user-defined functions.

Finally, you should use a standard naming scheme for functions and variables.

Physical Limits for Functions

The following physical limitations exist for functions:

- Total number of *required* arguments is less than 255
- Total number of *keyword/optional* arguments is less than 255
- Total number of local variables in a let is less than 255
- Max number of arguments a function can receive is less than 32KB
- Max size of code vector is less than 32KB

The limitation on the size of the code vector is new. In the past, there was no limit on the size of a SKILL function. Code vectors are limited to functions that can compile to less than 32KB words. This translates roughly into a limit of 20000 lines of SKILL code per function. The maximum number of arguments limit of 32KB is mostly applicable in the case when functions are defined to take an *@rest* argument or in the case of *apply* called on an argument list longer than 32KB elements.

“Cadence SKILL Lint” catches argument numbers greater than the limits with the following message:

```
NEXT RELEASE (DEF6): <filename - line number> (<funcname> :
definition for <funcname> cannot have more than 255 required or
optional arguments.
```

4

Data Structures

Overview information:

Access Operators

Several of the data access operators have a generic nature. That is, the syntax of accessing data for different data types can be the same. You can view the arrow operator as being a property accessor and the array reference operator [ ] as an indexer. The operators described below are used in examples throughout this chapter.

Arrow (->) Operator

The arrow (->) operator can be applied to disembodied property lists, defstructs, association tables, and user types (special application-supplied types) to access property values. The property must always be a symbol and the value of the property can be any valid Cadence® SKILL language type.

Squiggle Arrow (~>) Operator

The squiggle arrow (~>) operator is a generalization of the arrow operator. It works the same way as an arrow operator when applied directly to an object, but it can also accept a list of such objects. It walks the list applying the arrow operator whenever it finds an atomic object.

Array Access Syntax [ ]

The array access syntax [ ] can be used to access

- Elements of an array
- Key-value pairs in an association list

Symbols

Symbols in SKILL correspond to variables in C. In SKILL, the terms “symbol” and “variable” are often used interchangeably even though symbols in SKILL are used for other things as well. Each symbol has the following components:

- Print name
- Value
- Function binding
- Property list

Except for the name slot, all slots can be optionally empty. It is not advisable to give symbols both a value and a function binding.

Creating Symbols

The system creates a symbol whenever it encounters a text reference to the symbol for the first time. When the system creates a new symbol, the value of the symbol is set to *unbound*.

Normally, you do not need to explicitly create symbols. However, the following functions let you create symbols.

Creating a Symbol with a Given Base Name (gensym)

Use the *gensym* function to create a symbol with a given base name. The system determines the index appended to the base name to ensure that the symbol is new. The *gensym* function returns the newly created symbol, which has the value *unbound.* For example:

```
gensym( 'net ) => net2
```

Creating a Symbol from Several Strings (concat)

Use the *concat* function to create a symbol when you need to build the name from several strings.

The Print Name of a Symbol

Symbol names can contain alphanumeric characters (a-z, A-Z, 0-9), the underscore (_) character, and the question mark (?). If the first character of a symbol is a digit, it must be preceded by the backslash character (\). Other printable characters can be used in symbol names by preceding each special character with a backslash.

Retrieving the Print Name of a Symbol (get_pname)

Use the *get_pname* function to retrieve the print name of a symbol. This function is most useful in a program that deals with a variable whose value is a symbol. For example:

```
location = 'U235
get_pname( location ) => "U235"
```

The Value of a Symbol

The value of a symbol can be any type, including the type *symbol*.

Assigning a Symbol's Value

Use the = operator to assign a value to a symbol. The *setq* function corresponds to the = operator. The following are equivalent.

```
U235 = 100
setq( U235 100 )
```

Retrieving a Symbol's Value

Refer to the symbol's name to retrieve its value.

```
U235 => 100
```

Using the Quote Operator with a Symbol

If you need to refer to a symbol itself instead of its value, use the quote operator.

```
location = 'U235 => U235
```

Storing a Symbol's Value Indirectly (set)

You can assign a value indirectly to a symbol with the *set* function. There is no operator that corresponds to the *set* function. The following assigns 200 to the symbol *U235*.

```
set( location 200 )
```

Retrieving a Symbol's Value Indirectly (symeval)

You can retrieve a symbol's value indirectly with the *symeval* function. There is no operator that corresponds to the *symeval* function.

```
symeval( location ) => 200
```

Global and Local Values for a Symbol

Global and local variables and function parameters are handled differently in SKILL than they are in C and Pascal.

SKILL uses symbols for both global and local variables. A symbol's current value is accessible at any time from anywhere. SKILL manages a symbol's value slot transparently as if it were

a stack. The current value of a symbol is simply the top of the stack. Assigning a value to a symbol changes only the top of the stack. Whenever the flow of control enters a *let* or *prog* expression, the system pushes a temporary value onto the value stack of each symbol in the local variable list.

The Function Binding of a Symbol

When you declare a SKILL function, the system uses the function's name to determine a symbol to hold the function definition. The function definition is stored in the function slot.

If you are redefining the function, the same symbol is reused and the previous function definition is discarded.

Unlike the symbol's value slot, the symbol's function slot is not affected when the flow of control enters or exits *let* or *prog* expressions.

The Property List of a Symbol

A *property list* is a list containing property name/value pairs. Each name/value pair is stored as two consecutive elements on a property list. The *property name*, which must be a symbol, comes first. The *property value*, which can be of any data type, comes next.

When a symbol is created, SKILL automatically attaches to it a property list initialized to *nil*. A symbol property list can be accessed in the same way structures or records are accessed in C or Pascal, by using the dot operator and arrow operators.

Setting a Symbol's Property List (setplist)

The *setplist* function sets a symbol's property list. For example:

```
setplist( 'U235 '( x 200 y 300 ) ) => ( x 200 y 300 )
```

Retrieving a Symbol's Property Lst (plist)

The *plist* function returns the property list associated with a symbol.

```
plist( 'U235 ) => ( x 200 y 300 )
```

Using the Dot Operator to Retrieve Symbol Properties

The dot (.) operator gives you a simple way of accessing properties stored on a symbol's property list. The dot operator cannot be nested, and both the left and right sides of the dot operator must be symbols. For example:

```
U235.x => 200
```

The *getqq* function implements the dot operator. For example, the following behave identically.

```
U235.x
getqq( U235 x )
```

The *qq* suffix informally indicates that both arguments are implicitly quoted.

If you ask for the value of a particular property on a symbol and the property does not exist on the symbol's property list, *nil* is returned.

Using the Dot and Assignment Operators to Store Symbol Properties

If you assign a value to a property that does not exist, that property is created and put on the property list.

Using the arrow operator toRetrieve Symbol Properties

The arrow (->) operator gives you a simple way of indirectly accessing properties stored on a symbol's property list. The *getq* function implements the arrow operator. Both the left and right sides of the -> operator must be symbols. For example:

```
designator = 'U235
U235.x = 200
U235.y = 300
designator->x => 200
designator->y => 300
```

Using the Arrow Operator and the Assignment Operator to Store Symbol Properties

The arrow (->) operator and the assignment (=) operator work together to provide a simple way of indirectly storing properties on a symbol's property list. For example:

```
designator->x = 250
U235.x => 250
```

The *putpropq* function implements both the arrow operator and the assignment operator. For example, the following behave identically.

```
designator->x = 250
putpropq( designator 250 x )
```

Important Symbol Property List Considerations

Property lists attached to symbols are *globally visible to all applications*. Whenever you pass a symbol to a function, that function can add or alter properties on that symbol's property list.

In the following example, even though the sample property is established within a *let* expression, it is still available outside the *let* expression. In other words, when the flow of control enters and subsequently exits a *let* or *prog* expression, the property lists of local symbols are not affected.

```
x = 0
let( ( x )
      x = 2
      x.example = 5
      ) ; let
x => 0
plist( 'x ) => (example 5)
```

If you want to use symbol property lists to pass data from one function to another, you must make sure you choose unique names to avoid possible name collisions with other applications. Use setplist with caution because you might inadvertently destroy properties of importance to other applications.

Disembodied Property Lists

A disembodied property list is logically equivalent to a record. Unlike C structures or Pascal records, new fields can be dynamically added or removed. The arrow operator (->) can be used to store and retrieve properties in a disembodied property list.

A disembodied property list starts with a SKILL data object, usually *nil*, followed by alternating name/value pairs. The property names must satisfy the SKILL symbol syntax to be visible to the arrow operator. The first element of the disembodied list does not have to be *nil*. It can be any SKILL data object.

In the following example, a disembodied property list is used to represent a complex number.

```
procedure( trMakeComplex( @key ( real 0 ) ( imaginary 0 ) )
      let( ( result )
            result = ncons(nil)
            result->real = real
            result->imaginary = imaginary
            result
      ) ; let
) ; procedure
```

```
complex1 = trMakeComplex( ?real 2 ?imaginary 3 )
      => (nil imaginary 3 real 2)
complex2 = trMakeComplex( ?real 4 ?imaginary 5 )
      => (nil imaginary 5 real 4)

i = trMakeComplex( ?imaginary 1 )
      => (nil imaginary 1 real 0)

procedure( trComplexAddition( cmplx1 cmplx2 )
      trMakeComplex(
            ?real       cmplx1->real + cmplx2->real
            ?imaginary  cmplx1->imaginary + cmplx2->imaginary
      )
) ; procedure

procedure( trComplexMultiplication( cmplx1 cmplx2 )
      trMakeComplex(
            ?real
                  cmplx1->real * cmplx2->real -
                  cmplx1->imaginary * cmplx2->imaginary
            ?imaginary
                  cmplx1->imaginary * cmplx2->real +
                  cmplx1->real * cmplx2->imaginary
      )
) ; procedure

trComplexMultiplication( i i ) => (nil imaginary 0 real -1)
```

In several circumstances using a disembodied property list to represent a record has advantages over using a symbol's property list.

- An appropriate symbol to which you can attach a property list might not be available. For example, no symbol exists for complex numbers in the example above.
- If you create a symbol for each record your application tracks and your application requires many records, SKILL will have a lot of extra symbols to manage.
- It is easier to pass a disembodied property list as a parameter than it is to pass a symbol as a parameter.

You can store a disembodied property list as the value of a symbol without affecting the symbol's property list.

```
setplist( 'x '( example 0.5 ))
x = complex1     => (nil imaginary 3 real 2)
plist( 'x )      => ( example 0.5 )
```

Important Considerations

Be careful when you are assigning disembodied property lists to variables.

Property list functions that modify property lists modify the original list structures directly. If the property list is not to be shared, use the copy function to make a copy of the original property list.

This caution applies in general to assigning lists as values. Internally, SKILL uses pointers to the lists in virtual memory. For example, as a result of the following assignment

```
complex1 = complex2
```

both symbols *complex1* and *complex2* refer to the same list in virtual memory. Using the arrow operator to modify the *real* or *imaginary* properties of *complex2* is reflected in *complex1*.

Notice that

```
complex1 == complex2     => t
eq( complex1 complex2 )  => t
```

To avoid this problem, perform the assignment as follows.

```
complex1 = copy( complex2 )
```

Notice that

```
complex1 == complex2     => t
eq( complex1 complex2 )  => nil
```

Additional Property List Functions

Adding Properties to Symbols or Disembodied Property Lists (putprop)

putprop adds properties to symbols or disembodied property lists. If the property already exists, the old value is replaced with a new one. The *putprop* function is a *lambda* function, which means all of its arguments are evaluated.

```
putprop('s 1+2 'x)   => 3
s.x = 1+2            => 3
```

Both examples are equivalent expressions that set the property *x* on symbol *s* to 3.

Getting the Value of a Named Property in a Property List (get)

get returns the value of a named property in a property list. *get* is used with *putprop*, where *putprop* stores the property and *get* retrieves it.

```
putprop( 'U235 8 'pins )
```

Assigns the property *pins* on the symbol *U235* to a value of 8.

```
get( 'U235 'pins )   => 8
U235.pins             => 8
```

Adding Properties to Symbols or Disembodied Property Lists (defprop)

defprop adds properties to symbols or disembodied property lists, but none of its arguments are evaluated. *defprop* is the same as *putprop* except that none of its arguments are evaluated.

```
defprop(s 3 x)       => 3
```

Sets property *x* on symbol *s* to 3.

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

Sets property *x* on symbol *s* to the unevaluated expression 1+2.

Removing a Property and Restoring a Previous Value (remprop)

remprop removes a property from a property list. The return value is not useful.

```
setplist( 'U235 '( x 100 y 200 )) => (x 100 y 200)
```

Sets the property list to *(x 100 y 200)*.

```
putprop( 'U235 8 'pins )       => 8
```

Sets the value of the pins property to 8.

```
plist( 'U235 )       => (pins 8 x 100 y 200)
```

Verifies the operation.

```
get( 'U235 'pins )       => 8
```

Retrieves the pins property.

```
remprop( 'U235 'x )
```

Removes the *x* property.

```
plist( 'U235 )       => (pins 8 y 200)
```

Strings

A *string* is a specialized one-dimensional array whose elements are characters.

The string functions in this section are patterned after functions of the same name in the C run-time library. Strings can be compared, taken apart, or concatenated.

Concatenating Strings

Concatenating a List of Strings with Separation Characters (buildString)

buildString makes a single string from the list of strings. You specify the separation character in the third argument. A null string is permitted. If this argument is omitted, *buildString* provides a separating space as the default.

```
buildString( '("test" "il") ".")  => "test.il"
buildString( '("usr" "mnt") "/")  => "usr/mnt"
buildString( '("a" "b" "c"))      => "a b c"
buildString( '("a" "b" "c") "")   => "abc"
```

Concatenating Two or More Input Strings (strcat)

strcat creates a new string by concatenating two or more input strings. The input strings are left unchanged.

```
strcat( "l" "ab" "ef" ) => "labef"
```

You are responsible for any separating space.

```
strcat( "a" "b" "c" "d" )     => "abcd"
strcat( "a " "b " "c " "d " ) => "a b c d "
```

Appending a Maximum Number of Characters from Two Input Strings (strncat)

strncat is similar to *strcat* except that the third argument indicates the maximum number of characters from *string2* to append to *string1* to create a new string. *string1* and *string2* are left unchanged.

```
strncat( "abcd" "efghi" 2)   => "abcdef"
strncat( "abcd" "efghijk" 5) => "abcdefghi"
```

Comparing Strings

Comparing Two String or Symbol Names Alphabetically (alphalessp)

alphalessp compares two objects, which must be either a string or a symbol, and returns *t* if *arg1* is alphabetically less than the name of *arg2*. *alphalessp* can be used with the *sort* function to sort a list of strings alphabetically. For example:

```
stringList = '( "xyz" "abc" "ghi" )
sort( stringList 'alphalessp ) => ("abc" "ghi" "xyz")
```

The next example returns a sorted list of all the files in the login directory.

```
sort( getDirFiles( "~" ) 'alphalessp )
```

Comparing Two Strings Alphabetically (strcmp)

strcmp compares two strings. To simply test if two strings are equal or not, you can use the *equal* command. The return values for *strcmp* indicate

Return Value	Meaning
1	string1 is alphabetically greater than string2.
0	string1 is alphabetically equal to string2.
-1	string1 is alphabetically less than string2.

```
strcmp( "abc" "abb" )=> 1
strcmp( "abc" "abc")=> 0
strcmp( "abc" "abd")=> -1
```

Comparing Two String or Symbol Names Alphanumerically or Numerically (alphaNumCmp)

alphaNumCmp compares two string or symbol names. If the third optional argument is non-*nil* and the first two arguments are strings holding purely numeric values, a numeric comparison is performed on the numeric representation of the strings. The return values indicate

Return Value	Meaning
1	arg1 is alphanumerically greater than arg2.
0	arg1 is alphanumerically identical to arg2.
-1	arg2 is alphanumerically greater than arg1.

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

Comparing a Llimited Number of Characters (strncmp)

strncmp compares two strings alphabetically, but only up to the maximum number of characters indicated in the third argument. The return values indicate the same as for *strcmp* above.

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

Getting Character Information in Strings

Getting the Length of a String in Characters (strlen)

Refer to "Pattern Matching of Regular Expressions" on page 107 for information on the backslash notation used below.

```
strlen( "abc" ) => 3
strlen( "\007" )=> 1
```

Indexing with Character Pointers

Getting the Index Character of a String (getchar)

getchar returns an indexed character of a string or the print name if the string is a symbol.

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

getchar returns a symbol whose print name is the character, not a string.

SKILL represents an individual character by the symbol whose print name is the string consisting solely of the character. For example:

```
getchar( "1.2" 2 )              => \.
type( getchar( "1.2" 2 ) )      => symbol
get_pname( getchar( "1.2" 2 ) )  => "."
```

If you are familiar with C, you should note that the *getchar* SKILL function is totally unrelated to the C function of the same name.

Getting the Tail of a String (index, rindex)

index returns the remainder of *string1* beginning with the first occurrence of *string2*.

```
index( "abc" 'b )               => "bc"
index( "abcdabce" "dab" )       => "dabce"
```

```
index( "abc" "cba" )             => nil
index( "dandelion" "d")          => "dandelion"
```

rindex returns the remainder of *string1* beginning with the last occurrence of *string2*.

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

Getting the Character Index of a String (nindex)

nindex finds the symbol or string, *string2*, in *string1* and returns the character index, starting from one, of the first point at which *string2* matches part of *string1*.

```
nindex( "abc" 'b )         => 2
nindex( "abcdabce" "dab" ) => 4
nindex( "abc" "cba" )      => nil
```

Creating Substrings

Copying Substrings (substring)

substring creates a new substring from an input string, starting at an index point (*arg2*) and continuing for a given length (*arg3*).

```
substring("abcdef" 2 4)=> "bcde"
substring("abcdef" 4 2)=> "de"
```

Breaking Lists Into Substrings (parseString)

parseString breaks a string into a list of substrings with specified break characters, which are indicated by an optional second argument.

```
parseString( "Now is the time" ) => ("Now" "is" "the" "time")
```

Space is the default break character

```
parseString( "prepend" "e" )      => ("pr" "p" "nd" )
```

e is the break character.

```
parseString( "feed" "e")          => ("f" "d")
```

A sequence of break characters in *t_string* is treated as a single break character.

```
parseString( "~/exp/test.il" "./") => ("~" "exp" "test" "il")
```

Both . and / are break characters.

```
parseString( "abc de" "")         => ("a" "b" "c" " " "d" "e")
```

The single space between c and d contributes " " in the return result.

Converting Case

Converting to Upper Case (*upperCase*)

upperCase replaces lowercase alphabetic characters with their uppercase equivalents. If the parameter is a symbol, the name of the symbol is used.

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

symName = "nameofasymbol"    => "nameofasymbol"
upperCase(symName)           => "NAMEOFASYMBOL"
```

Converting to Lower Case (*lowerCase*)

lowerCase replaces uppercase alphabetic characters with their lowercase equivalents. If the parameter is a symbol, the name of the symbol is used.

```
lowerCase("Hello World!")    => "hello world!"
```

Pattern Matching of Regular Expressions

In many applications, you need to match strings or symbols against a pattern. SKILL provides a number of pattern matching functions that are built on a few primitive C library routines with a corresponding SKILL interface.

A *pattern* used in the pattern matching functions is a string indicating a regular expression. Here is a brief summary of the rules for constructing regular expressions in SKILL:

Rules for Constructing Regular Expressions

Synopsis	Meaning
c	Any ordinary character (not a special character listed below) matches itself.
.	A dot matches any character.
\	A backslash when followed by a special character matches that character literally. When followed by one of <, >, (,), and 1,...,9, it has a special meaning as described below.

Rules for Constructing Regular Expressions

Synopsis	Meaning
[c...]	A nonempty string of characters enclosed in square brackets (called a set) matches one of the characters in the set. If the first character in the set is ^, it matches a character not in the set. A shorthand S-E is used to specify a set of characters S up to E, inclusive. The special characters] and - have no special meaning if they appear as the first character in a set.
*	A regular expression in the above form, followed by the closure character * matches zero or more occurrences of that form.
+	Similar to *, except it matches *one* or more times.
\(...\)	A regular expression wrapped as \(form \) matches whatever *form* matches, but saves the string matched in a numbered register (starting from one, can be up to nine).
\n	A backslash followed by a digit *n* matches the contents of the *n*th register from the current regular expression.
\<...\>	A regular expression starting with a \< and/or ending with a \> restricts the pattern matching to the beginning and/or the end of a word. A word defined to be a character string can consist of letters, digits, and underscores.
rs	A composite regular expression *rs* matches the longest match of *r* followed by a match for *s*.
^, $	A ^ at the beginning of a regular expression matches the beginning of a string. A $ at the end matches the end of a string. Used elsewhere in the pattern, ^ and $ are treated as ordinary characters.

How Pattern Matching Works

The mechanism for pattern matching

- Compiles a pattern into a form and saves the form internally
- Uses that internal form in every subsequent matching against the targets until the next pattern is supplied

The *rexCompile* function does the first part of the task, that is, the compilation of a pattern. The *rexExecute* function takes care of the second part, that is, actually matching a target against the previously compiled pattern. Sometimes this two-step interface is too low-level

and awkward to use, so functions for higher-level abstraction (such as *rexMatchp*) are also provided in SKILL.

Avoiding Null and Backslash Problems

- A null string ("") is interpreted as no pattern being supplied, which means the previously compiled pattern is still used. If there was no previous pattern, an error is signaled.
- To put a backslash character (\) into a pattern string, you need an extra backslash (\) to escape the backslash character itself.

For example, to match a file name with dotted extension ".il", the pattern "^[a-zA-Z]+\\.il$" can be used, but "^[a-zA-Z]\.il$" gives a syntax error. However, if the pattern string is read in from an input function such as *gets* that does not interpret backslash characters specifically, you should *not* add an extra backslash to enter a backslash character.

Pattern Matching Functions

Finding a Pattern Within a String or Symbol (rexMatchp)

```
rexMatchp("[0-9]*[.][0-9][0-9]*" "100.001")   => t
rexMatchp("[0-9]*[.][0-9]+" ".001")           => t
rexMatchp("[0-9]*[.][0-9]+" ".")              => nil
rexMatchp("[0-9]*[.][0-9][0-9]*" "10.")     => nil

rexMatchp("[0-9" "100")
=> *Error* rexMatchp: Missing ] - "[0-9"
```

Compiling a Regular Expression String Pattern (rexCompile)

rexCompile compiles a regular expression string pattern into an internal representation to be used by succeeding calls to *rexExecute*.

```
rexCompile("^[a-zA-Z]+")                  => t
rexCompile("\\([a-z]+\\)\\.\\1")          => t
rexCompile("^\\([a-z]*\\)\\1$")           => t
rexCompile("[ab") => *Error* rexCompile: Missing ] - "[ab"
```

Matching Against a Previously Compiled Pattern (rexExecute)

rexExecute matches a string or symbol against the previously compiled pattern created by the last *rexCompile* call.

```
rexCompile("^[a-zA-Z][a-zA-Z0-9]*")       => t
rexExecute('Cell123)                      => t
rexExecute("123 cells")                  => nil
```

The target "123 cells" does not begin with a-z/A-Z.

```
rexCompile("\\([a-z]+\\)\\.\\1")          => t
rexExecute("abc.bc")                      => t
rexExecute("abc.ab")                     => nil

rexCompile("\\(^[a-z]+\\)\\.\\1")         =>t
rexExecute("abc.bc")                     => nil
```

The caret (^) in the *rexCompile* pattern requires that the pattern must match from the beginning of the input string.

Matching a List of Strings or Symbols (rexMatchList)

rexMatchList matches a list of strings or symbols against a regular expression string pattern and returns a list of the strings or symbols that match.

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

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

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

Creating an Association List Made of Matching Strings (rexMatchAssocList)

rexMatchAssocList returns a new association list created out of those elements of an association list whose key matches a regular expression string pattern.

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

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

Turning Meta-Characters On and Off (rexMagic)

rexMagic turns on or off the special interpretation associated with the meta-characters (^, $, *, +, \, [,], and so forth) in regular expressions. Users of *vi* will recognize this as equivalent to the *set magic/set nomagic* commands.

```
rexCompile( "^[0-9]+" )      => t
rexExecute( "123abc" )       => t
rexSubstitute( "got: \\0")   => "got: 123"
rexMagic( nil )               => nil
rexCompile( "^[0-9]+" )      => t;Recompile w/o magic.
rexExecute( "123abc" )       => nil
```

```
rexExecute( "**^[0-9]+!**")  => t
rexSubstitute( "got: \\0")   => "got: \\0"
rexMagic( t )                 => t
rexSubstitute( "got: \\0")   => "got: ^[0-9]+"
```

Replacing a Substring (rexReplace)

rexReplace replaces the substring(s) in the source string that matched the last regular expression compiled by the replacement string. The third argument tells which occurrence of the matched substring is to be replaced. If it's 0 or negative, all the matched substrings will be replaced. Otherwise only the specified occurrence is replaced. *rexReplace* returns the source string if the specified match is not found

```
rexCompile( "[0-9]+" )=> t
rexReplace( "abc-123-xyz-890-wuv" "(*)" 1)=> "abc-(*)-xyz-890-wuv"
rexReplace( "abc-123-xyz-890-wuv" "(*)" 2)=> "abc-123-xyz-(*)-wuv"
rexReplace( "abc-123-xyz-890-wuv" "(*)" 3)=> "abc-123-xyz-890-wuv"
rexReplace( "abc-123-xyz-890-wuv" "(*)" 0)=> "abc-(*)-xyz-(*)-wuv"

rexCompile( "xyz" )              => t
rexReplace( "xyzzyxyzz" "xy" 0)  => "xyzyxyz" ; No rescanning!
```

Defstructs

Defstructs are collections of one or more variables. They can be of different types and grouped together under a single name for easy handling. They are the equivalent of structs in C.

The following template for *defstruct* defines a data structure with the named slots:

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

The *defstruct* also creates a constructor function, *make_name*, where *name* is the structure name supplied to *defstruct*. The constructor function takes keyword arguments: one for each slot in the structure. All arguments are symbols and none need to be quoted.

Behavior Is Similar to Disembodied Property Lists

Once created, structures behave just like disembodied property lists, but are more efficient in space utilization and access times. Structures can have new slots added at any time. However, these dynamic slots are less efficient than the statically declared slots, both in access time and space utilization.

Defstructs respond to the following operations, assuming *struct* is an instance built from a constructor function:

```
struct->slot
```

Returns the value associated with a slot of an instance.

```
struct->slot = newval
```

Modifies the value associated with a slot of an instance.

```
struct->?
```

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

```
struct->??
```

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

Additional Defstruct Functions

Testing a SKILL Object (defstructp)

```
defstructp( g_object [st_name] ) => t / nil
```

defstructp tests a SKILL object, returning *t* if it's a structure instance, otherwise *nil*. The second argument is optional and denotes the name of the structure to test for. The test in this case is stronger and only returns *t* if *g_object* is an instance of defstruct *st_name*. The name can be passed either as a symbol or a string.

Printing the Contents of a Structure (printstruct)

```
printstruct( r_structureInstance ) => t
```

For debugging, the *printstruct* function prints the contents of a structure in an easily readable form. It recursively dumps out any slot value that is also a structure instance. The *printstruct* function dumps out a structure instance.

Beware of Structure Sharing (copy_<name>)

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

Making a Deep or Recursive Copy (copyDefstructDeep)

```
copyDefstructDeep( r_object ) => r_object
```

Performs a deep or recursive copy on defstructs with other defstructs as sub-elements, making copies of all the defstructs encountered. The various *copy_name* functions are called to create copies for the defstructs encountered in the deep copy.

Accessing Named Slots in SKILL Structures

Slot Access Example

This example defines a *card* structure and allocates an instance of *card*.

```
defstruct( card rank suit faceUp ) => t
```

This structure has three slots: *rank suit faceUp*. Next, allocate an instance of *card* and store a reference to it in *aCard*.

```
aCard = make_card( ?rank 'ace ?suit 'spades )
=> array[5]:21556040
```

Structure instances are implemented as arrays. Refer to "Arrays" on page 115.

```
aCard => array[5]:21556040
```

The *type* function returns the structure name.

```
type( aCard ) => card
```

Use the Arrow Operator -> and ~> to Access Slots

```
aCard->rank => ace
aCard->faceUp = t => t
```

Use ->? to Get a List of the Slot Names

```
aCard->? => ( faceUp suit rank )
```

Use ->?? to Get a List of Slot Names and Values

```
aCard->?? => ( faceUp t suit spades rank ace )
```

Slots can be created dynamically for an instance by simply referencing them with the -> operator.

If you have a list of instances of defstructs and you wish access the same slot in all the instances in that list use the ~> operator.

```
cardList = list( make_card( ?rank 'ace ?suit 'spades )
                 make_card( ?rank 'king ?suit 'diamonds))

cardList~>rank       => (ace king)
```

```
cardList~>faceUp     => (nil nil)
cardList~>faceUp = t => (t t)
cardList~>faceUp     => (t t)
```

Extended defstruct Example

1. Define a structure.

```
defstruct(point x y)            => t
```

2. Define another structure.

```
defstruct(bbox ll ur)           => t
```

3. Make an instance.

```
p1 = make_point(?x 100 ?y 200)  => array[4]:xxx
```

4. Make another instance.

```
p2 = make_point(?x 0 ?y 0)      => array[4]:xxxx
```

5. Make a *bbox* instance.

```
b1 = make_bbox()                => array[4]:xxxx
```

6. Set a field in *b1*.

```
b1->ll = p2                     => array[4]:xxxx
```

7. Set the other field.

```
b1->ur = p1                     => array[4]:xxxx
```

8. Look inside and note the recursive printing.

```
printstruct( b1 )               ; Look inside
 Structure of type bbox :       ; Note the recursive printing
      ll :
       Structure of type point:
                  x:0
            y:0


ur:
      Structure of type point:
            x: 100
      y: 200
b1->ll->x = 12
=> 12
printstruct( p2 )
Structure of type point :
      x: l2
      y: 0
p1->??
=> (y 200 x 100)
b1->??
=> (ur array[4]:xxx ll array[4]:xxx)
```

9. Add a previously undefined slot.

```
b1->color = 'blue
printstruct( b1 )
      Structure of type bbox :
            ll :
                  Structure of type point :
            x: 12
            y: 0
      ur:
      Structure of type point  :
            x: 100
            y: 200
      color: blue

b1->?
=> (color ll ur)
```

Returns the list of currently defined fields.

Arrays

An *array* represents aggregate data objects in SKILL. Unlike simple data types, you must explicitly create arrays before using them so the necessary storage can be allocated. SKILL arrays allow efficient random indexing into a data structure using familiar syntax.

- Arrays are not typed. Elements of the same array can be different data types.
- SKILL provides run-time array bounds checking.
- Arrays are one dimensional. You can implement higher dimensional arrays using single dimensional arrays. You can create an array of arrays.
- The array bounds are checked with each array access during run-time. An error occurs if the index is outside the array bounds.

Allocating an Array of a Given Size

Use the *declare* function to allocate an array of a given size.

```
declare( week[7] )       => array[7]:9780700
week                     => array[7]:9780700
type( week )             => array
arrayp( week )           => t
days = '(monday tuesday wednesday
                 thursday friday saturday sunday)
for( day 0 length(week)-1
      week[day] = nth(day days))
```

- The *declare* function returns the reference to the array storage and stores it as the value of *week*.

- The *type* function returns the symbol *array*.
- The *arrayp* function returns *t*.

Accessing Arrays

When the name of an array appears without an index on the right side of an assignment statement, only the array object is used in the assignment; the values stored in the array are not copied. It is therefore possible for an array to be accessible by different names. Indices are used to specify elements of an array and always start with 0; that is, the first element of an array is element 0. SKILL normally checks for an out-of-bounds array index with each array access.

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

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

```
b = a
```

b now refers to the same array as *a*.

```
declare(c[10])
```

Declares another array of 10 elements.

```
declare(d[2])
```

Declares *d* as an array of 2 elements.

```
d[0] = b
```

d[0] now refers to the array pointed to by *b* and *a*.

```
d[1] = c
```

d[1] is the array referred to by *c*.

```
d[0][2]
```

Accesses element 2 of the array referred to by *d[0]*.
This is the same element as *a[2]*.

Brackets ([ ]) are used to represent array references and are part of the statement syntax. The *declare* function is also an example of an *nlambda* function. The arguments of *nlambda* functions are passed literally (that is, not evaluated). It is up to the called function to evaluate selected arguments when necessary.

Association Tables

An association table is a generalized array, a collection of key/value pairs. The SKILL data types that can be used as keys in a table are integer, float, string, list, symbol, and instances of certain user-defined types. An association table is implemented internally as a hash table.

An association table lets you look up any entry with valid instances of SKILL data types. Data is stored in key/value pairs that can be quickly accessed with syntax for standard array access and various iterative functions. This access is based on a system that uses the SKILL *equal* function to compare keys.

Association tables offer convenience and performance not available in disembodied property lists, arrays, or association lists. Disembodied property lists and association lists are not efficient in situations where their contents can expand greatly. In addition, using symbols for properties or keys in a disembodied property list can be wasteful. A simple conversion process converts disembodied property lists and association lists to association tables. An association table can also be converted to a list of association pairs.

Initializing Tables

The *makeTable* function defines and initializes the association table. This function takes a single string argument as the table name for printing purposes. An optional second argument provides the default value that is returned when a query to the table yields no match. The *tablep* predicate verifies the data type of a table, and the *length* function determines the number of keys in the table. To refer to and add elements, use the syntax for standard array access.

The following example creates a table and loads it with keys and related values that pair numbers and colors for a color map. The keys can be any of the data type mentioned earlier; they are not restricted to numeric data types.

```
myTable = makeTable("atable1" 0) => table:atable1
tablep(myTable)                  => t
myTable[1] = "blue"              => "blue"
myTable["two"] = '(r e d)        => (r e d)
myTable["three"] = 'green        => green
length(myTable)                  => 3
```

If a new pair is added to the table but its key already exists, the new value replaces the existing value in the table.

If a key to be accessed does not exist, the process returns either the default value specified at table creation or the symbol *unbound*, if no default value was specified.

Manipulating Table Data

The *foreach, forall,* and *setof* functions scan the contents of an association table and perform iterative programming functions on each key and its associated value. Standard input and output functions are available through the *readTable, writeTable*, and *printstruct* functions.

The *append* function appends data from existing disembodied property lists or association lists to an existing association table. You specify the association table (created with the *makeTable* function) as the first argument for this function. For the second argument, you specify the disembodied property list, association list, or other association table whose data is to be appended.

Association Table Functions

Several list-oriented functions also work on tables, including iteration.

List-Oriented Functions for Tables

Use this	To do this
Syntax for array access	To store and retrieve entries in a table
makeTable function	To create and initialize the association table. The arguments are the table name (required) and (optional) the default value to return for keys not present in the table. The default is *unbound*.
foreach, foreach mapcar functions	To execute a collection of SKILL expressions for each key/value pair in a table
setof function	To return a list of keys in a table that satisfy a criterion.
length function	To return the number of key/value pairs in a table
remove function	To remove a key from a table

Testing Whether a Data Value is a Table (tablep)

Use the *tablep* function to test whether a data value is a table.

```
myTable = makeTable("atable1" 0)  => table:atable1
tablep(myTable)                   => t
tablep(9)                         => nil
```

Converting the Contents of an Association Table to an Association List (tableToList)

This function eliminates the efficiency that you gain from referencing data in an association table. Do not use this function for processing data in an association table. Instead, use this function interactively to look at the contents of a table.

```
tableToList(myTable)
=> (("two" (r e d)) ("three" green) (1 "blue"))
```

Writing the Contents of an Association List to a File (writeTable)

The *writeTable* function is for writing basic SKILL data types that are stored in an association table. The function cannot write database objects or other user-defined types that might be stored in association tables.

```
writeTable("inventory.log" myTable) => t
```

Appending the Contents of a File to an Existing Association Table (readTable)

The file must have been created with the *writeTable* function so that the contents are in a usable format.

```
readTable("inventory.log" myTable)=> t
```

Printing the Contents of an Object in a Tabular Format (printstruct)

For debugging, the *printstruct* function prints the contents of a structure in an easily readable form. It recursively prints nested structures.

```
printstruct(myTable)
=>      1: "blue"
      "three": green
      "two": (r e d)
```

Traversing Association Tables

Use the *foreach* function to visit every key in an association table. For example, use the following function call to print each key/value pair in a table.

```
foreach( key myTable
            println(
                  list( key myTable[ key ] )
                  )
      )
```

You can also use the functions *forall, exists* and *setof* to traverse association tables. (These functions are described in detail in “Advanced List Operations” on page 177)

For example, use the following function call to test if every key/value pair in a table are such that the key is a string and value is an integer.

```
forall( key myTable
            stringp(key) && fixp(myTable[key])
            )
```

To check if there is a single pair that satisfies the above expression, call the following function.

```
exists( key myTable
            stringp(key) && fixp(myTable[key])
            )
```

- To write the entire contents of a table to a file, use *writeTable*.
- To read a file (created using *writeTable*), use *readTable*.
- To view the contents of a table, use *printstruct*.

The *append* function appends data from existing disembodied property lists or association lists to an existing association table.

Implementing Sparse Arrays

A sparse array is an indexed collection, most of whose entries are unused. For large sparse arrays, it is wasteful to allocate the entire array. Instead, you can use an association table for a one-dimensional array. Use integers as the keys. To implement a two-dimensional sparse array, use lists of index pairs as keys.

```
procedure( tr2DSparseArray()
            makeTable( gensym( 'trSparseArray ) )
      ) ; procedure

trSparseTimesTable = tr2DSparseArray( )
for( i 0 3
      for( j 0 6
            trSparseTimesTable[ list( i j ) ] = i*j
            ) ; for
      ) ; for
```

Association Lists

A list of key/value pairs is a natural means to record associations. An association list is a list of lists. The first element of each list is the key. The key can be an instance of any of SKILL's basic types.

```
assocList = '( ( "A" 1 ) ( "B" 2 ) ( "C" 3 ) )
```

The *assoc* function retrieves the list given the index.

```
assoc( "B" assocList ) => ( "B" 2 )
assoc( "D" assocList ) => nil
```

Use the *rplaca* function to destructively update an entry. The following replaces the *car* of the *cdr* of the association list entry.

```
rplaca( cdr( assoc( "B" assocList ) ) "two" )
=> ( "two" )

assocList => (( "A" 1 ) ( "B" "two" ) ( "C" 3 ))
```

Association lists behave the same way as association tables. For lists with less than ten pairs, it is more efficient to use association lists than association tables. For lists likely to grow beyond ten pairs, it is more efficient to use association tables.

User-Defined Types

User-defined types are special foreign or external data types exported into SKILL by various applications. Their behavior is predetermined by the applications that own them. For example, database and window objects are generally implemented as C-structs and exported into SKILL as user-defined types.

The application that defines the SKILL behavior for the user-defined types provides methods for SKILL to apply in various situations. For example, when you apply the accessor operators -> or ~> to a user-defined type, the SKILL engine resolves the operation by calling the accessor method implemented for that type by an application.

There are other methods to support a user-defined type's behavior in SKILL. For example, to test two user-defined types for equality (*equal*), the application exporting the type provides a method to overload the SKILL *equal* function just for that type. The equal method takes two arguments and returns *t* or *nil*.

The application exporting the type determines what methods are needed to support the type in SKILL. If the application does not supply a method, SKILL applies a default behavior. In general, to a user, instances of a user-defined type look and feel very similar to instances of *defstructs*.

Specific information on user-defined types is supplied by the applications exporting the types. For example, creating instances of user-defined types happens when certain application functions are called, such as *dbOpen*.

5

Arithmetic and Logical Expressions

Overview information:

Evaluating Expressions

Expressions are Cadence® SKILL language objects that also evaluate to SKILL objects. SKILL performs a computation as a sequence of function evaluations. A SKILL *program* is a sequence of expressions that perform a specified action when evaluated by the SKILL interpreter.

There are three types of primitive expressions in SKILL: constants, variables, and function calls.

Constants

A *constant* is an expression that evaluates to itself. That is, evaluating a constant returns the constant itself. Examples of constants are

`123`, `10.5`, and `"abc"`

Variables

A *variable* stores values used during the computation. It returns its value when evaluated. Examples of variables are

`a`, *`x`*, and *`init_var`*

When the interpreter evaluates a variable whose value has not been initialized, it displays an error message telling you that you have an *unbound* variable. Users often come upon the error message when they misspell a variable because the misspelling creates a new variable.

```
myVariable
```

causes an error message because it has been referenced before being assigned whereas

```
myVariable = 5
```

works.

When SKILL creates a variable, it gives the variable an initial value of *unbound*. It is an error to evaluate a variable with this value because the meaning of *unbound* is that-value-which-represents-no-value. *unbound* is not the same as *nil*.

Function Calls

When evaluated, a *function call* applies the named function to the list of arguments and returns the result.

Examples of function calls are

f(a b c d), abs(-123), and exit()

Creating Arithmetic and Logical Expressions

Constants, variables, and function calls can be combined with the *infix* operators, such as less than (<), colon (:), and greater than (>) to form arithmetic and logical expressions. For example:

1+2, a*b+c,and x>y

You can form arbitrarily complicated expressions by combining any number of the primitive expressions described above.

Role of Parentheses

Parentheses ( ) delimit the names of functions from their argument lists and delimit nested expressions. In general, the innermost expression of a nested expression is evaluated first, returning a value used in turn to evaluate the expression enclosing it, and so on until the expression at the top level is evaluated.

Parentheses resemble natural mathematical notation and are used in both arithmetic and control expressions.

Quoting to Prevent Evaluation

Occasionally you might want to prevent expressions from being evaluated. This is done by “quoting” the expression, that is, putting a single quote just before the expression.

Quoting Variables

Quoting is often used with the names of variables and their values. For example, putting a single quote before the variable *a* (that is, *'a*) prevents *a* from being evaluated. Instead of returning the value of *a*, the name of the variable *a* is returned when *'a* is evaluated.

Quoting Lists

You generally specify lists of data within a program by quoting. Quoting is necessary because of the common list representation used for both program and data. For example, evaluating the list *(f a b c)* leads to the interpretation that *f* is the name of a function and *(a b c)* is a list of arguments for *f*. By quoting the same list, *'(f a b c)*, the list is instead treated as a data list containing the four elements *f, a, b,* and *c*.

Try It Out

The best way to understand evaluation and quoting is by trying them out in SKILL. An interactive session with the SKILL interpreter will help to clarify and reinforce many of the concepts just described.

Arithmetic and Logical Operators

All arithmetic operators are translated into calls to predefined SKILL functions. These operators are listed in the table below in descending order of operator precedence. The table also lists the names of the functions, which can be called like any other SKILL function.

Error messages report problems using the name of the function. The letters in the Synopsis column refer to data types. Refer to the Data Types on page 71 for a discussion of data type characters.

Arithmetic and Logical Operators

Name of Function(s)	Synopsis	Operator
Data Access		
arrayref	a[index]	[ ]
setarray	a[index] = expr	
bitfield1	x<bit>	<>
setqbitfield1	x<bit>=expr	
setqbitfield	x<msb:lsb>=expr	
quote	'expr	'
getqq	g.s	.
getq	g–>s	–>
putpropqq	g.s = expr, g–>s = expr	~>

Arithmetic and Logical Operators

Name of Function(s)	Synopsis	Operator
putpropq	d~>s, d~>s = expr	
Unary		
preincrement	++s	++
postincrement	s++	++
predecrement	--s	--
postdecrement	s--	--
minus	–n	–
null	!expr	!
bnot	~x	~
Binary		
expt	n1 ** n2	**
times	n1 * n2	*
quotient	n1 / n2	/
plus	n1 + n2	+
difference	n1 - n2	–
leftshift	x1 << x2	<<
rightshift	x1 >> x2	>>
lessp	n1<n2	<
greaterp	n1>n2	>
leqp	n1<=n2	<=
geqp	n1>=n2	>=
equal	g1 == g2	==
nequal	g1 != g2	!=
band	x1 & x2	&
bnand	x1 ~& x2	~&
bxor	x1 ^ x2	^
bxnor	x1 ~^ x2	~^

Arithmetic and Logical Operators

Name of Function(s)	Synopsis	Operator
bor	x1 | x2	
bnor	x1 ~| x2	|, ~|
and	rel. expr && rel. expr	&&
or	rel. expr || rel. expr	||
range	g1 : g2	:
setq	s = expr	=

The following table gives more details on some of the arithmetic operators.

More on Arithmetic Operators

Arithmetic Operator	Comments
+, –, *, and /	Perform addition, subtraction, multiplication, and division operations.
Exponentiation operator **	Has the highest precedence among the binary operators.
Shift operators (<<, >>)	Shift their first arguments left or right by the number of bits specified by their second arguments. Both the left and right shifts are logical (that is, vacated bits are 0-filled).
Preincrement operator (++ appearing before the name of a variable)	Takes the name of a variable as its argument, increments its value (which must be a number) by one, stores it back into the variable, and then returns the incremented value.
Postincrement operator (++ appearing after the name of a variable)	Takes the name of a variable as its argument, increments its value (which must be a number) by one, and stores it back into the variable. However, it returns the original value stored in the variable as the result of the function call.
Predecrement and postdecrement operators	Similar to pre- and postincrement, but they decrement instead of increment the values of their arguments by one.
Range operator (:)	Evaluates both of its arguments and returns the results as a two-element list. It provides a very convenient way of grouping a pair of data values for subsequent processing. For example, 1:3 returns the list *(1 3)*.

Predefined Arithmetic Functions

In addition to the basic *infix* arithmetic operators, several functions are predefined in SKILL.

Predefined Arithmetic Functions

Synopsis	Result
General Functions	
add1(n)	n + 1
sub1(n)	n – 1
abs(n)	Absolute value of n
exp(n)	e raised to the power n
log(n)	Natural logarithm of n
max(n1 n2 ...)	Maximum of the given arguments
min(n1 n2 ...)	Minimum of the given arguments
mod(x1 x2)	x1 modulo x2, that is, the integer remainder of dividing x1 by x2
round(n)	Integer whose value is closest to n
sqrt(n)	Square root of n
sxtd(x w)	Sign-extends the rightmost w bits of x, that is, the bit field x<w-1:0> with x<w-1> as the sign bit
zxtd(x w)	Zero-extends the rightmost w bits of x, executes faster than doing x<w-1:0>
Trigonometric Functions	
sin(n)	sine, argument n is in radians
cos(n)	cosine
tan(n)	tangent
asin(n)	arc sine, result is in radians
acos(n)	arc cosine
atan(n)	arc tangent
Random Number Generator	

Predefined Arithmetic Functions

Synopsis	Result
random(x)	Returns a random integer between 0 and x-1. If random is called with no arguments, it returns an integer that has all of its bits randomly set.
srandom(x)	Sets the initial state of the random number generator to x

Bitwise Logical Operators

The *bnot, band, bnand, bxor, bxnor, bor,* and *bnor* operators all perform bitwise logical operations on their integer arguments.

Bitwise Logical Operators

Meaning	Operator
bitwise AND	&
bitwise inclusive OR	|
bitwise exclusive OR	^
left shift	>>
right shift	<<
one's complement (unary)	~

Bit Field Operators

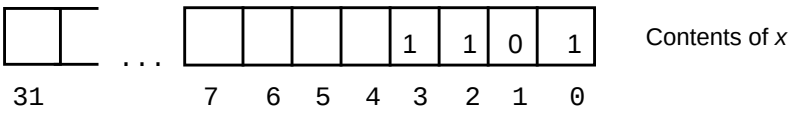
Bit field operators operate on bit fields stored inside 32-bit words. To avoid confusion in naming bits, SKILL uses the uniform convention that the least significant bit in an integer is bit 0, with the bit number increasing as you move left in the direction of the most significant bit.

- You can select bit fields by naming the leftmost and the rightmost bits in the bit field or by just naming the bit if the bit field is only one bit wide.
- You can use either integer constants or integer variables to specify bit positions, but expressions are not allowed.
- All bit fields are treated as unsigned integers.
- Use the *sxtd* function for sign-extending a bit field.

Bit Field Examples

```
x = 0b01101     => 13
```

Assigns *x* to 13 in binary.




```
x<0>      => 1
```

The contents of the rightmost bit of *x* is 1.

```
x<1>      => 0
```

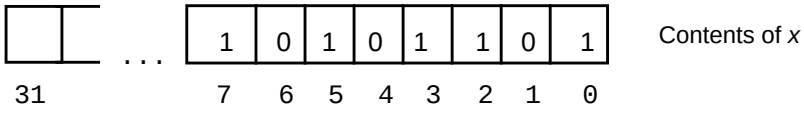
The contents of bit one of *x* is 0.

```
x<2:0>    => 5
```

Extracts the contents of the rightmost three bits of *x*. These are *101* or 5 in decimal.

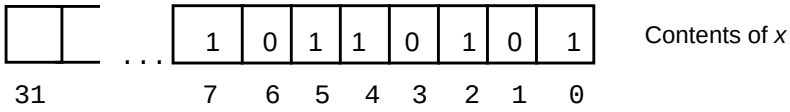
```
x<7:4> = 0b1010 => 173
```

Stores the bit pattern into the bits 4 through 7.




```
(x + 4)<4:3>    => 2
```

Adds 4 to *x*, then extracts the 3rd and 4th bits. 173 plus 4 is 177. SKILL returns the result of the last expression, which is binary *10* or 2 decimal.



Calling Conventions for Bit Field Functions

Because of limitations in the grammar, only integer constants or names of variables are permitted inside the angle brackets. To use the value of an expression to specify a bit position, you must either first assign the value of the expression to a variable or directly call the bit field functions without using the less than (<), colon (:), and greater than (>) *infix* operators. The calling conventions for the bit field functions are as follows:

```
bitfield1(
      x_value
      x_bitPosition )
setqbitfield1(
      s_name
      x_newvalue
      x_bitPosition )
bitfield(
      x_value
      x_leftmostBit
      x_rightmostBit )
setqbitfield(
      s_name
      x_newvalue
      x_leftmostBit
      x_rightmostBit )
```

Mixed-Mode Arithmetic

SKILL makes a distinction between integers and floating-point numbers.

- Integer arithmetic is used if all the arguments given to an arithmetic operator are integers.
- Floating-point arithmetic is used if all arguments are floating-point numbers.
- When integers and floating-point numbers are mixed, SKILL tries to stay with integer arithmetic until it encounters a floating-point number. SKILL then switches to floating-point arithmetic and returns a floating-point number as the result.

Integer vs. Floating-Point Division

The division operator requires special attention because

- Integer division truncates its results
- Floating-point division computes an exact number

In fact, two numbers can only be equal if they are of the same type (that is, *integer* or *float*) and have identical values. To compare an integer to a floating-point number, explicit type conversion is needed before a meaningful comparison can be made.

Type Conversion Functions (fix and float)

The *fix* function converts a floating-point number to an integer. The *float* function converts an integer to a floating-point number. If the argument given to *fix* or *float* is already of the desired type, the argument is returned.

Comparing Floating-Point Numbers

Unless two floating-point numbers are assigned identical constants or are generated by exactly the same sequence of computations, it is unlikely they are exactly equal when you compare them.

Two floating-point numbers might appear the same when printed out, and yet differ internally in their least significant bits. This difference is further compounded because SKILL stores all floating-point numbers in double precision. Some applications only store floating-point numbers in single precision, which causes a loss of precision when a floating-point number is stored or retrieved. For example:

```
if( (a == b) println("same"))
```

Simple comparison rarely works for floating-point numbers.

```
if( (abs(a - b)/a < 1e-6) println("same"))
```

Range comparison is much more robust.

Function Overloading

Some applications that are based on SKILL overload the arithmetic and/or bit-level functions with new or modified semantics. While *sqrt(-1)* normally signals an error in SKILL, it returns a valid result as a complex number when some applications are running.

Because the arithmetic and bit-level operators are just simpler syntax for calling their corresponding functions, by overloading these functions with extended semantics for new data types, you can use the same familiar notation in writing expressions involving objects of these new data types.

By overloading the *plus*, *difference*, *times*, and *quotient* functions for a complex-number data type, you can use +, -, *, and / in forming arithmetic expressions involving complex numbers as you normally do in writing mathematical formulas.

This kind of function overloading is done by the individual application. There is no support for user-definable function overloading in SKILL. Refer to the reference manuals of the individual applications for more details about which functions/operators have been overloaded and what semantics to use.

Integer-Only Arithmetic

In addition to standard arithmetic functions that can handle integers and floating-point numbers, SKILL provides several integer-only arithmetic functions that are slightly faster than the standard functions. These functions are named by prepending *x* to the names of the corresponding standard functions: *xdifference*, *xplus*, *xquotient*, and *xtimes*.

When integer mode is turned on using the *sstatus* function, the SKILL parser translates all arithmetic operators into calls on integer-only arithmetic functions. This results in small execution time savings and makes sense only for compute-intensive tasks whose inner loops are dominated by integer arithmetic calculations.

```
sstatus( integermode t)=> t
```

Turns on integer mode.

```
status( integermode )=> t
```

Checks the status of *integermode* and returns *t* if *integermode* is on. The default is off.

The internal variables are typically Boolean switches that accept only the Boolean values of *t* and *nil*. For efficiency and security, system variables are stored as internal variables that can only be set by *status*, rather than as SKILL variables you can set directly. Refer to "Names of Variables" on page 64 for a discussion of internal variables.

True (non-nil) and False (nil) Conditions

Unlike C or other programming languages that use integers to represent true and false conditions, SKILL uses the nonnumeric special atom *nil* to represent the false condition. The true condition is represented by the special atom *t* or anything other than *nil.*

Relational Operators

The relational operators *lessp* (<), *leqp* (<=), *greaterp* (>), *geqp* (>=), *equal* (==), and *nequal* (!=) operate on numeric arguments and return either *t* or *nil* depending on the results.

Logical Operators

The logical operators *and* (&&), *or* (||), and *null* (!), on the other hand, operate on nonnumeric arguments that represent either the false (*nil*) or true (non-*nil*) conditions.

False/True Conditions Do Not Equal Constants 0 and 1

If you program in C, be especially careful not to interpret the false/true conditions as equivalent to the integer constants 0 and 1.

Testing for Equality and Inequality

You can also use the *equal* (==) and the *nequal* (!=) operators to test for the equality and inequality of nonnumeric atoms.

- Two atoms are equal if they are the same type and have the same value.
- Two lists are equal if they contain exactly the same elements.

Controlling the Order of Evaluation

The binary operators && and || are often used to control the order of evaluation.

The && Operator

The && operator evaluates its first argument and, if the result is *nil*, returns *nil* without evaluating the second argument. If the first argument evaluates to non-*nil*, && proceeds to evaluate the second argument and returns that value as the value of the function.

The || Operator

The || operator also evaluates its first argument to see if the result is non-*nil*. If so, || returns that result as its value and the second argument is not evaluated. If the first argument evaluates to *nil*, || proceeds to evaluate the second argument and returns that value as the value of the function.

Testing Arithmetic Conditions

In addition to the six *infix* relational operators, several arithmetic predicate functions are available for efficient testing of arithmetic conditions. These predicates are listed in the table below.

Arithmetic Predicate Functions

Synopsis	Result
minusp(n)	*t* if n is a negative number, *nil* otherwise
plusp(n)	*t* if n is a positive number, *nil* otherwise
onep(n)	*t* if n is equal to 1, *nil* otherwise
zerop(n)	*t* if n is equal to 0, *nil* otherwise
evenp(x)	*t* if x is an even integer, *nil* otherwise
oddp(x)	*t* if x is an odd integer, *nil* otherwise

Differences Between SKILL and C Syntax

Arithmetic and logical expressions in SKILL are the same as in the C programming language with the following minor differences.

- An exponentiation operator, represented by two asterisks (**), has been added to SKILL.
- The modulo operator “%” has been replaced by the function *mod* so that you do not have to use “%” as a special character for an infrequently used function. Instead of “i % j”, use “mod(i j).”
- The conditional expression operators “?” and “:” are made obsolete by the more general *if/then/else* control construct.
- The indirection operator “*” and the address operator “&” do not have any meaning in SKILL and are therefore not supported.
- The bitwise operators have been augmented by operators implementing the nand (~&), nor (~|), and xnor (~^) functions.
- Logical expressions return either the special atom *nil* or a non-*nil* value (usually the special atom *t*) depending on whether the expression evaluates to false or true, respectively.

SKILL Predicates

The following predicate functions test for a condition.

The atom Function

atom checks if an object is an atom. Atoms are all SKILL objects (except nonempty lists). The special symbol *nil* is both an atom and a list.

```
atom( 'hello )   => t

x = '(a b c)
atom( x )        => nil
atom( nil )       => t
```

The boundp Function

boundp checks if a symbol is bound, that is, has been assigned a value.

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

boundp( 'x )
=> t

boundp( 'y )
=> nil

y = 'x               ; Binds y to the constant x.
boundp( y )
=> t                 ; Returns t because y evaluates to x,
                     ; which is bound.
```

Using Predicates Efficiently

Some predicates are faster than others. For example, the *eq*, *neq, memq,* and *caseq* functions are faster than their close relatives the *equal, nequal, member,* and *case* functions.

The *equal, nequal, member*, and *case* functions compare values while the *eq, neq, memq,* and *caseq* functions test if the objects are the same. That is, they test whether the objects reside in the same location in virtual memory.

These functions result in quicker tests but might require that you alter your application to take advantage of them.

The eq Function

eq checks addresses when testing for equality. The *eq* function returns *t* if both arguments are the same object in virtual memory. You can test for equality between symbols using *eq* more efficiently than using the == operator. The following example illustrates the differences between *equal* (==) and *eq* for lists.

```
list1 = '( 1 2 3 )          => ( 1 2 3 )
list2 = '( 1 2 3 )          => ( 1 2 3 )
list1 == list2              => t
eq( list1 list2 )           => nil

list3 = cons( 0 list1 )     => ( 0 1 2 3 )
list4 = cons( 0 list1 )     => ( 0 1 2 3 )
list3 == list4              => t
eq( cdr( list3 ) list1 )    => t

aList = '( a b c )          => a b c
eq( 'a car( aList ) )       => t
```

The equal Function

equal tests for equality.

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

This test is slower than using *eq* but works for comparing objects other than symbols.

```
x = 'cat
equal( x 'cat )      => t
x == 'dog            => nil; == is the same as equal.
x = "world"
equal( x "world" )  => t
x = '(a b c)
equal( x '(a b c))  => t
```

The neq Function

neq checks if two arguments are *not* equal, and returns *t* if they are not. Any two SKILL expressions are tested to see if they point to the same object.

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

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

The nequal Function

nequal checks if two arguments are *not* logically equivalent and returns *t* if they are not.

```
x = "cow"
nequal( x "cow" )    => nil
nequal( x "dog" )    => t

z = '(1 2 3)
nequal(z z)          => nil
nequal('(1 2 3) z)   => nil
```

The member and memq Functions

These functions test for list membership. *member* tests using *equal* while *memq* uses *eq* and is therefore faster. These functions return a non-*nil* value if the first argument matches a member of the list passed in as the second argument.

```
x = 'c
memq( x '(a b c d))              => (c d)
memq( x '(a b d))                 => nil

x = "c"
member( x '("a" "b" "c" "d"))    => (“c” “d”)
memq('c '(a b c d c d))          => (c d c d)
memq( concat( x ) '(a b c d ))   => (c d)
```

The tailp Function

`tailp` returns the first argument if a list cell `eq` to the first argument is found by `cdr`’ing down the second argument zero or more times, `nil` otherwise. Because `eq` is being used for comparison, the first argument must actually point to a tail list in the second argument for this predicate to return a non-`nil` value.

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

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

Type Predicates

Many predicate functions are available for testing the data type of a data object. The suffix “p” is usually added to the name of a function to indicate that it is a predicate function.

These functions are listed in the table below. *g* (general) can be any data type.

Type Predicates

Function	Value Returned
arrayp(g)	*t* if g is an array, *nil* otherwise
bcdp(g)	*t* if g is a binary function, *nil* otherwise
dtpr(g)	*t* if g is a non-empty list, *nil* otherwise (note that dtpr (*nil*) returns *nil*)
fixp(g)	*t* if g is a fixnum, *nil* otherwise
floatp(g)	*t* if g is a flonum, *nil* otherwise
listp(g)	*t* if g is a list, *nil* otherwise (note that listp(*nil*) returns *t*)
null(g)	*t* if g is *nil*, *nil* otherwise
numberp(g)	*t* if g is a number (that is, fixnum or flonum), *nil* otherwise
otherp(g)	*t* if g is a foreign data pointer, *nil* otherwise
portp(g)	*t* if g is an I/O port, *nil* otherwise
stringp(g)	*t* if g is a string, *nil* otherwise
symbolp(g)	*t* if g is a symbol, *nil* otherwise
symstrp(g)	*t* if g is either a symbol or a string, *nil* otherwise
type(g)	a symbol whose name describes the type of g
typep(g)	same as type(g)

6

Control Structures

Overview information:

Control Functions

The Cadence® SKILL language control functions provide a great deal of functionality familiar to users of a language such as C. These high-level control functions give SKILL additional power over most Lisp languages.

The control functions are also the biggest cause of inefficient code in the SKILL language. One of the inevitabilities of providing so many control structures is that some are more efficient than others and that there is a great deal of overlap between the functions. This means that it is easy for a programmer to choose a structure that works perfectly well for the task in hand, but is not in fact the best structure to use as far as efficiency and (even occasionally) readability are concerned.

A control function is any function that controls the evaluation of expressions given to it as arguments. The order of evaluation can depend on the result of evaluating test conditions, if any, given to the function. In addition to supporting standard control constructs such as *if/ while/for*, SKILL makes it easy for you to define control functions of your own. Because control functions in SKILL correspond to “statements” in conventional languages, this manual sometimes uses the terms interchangeably.

Conditional Functions

Conditional functions test for a condition and perform operations when that condition is found.

There are four conditional functions available to the SKILL programmer: *if*, *when*, *unless*, and *cond*. These each have their own distinct characteristics and uses. Because the four functions carry out very similar tasks, it is very easy for the programmer to choose an inappropriate function. Choose a conditional function according to the following criteria:

if	There are exactly two values to consider, true and false.
when	There are statements to carry out when the test proves true.
unless	There are statements to carry out unless the test proves true.
cond	There is more than one test condition, but only the statements of one test are to be carried out.

The *cond* function is discussed here. For a discussion of the *if, when,* and *unless* functions, refer to “Getting Started” on page 31.

The cond Function

The *cond* function offers multiway branching.

```
cond(
      ( condition1 exp11 exp12 ... )
      ( condition2 exp21 exp22 ... )
      ( condition3 exp31 exp32 ... )
      ( t expN1 expN2 ... )
      ) ; cond
```

The *cond* function

- Sequentially evaluates the conditions in each branch until it finds one that is non-*nil*. It then executes all the expressions in the branch and exits.
- Returns the last value computed in the branch it executes.

The *cond* function is equivalent to

```
if           condition1       exp11   exp12 ...
else if      condition2       exp21   exp22 ...
else if      condition3       exp31   exp32 ...
...
else     expN1expN2 ....
```

For example, this version of the *trClassify* function is equivalent to the one using the *prog* and *return* functions in “The return Function” on page 146.

```
procedure( trClassify( signal )
      cond(
            ( !signal nil )
            ( !numberp( signal ) nil )
            ( signal >= 0 && signal < 3 'weak )
            ( signal >= 3 && signal < 10 'moderate )
            ( signal >= 10 'extreme )
            ( t          'unexpected )
            ) ; cond
      ) ; procedure
```

Iteration Functions

There are two basic iteration functions available in the SKILL language: *while* and *for*. These are both very general functions that have many uses.

The while Function

The *while* function is the more general function because everything that can be done with a *for* can be done with a *while*.

When using the *while* function remember that all parts of the test condition are evaluated on each pass of the loop. This means that if there are parts of the test that do not depend on the contents of the loop, they should be moved outside of the loop. Consider the following code:

```
while( i < length(myList)
      ...
      i++
)
```

If the value of the symbol *myList* does not change within this loop, the value of *length(myList)* is being re-evaluated on each loop for no reason. It would be better to assign the value of *length(myList)* to a variable before starting the *while* loop.

When using a *while* loop, consider whether it would be better to use one of the list iteration and quantifier functions such as *foreach*, *setof*, or *exists*.

The for Function

The main advantage of the *for* function is that it automatically declares the loop variable. This means that the variable does not need to be declared in a local variable section of a structure such as *prog* or *let*. It also means that the variable cannot be used outside the loop, which differs from the case in C. Consider the following code:

```
for( i 1 length(myList)
      evaluateList(i)
)
if( i == 0
      printf("The list was empty!\n")
)
```

The *if* test is incorrect because the variable *i* will be unbound by the time it is executed.

Selection Functions

There are two selection functions in SKILL: *caseq* and *case*. The difference between these functions is the range of values that are allowed within the test conditions.

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

The *caseq* and *case* functions allow lists of elements within the test parts and match if the test value is *eq* or *equal* to one of those elements, as appropriate.

One common fault with the use of the *caseq* function is the misconception that the values in the conditional part of the function are evaluated. Consider the following call to *caseq*:

```
caseq( x
 ('a "a")
 ('b "b")
)
```

The conditional parts of this, *'a* and *'b*, are *not* evaluated, so this code equates to

```
caseq( x
 ((quote a) "a")
 ((quote b) "b")
)
```

That is, if the value of *x* is the symbol *a* or is the symbol *quote*, *caseq* returns the value *a*. This is clearly not what was actually required.

Be careful when using symbols in these selection functions because the symbol *t* indicates the default case and should not therefore be used. For example, consider the case where a function returns one of the values *t*, *nil*, or *indeterminate*.

It might be tempting to write a function such as

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

But this function will not work because the *t* case is the default and always matches. The correct way to write this test is

```
caseq( value
 (nil               printf("Failed.\n"))
 (indeterminate     printf("Indeterminate.\n"))
 (t                 when( eq(value,t)
                        printf("Succeeded.\n")
                    )
      )
) /* caseq */
```

The problem can also be avoided by putting the *t* within parentheses because the default case only matches against a single *t*. This is not recommended because it is an implementation dependency. The SKILL Lint program always warns of dubious uses of the *t* case in a selection function.

Declaring Local Variables with prog

All variables that appear in a SKILL program are global to the whole program unless they are explicitly declared as local variables. You declare local variables using the *prog* control construct, which initializes all its local variables to *nil* upon entry and restores their original values (that is, the values of the variables before the *prog* was executed) upon exit from the *prog*.

A symbol's current value is accessible at any time from anywhere. The SKILL interpreter transparently manages a symbol's value slot as if it were a stack.

- The current value of a symbol is simply the top of the stack.
- Assigning a value to a symbol changes only the top of the stack.

Whenever your program invokes the *prog* function, the system pushes a temporary value onto the value stack of each symbol in the local variable list. When the flow of control exits, the system pops the temporary value off the value stack, restoring the previous value.

The prog Function

The *prog* function allows an explicit loop to be written since *go* is supported within the *prog*. In addition, *prog* allows you to have multiple return points through use of the function *return*. If you are not using either of these two features, *let* is much simpler and faster.

If you need to conditionally exit a collection of SKILL statements, use the *prog* function. A list of local variables and your SKILL statements make up the arguments to the *prog* function.

```
prog( ( local variables ) your SKILL statements )
```

The return Function

Use the *return* function to force the *prog* to immediately return a value skipping over subsequent statements. If you do not call the *return* function, the *prog* expression returns *nil*.

Example: The *trClassify* function returns either *nil, weak, moderate, extreme,* or *unexpected* depending on *signal*. It does not use any local variables.

```
procedure( trClassify( signal )
      prog( ()
            unless( signal return( nil ))
            unless( numberp( signal )return( nil ))
            when( signal >= 0 && signal < 3return( 'weak ))
            when( signal >= 3 && signal < 10return( 'moderate ))
            when( signal >= 10return( 'extreme ))
            return( 'unexpected )
            ) ; prog
      ) ; procedure
```

Use the *prog* function and the *return* function to exit early from a *for* loop. This example finds the first odd integer less than or equal to 10.

```
prog( ( )
      for( i 0 10
            when( oddp( i )
                  return( i )
                  ) ; when
```

```
        ) ; for
    ) ; prog
```

Grouping Functions

Three main functions allow the grouping of statements where only a single statement would otherwise be allowed. These functions are *prog*, *let*, and *progn*. In addition, the *let* and *prog* functions allow for the declaration of local variables. The *prog* function also allows for the use of *return* statements to jump out from within a piece of code and the *go* function, along with labels, to jump around within the code.

When considering whether to use *prog*, *let*, or *progn*, the function with the least extra functionality should be used at all times because the functions are progressively more expensive in terms of run time. Use the functions as follows:

- If local variables and jumps are not needed, use a *progn*.
- If local variables are needed but not jumps, use a *let*.
- Only if jumps are really needed, use a *prog*.

Using prog, return, and let

The *prog* statement should be used only when it is absolutely necessary. Its overuse is one of the biggest causes of inefficiency in all SKILL code. Returning from the middle of a piece of code is not only highly expensive, but can also lead to code that is difficult to read and understand. As with all high level programming languages, the use of *go* (the SKILL ‘goto’ statement) is highly discouraged. There are cases when it is necessary, but these are few.

A programmer usually uses the *prog* form when a certain amount of error checking must be done at the start of a function, with the rest of the function only being carried out if the error checking succeeds. Consider the following piece of code:

```
procedure(check(arg1 arg2)
 prog( ()
          when(illegal_val(arg1)
                  printf("Arg1 in error.\n")
                  return()
            ) /* end when */

            when(illegal_val(arg2)
                  printf("Arg2 in error.\n")
                  return()
            ) /* end when */

            Rest of code ...
 ) /* end prog */
) /* end check */
```

This code is reasonably clear, except that it is easy for someone to miss the *return* statements, and it uses the *prog* form. Consider the following alternative. This code seems to be a longer procedure, but it is clearer, faster, and more maintainable:

```
procedure(check(arg1 arg2)
      cond(
            ( illegal_val(arg1)
                  printf("Arg1 in error.\n")
                  nil
            ) /* check arg1 */
            (illegal_val(arg2)
                  printf("Arg2 in error.\n")
                  nil
            ) /* check arg2 */
            (t
                  Rest of code ...
      )
 ) /* end cond */
) /* end check */
```

A separate function could be called from within the *t* condition, which could expect its arguments to be correct. This would, at the small cost of an extra function call, separate the error checking code completely from the main body of the function, thereby making it even easier for programmer maintaining the code to see exactly what is involved in the function, without having to worry about the peripheral interfaces.

Another common mistake with the use of the *prog* and *let* functions is the initialization of the local variables to *nil*. All local variables in a *prog* or *let* are automatically initialized to *nil*. Remember that the *let* function allows local variables to be initialized within the declaration. This saves both time and space, and, as long as care is taken over the layout of the code, can be no less readable:

```
procedure(test()
      let( ((localvar1 initvalue1)
            (localvar2 initvalue2)
            localvar3                  /* initial value nil */
            )
      Rest of code ...
      ) /* end let */
) /* end procedure */
```

When setting initial values within the declaration, reference cannot be made to other local variables. For example, the following is wrong:

```
procedure(incorrect(list)
 let( ((listHead         car(list))
            (headval       car(listHead)) /* WRONG!!! */
            )
            Rest of code ...
```

Note that the *prog* and *let* functions have different return values.

- The *prog* function returns the value given in a *return* statement or, if it exits without a *return*, returns *nil*.

- The *let* function always returns the value of the last statement.

This means that in converting a *prog* to a *let*, it might be necessary to add an extra *nil* to the end of the function.

Using the progn Function

The *progn* function is a simple means of grouping statements where multiple statements are required, but only one is expected.

An example is the *setof* function, which only allows a single statement in the conditional part.

Remember that there is an overhead in using *progn*. It should only be used where there is more than one statement, and only one statement is allowed.

Using the prog1 and prog2 Functions

Two minor grouping functions that have roughly the same overhead as the *progn* function are *prog1* and *prog2*.

The prog1 Function

prog1 evaluates expressions from left to right and returns the value of the first expression.

```
prog1(
      x = 5
      y = 7 )
=> 5
```

The prog2 Function

prog2 evaluates expressions from left to right and returns the value of the second expression.

```
prog2(
      x = 4
      p = 12
      x = 6 )
=> 12
```

prog1 and *prog2* are often useful when a local variable would otherwise be needed to hold a temporary variable before that variable is returned. These two functions should be used with caution, because they can detract from the readability of the program, and they are generally only useful where otherwise a *let* would be necessary. For example:

```
procedure(main()
      let( (status)
```

```
            initialize()
            /* Main body of program */
            status = analyzeData()
             wrapUp()
/* Return the status */
            status
      ) /* end let */
) /* end main */
```

This code can be more efficiently (but less clearly) implemented using a *prog2*:

```
procedure(main()
     prog2(
            initialize()
            /* Main body of program.
            * The value of this will be returned by prog2.
            */
            nalyzeData()
            wrapUp()
      ) /* end prog2 */
) /* end main */
```

7

I/O and File Handling

Overview information:

File System Interface

All input and output in the Cadence® SKILL language is defined with respect to the UNIX file system. Writing I/O statements in SKILL requires an understanding of files, directories, and paths.

Files

A *file* contains data, usually organized in multiple records, and has several attributes such as name, the date the file was created, the last time it was accessed, access permissions, and so on. A *device* is a file with special attributes.

Directories

A *directory* has a name, just like a file, but it contains a list of other files. Directories can be nested to as many levels as desired. A directory allows related files to be grouped together. Because of the thousands of files that can exist on a single disk, using directories helps to avoid chaos. Most network-wide file systems are dependent on directories.

Directory Paths

Often, there are several directories you want to search in a particular order by specifying a set of directory paths. You can specify a file name in an absolute sense or in a relative sense.

The following description uses *path* as a generic term where either a file name or a directory name can be used. However, because a directory under UNIX is just a special kind of file, file name is often used as a synonym for path.

Absolute Paths

You can specify the path with a slash character (/). When used as the first character of a name, it represents the system root directory. Intermediate levels of directories can use the slash character again as a separator.

Relative Paths

Any path that does not begin with a slash is a relative name.

If the path begins with a tilde followed by a slash (~/), the search path begins in your home directory.

If the tilde is followed by a name, the name is interpreted as a user name. That is, *~jones/file1* specifies a file named *file1* in jones' home directory.

If the path begins with a period and a slash (./), the search begins with the current working directory.

If the file name begins with two periods and a slash (../), the search begins with the parent of the current working directory.

If you are using a function that refers to the SKILL path, refer to the following section.

The SKILL Path

SKILL provides a flexible mechanism for using relative paths. An internal list of paths, referred to as the *SKILL path*, is used in many file-related functions.

Importance of the First Path Character

When a relative path that does not begin with ~ or ./ is given to a function, the paths in the SKILL path are used as directory names and prepended to the given path (with a / separator if needed) to form possible paths. The *setSkillPath* and *getSkillPath* functions access and change this internal SKILL path setup.

Path Order when the Same File Name Exists in Multiple Directories

The order of the paths on the SKILL path is very important when the same file name is in multiple directories. If a file is opened for input or queried for status, all readable directories in the SKILL path are checked, in order, for the given file name. The first one found is taken to be the intended path.

Path Order when a File is Updated or Written for the First Time

The order of the paths is also very important when a file is updated or written for the first time. If you open an output file, all directory paths in the SKILL path are checked, in the order specified, for that file name. If found, the system overwrites the first updateable file in the list. If no updateable file is found, it places a new file of that name in the first writable directory.

Know Your SKILL Path

Having an implicit list of search paths provides a powerful shortcut in many situations, but it can also be a source of possible confusion. When in doubt, double check the current setup of your SKILL path or set it to *nil*.

When you start your system, the SKILL path might be set to a default value. You can use the *setSkillPath* function to make sure it is set up correctly.

Working with the SKILL Path

Setting the Internal SKILL Path (setSkillPath)

setSkillPath sets the internal SKILL path. You can specify the directory paths either as a string, where each alternate path is separated by spaces, or as a list of strings. The system tests the validity of each directory path as it puts the input into standard form.

- If all directory paths exist, it returns *nil*
- If any path does not exist, it returns a list in which each element is an invalid path

The paths on the SKILL path are always searched for in the path order you specify. Even if a path does not exist (and hence appears in the returned list), it remains on the new SKILL path. The use of the SKILL path in other file-related functions can be effectively disabled by calling *setSkillPath* with *nil* as the argument.

```
setSkillPath('("." "~" "~/cpu/test1"))
=> nil                        ; If "~/cpu/test1" exists.
=> ("~/cpu/test1")            ; If "~/cpu/test1" does not exist.
```

The same task can be done with the following call that puts all paths in one string.

```
setSkillPath(". ~ ~/cpu/test1")
```

Finding the Current SKILL Path (getSkillPath)

getSkillPath returns directory paths from the current SKILL path setting. The result is a list where each element is a path component as specified by *setSkillPath*.

```
setSkillPath('("." "~" "~/cpu/test1"))=> nil
getSkillPath()=> ("." "~" "~/cpu/test1")
```

The example below shows how to add a directory to the beginning of your search path (assuming a directory “~/lib”).

```
setSkillPath(cons("~/lib" getSkillPath()))=> nil
getSkillPath()=> ("~/lib" "." "~" "~/cpu/test1")
```

Working with the Installation Path

Finding the Installation Path (getInstallPath)

getInstallPath returns the system installation path (that is, the root directory where the Cadence products are installed in your file system) as a list of a single string, where

- The path is always returned in absolute format
- The result is always a list of one string

```
getInstallPath()=> ("/usr5/cds/4.2")
```

Attaching the Installation Path to a Given Path (prependInstallPath)

prependInstallPath prepends the Cadence installation path to the given path (possibly adding a slash (/) separator if needed) and returns the resulting path as a string. The typical use of this function is to compute one member of a list passed to *setSkillPath*.

```
getInstallPath()
=> ("/usr5/cds/4.2")
```

Assume this is your install path.

```
prependInstallPath( "etc/context" )
=> "/usr5/cds/4.2/etc/context"
```

A slash (/) is added.

```
prependInstallPath( "/bin" )
=> "/usr5/cds/4.2/bin"
```

```
setSkillPath( list("." prependInstallPath("bin")
                prependInstallPath("etc/context")) )
=> nil
```

Finding the Root of the Hierarchy (cdsGetInstPath)

getInstallPath returns the root of the *dfII* hierarchy whereas *cdsGetInstPath* returns the root of the hierarchy. *cdsGetInstPath* is more general and is meant to be used by all *dfII* and non-*dfII* applications. For example:

```
getInstallPath() => ("/usr/mnt/hamilton/9304/tools/dfII")
cdsGetInstPath() => "/usr/mnt/hamilton/9304"
```

Checking File Status

Checking if a File Exists (isFile, isFileName)

isFileName checks if a file exists. The file name can be specified with either an absolute path or a relative path. In the latter case, the current SKILL path is used if it's not `nil`. Only the presence or absence of the name is checked. If found, the name can belong to either a file or a directory. *isFileName* differs from *isFile* in this regard.

```
isFileName("myLib")=> t
```

A directory is just a special kind of file.

```
isFileName("triadc")=> t
isFileName("triad1")=> nil
```

Result if *triad1* is not in the current working directory.

isFile checks if a file exists. `isFile` is identical to `isFileName`, except that directories are not viewed as (regular) files. Uses the current SKILL path for relative paths.

```
isFile( "triadc")=> t
```

Checking if a Path Exists and if it is the Name of a Directory (isDir)

isDir checks if a path exists and if it is the name of a directory. When the path is a relative path, the current SKILL path is used if it's non-`nil`.

```
isDir("myLib")   => t
isDir("triadc")  => nil
```

Assumes *myLib* is a directory and *triadc* is a file under the current working directory and the SKILL path is *nil*.

```
isDir("test")=> nil
```

Result if *test* does not exist.

Checking if You Have Permission to Read a File or List a Directory (isReadable)

isReadable checks if you have permission to read the file or list the directory you specify. Uses the current SKILL path for relative paths.

```
isReadable("./") => t
```

Result if current working directory is readable.

```
isReadable("~/myLib")=> nil
```

Result if *~/myLib* is not readable or does not exist.

Checking for Permission to Write a File or Update a Directory (isWritable)

isWritable checks if you have permission to write a file or update a directory that you specify. It uses the current SKILL path for relative paths.

```
isWritable("/tmp")           => t
isWritable("~/test/out.1")   => nil
```

Result if *out.1* does not exist or there is no write permission to it.

Checking for Permission to Execute a File or Search a Directory (isExecutable)

isExecutable checks if you have permission to execute a file or search a directory. A directory is executable if it allows you to name that directory as part of your UNIX path in searching files. It uses the current SKILL path for relative paths.

```
isExecutable("/bin/ls")      => t
isExecutable("/usr/tmp")     => t
isExecutable("attachFiles")  => nil
```

Result if *attachFiles* does not exist or is not executable.

Determining the Number of Bytes in a File (fileLength)

fileLength determines the number of bytes in a file. A directory is viewed just as a file in this case. `fileLength` uses the current SKILL path if a relative path is given.

```
fileLength("/tmp")            => 1024
```

Return value is system-dependent.

```
fileLength("~/test/out.1")    => 32157
```

This examples assumes the file exists. If the file does not exist, you get an error message, such as

```
*Error* fileLength: no such file or directory - "~/test/out.1"
```

Getting Information About Open Files (numOpenFiles)

numOpenFiles returns the number of files that are open and the maximum number of files that a process can open. The numbers are returned as a two-element list.

```
numOpenFiles()                => (6 64)
```

Result is system-dependent.

```
f = infile("/dev/null")      => port:"/dev/null"
numOpenFiles()                 => (7 64)
```

One more file is open now.

Working with File Offsets (fileTell, fileSeek)

fileTell returns the current offset (from the beginning of the file) in bytes for the file opened on a port.

fileSeek sets the position for the next operation to perform on the file opened on a port. The position is specified in bytes. *fileSeek* takes three arguments. The first two are for port and for offset designated in number of bytes to move forward (or backward with a negative argument). The valid values for the third argument are

0 Offset from the beginning of the file
1 Offset from current position of file pointer
2 Offset from the end of the file.

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

```
0123456789 test xyz
```

```
p = infile("test.data")  => port:"test.data"
fileTell(p)              => 0
for(i 1 10 getc(p))      => t     Skip first 10 characters
fileTell(p)              => 10

fscanf(p "%s" s)         => 1    s = "test" now
fileTell(p)              => 15

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

fileSeek(p 6 1)          => t
fscanf(p "%s" s)         => 1     s = "xyz" now

fileSeek(p -12 2)        => t
fscanf(p "%d" x)         => 1     x = 89 now
```

Working with Directories

Creating a Directory (createDir)

createDir creates a directory. The directory name can be specified with either an absolute or relative path; the SKILL path is used in the latter case. You get an error message if the directory cannot be created because you do not have permission to update the parent directory or a parent directory does not exist.

```
createDir("/usr/tmp/test")   => t
createDir("/usr/tmp/test")   => nil
```

Directory already exists.

Deleting a Directory (deleteDir)

deleteDir deletes a directory. The directory name can be specified with either an absolute or relative path; the SKILL path is used in the latter case. You get an error message if you do not have permission to delete a directory or the directory you want to delete is not empty.

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

If you do not have permission to delete */bin*, you get an error message about permission violation.

```
deleteDir("~")
```

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

Deleting a File (deleteFile)

deleteFile deletes a file. The file name can be specified with either an absolute or relative path; the SKILL path is used in the latter case. If a symbolic link is passed in as the argument, it is the link itself, not the file or directory referenced by the link, that gets removed.

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

If the file exists and is deleted.

```
deleteFile("~/test/out.2")=> nil
```

If the file does not exist.

```
deleteFile("/bin/ls")
```

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

Creating a Unique File Name (makeTempFileName)

makeTempFileName appends a string suffix to the last component of a path template such that the resultant composite string does not duplicate any existing file name. (That is, it checks that the file does not exist; the SKILL path is not used in this checking.)

Successive calls to *makeTempFileName* return different results *only if the first name returned is actually used to create a file* in the same directory before a second call is made.

- The last component of the resultant path is guaranteed to be no more than 14 characters. The example below requests a “file” with 15 characters

```
makeTempFileName("/tmp/123456789123456")
=> "/tmp/12345678a08717"
```

- If the original template has a long last component, it is truncated from the end if needed

```
makeTempFileName("/tmp/123456789.123456")
=> "/tmp/12345678a08717"
```

- Any trailing Xs are removed from the template before the new string suffix is appended

You should follow the convention of placing temporary files in the */tmp* directory on your system.

```
d = makeTempFileName("/tmp/testXXXX") => "/tmp/testa00324"
```

Trailing Xs are removed.

```
createDir(d)                              => t
```

The name is used this time.

```
makeTempFileName("/tmp/test")         => "/tmp/testb00324"
```

A new name is returned this time.

Listing the Names of All Files and Directories (getDirFiles)

getDirFiles lists the names of all files and directories (including . and ..) in a directory. Uses the current SKILL path for relative paths.

```
getDirFiles(car(getInstallPath())) =>
("." ".." "bin" "cdsuser" "etc" "group" "include" "lib" "pvt"
"samples" "share" "test" "tools" "man" "local" )
```

Expanding the Name of a File to its Full Path (simplifyFilename)

simplifyFilename returns the fully expanded name of a file. Tilde expansion is performed, ./ and ../ are compressed, and redundant slashes are removed. Symbolic links are also resolved by default, unless the second (optional) argument *g_dontResolveLinks* is specified to non-nil. If the file you supply is not absolute, the current working directory is prefixed to the returned file name.

```
simplifyFilename("~/test") => "/usr/mnt/user/test"
```

Returns the fully expanded name of *test*, assuming the user's home directory is */usr/mnt/user.*

Getting the Current Working Directory (getWorkingDir)

getWorkingDir returns the current working directory as a string. The result is put into a "~/ prefixed" form if possible by testing for commonality with the current user's home directory. For example, *~/test* is returned in preference to */usr/mnt/user1/test*, assuming that the home directory for *user1* is */usr/mnt/user1* and the current working directory is */usr1/mnt/user1/test*.

```
getWorkingDir() => "~/project/cpu/layout"
```

Changing the Current Working Directory (changeWorkingDir)

Changes the working directory to the name you supply. The name can be specified with either a relative or absolute path. If you supply a relative path, the *cdpath* shell variable is used to search for the directory, not the SKILL path.

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

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

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

```
changeWorkingDir( "/usr5/design/cpu") => t
changeWorkingDir( "test")
```

Signals an error that no such directory exits.

Ports

All input and output in SKILL goes through a data type called a port. A port can be opened either for reading (an input port) or writing (an output port). Ports are analogous to FILE* variables used by the *stdio* library in C. Most implementations of the UNIX operating system impose a strict limit (typically between 30 and 64) on the number of files that can be open at any time.

Your application typically needs to use some of these scarce file descriptors, leaving you with only a few free ports with which to work. You should therefore always close ports that are no longer in use and avoid using an excessive number of ports; you might otherwise run out of ports when your code is moved to a different UNIX machine.

Predefined Ports

Most I/O functions in SKILL accept a port as an optional argument. If a port is not specified, the *piport* and *poport* are used as default ports for input and output respectively. The table below lists the names and the use of the input/output ports predefined in SKILL.

You can redefine the default values, if necessary, but many internal SKILL functions are hard wired to use particular ports and you should be careful not to assign any of them an illegal value.

The *stdin*, *stdout*, and *stderr* ports are also predefined. These ports correspond to the standard input, standard output, and standard error streams available to every UNIX program.

Predefined Input/Output Ports

Name	Usage
piport	Standard input port, analogous to and initialized to stdin in C
poport	Standard output port, analogous to and initialized to stdout in C
errport	Output port for printing error messages, analogous to and initialized to stderr in C
ptport	Output port for printing trace information; initialized to stderr in C

Opening and Closing Ports

The following functions work with opening and closing ports. Both of the file opening functions use the SKILL path variable.

Opening an Input Port to Read a File (infile)

infile opens an input port ready to read a file you specify. The file name can be specified with either an absolute path or a relative path. In the latter case, the current SKILL path is used if it's not *nil.*

```
infile("~/test/input.il")=> port:"~/test/input.il"
```

Result if such a file exists and is readable.

```
infile("myFile") => nil
```

Result if *myFile* does not exist according to the SKILL path or exists but is not readable.

Opening an Output Port to Write a File (outfile)

`outfile` opens an output port ready to write to the file you specify. The file name can be specified with either an absolute path or a relative path.

- If a relative path is given and the current SKILL path setting is not *nil*, all directory paths from the SKILL path are checked, in the order specified, for that file name
- If found, the system overwrites the first updateable file in the list
- If no updateable file is found, it places a new file of that name in the first writable directory

```
p = outfile("~/test/out.il" "w") => port:"~/test/out.il"
```

Returns the name of the output port ready to write to the file.

```
outfile("/bin/ls")                => nil
```

Returns *nil* if the file cannot be opened for writing.

Writing Out All Characters in the Output Buffer of a Port (drain)

drain writes out all characters that are in the output buffer of a port. *drain* is analogous to a combination of *fflush* and *fsync* in C. You get an error message if the port to drain is an input port or has been closed.

```
drain()          => t
drain(poport)    => t
```

Draining, Closing, and Freeing a Port (close)

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

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

Output

SKILL provides functions for unformatted and formatted output.

Unformatted Output

Printing the Value of an Expression in the Default Format (print, println)

print prints the value of an expression using the default format for the data type of the value (for example, strings are enclosed in double quotes).

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

Prints to *poport* and returns *nil*.

println prints the value of an expression just like `print`, but a newline character is automatically printed after printing the input value. `println` flushes the output port after printing each newline character.

```
println("Hello World!")
"Hello World!"
=> nil
```

Printing a Newline (\n) Character (newline)

Prints a newline (\n) character. If you do not specify the output port, it defaults to *poport*, the standard output port. The `newline` function flushes the output port after printing each newline character.

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

Printing a List with a Limited Number of Elements and Levels of Nesting (printlev)

printlev prints a list with a limited number of elements and levels of nesting. Lists are normally printed in their entirety no matter how many elements they have or how deeply nested they are.

Applications can, however, set upper limits on the number of elements and the levels of nesting shown when printing lists using *printlev*. These limits are sometimes necessary to control the volume of interactive output because the SKILL top-level automatically prints the results of expression evaluation. Limits can also protect against infinite looping on circular lists possibly created when novices use the destructive list modification functions, such as *rplaca* or *rplacd*, without a thorough understanding of how they work. *printlev* uses the following syntax:

```
printlev( g_value x_level x_length [p_outputPort] ) => nil
```

Two integer variables, print length and print level (specified by *x_length* and *x_level*), control the maximum number of elements and the levels of nesting that are printed. List elements beyond the maximum specified by print length are abbreviated as "..." and lists nested deeper than the maximum level specified by print level are abbreviated as "&." Both print length and print level are initialized to *nil* (meaning no limits are imposed) by SKILL, but each application can set its own limits.

The *printlev* function is identical to *print* except that it takes two additional arguments specifying the maximum level and length to use in printing the expression.

```
List = '(1 2 (3 (4 (5))) 6)
printlev(List 100 2)
(1 2 ...)
=> nil

printlev(List 3 100)
(1 2 (3 (4 &)) 6)
=> nil

printlev(List 3 3 p)
(1 2 (3 (4 &)) ...)
=> nil
```

Assumes port p exists. Prints to port p.

Formatted Output

You can precede format characters with a field width specification. For example, %5d prints an integer in a field that is 5 columns wide. If the field width begins with the digit "0", zero padding is done instead of blank padding. For the format characters *f* and *e*, the width specification can be followed by a period "." and an integer specifying the precision, that is, the number of digits to print after the decimal point.

Output is right justified within a field by default unless an optional minus sign "-" immediately follows the "%" character, which will then be left justified. To print a percent sign, you must use

two percent signs in succession. You must explicitly put “\n” in your format string to print a newline character and “\t” for a tab.

Common Output Format Specifications

Format Specification	Type(s) of Argument	Prints
%d	fixnum	Integer in decimal radix
%o	fixnum	Integer in octal
%x	fixnum	Integer in hexadecimal
%f	flonum	Floating-point number in the style [-]ddd.ddd
%e	flonum	Floating-point number in the style [-]d.ddde[-]ddd
%g	flonum	Floating-point number in style f or e, whichever gives full precision in minimum space
%s	string, symbol	Prints out a string (without quotes) or the print name of a symbol
%c	string, symbol	The first character
%n	fixnum, flonum	Number
%L	list	Default format for the data type
%P	list	Point
%B	list	Box

For formatted output, SKILL makes available the standard C *stdio* library routines *printf*, *fprintf*, and *sprintf*. SKILL provides a robust interface to these routines. Below is a brief description of each routine in the context of the SKILL runtime environment. If more detailed descriptions are needed for these functions, consult your C programming manual.

Writing Formatted Output to Ooport (printf)

printf writes formatted output to *poport*. Optional arguments following the format string are printed according to their corresponding format specifications. *printf* is identical to *fprintf* except that it does not take a port argument and the output is written to *poport*.

```
x = 197.9687
printf("The test measures %10.2f.\n" x)
```

Prints the following line to *poport* and returns *t*.

```
The test measures            197.97. => t
```

Writing Formatted Output to a Port (fprintf)

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

```
x = 197.9687
fprintf(p "The test measures %10.2f.\n" x)
```

Prints the following line to port p and returns *t*.

```
The test measures            197.97. => t
```

Writing Formatted Output to a String Variable (sprintf)

sprintf formats the output and puts the resultant string into the variable given as the first argument. If *nil* is specified as the first argument, no assignment is made. The formatted string is returned. Because of internal buffering in *sprintf*, there is a limit to how many characters *sprintf* can handle, but the limit is large enough (8192 characters) that it should not present any problem.

```
sprintf(s "Memorize %s number %d!" "transaction" 5)
=> "Memorize transaction number 5!"

s
=> "Memorize transaction number 5!"

p = outfile(sprintf(nil "test%d.out" 10))
=> port:"test10.out"
```

Pretty Printing

SKILL provides functions for “pretty printing” function definitions and long data lists with proper indenting to make them more readable and easier to manipulate in text form.

You need the SKILL Development Environment license to pretty print function definitions using the *pp* SKILL function below.

Pretty Printing a Function Definition (pp)

pp pretty prints a function definition. The function must be a readable interpreted function. (Binary functions cannot be pretty printed.) Each function definition is printed so it can be read back into SKILL. *pp* does not evaluate its first argument but does evaluate the second argument, if given.

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

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

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

Pretty Printing Long Data Lists (pprint)

pprint is identical to *print* except that it tries to pretty print the value whenever possible. (*pprint* does not work the same as the *pp* function. *pp* is an *nlambda* and only takes a function name whereas *pprint* is a lambda and takes an arbitrary SKILL object.)

The *pprint* function is useful, for example, when printing out a long list where *print* simply prints the list on one (possibly huge) line but *pprint* limits the output on a single line and produces a multiple-line printout if necessary. This multiple-line printout makes later input much easier.

```
pprint '(1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j k)
(1 2 3 4 5
      6 7 8 9 0
      a b c d e
      f g h i j
      k
)
=> nil
```

Input

When describing input functions, this manual often uses the term “form” to refer to a logical unit of input. A form can be an expression, such as source code or a data list that can span multiple input lines. Input functions such as *lineread* read in one input line at a time but continue reading if they do not find a complete form at the end of a line.

You can think of input forms and how the SKILL functions work with them in the following ways.

Input Functions

Input Source	SKILL Evaluated	SKILL Not Evaluated	Application-Specific Formats
File	load loadi	lineread	infile gets getc fscanf close
String	evalstring loadstring errsetstring	linereadstring	instring gets getc fscanf close

SKILL forms read from a file are either evaluated or not evaluated. Input strings can have an application-specific syntax of their own, such as a netlist syntax. It is the programmer's responsibility to open a port, understand the application-specific syntax, process the input, and then close the port.

Reading and Evaluating SKILL Formats

Reading and Evaluating an Expression Stored in a String (evalstring)

evalstring reads and evaluates an expression stored in a string. The resulting value is returned. Notice that *evalstring* does not allow the outermost set of parentheses to be omitted, as in the top level. Refer to the "Top Levels" on page 220 for a discussion of the top level.

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

evalstring("car '(1 2 3)")
```

Signals that *car* is an unbound variable.

Opening a String and Executing its Expressions (loadstring)

loadstring opens a string for reading, then parses and executes expressions stored in the string just as *load* does in loading a file. *loadstring* is different from *evalstring* in two ways. *loadstring*

- Uses *lineread* mode
- Always returns *t* if it evaluates successfully

```
loadstring "1+2"                        => t
loadstring "procedure( f(n) x=x+n )"  => t
loadstring "x=10\n f 20\n f 30"       => t
x                                       => 60
```

Reading and Evaluating an Expression then Checking for Errors (errsetstring)

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

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

Returns *nil* because an error occurred.

```
errsetstring("1+'a" t)        => nil
```

```
Prints out an error message:
*Error* plus: can't handle (1 + a)
```

Loading Files (load, loadi)

load opens a file, repeatedly calls *lineread* to read in the file, and immediately evaluates each form after it is read in. It closes the file when end of file is reached. Unless errors are discovered, the file is read in quietly. If *load* is interrupted by pressing Control-c, the function skips the rest of the file being loaded.

SKILL has an autoload feature that allows applications to load functions into SKILL on demand. If a function being executed is undefined, SKILL checks if the name of the function (a symbol) has a property called *autoload* attached to it. If the property exists, its value, which must be either a string or a function call that returns a string, is used as the name of a file to load. The file should contain a definition for the function that triggered the autoload. Execution proceeds normally after the function is defined. The whole autoload sequence is functionally transparent. Refer to "Delivering Products" on page 225

```
load( "testfns.il")                ; Load file testfns.il
fn.autoload = "myfunc.il"          ; Declares an autoload property.
fn(1)
```

fn is undefined at this point, so this call triggers an autoload of *myfunc.il*, which contains the definition of *fn*.

```
fn(2)              ; fn is now defined and executes normally.
```

loadi is identical to *load*, except that *loadi* ignores errors encountered during the load, prints an error message, and then continues loading.

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

Loads the *testfns.il* file.

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

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

Reading but Not Evaluating SKILL Formats

Parsing the Next Line in the Input Pport into a List (lineread)

lineread parses the next line in the input port into a list that you can further manipulate. It is used by the interpreter's top level to read in all input and understands only SKILL syntax.

Only one line of input is read in unless there are still open parentheses pending at the end of the first line, or binary *infix* operators whose right-hand argument has not yet been supplied, in which case additional input lines are read until all open parentheses have been closed and all binary *infix* operators satisfied. The symbol *t* is returned if *lineread* reads a blank input line and *nil* is returned at the end of the input file.

```
lineread(piport)          ; Reads in the next input expression
f 1 2 +                   ; First input line of the file being read
3                         ; Second input line
=> f (1 (2 + 3))

lineread(piport)
f(a b c)                  ; Another input line of the file
=> ((f a b c))            ; Returns a list of input objects
```

Reading a String into a List (linereadstring)

linereadstring executes *lineread* on a string and returns the form read in. Anything after the first form is ignored.

```
linereadstring "abc"                    => (abc)
linereadstring "f a b c"                => (f a b c)
linereadstring "x + y"                  => ((x + y))
linereadstring "f a b c\n g 1 2 3"      => (f a b c)
```

In the last example, only the first form is read in.

Reading Application-Specific Formats

The following input functions are helpful when you must read input from a file that was not written in SKILL-compatible format.

Reading Formatted Input (fscanf)

fscanf reads the input from a port according to format specifications in a format string. The results are stored in corresponding variables in the call. *fscanf* can be considered the inverse of the *fprintf* output function. *fscanf* returns the number of input items it successfully matches with its format string. It returns *nil* if it encounters an end of file.

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

The input formats accepted by *fscanf* are summarized below.

Common Input Format Specifications

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

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

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

Assume there is a file *testcase* with one line:

```
hello 2 3 world
```

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

Reading a Line and Storing it in a Variable (gets)

gets reads a line from the input port and stores it as a string in a variable. The string is also returned as the value of *gets*. The terminating newline character of the line becomes the last

character in the string. *gets* returns *nil* if EOF is encountered and the variable maintains its last value. Assume the *test1.data* file has the following first two lines:

```
#This is the data for test1
0001 1100 1011 0111
```

```
p = infile("test1.data") => port:"test1.data"
gets(s p)                => "#This is the data for test1\n"
gets(s p)                => "0001 1100 1011 0111\n"
s                          => "0001 1100 1011 0111\n"
```

Reading and Returning a Single Character from an Input Port (getc)

getc reads a single character from the input port and returns it as the value of *getc*. If the character returned is a non-printable character, its octal value is stored as a symbol. If you are familiar with C, you should note that the *getc* and *getchar* SKILL functions are totally unrelated. *getc* returns *nil* if EOF is encountered.

The input port arguments for both *gets* and *getc* are optional. If the port is not given, the functions take their input from *piport*. In the following example assume the file *test1.data* has its first line read as:

```
#This is the data for test1
```

```
p = infile("test1.data") => port:"test1.data"
getc(p)                  => \#
getc(p)                  => T
getc(p)                    => h
```

Reading Application-Specific Formats from Strings

In addition to being able to accept input from the terminal and from text files, SKILL can also take its input directly from strings. Some applications store programs internally as strings and then parse the strings into their corresponding internal SKILL representations as needed.

Because parsing is a relatively expensive operation, you should avoid calling any of the following functions repeatedly on the same string. It is a good practice to convert each string into its internal SKILL representation before using it more than once.

Opening a String for Reading (instring)

Opens a string for reading just as *infile* opens a file. An input port that can be used to read the string is returned.

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

```
b                               => "World!"
close(p)                          => t
```

Always remember to close the port when you are done.

System-Related Functions

Various SKILL functions are available to interact with and query the system environment.

Executing UNIX Commands

From within SKILL, you can execute individual UNIX commands or invoke the *sh* or *csh* UNIX shell.

Starting the UNIX Bourne-Shell (sh, shell)

Starts the UNIX Bourne-shell *sh* as a child process to execute a command string. If the *sh* function is called with no arguments, an interactive UNIX shell is invoked that prompts you for UNIX command input (available only in nongraphic applications).

```
sh( "rm /tmp/junk")
```

Removes the *junk* file from the */tmp* directory and returns *t* if it is removed successfully.

Starting the UNIX C-Shell (csh)

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

```
csh( "mkdir ~/tmp" )
```

Creates a directory called *tmp* in your home directory.

System Environment

The following functions find and compare the current time, retrieve the version number of the software you are using, and determine the value of a UNIX environment variable.

Getting the Current Time (getCurrentTime)

getCurrentTime returns the current time in the form of a string. The format of the string is *month day hour:minute:second year*.

```
getCurrentTime( ) => "Jan 26 18:15:18 1993"
```

Comparing Times (compareTime)

compareTime compares two string arguments, representing a clock-calendar time. The format of the string is *month day hour:minute:second year*. The units are seconds.

```
compareTime( "Apr 8 4:21:39 1991" "Apr 16 3:24:36 1991")
=> -687777.
```

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

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

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

Getting the Current Version Number of Cadence Software (getVersion)

Returns the version number of software you are using.

```
getVersion()
=> "cds3 version 4.2.2 Fri Jan 26 20:40:28 PST 1993"
```

Getting the Value of a UNIX Environment Variable (getShellEnvVar)

getShellEnvVar returns the value of a UNIX environment variable, if it has been set.

```
getShellEnvVar("SHELL") => "/bin/csh"
```

Returns the current value of the *SHELL* environment variable.

Setting a UNIX Environment Variable (setShellEnvVar)

setShellEnvVar sets the value of a UNIX environment variable to a new value.

```
setShellEnvVar("PWD=/tmp") => t
```

Sets the parent working directory to the */tmp* directory .

```
getShellEnvVar("PWD")=> "/tmp"
```

Gets the parent working directory.

8

Advanced List Operations

Overview information:

Conceptual Background

"Getting Started" on page 31 introduces you to building Cadence® SKILL language lists. This chapter fills you in on more of the details of constructing lists.

Understanding how lists are stored in virtual memory helps you better understand SKILL functions, such as *car* and *cdr*, that manipulate lists and the issues behind building large lists efficiently.

How Lists Are Stored in Virtual Memory

SKILL functions that manipulate lists and symbols are actually dealing with memory pointers. When you assign a list to a variable, the variable is internally assigned a pointer to the head of the list. When a list is taken apart by functions such as *car* and *cdr*, only pointers to various parts of the list are returned and no new list cells are created.

SKILL suppresses your awareness of pointers by how it displays lists and symbols. In general, when SKILL displays a supported data type, it uses a characteristic syntax to suppress irrelevant detail and focus on the essentials of the data. This syntax has implications for the *list* and *symbol* data types. Instead of displaying memory addresses, SKILL displays

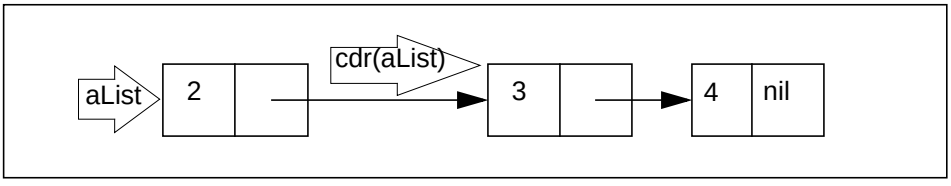
- The elements of a list surrounded by parentheses
- The name of a symbol

A SKILL List as a List Cell

SKILL represents a list by means of a *list cell.* A list cell occupies two locations in virtual memory.

- The first location holds a reference to the first element in the list.
- The second location holds a reference to the tail of the list, that is, another list cell or *nil*.

The expression *aList = '(2 3 4)* allocates the following three list cells.



The *car* function returns the contents of the first location of a list cell.

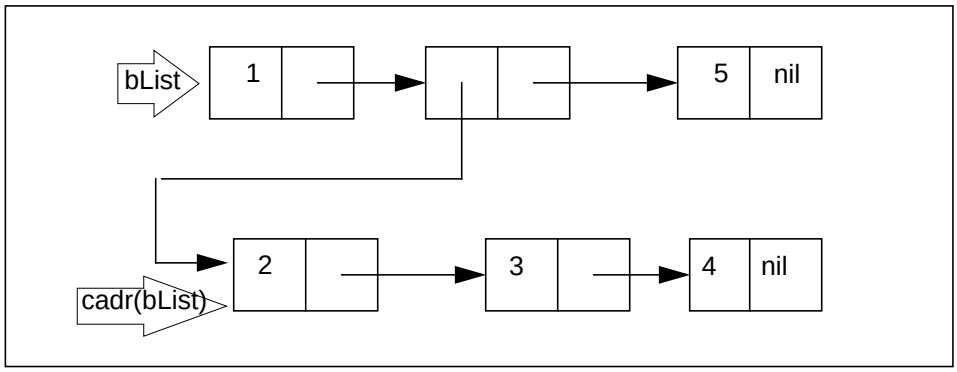
```
car( aList) => 2
```

The *cdr* function returns the contents of the second location of a list cell.

```
cdr( aList) => (3 4)
```

Lists Containing Sublists

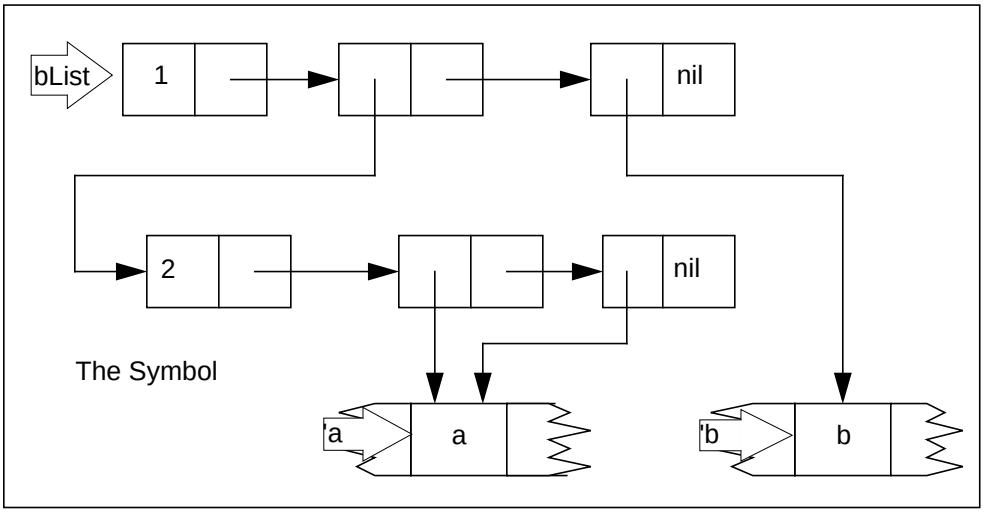
The expression *bList = '(1 (2 3 4) 5)* allocates the following list cells.




```
cadr(bList) => (2 3 4)
```

Lists Containing Symbols

The expression *bList = '(1 (2 a a) b)* allocates the following list cells.



Internally the expression *'a* returns the pointer to the symbol *a* in the symbol table.

Destructive versus Non-Destructive Operations

Non-Destructive Operations

The term *non-destructive modification* refers to any operation that allocates a copy of a list that reflects the desired alteration. The original list is not altered. It is your responsibility to update any variables that need to reflect the operation.

Such operations are generally easier for you to implement than destructive operations that do alter the original list. The disadvantage is that making an altered copy of the original list might be significantly time-consuming.

Destructive Operations

The term *destructive list modification* refers to any operation that alters either the *car* or *cdr* of a list cell. Destructive modification functions do not need to create new list structures. They are therefore considerably faster than equivalent nondestructive modification functions.

Depending on the operation, any variable referring to the original list can be affected. Many subtle problems can arise when these functions are used without a thorough understanding of the implications.

You should only use the destructive modification functions described in this chapter with a very good understanding of how the SKILL language represents lists in virtual memory.

Summary of List Operations

The following table summarizes the list operations that are discussed in this chapter. Use the destructive modification functions with great care.

List Operations

Operation	Function	Non-destructive	Destructive
Altering List Cells	rplaca, rplacd		x
Accessing a List	nthelem, nthcdr, last	x	

List Operations

Operation	Function	Non-destructive	Destructive
Building a List	cons, ncons, xcons, append1	x	
	tconc, nconc, lconc		x
Reorganizing a List	reverse	x	
	sort, sortcar		x
Removing Elements	remove, remq	x	
	remd, remdq		x
Searching Lists	member, memq, exists	x	
Filtering Lists	setof	x	
Substituting	subst	x	
Traversal	mapc, map, mapcar, maplist, mapcan	x	
Traversal	mapcan		x

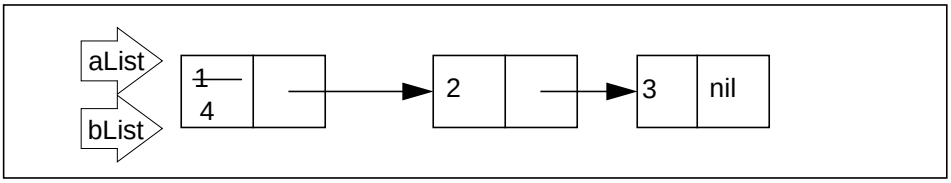
Altering List Cells

The most fundamental destructive operations concern altering a list cell. You can change either the *car* cell or the *cdr* cell. *rplac*a and *rplacd* are destructive operations.

The rplaca Function

Use the *rplaca* function to replace the first element of a list.

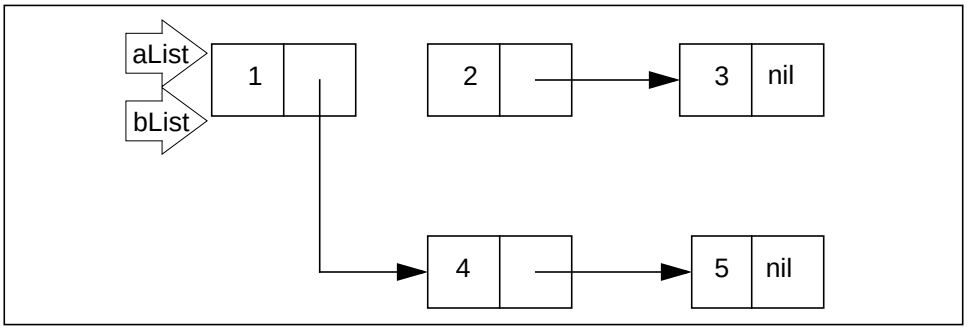
```
aList = '( 1 2 3) => ( 1 2 3 )
bList = rplaca( aList 4 ) => ( 4 2 3 )
aList => ( 4 2 3 )
eq( aList bList ) => t
```



The rplacd Function

Use the *rplacd* function to replace the tail of a list.

```
aList = '( 1 2 3 ) => ( 1 2 3 )
bList = rplacd( aList '( 4 5 ) ) => ( 1 4 5 )
aList => ( 1 4 5 )
eq( aList bList ) => t
```



Notice that the *rplacd* function returns a list with the desired modifications. An important point to remember about destructive operations is that the modified list is literally the same list in virtual memory as the original list. To verify this fact, use the *eq* function, which returns *t* if both arguments are the same object in virtual memory.

Accessing Lists

The following functions are convenient variations and extensions of the *cdr* and *nth* functions introduced in “Getting Started” on page 31.

Selecting an Indexed Element from a List (nthelem)

nthelem returns an indexed element of a list, assuming a one-based index. Thus *nthelem(1 l_list)* is the same as *car(l_list)*.

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

Applying cdr to a List a Given Number of Times (nthcdr)

You supply the iteration count and the list of elements.

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

Getting the Last List Cell in a List (last)

last returns the last list cell in a list. The *car* of the last list cell is the last element in the list. The *cdr* of the last list cell is *nil*.

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

Building Lists Efficiently

To build lists efficiently, you must understand how lists are constructed. Using a function like *append* involves searching for the end of a list, which can be unacceptably slow for large lists.

Adding Elements to the Front of a List (cons, xcons)

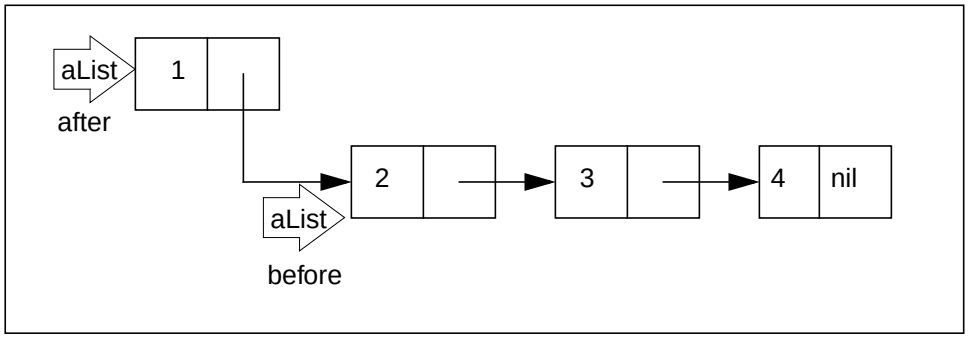
If order is unimportant, the easiest way to build a list is to repeatedly call *cons* to add new elements at the head of the list.

The *cons* function allocates a new list cell consisting of two memory locations. It stores its first argument in the first location and its second argument in the second location.

The *cons* function returns a list whose first element is the one you supplied (1 below) and whose *cdr* is the list you supplied (*aList* below). The expressions

```
aList = '( 2 3 4 )
aList = cons( 1 aList ) => (1 2 3 4 )
```

allocate the following.



xcons accepts the same arguments as the *cons* function, but in reverse order. *xcons* adds an element to the beginning of a list, which can be *nil*.

```
xcons( '( b c ) 'a ) => ( a b c )
```

Building a List with a Given Element (ncons)

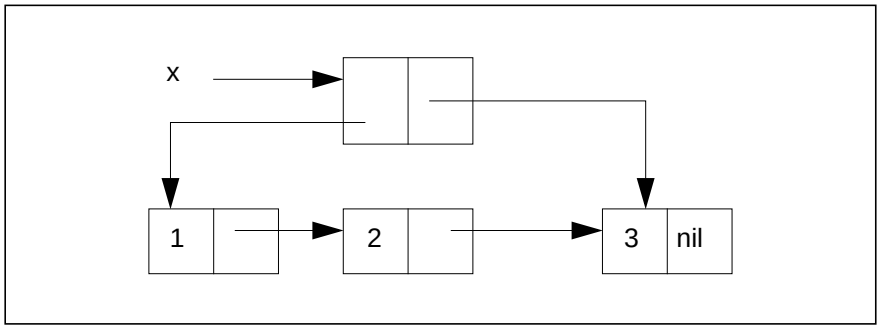
ncons builds a list by adding the *element* you supply to the beginning of an empty list. It is equivalent to *cons(g_element nil)*.

```
ncons( 'a )      => ( a )         ;;; equivalent to cons( 'a nil )
z = '( 1 2 3 )
ncons( z )      => ( ( 1 2 3 ) ) ;;; equivalent to cons( z nil )
```

Adding Elements to the End of a List (tconc)

Because lists are singly linked in only one direction, searching for the end of a list typically requires the traversal of every list cell in the list, which can be quite slow with a long list containing many list cells. This long traversal poses a problem when you want to build a list by adding elements at the end of a list. If a list must be built by adding new elements at the end of the list, the most efficient way is to use *tconc*.

The *tconc* function creates a list cell (known as a *tconc* structure) whose *car* points to the head of the list being built and whose *cdr* points to the last element of the list.



The *tconc* structure allows subsequent calls to *tconc* to find the end of a list instantly without having to traverse the entire list. For this reason, call *tconc* once to initialize a special list cell and pass this special list cell to subsequent calls on *tconc*. Finally, to obtain the actual value of the list you have been building, take the *car* of this special list cell. The typical steps required to use *tconc* are as follows:

1. Create the *tconc* structure by calling *tconc* with *nil* as its first argument and the first element of the list being built as the second argument.

   ```
   x = tconc(nil 1 )
   ```

2. Repeatedly call *tconc* with other elements to be added to the end of the list, each time giving the *tconc* structure as the first argument to *tconc*. There is no need to assign the value returned by *tconc* to a variable because *tconc* modifies the *tconc* structure. For example:

   ```
   tconc(x 2 ), tconc(x 3 ) ...
   ```

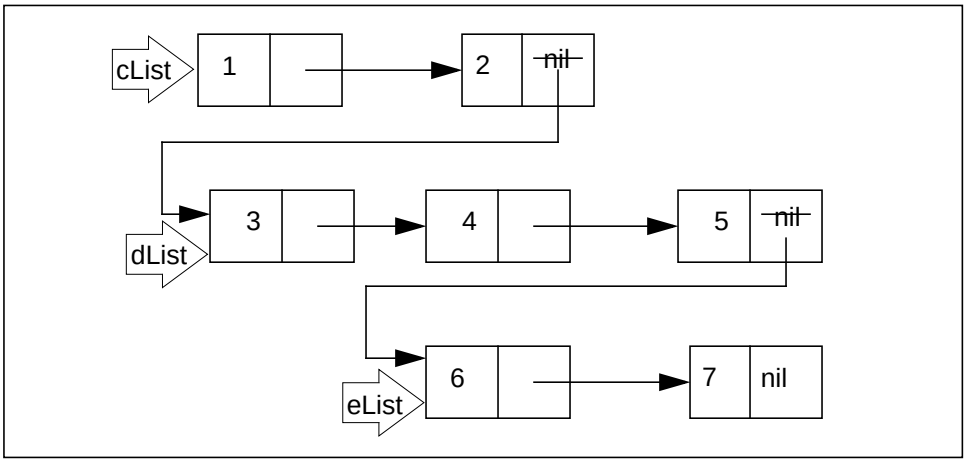
3. After the list has been built, take the *car* of the *tconc* structure to get the actual value of the list being built. For example:

```
x = car(x)
```

Appending Lists

The nconc Function

Use the *nconc* function to quickly append lists destructively. The *nconc* function takes two or more lists as arguments. Only the last argument list is unaltered.




```
cList = '( 1 2 )
dList = '( 3 4 5 )
eList = '( 6 7 )
nconc( cList dList eList ) => ( 1 2 3 4 5 6 7 )
cList => ( 1 2 3 4 5 6 7 )
dList => ( 3 4 5 6 7 )
eList => ( 6 7 )
```

Use the *apply* function and the *nconc* function to append a list of lists.

```
apply( 'nconc '( ( 1 2 ) ( 3 4 5 ) ( 6 7 )) )
=> (1 2 3 4 5 6 7 )
```

The lconc Function

lconc uses a *tconc* structure to efficiently splice lists to the end of lists. The *tconc* structure must initially be created using the *tconc* function. See the example below.

```
x = tconc(nil 1)            ; x is initialized ((1) 1)
lconc(x '(2 3 4))           ; x is now ((1 2 3 4) 4)
lconc(x nil)                ; Nothing is added to x.
```

```
lconc(x '(5))                ; x is now ((1 2 3 4 5) 5)
x = car( x )                 ; x is now (1 2 3 4 5)
```

Reorganizing a List

SKILL provides several functions that reorganize a list. Sometimes the most efficient way to build a list is to reverse it or sort it after you have built it incrementally with the *cons* function. These functions change the sequence of the top-level elements of your list.

Reversing a List

The following is a non-destructive operation.

Reversing the Order of Elements in a List (reverse)

reverse returns the top-level elements of a list in reverse order.

```
aList = '( 1 2 3 )
aList = reverse( aList ) => ( 3 2 1 )
anotherList = '( 1 2 ( 3 4 5 ) 6 )
reverse( anotherList ) => ( 6 ( 3 4 5 ) 2 1 )
```

Although *reverse(anotherList)* returns the list in reverse order, the value of *anotherList*, the original list, is not modified. It is your responsibility to update any variables that you want to reflect this reversal.

```
anotherList => ( 1 2 ( 3 4 5 ) 6 )
```

Sorting Lists

The following functions are helpful when you must sort lists according to various criteria and locate elements within lists. They are destructive operations.

The sort Function

The syntax statement for the *sort* function is

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

sort sorts a list of objects (*l_data*) according to the *sort* function (*u_comparefn*) you supply. *u_comparefn(g_x g_y)* returns non-*nil* if *g_x* can precede *g_y* in sorted order and *nil* if *g_y* must precede *g_x*. If *u_comparefn* is *nil*, alphabetical order is used. The algorithm in *sort* is based on recursive merge sort.

```
sort( '(4 3 2 1) 'lessp )    => (1 2 3 4)
sort( '(d b c a) 'alphalessp) => (a b c d)
```

The sortcar Function

sortcar is similar to *sort* except that only the *car* of each element in a list is used for comparison by the sort function.

```
sortcar( '((4 four) (3 three) (2 two)) 'lessp )
=> ((2 two) (3 three) (4 four)
sortcar( '((d 4) (b 2) (c 3) (a 1)) nil )
=> ((a 1) (b 2) (c 3) (d 4))
```

The list is modified in place and no new storage is allocated. Pointers previously pointing to the list might not be pointing at the head of the sorted list.

Searching Lists

SKILL provides several functions for locating elements within a list that satisfy a criterion. The most basic criterion is equality, which in SKILL can mean either *value equality* or *memory address equality*.

The member Function

The *member* function is briefly discussed in “Getting Started” on page 31. It uses the *equal* function as the basis for finding a top-level element in a list. The *member* function

- Returns *nil* if the element is not equal to any top-level element in the list.
- Returns the first tail of the list that starts with the element.

Some examples include

```
member( 3 '( 2 3 4 3 5 )) => (3 4 3 5)
member( 6 '( 2 3 4 3 5 )) => nil
```

The *member* function resembles the *cdr* function in that it internally returns a pointer to a list cell. The *car* of the list cell is equal to the element. You can use the *member* function with the *rplaca* function to destructively substitute one element for the first top-level occurrence of another in a list. For example, find the first occurrence of 3 and replace it with 6:

```
rplaca(
      member( 3 '( 2 3 4 3 5 ))
      6 )
```

SKILL provides a non-destructive *subst* function for substituting an element at all levels of a list. See “Substituting Elements” on page 191.

The memq Function

The *memq* function is the same as the *member* function except that it uses the *eq* function for finding the element. Because the *eq* function is more efficient than the *equal* function, use *memq* whenever possible based on the nature of the data. For example, if the list to search contains only symbols, then using the *memq* function is more efficient than using the *member* function.

The exists Function

The *exists* function can use an application-specific testing function to locate the first occurrence of an element in a list. The *exists* function generalizes the *member* and *memq* functions, which locate the first occurrence of an element in a list based on equality.

```
exists( x '( 2 4 7 8 ) oddp( x ) ) => ( 7 8 )
exists( x '( 2 4 6 8 ) evenp( x ) ) => ( 2 4 6 8 )

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

Copying Lists

Sometimes it is more efficient to apply a destructive operation to a copy of a list than it is to apply a non-destructive operation. First determine whether a shallow copy of only the top-level elements is sufficient.

The copy Function

copy returns a copy of a list. *copy* only duplicates the top-level list cells. All lower-level objects are still shared. You should consider making a copy of any list before using a destructive modification on the list.

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

z and x have the same value.

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

z and x are not the same list.

Copying a List Hierarchically

The following function recursively copies a list.

```
procedure( trDeepCopy( arg )
      cond(
            ( !arg nil )
            ( listp( arg );;; argument is a list
                  cons(
                        trDeepCopy( car( arg ) )
                        trDeepCopy( cdr( arg ) )
                        )
                  )
            ( t arg ) ;;; return the argument itself
            ) ; cond
      ) ; procedure
```

Filtering Lists

Many list operations can be abstractly considered as making a filtered copy of a list. The filter can be any function that accepts a single argument. If the filter function returns non-*nil,* the element is included in the new list. If the filter function returns *nil*, the element is excluded from the new list.

The setof Function

setof makes a filtered copy of the top-level elements of a list, including all elements that satisfy a given criteria. For example, contrast the following two approaches to computing the intersection of two lists.

- One way to proceed is to use the *cons* function as follows:

```
procedure( trIntersect( list1 list2 )
      let( ( result )
            foreach( element1 list1
                  when( member( element1 list2 )
                        result = cons( element1 result )
                        ) ; when
                  ) ; foreach
            result
            ) ; let
      ) ; procedure
```

- The more efficient way is as follows:

```
procedure( trIntersect( list1 list2 )
      setof(
            element list1
            member( element list2 ) )
      ) ; procedure
```

The criteria is used to decide whether to include each element of the list. The copied element is not transformed.

Removing Elements from a List

SKILL has several functions that remove all top-level occurrences of an element from a list.

The Removal Function

	Non-destructive	Destructive
Uses equal	remove	remd
Uses eq	remq	remdq

Non-Destructive Operations

The remove Function

remove returns a copy of an argument with all top-level elements equal to a given SKILL object removed. The *equal* function, which implements the = operator, is used to test for equality.

```
aList = '( 1 2 3 4 5 )
remove( 3 aList ) => ( 1 2 4 5 )
aList => ( 1 2 3 4 5 )
```

It is your responsibility to make the appropriate assignment so that *aList* reflects the removal.

```
aList = remove( 3 aList )
```

The element to remove can itself be a list.

```
remove( '( 1 2 ) '( 1 ( 1 2 ) 3 )) => ( 1 3 )
```

The remq Function

remq returns a copy of an argument list with all top-level elements equal to a given SKILL object removed. The *eq* function is used to test for equality. This function is faster than the *remove* function because the *eq* equality test is faster than the *equal* equality test. However, the *eq* test is only meaningful for certain data types, such as symbols and lists. The *remq* function is appropriate, for example, when dealing with a list of symbols.

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

The *remq* function does not work on association tables.

Destructive Operations

The remd Function

remd removes all top-level elements equal to a given SKILL object from a list. *remd* uses *equal* for comparison. This is a destructive removal.

```
remd( "x" '("a" "b" "x" "d" "f")) => ("a" "b" "d" "f")
```

The remdq Function

remdq removes all top-level elements equal to the first argument from a list. *remdq* uses *eq* instead of *equal* for comparison. This is a destructive removal.

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

Substituting Elements

The *subst* function is a non-destructive operation. It is your responsibility to update any variables that you want to reflect this substitution.

Substituting One Object for Another Object in a List (subst)

subst returns the result of substituting the *new object* (first argument) for all equal occurrences of the *previous object* (second argument) at all levels in a list.

```
aList = '( a b c )            => ( a b c )
subst( 'a 'b aList )          => ( a a c )
anotherList = '( a b y ( d y ( e y )))
subst('x 'y anotherList )     => ( a b x ( d x ( e x )))
```

Although *subst('x 'y anotherList)* returns a list reflecting the desired substitutions, the value of *anotherList*, the original list, is not modified.

```
anotherList => ( a b y ( d y ( e y )))
```

Transforming Elements of a Filtered List

Many list operations can be modeled as a filtering pass followed by a transformational pass. For example, suppose you are given a list of integers and you want to build a list of the squares of the odd integers.

Phase I: Filter the odd integers into a list.

You can use the *setof* function here.

```
setof( x '( 1 2 3 4 5 6 ) oddp(x)) => ( 1 3 5 )
```

Phase II: Square each element of the list.

You can use the *mapcar* function together with a function that squares its argument:

```
mapcar(
      lambda( (x) x*x ) ;; square my argument
      '( 1 3 5 )
      ) => ( 1 9 25 )
```

or use the *foreach* function:

```
foreach( mapcar x '( 1 3 5 ) x*x ) => ( 1 9 25 )
```

The *trListOfSquares* function summarizes this approach.

```
procedure( trListOfSquares( aList )
      let( ( filteredList )
            filteredList =
                  setof( element aList oddp( element ))
            foreach( mapcar element filteredList
                  element * element
                  ) ; foreach
            ) ; foreach
      ) ; procedure

trListOfSquares( '( 1 2 3 4 5 6 )) => ( 1 9 25 )
```

Validating Lists

A predicate is a function that validates that a single SKILL object satisfies a criterion. SKILL provides many basic predicates. (Refer to “SKILL Predicates” on page 137 and “Type Predicates” on page 140.) Predicates return *t* or *nil*.

SKILL provides the *forall* function and the *exists* function so you can check whether all elements or some elements in a list satisfy a criterion. These two functions correspond to quantifiers in mathematical logic.

- *forall* is represented mathematically as ∀.
- *exists* is represented mathematically as ∃.

The criterion is represented by a SKILL expression with a single argument that you identify as the first argument to the *forall* or *exists* function.

The forall Function

forall verifies that an expression remains true for every element in a list. The *forall* function can also be used to verify that an expression remains true for every key/value pair in an association table. (Refer to "Association Tables" on page 117 for further details.)

```
forall( x '( 2 4 6 8 ) evenp( x ) ) => t
forall( x '( 2 4 7 8 ) evenp( x ) ) => nil
```

The exists Function

exists can use an application-specific testing function to locate the first occurrence of an element in a list based on equality. The *exists* function generalizes the *member* and *memq* functions, which locate the first occurrence of an element in a list based on equality.

```
exists( x '( 2 4 7 8 ) oddp( x ) ) => ( 7 8 )
exists( x '( 2 4 6 8 ) evenp( x ) ) => ( 2 4 6 8 )

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

Using Mapping Functions to Traverse Lists

SKILL provides a family of very powerful functions for iterating over lists. For historical reasons, these are called mapping functions. The five mapping functions are *map*, *mapc*, *mapcar*, *mapcan*, and *maplist*.

All five *map** functions have the same arguments.

- A function, which must take a single argument.
- A list.

Using lambda with the map* Functions

It is often convenient to use the *lambda* construct to define a nameless function to be used as the first argument.

For example:

```
mapcar(
      lambda( ( x ) list( x x**2 )) ;;; return pair of x x**x
      '(0 1 2 3 ))
      => ( (0 0) (1 1) (2 4) (3 9) )
```

Refer to "Syntax Functions for Defining Functions" on page 81 for further details on the *lambda* construct.

Using the map* Functions with the foreach Function

Alternatively, you can use each mapping function as an option to the *foreach* function. Often, using the *foreach* function results in more understandable code.

For example, the following are equivalent.

```
foreach( mapcar x '( 0 1 2 3 )
      list( x x**2 ) ;;; build 2 element list of a x and x*x
      )
      => ( (0 0) (1 1) (2 4) (3 9) )

mapcar(
      lambda( ( x ) list( x x**2 )) ;;; return pair of x x**x
      '(0 1 2 3 ))
```

The relationship between the two usages of the mapping functions is very tight. When you use the *foreach* function, SKILL actually incorporates the expressions within the *foreach* body in a *lambda* function as illustrated below. This *lambda* function is passed to the mapping function. For example, the following are equivalent:

```
foreach( mapcar x aList
      exp1()
      exp2()
      exp3()
      )

mapcar(
      lambda( ( x )
            exp1()
            exp2()
            exp3()
            )
      aList
      )
```

The following descriptions illustrate both approaches to using each mapping function.

The mapc Function

The *mapc* function is the default mapping function used by the *foreach* macro. When used with the *foreach* function

- The *foreach* function iterates over each element of the argument list.
- At each iteration, the current element is available in the loop variable of the *foreach*.
- The *foreach* function returns the original argument list as a result.

For example:

```
foreach( mapc x '(1 2 3 4 5)
      println(x)
)
mapc(
      lambda( ( x ) println( x ) )
      '( 1 2 3 4 5 )
      )
```

displays the following:

```
1
2
3
4
5
```

The return value is *(1 2 3 4 5)*.

The map Function

The *map* function is useful for processing each list cell because it uses *cdr* to step down the argument list. When used with the *foreach* function

- The *foreach* function iterates over each list cell of its argument.
- At each iteration, the current list cell is available in the loop variable of the *foreach*.
- The *foreach* function returns the original argument list as a result.

For example, suppose you want to make substitutions at the top-level of a list using a look-up table such as the following:

```
trLookUpList = '(
      ( 1 "one" )
      ( 2 "two" )
      ( 3 "three" ))
```

By using the *map* function, you gain access to each successive list cell, allowing you to use the *rplaca* function to make the desired substitution. Notice that the lookup list is an association list so that you can use the *assoc* function to retrieve the appropriate substitution.

```
assoc( 2 trLookUpList ) => ( 2 "two" )
assoc( 5 trLookUpList ) => nil
procedure( trTopLevelSubst( aList aLookUpList )
      let( ( currentElement substValue )
            foreach( map listCell aList
                  currentElement = car( listCell )
                  substValue = cadr(
                        assoc( currentElement aLookUpList ) )
                  when( substValue
                        rplaca( listCell substValue )
                        ) ; when
                  ) ; foreach
            aList
```

```
            ) ; let
      ) ; procedure
trList = '( 1 4 5 3 3 )
trTopLevelSubst( trList trLookUpList ) =>
            ("one" 4 5 "three" "three")
```

The mapcar Function

The *mapcar* function is useful for building a list whose elements can be derived one-for-one from the elements of an original list. When used with the *foreach* function

- The *foreach* function iterates over each element of the argument list.
- At each iteration, the current element is available in the loop variable of the *foreach*.
- Each iteration produces a result value.
- These values are returned in a list as the result of the function.

For example:

```
foreach(mapcar x '(1 2 3 4 5)
      println(x)
      x*3
)
```

displays the following:

```
1
2
3
4
5
```

The return value is *(3 6 9 12 15).*

The maplist Function

The *maplist* function is useful for processing each list cell because it uses *cdr* to step down the argument list. Like the *mapcar* function, it returns the list of results that it collects for each iteration. When used with the *foreach* function

- The *foreach* function iterates over each list cell of the argument list.
- At each iteration, the current list cell is available in the loop variable of the *foreach*.
- Each iteration produces a result value, and these values are returned in a list as the result of the *foreach* function.

For example, consider the substitution example illustrating the *map* function. In addition to making the substitutions, suppose that the function must display a message giving the number of substitutions made.

By using *maplist* instead of *map* you can build a list of 1s and Os reflecting the substitutions made. Use the *apply* function with *plus* to add up the numbers to produce the desired count.

```
procedure( trTopLevelSubst( aList aLookUpList )
            let( ( currentElement substValue substCountList )
            substCountList = foreach( maplist listCell aList
                  currentElement = car( listCell )
                  substValue = cadr(
                        assoc( currentElement aLookUpList ) )
                  if( substValue
                        then
                              rplaca( listCell substValue )
                              1
                        else
                              0
                        ) ; if
                  ) ; foreach
            printf( "There were %d substitutions\n"

                  apply( 'plus substCountList )
                  )
            aList
            ) ; let
      ) ; procedure
trList = '( 1 4 5 3 3 )
trTopLevelSubst( trList trLookUpList ) =>
      ("one" 4 5 "three" "three")
There were 3 substitutions
```

The mapcan Function

Like the *mapcar* function, the *mapcan* function is useful for building a list by transforming the elements of the original list. However, instead of collecting the intermediate results into a list as *mapcar* does, *mapcan* appends them. The intermediate results must be lists. Notice that the resulting list need not be in 1-1 correspondence with the original list.

The *mapcan* function iterates over each element of the argument list. When used with the *foreach* function

- At each iteration, the current element is available in the loop variable of the *foreach*.
- Each iteration produces a result value, which must be a list. These lists are then concatenated by the destructive modification function *nconc*, and the new list is returned as the result of the function as a whole.

For example, flattening a list of lists can be easily done with

```
foreach( mapcan x '( ( 1 2 ) ( 3 4 5 ) ( 6 7 ) )
      x
      ) => ( 1 2 3 4 5 6 7 )

mapcan(
      lambda( ( x ) x )
      '( ( 1 2 ) ( 3 4 5 ) ( 6 7 ) )
      ) => ( 1 2 3 4 5 6 7 )
```

For another example, the following builds a list of squares of the odd integers in the list *(1 2 3 4 5 6)*.

```
foreach( mapcan x '( 1 2 3 4 5 6 )
      when( oddp( x )
            list( x )
            )
      ) => ( 1 3 5 )

mapcan(
      lambda( ( x ) when( oddp( x ) list( x )) )
      '( 1 2 3 4 5 6 )
      ) => ( 1 3 5 )
```

Summarizing the List Traversal Operations

The table below summarizes how the mapping functions work. The term *Function* refers to the function you pass to the mapping function. Each table entry is a mapping function that behaves according to the row and column headings. In one case, there is no such mapping function.

Summary of the Mapping Functions

Mapping Function Return Value	Function is Passed Successive Elements	Function is Passed Successive List Cells
Ignores the result of each iteration. Returns the original list.	mapc (default option)	map
Collects the result of each iteration. Returns the list of results.	mapcar	maplist
Collects the result of each iteration. Uses *nconc* to append the result of each iteration.	mapcan	No such function

List Traversal Case Studies

The most useful mapping functions are *mapc*, *mapcar*, and *mapcan*. A good understanding of these functions simplifies most list handling operations in SKILL.

Handling a List of Strings

Suppose you need to calculate the length of each string in a list of strings so that the lengths are returned in a new list. That is, the function should perform the following transformation:

```
( "sam" "francis" "nick" ) => ( 3 7 4 )
```

Someone not familiar with the *mapcar* option to *foreach* might code this as follows:

```
procedure( string_lengths( stringList )
      let( (result)
            foreach( element stringList
                  result = cons( strlen(element) result)
                  ) ; foreach
            reverse( result )
            ) ; let
      ) ; procedure
```

Using the *mapcar* function allows this code to be written as

```
procedure(string_lengths(stringList)
      foreach(mapcar element stringList
            strlen(element)
            ) ; foreach
      ) ; procedure
```

In fact, the *foreach* function is implemented as a macro that expands to one of the mapping functions, so the same example can be written using the mapping function directly as follows:

```
procedure(string_lengths(stringList)
      mapcar( 'strlen stringList)
      ) ; procedure
```

Making Every List Element into a Sublist

You can perform this operation with either version of the *trMakeSublists* function.

```
procedure( trMakeSublists( aList )
      foreach( mapcar element aList
            list( element )
            ) ; foreach
      ) ; procedure
```

```
procedure( trMakeSublists( aList )
      mapcar(
            'ncons              ;;; return argument in a list
            aList
```

```
      )
) ; procedure
trMakeSublists( '(1 2 3)) => ( ( 1 ) ( 2 ) ( 3 ) )
```

Using mapcan for List Flattening

The *mapcan* function is useful when a new list is derived from an old one, and each member of the old list produces a *number* of members in the new list. (Note that using *mapcar* allows *only a one-to-one mapping*). One application of *mapcan* is in list flattening. Suppose we have a list that contains lists of numbers:

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

This list can be flattened using the following procedure:

```
procedure(flatten(numberList)
      foreach( mapcan element numberList
            element
            ) ; foreach
      ) ; procedure

flatten(x) => ( 1 2 3 4 5 6 7 8 9 )
```

This function simply concatenates all sublists of the argument. Remember that *nconc* is a *destructive* modification function, so variable *x* no longer holds useful information after the call.

To preserve the value of *x*, each sublist should be *copied* within the *foreach* function to produce a new sublist that can be harmlessly modified:

```
procedure( flatten( numberList )
      foreach( mapcan element numberList
            copy(element)
            ) ; foreach
      ) ; procedure

x = '( (1 2 3) (4) (5 6 7 8) () (9) )
flatten(x) => ( 1 2 3 4 5 6 7 8 9 )
x => ((1 2 3) (4) (5 6 7 8) nil (9))
```

Flattening a List with Many Levels

By using a type predicate and a recursive step, this procedure can be modified to flatten a list with many levels:

```
procedure(flatten(numberList)
      foreach(mapcan element numberList
            if( listp( element )
                  flatten(copy(element)) ;; then
                  ncons(element)
                  ) ; if
            ) ; foreach
      ) ; procedure
```

```
x = '((1) ((2 (3) 4 ((5)) () 6) ((7 8 ()))) 9)
flatten(x)       => (1 2 3 4 5 6 7 8 9)
x                => ((1) ((2 (3) 4 ((5)) nil 6) ((7 8 nil))) 9)
```

The body of the *foreach* first checks the type of the current list member.

- If the member is a list, the result is a list obtained by flattening a copy of it.
- If the member is not a list, it cannot be flattened further, and the result is a *one-element list* containing the member.

Remember that when using *mapcan*, the result of each iteration must be a list so that the results can be concatenated using *nconc*.

Manipulating an Association List

Mapping functions can be very powerful when used together. For example, suppose there is a database of names and extension numbers:

```
thePhoneDB = '(
            ("eric" 2345 6472 8857)
            ("sam" 4563 8857)
            ("julie" 7765 9097 5654 6653)
            ("francis")
            ("pat" 5527)
            )
```

The database is stored as a list with one entry for each person. An entry consists of a list of the person's name followed by a list of extensions at which the person can be reached. This type of list is known as an association list. Some people can be reached at several extensions, and some people at none at all. An automated dialing system has been introduced that accepts only name-number pairs: in other words, it requires data in the following format:

```
(("eric" 2345)
 ("eric" 6472)
 ("eric" 8857)
 ("sam" 4563)
 . . . . . .
```

How can the information be transformed from one format to another? Each *person* entry in the original database can produce several entries in the new database, so *mapcan* must be used to traverse person entries. Each *number* in the old database produces a single entry in the new database, so *mapcar* can be used to traverse the numbers.

From this information, a function can be written to translate the database:

```
procedure( translate( phoneDB )
      let((name)
            foreach(mapcan personEntry phoneDB
                  name = car( personEntry )
```

```
                foreach( mapcar number cdr( personEntry )
                      list( name number )
                      ) ; foreach
                ) ; foreach
           ) ; let
      ) ; procedure
```

To show that this works, consider the innermost *foreach* loop first. This loop is called for each person entry in the database. Suppose that the entry *("sam" 4563 8857)* is being processed. In this case, *name* is set to *"sam"* and the *foreach* iterates over the list of numbers *(4563 8857)*.

Because this iteration is a *mapcar*, the result of the *foreach* is a new list with one entry for each number in the old list. Each entry in this new list is a name-number pair constructed by the expression *list(name number)*. Hence, the result of this *foreach* is the list

```
(("sam" 4563) ("sam" 8857))
```

The outermost *foreach* concatenates these lists of name-number pairs: it works exactly like the first example of *flatten* given previously. Notice that there is no need to copy the elements before concatenating them (as was the case in *flatten*) because they have just been created.

Using the exists Function to Avoid Explicit List Traversal

SKILL provides a large number of list iteration functions. The most basic of these is the *foreach* function, which traverses all elements of a list. This traversal is useful, but often what is required is to traverse just part of a list, to check for a certain condition. In this situation, correct use of the *exists*, *forall*, and *setof* functions can greatly improve your code. Generally the alternatives are less efficient.

Consider writing a simple function that iterates over all integers in a list and checks whether there is any even integer. There are several approaches.

One approach is to use a *while* loop that explicitly tests a Boolean variable found.

```
procedure( contains_even( list )
 let( (found)
          while( list && !found
                  when( evenp( car(list) )
                   found = t
             )
             list = cdr(list)
             ) /* end while */
          /* Return whether found. */
          found
 ) /* end let */
) /* end contains_pass */
```

```
contains_even( '(1 2 3 4 )) => t
contains_even( '(1 7 3 9 )) => nil
contains_even( nil ) => nil
```

Another approach is to jump out of the *while* loop:

```
procedure(contains_even( list )
      prog( ()
            while(list
                  if( evenp( car(list) )
            return(t)
                  )
                  list = cdr(list)
            ) /* end while */
            return(nil)
 ) /* end prog */
) /* end contains_pass */
```

This approach might appear to be better because there is no need for the local variable.

A third approach involves jumping out of a *foreach* loop:

```
procedure(contains_even(list)
      prog( ()
      foreach(element list
                  if( evenp(element)
                        return(t)
                  )
            ) /* end foreach */
            return(nil)
 ) /* end prog */
) /* end contains_pass */
```

But this approach still requires the *prog* to allow the jump out of the *foreach* loop once the element has been found.

A much simpler way of writing this procedure is as follows:

```
procedure(contains_even(list)
      when( exists( element list evenp( element ))
            t
      ) /* when */
) /* end contains_pass */
```

This approach has neither the *prog* nor any local variables, and is just as intuitive. The *exists* function terminates as soon as a matching list member has been found, so this example is just as efficient in this respect as the previous ones (which had to use *return* to achieve this result). The result of an *exists* is *nil* if no matching entry is found. Otherwise the result is the sublist of the argument that contains the matching member as its head. For example:

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

Because SKILL considers *nil* to be equivalent to false and non-*nil* to be equivalent to true, the result of an *exists* can be treated as a boolean if desired. Note that if the *contains_even* function was defined to return either *nil* or non-*nil* (rather than *t*), it could be further simplified to

```
procedure( contains_even( list )
      exists( element list evenp( element ))
) /* end contains_pass */
```

As with all the other iterative functions, there is no need to declare the loop variable for *exists* because the loop variable is local to the function and automatically declared.

Commenting List Traversal Code

The examples above demonstrate the power of these mapping functions. Generally, wherever a *foreach* loop is used to build a list, a mapping function can usually be used instead. Because the mapping functions collect all the results and apply a single list building operation, they are faster than the equivalent iterative function and often look more succinct.

However, the mapping functions all look very similar, and it is often difficult to see exactly what a piece of code using a mapping function is trying to do. For this reason whenever a mapping function is used, you should write a comment detailing exactly what the function is expected to return.

9

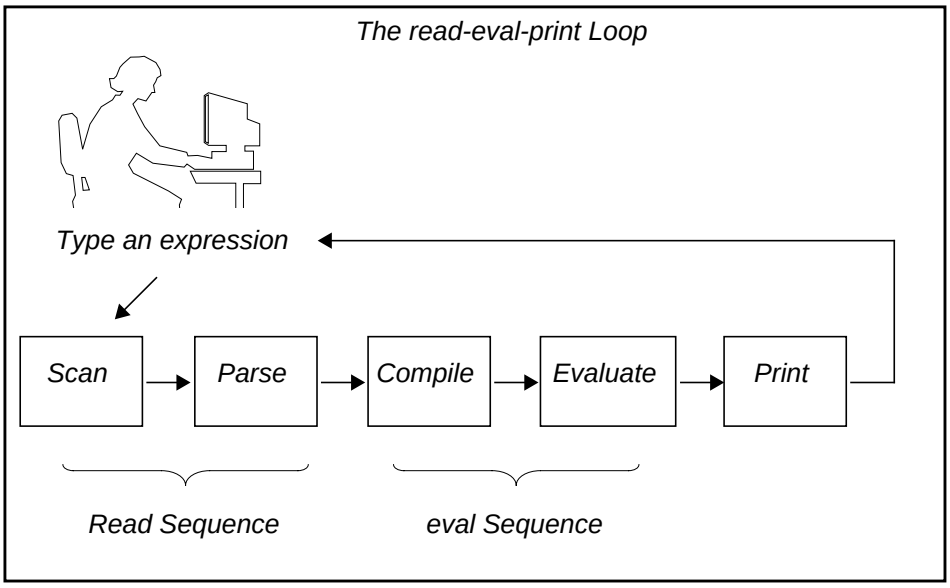
Advanced Topics

Overview information:

Cadence SKILL Language Architecture and Implementation

When you first encounter the Cadence® SKILL language in an application and begin typing expressions to evaluate, you are encountering what is known as the *read-eval-print* loop.

The expression you type in is first “read” and converted into a format that can be evaluated. The evaluator then does an “eval” on the output from the “read.” The result of the “eval” is a SKILL data value, which is then printed. The same sequence is repeated when other expressions are entered.



The “read” in this case performs the following tasks.

- The expression is parsed, resulting in the generation of a parse tree.
- The parse tree is then compiled into a body of code, known as a function object, made of a set of instructions that, when executed, results in the desired effect from the expression.

The instructions generated are not those of a particular machine architecture. They are the instructions of an abstract machine. Generally, this set of instructions might be referred to as byte-code or p-code. (p-code was the target instruction set of some of the early Pascal compilers and lent the name to this technique.)

The evaluator executes the byte-code generated by the compiler. In a sense, the evaluator emulates in software the operations of a hardware CPU. This technique of using an abstract

instruction set to be executed by an engine written in software has several advantages for the implementation of an extension language.

- This technique lends itself well to an interpreter-based implementation.
- This technique offers faster performance than direct source interpretation.
- Once SKILL code is compiled into contexts (refer to Delivering Products on page 225) the context files are faster to load than original source code and are portable from one machine to another.

That is, if the context file is generated on a machine with architecture A from vendor X, it can be copied onto a machine with architecture B from vendor Y and SKILL will load the file without the need for recompilation or translation.

Evaluation

SKILL provides functions that invoke the evaluator to execute a SKILL expression. You can therefore store programs as data to be subsequently executed. You can dynamically create, modify, or selectively evaluate function definitions and expressions.

Evaluating an Expression (eval)

eval accepts any SKILL expression as an argument. *eval* evaluates an argument and returns its value.

```
eval( '( plus 2 3 ) )=> 5
```

Evaluates the expression *plus(2 3)*.

```
x = 5
eval( 'x )=> 5
```

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

```
eval( list( 'max 2 1 ) ) => 2
```

Evaluates the expression *max(2 1)*.

Getting the Value of a Symbol (symeval)

symeval returns the value of a symbol. *symeval* is slightly more efficient than *eval* and can be used in place of *eval* when you are sure that the argument being evaluated is indeed a symbol.

```
x = 5
symeval( 'x )=> 5
y = 'unbound
symeval( 'y )=> unbound
```

Returns *unbound* if the symbol is *unbound*.

Use the *symeval* function to evaluate symbols you encounter in lists. For example, the following *foreach* loop returns *aList* with the symbols replaced by their values.

```
a = 1
b = 2
aList = '( a b 3 4 )
anotherList = foreach( mapcar element aList
      if( symbolp( element )
            then symeval( element )
            else element
            ) ; if
      )
=> ( 1 2 3 4 )
```

Applying a Function to an Argument List (apply)

apply is a function that takes two or more arguments. The first argument must be either a symbol signifying the name of a function or a function object. (Refer to "Declaring a Function Object (lambda)" on page 210.) The rest of the arguments to apply are passed as arguments to the function.

apply calls the function given as the first argument, passing it the rest of the arguments. *apply* is flexible as to how it takes the arguments to pass to the function. For example, all the calls below have the same effect, that of applying *plus* to the numbers 1, 2, 3, 4, and 5:

```
apply('plus '(1 2 3 4 5) )
=> 15
apply('plus 1 2 3 '(4 5) )
apply('plus 1 2 3 4 5 nil)
=> 15
```

The last argument to *apply* must always be a list.

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

```
apply('plus (list 1 2) )         ; Apply plus to its arguments.
=> 3
defmacro( sum (@rest nums) `(plus ,@nums)) ; Define a macro.
=> sum
apply('sum '(sum 1 2))
=> (1 + 2)                       ; Returns expanded macro.
eval('(sum 1 2))
=> 3
```

Function Objects

When you use the procedure function to define a function in SKILL, the byte-code compiler generates a block of code known as a function object and places that object on the function property of a symbol.

Subsequently, when SKILL encounters the symbol in a function call, the function object is retrieved and the evaluator executes the instructions.

Function objects can be used in assignment statements and passed as arguments to functions such as *sort* and *mapcar*.

SKILL provides several functions for manipulating function objects.

Retrieving the Function Object for a Symbol (getd)

You can use the *getd* function to retrieve the function object that the procedure function associates with a symbol.

For example:

```
procedure( trAdd( x y )
      printf( "Adding %d and %d ... %d \n" x y x+y )
      x+y
      )
=> trAdd
getd( 'trAdd ) => funobj:0x1814bc0
```

If there is no associated function object, *getd* returns nil. The following table shows several other possible return values.

The getd Function

Function	Return Value	Explanation
trAdd	funobj:0x1814bc0	Application SKILL function.
edit	t	Read-protected SKILL function. A function is read protected when it is loaded from a context or from an encrypted file.
max	lambda:0xf6f25c	Built-in lambda function.
breakpt	nlambda:0xf7a784	Built-in nlambda function.

Assigning a New Function Binding (putd)

The *putd* function binds a function object to a symbol. You can undefine a function by setting its function binding to *nil*. You cannot change the function binding of a write-protected function using *putd*.

For example, you can copy a function definition into another symbol as follows:

```
putd( 'mySqrt getd( 'sqrt ))=> lambda:0x108b8
```

Assigns the function *mySqrt* the same definition as *sqrt*.

```
putd( 'newFun
      lambda( ( x y ) x + y )
      )
=> funobj:0x17e0b3c

newFun( 5 6 )
=> 11
```

Assigns the symbol *newFun* a function definition that adds its two arguments.

Declaring a Function Object (lambda)

The word *lambda* in SKILL is inherited from Lisp, which in turn inherits it from *lambda calculus*, a mathematical compute engine on which Lisp is based.

The *lambda* function builds a function object. The arguments to the *lambda* function are

- The formal arguments
- The SKILL expressions that make up the function body (these expressions are evaluated when the function object is passed to the *apply* function or the *funcall* function)

Unlike the procedure function, the *lambda* function does not associate the function object with any particular symbol. For example:

```
(lambda (x y) (sqrt (x*x + y*y)))
```

defines an unnamed function capable of computing the length of the diagonal side of a right-angled triangle.

Evaluating a Function Object

Unnamed or anonymous functions are useful in various situations. For example, mapping functions such as *mapcar* require a function as the first argument. You can pass either a symbol or the function object itself.

```
mapcar( 'get_pname '( sin cos tan ))
=> ("sin" "cos" "tan")
mapcar( lambda(( x ) strlen( get_pname(x)) )
      '( sin cos tan ))
=> ( 3 3 3 )
```

Note that a *quote* before a *lambda* construct is not needed. In fact, a *quote* before a *lambda* construct used as a function is slower than one without a *quote* because the construct is compiled every time before it is called. That is, the *quote* prevents the *lambda* construct from being compiled into a function object when the code is loaded. You can save function objects in data structures. For example:

```
var = (lambda (x y) x + y)
=> funobj:0x1eb038
```

The result is a function object stored in the variable *var*. Function objects are first class objects. That is, you can use function objects just like an instance of any other type to pass as an argument to other functions or to assign as a value to variables. You can also use function objects with *apply* or *funcall*. For example:

```
apply(var '(2 8))
=> 10
funcall(var 2 8)
=> 10
```

Efficiently Storing Programs as Data

Whenever possible, store SKILL programs as function objects instead of text strings. Function objects are more efficient because calls to *eval, errset, evalstring,* or *errsetstring* require the compiler and generate garbage parsing the text strings. On the other hand, unquoted *lambda* expressions are compiled once. Use *apply* or *funcall* to do the evaluation.

Converting Strings to Function Objects (stringToFunction)

To convert an expression represented as a string into a function object with zero arguments, use *stringToFunction.* For example:

```
f = stringToFunction("1+2")  => funobj:0x220038
apply(f nil) => 3
```

To convert an expression represented as a list, you can construct a list with *lambda* and *eval* it. Make sure you account for any parameters:

```
expr = '(x + y)
f = eval( ‘( lambda ( x y ) ,expr )) => funobj:0x33ab00
apply( f '( 5 6 ) ) => 11
```

You can always construct the expression as a *lambda* construct at the outset to avoid an unnecessary call to *eval*.

Macros

Macros in SKILL are different from macros in C.

- In C, macros are essentially syntactic substitutions of the body of the macro for the call to the macro.
- A macro function allows you to adapt the normal SKILL function call syntax to the needs of your application.

Benefits of Macros

SKILL macros can be used in various situations:

- To gain speed by replacing function calls with in-line code
- To expand constant expressions for readability
- As convenience wrappers on top of existing functions

Note that in-line expansion increases the size of the code using macros, so use macros sparingly and only in those situations where they clearly add value to the code.

Macro Expansion

When SKILL encounters a macro function call, it evaluates the function call immediately and the last expression computed is compiled in the current function object. This process is called macro expansion. Macro expansion is inherently recursive: the body of a macro function can refer to other macros including itself.

Macros should be defined before they are referenced. This is the most efficient way to process macros. If a macro is referenced before it is defined, the call is compiled as “unknown” and the evaluator expands it at run-time, incurring a serious penalty in performance for macros.

Redefining Macros

If you are in development mode and you redefine a macro, make sure all code that uses that macro is reloaded or redefined. Otherwise, wherever the macro was expanded, the previous definition continues to be used.

defmacro

To define a macro in SKILL, use *defmacro*.

For example, if you want to check the value of a variable to be a string before calling *printf* and you don't want to write the code to perform the check at every place where *printf* is called, you might consider writing a macro:

```
(defmacro myPrintf (arg) `when((stringp ,arg)
 printf( "%s" ,arg)))
```

As you can see, the macro *myPrintf* returns an expression constructed using backquote, which is substituted for the call to *myPrintf* at the time a function calling *myPrintf* is defined.

mprocedure

The *mprocedure* function is a more primitive facility on which *defmacro* is based. Avoid using *mprocedure* in new code that you are developing. Use *defmacro* instead. While compiling source code, when SKILL encounters an *mprocedure* call, the entire list representing the function call is passed to the *mprocedure* immediately, bound to the single formal argument. The result of the last expression computed within the *mprocedure* is compiled.

Using the Backquote (') Operator with defmacro

Here is a sample macro that highlights the effect of compile time macro expansion in SKILL. It assumes *isMorning* and *isEvening* are defined.

```
(defmacro myGreeting (_arg)
      let((_res)
            cond( (isMorning()
                              res= sprintf(nil "Good morning %s" _arg))
                        (isEvening()
                              _res= sprintf(nil "Good evening %s" _arg)))
            'println(,_res))
)
```

Use the utility function *expandMacro* to test the expansion, for example,

```
expandMacro('myGreeting("Sue")) ; using the above definition
=> println("Good morning Sue")  ; if isMorning returns t
```

When called, *myGreeting* returns a *println* statement with the desired greeting to be in-line substituted for the call to *myGreeting*. Because the call to *sprintf* inside *myGreeting* is performed outside of the expression returned, the greeting message reflects the time when the code was compiled, and not when it was run.

This is how *myGreeting* should be rewritten to have the greeting message reflect the time the code was run:

```
(defmacro myGreeting (_arg)
     'let((_res)
        cond( (isMorning()
                    _res= sprintf(nil "Good morning %s" ,_arg))
                (isEvening()
                    _res= sprintf(nil "Good evening %s" ,_arg)))
        println(_res))
)
```

The above, when compiled, substitutes the entire *let* expression in-line for the macro call.

Using an @rest Argument with defmacro

The next macro example shows how a functionality can be implemented efficiently by exploiting in-line expansion. The macro implements *letStar (read: let-star)*. It differs from regular *let* in that it performs the bindings for the declared local variables in sequence, so an earlier declared variable can be used in expressions evaluated to bind subsequent variable declarations, which is not safe to do in a *let*. For example:

```
(letStar ((x 1) (y x+1) ...)
```

guarantees that *x* is first bound to 1 when the binding for *y* is done.

The *letStar* macro

```
defmacro(letStar (decl @rest body)
                 cond(( zerop(length(decl))
                                cons(‘progn body))
                        ( t ‘let((,car(decl))
                                        letStar(,cdr(decl),@body)))
                 )
)
```

is defined recursively. For each variable declaration, it nests *let* statements, thereby guaranteeing that all bindings are performed sequentially. For example:

```
procedure( foo()
      letStar(((x 1) (y x+1))
                      y+x))
```

expands to

```
procedure(foo()
      let(((x 1))
            let(((y x+1))
                        progn( y+x ))))
```

Using @key Arguments with defmacro

The following example illustrates a custom syntax for a special way to build a list from an original list by applying a filter and a transformation. To build a list of the squares of the odd integers in the list

```
      ( 0 1 2 3 4 5 6 7 8 9 )
```

you write

```
trForeach(
      ?element          x
      ?list             '( 0 1 2 3 4 5 6 7 8 9 )
      ?suchThat         oddp(x)
      ?collect          x*x
      ) => ( 1 9 25 49 81 )
```

instead of the more complicated

```
foreach( mapcar x
      setof(x '(0 1 2 3 4 5 6 7 8 9) oddp(x))
      x * x
      ) => ( 1 9 25 49 81 )
```

Implementing an easy-to-maintain macro requires knowledge of how to build SKILL expressions dynamically using the backquote (‘),
comma (,), and comma-at (,@) operators.

The definition for *trForeach* follows.

```
defmacro( trForeach ( @key element list suchThat collect )
      ‘foreach( mapcar ,element
            setof( ,element ,list ,suchThat )
                  ,collect
            ) ; foreach
) ; defmacro
```

Variables

SKILL uses symbols for both global and local variables. In SKILL, global and local variables are handled differently from C and Pascal.

Lexical Scoping

In C and Pascal, a program can refer to a local variable only within certain textually defined regions of the program. This region is called the *lexical scope* of the variable. For example, the lexical scope of a local variable is the body of the function. In particular

- Outside of a function, local variables are inaccessible
- If a function refers to non-local variables, they must be global variables

Dynamic Scoping

SKILL does not rely on lexical scoping rules at all. Instead

- A symbol's current value is accessible at any time from anywhere within your application
- SKILL transparently manages a symbol's value slot as if it were a stack
- The current value of a symbol is simply the top of the stack
- Assigning a value to a symbol changes only the top of the stack
- Whenever the flow of control enters a *let* or *prog* expression, the system pushes a temporary value onto the value stack of each symbol in the local variable list (he local variables are normally initialized to *nil*)
- Whenever the flow exits the *let* or *prog* expression, the prior values of the local variables are restored

Dynamic Globals

During the execution of your program, the SKILL programming language does not distinguish between global and local variables.

The term *dynamically scoped variable* refers to a variable used as a local variable in one procedure and as a global in another procedure. Such variables are of concern because any function called from within a *let* or *prog* expression can alter the value of the local variables of that *let* or *prog* expression. For example:

```
procedure( trOne()
      let( ( aGlobal )
            aGlobal = 5 ;;; set the value of trOne's local variable
            trTwo()
            aGlobal     ;;; return the value of trOne's local variable
            ) ;let
      ) ; procedure

procedure( trTwo()
```

```
printf("\naGlobal: %L" aGlobal )
aGlobal = 6
printf("\naGlobal: %L\n" aGlobal )
aGlobal
) ; procedure
```

The *trOne* function uses the *let* function to define *aGlobal* to be a local variable. However, *aGlobal*'s temporary value is accessible to any function *trOne* calls. In particular, the *trTwo* function changes *aGlobal*.

This change is not intuitively expected and can lead to problems that are very difficult to isolate. SKILL Lint reports this type of variable as an "Error Global." It is generally recommended that users should not rely on the dynamic behaviour of variable bindings.

Error Handling

SKILL has a robust error handling environment that allows functions to abort their execution and recover from user errors safely. When an error is discovered, you can send an error signal up the calling hierarchy. The error is then caught by the first error catcher that is active. The default error catcher is the SKILL top level, which catches all errors that were not caught by your own error catchers.

The errset Function

The *errset* function catches any errors signalled during the execution of its body. The *errset* function returns a value based on how the error was signalled. If no error is signalled, the value of the last expression computed in the *errset* body is returned in a list.

```
errset( 1+2 )(3)
```

In the following example, without the *errset* wrapper, the expression

```
1+"text"
```

signals an error and display the messages

```
*Error* plus: can't handle (1 + "text")
```

To trap the error, wrap the expression in an *errset*. Trapping the error causes the *errset* to return *nil*.

```
errset( 1+"text" ) => nil
```

If you pass *t* as the second argument, any error message is displayed.

```
errset( 1+"text" t ) => nil
*Error* plus: can't handle (1 + "text")
```

Information about the error is placed in the *errset* property of the *errset* symbol. Programs can therefore access this information with the *errset.errset* construct after determining that *errset* returned *nil.*

```
errset( 1+"text" )=> nil
errset.errset =>
("plus" 0 t nil
      ("*Error* plus: can't handle (1 + \"text\")"))
```

Using err and errset Together

Use the *err* function to pass control from the point at which an error is detected to the closest *errset* on the stack. You can control the return value of the *errset* by your argument to the *err* function.

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

```
procedure( trDivide( x )
      cond(
            ( !numberp( x ) err() )
            ( zerop( x ) err( 'trDivideByZero ) )
            ( t 1.0/x )
            )
      ) ; procedure

errset( trDivide( 5 ) )      => ( 0.2 )
errset( trDivide( 0 ) )      => (trDivideByZero)
errset( trDivide( "text" ) ) => nil

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

The error Function

error prints the error messages, if any are given, and then calls *err*, causing an error. The first argument can be a format string, which causes the rest of the arguments to print in that format.

```
error( "myFunc" "Bad List" )
```

Prints **Error* myFunc: Bad List*.

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

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

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

Prints **Error* test.*

The warn Function

warn queues a warning message string. After a function returns to the top level, all queued warning messages are printed in the Command Interpreter Window and the system flushes the warning queue. Arguments to *warn* use the same format specification as *sprintf*, *printf*, and *fprintf*.

This function is useful for printing SKILL warning messages in a consistent format. You can also suppress a message with a subsequent call to *getWarn*.

```
arg1 = 'fail
warn( "setSkillPath: first argument must be a string or list of strings - %s\n"
arg1)
=> nil

*WARNING* setSkillPath: first argument must be a string or list of strings - fail
```

The getWarn Function

getWarn dequeues the most recently queued warning from a previous *warn* function call and returns that warning as its return result.

```
procedure( testWarn( @key ( dequeueWarn nil ) )
      warn("This is warning %d\n" 1 ) ;;; queue a warning
      warn("This is warning %d\n" 2 ) ;;; queue a warning
      warn("This is warning %d\n" 3 ) ;;; queue a warning
      when( dequeueWarn
            getWarn() ;;; return the most recently queued warning
            )
      ) ; procedure
```

The *testWarn* function prints the warning if *t* is passed in and gets the warning if *nil* is given as an argument.

```
testWarn( ?dequeueWarn nil)
=> nil
*WARNING* This is warning 1
*WARNING* This is warning 2
*WARNING* This is warning 3
```

Returns *nil* and the system prints all the queued warnings.

```
testWarn( ?dequeueWarn t)
=> "This is warning 3\n"
*WARNING* This is warning 1
*WARNING* This is warning 2
```

Returns the dequeued (most recent) warning and the system prints the remaining queued warnings.

Top Levels

When you run SKILL or non-graphical applications built on top of SKILL, you are talking to the SKILL top level, which reads your input from the terminal, evaluates the expressions, and prints the results. If an error is encountered during the evaluation of expressions, control is usually passed back to the top level.

When you are talking to the top level, any complete expression that you type (followed by typing the Return key to signal the end of your input) is evaluated immediately. Following the evaluation, the value of the expression is pretty printed.

If only the name of a symbol is typed at the top level, SKILL checks if the variable is *bound*. If so, the value of the variable is printed. Otherwise, the symbol is taken to be the name of a function to call (with no arguments). The following examples show how the outer pair of parentheses can be omitted at the top-level.

```
if (ga > 1) 0 1
if( (a > 1) 0 1 )

loadi "file.ext"
loadi("file.ext")

exit                ; Assuming exit has no variable binding
exit()
```

The default top level uses *lineread*, so it quietly waits for you to complete an expression if there are any open parentheses or any binary *infix* operators that have not yet been assigned a right operand. If SKILL seems to do nothing after you press Return, chances are you have mistyped something and SKILL is waiting for you to complete your expression.

Sometimes typing a super right bracket (]) is all you need to properly terminate your input expression. If you mistype something when entering a form that spans multiple lines, you can cancel your input by pressing *Control-c*. You can also press *Control-c* to interrupt function execution.

Memory Management (Garbage Collection)

In SKILL all memory allocation and deallocation is managed automatically. That is, the developer using SKILL does not have to remember to deallocate unused structures. For example, when you create an array or an instance of a *defstruct* and assign it as a value to a variable declared locally to a procedure, if the structure is no longer in use after the procedure exits, the memory manager reclaims that structure automatically. In fact, reclaimed structures are subsequently recycled. For users programming in SKILL, garbage collection simplifies bookkeeping to the point that most users do not have to worry about storage management at all.

The allocator keeps a pool of memory for each data type and, on demand, it allocates from the various pools and reclaimed structures are returned to the pool. The process of reclaiming unused memory - garbage collection (GC) - is triggered when a memory pool is exhausted.

Garbage collection replenishes the pool by tracking all unused or unreferenced memory and making that memory available for allocation. If garbage collection cannot reclaim sufficient memory, the allocator applies heuristics to expand the memory pools by a factor determined at run time.

Garbage collection is transparent to SKILL users and to users of applications built on top of SKILL. The system might slow down for a brief moment when garbage collection is triggered, but in most cases it should not be noticeable. However, unrestrained use of memory in SKILL applications can result in more time spent in garbage collection than intended.

How to Work with Garbage Collection

The garbage collector uses a heuristic procedure that dynamically determines when and if additional working memory should be allocated from the operating system. The procedure works well in most cases, but because optimal time/space trade-offs can vary from application to application, you might sometimes want to override the default system parameters.

When an application is known to use certain SKILL data types more than others, you can measure the amount of memory pools needed for the session and preallocate that pool. This allocation helps reduce the number of garbage collection cycles triggered in a session. However, because the overhead caused by garbage collection is typically only several percent of total run time, such fine-tuning might not be worthwhile for many applications.

First you need to analyze your memory usage by using *gcsummary*. This function prints a breakdown of memory allocation in a session. See the next section for a sample output. Once you have determined how many instances of a data type you need for the session, you can preallocate memory for that data type by using *needNCells* (described at the end of this section).

For the most part, you do not need to fine tune memory usage. You should first use memory profiling (refer to ”Cadence SKILL Profiler” in *SKILL Development Help*) to see if you can track down where the memory is generated and deal with that first. Use the memory tuning technique described in this section as a last resort. Remember, because all memory tuning is global, you can’t just tune the memory for your application. All other applications running in the same currently running binary are affected by your tuning.

Printing Summary Statistics

The *gcsummary* function prints a summary of memory allocation and garbage collection statistics in the current SKILL run.

How to Interpret the Summary Report

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

```
************* SUMMARY OF MEMORY ALLOCATION *************
Maximum Process Size (i.e., voMemoryUsed) = 3589448
Total Number of Bytes Allocated by IL = 2366720
Total Number of Static Bytes          = 1605632
-------------------------------------------------------------
Type       Size   Allocated     Free      Static   GC count
-------------------------------------------------------------
list         12      339968    42744     1191936          9
fixnum        8       36864    36104       12288          0
flonum       16        4096     2800       20480          0
string        8       90112    75008       32768          0
symbol       28           0        0      303104          0
binary       16           0        0        8192          0
port         60        8192     7680           0          0
array        16       20480     8288        8192          0
TOTALS       --      516096   188848     1576960          9
-------------------------------------------------------------
User Type (ID)             Allocated     Free    GC count
-------------------------------------------------------------
hiField              (20)       8192     7504           0
hiToggleItem         (21)       8192     7900           0
hiMenu               (22)       8192     7524           0
hiMenuItem           (23)       8192     5600           0
TOTALS                --       32768    28528           0
-------------------------------------------------------------
Bytes allocated for :
      arrays       = 38176
      strings      = 43912
      strings(perm)= 68708
      IL stack     = 49140
      (Internal)   = 12288
```

```
TOTAL GC COUNT 9
----- Summary of Symbol Table Statistics -----
Total Number of Symbols = 11201
Hash Buckets Occupied = 4116 out of 4499
Average chain length = 2.721331
```

Allocating Space Manually

The *needNCells* function takes a cell count and allocates the appropriate number of pages to accommodate the cell count. The name of the user type can be passed in as a string or a symbol. However, internal types, like *list* or *fixnum,* must be passed in as symbols. For example:

```
needNCells( 'list 1000 )
```

guarantees there will always be 1000 list cells available in the system.

Exiting SKILL

Normally you exit SKILL indirectly by selecting the *Quit* menu command from the CIW while running the Cadence software in graphic mode, or by typing *Control-d* at the prompt while running in non-graphic mode. However, you can also call the *exit* function to exit a running SKILL application with or without an explicit status code. Actually, both the *Quit* menu command in the CIW and *Control-d* in standalone SKILL trigger a call to *exit*.

Sometimes you might like to do certain cleanup actions before exiting SKILL. You can do this by registering exit-before and/or exit-after functions, using the *regExitBefore* and *regExitAfter* functions. An exit-before function is called before *exit* does anything, and an exit-after function is called after *exit* has performed its bookkeeping tasks and just before it returns control to the operating system. The user-defined exit functions do not take any arguments.

To give you even more control, an exit-before function can return the atom *ignoreExit* to abort the exit call totally. When *exit* is called, first all the registered exit-before functions are called in the reverse order of registration. If any of them returns the special atom *ignoreExit*, the exit request is aborted and it returns *nil* to the caller. After calling the exit-before functions, it does some bookkeeping tasks, calls all the registered exit-after functions in the reverse order of their registration, and finally exits to the operating system.

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

10

Delivering Products

Overview information:

Contexts

This information is for users who are writing a large volume of using the Cadence® SKILL language code and are interested in modularizing the code and possibly autoloading it at run time.

What Are Contexts?

Contexts are binary representations of the internal state of the interpreter. Their primary purpose is to help speed the loading of SKILL files. They are best used when the set of SKILL files to load is large.

A SKILL Development license is required to build contexts using the saveContext command.

All SKILL-related structures can be saved in a context except those with values meaningless outside of a session. For example, port values, database handles, and window types are meaningful only to current sessions, whereas lists, integers, floats, defstructs, and so forth are transportable from one session to another.

Contexts Load and Initialize Product Code Quickly

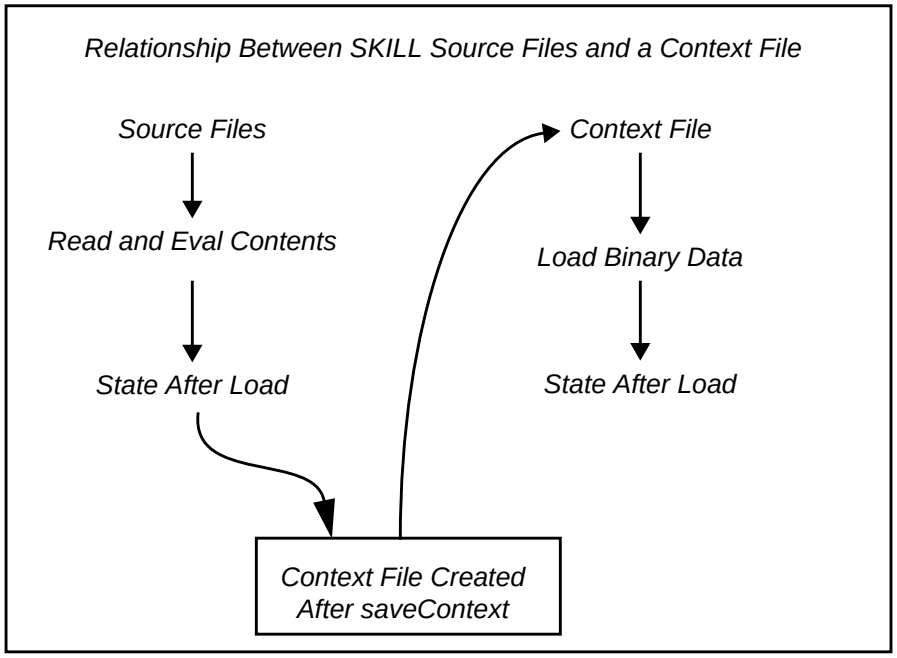
SKILL contexts contain source code and data. Their main purpose is to allow for fast loading and initializing of product code. One way of looking at contexts is to view them as snapshots of the internal state of the interpreter, much like a core file in UNIX is an image of the running process.

What You Cannot Put in Contexts

Contexts, however, cannot store at save time and retrieve at load time structures that are process-dependent. For example, file descriptors stored in port structures or process IDs cannot be saved in contexts. All other non process-dependent SKILL structures can be safely stored. For example, lists, numbers, strings, arrays, and so forth can be saved into and retrieved from a context.

Difference Between Loading SKILL Files and Contexts

When a SKILL source file is loaded, it is first parsed. The evaluator is called for each complete expression read in. This is how procedures are defined. Context files, on the other hand, contain binary data that is loaded directly into memory. The binary data stored into a context has been parsed and evaluated before being saved.



When to Use Contexts

The developer must decide when it is better to use contexts than straight or encrypted SKILL code.

Context Size

Context files should be kept small. Context files greater than five megabytes are not recommended.

Is the Code Likely to Become a Product?

Generally speaking, the first criteria is whether the code is likely to become a product that will be shipped. This is important because contexts offer a good vehicle for productizing code.

How Long Does the Code Take to Load?

The second criteria is whether there is enough SKILL code to warrant being in a context. This is difficult to measure. A simple test is to load the source code and see if the time it takes is likely to be unacceptable to a user. For example, if the code is likely to be auto-loaded during the physical manipulation of graphics, the impact of the load and initialization should be minimized. Contexts can help in this case.

The more code and initialization needed at load time, the better it is to use contexts. For very small amounts of code (200 lines), contexts might be overkill. To load and initialize 20,000 lines of SKILL code without using contexts takes approximately 30 seconds, whereas loading the same code using a context takes approximately 4 seconds. In this case, the perceptible impact is high, so using contexts makes sense.

Modularizing Code

Sometimes it is necessary to modularize code according to predefined capabilities. There might be a need to have these capabilities loaded incrementally at run-time. Thus it is not necessary to load all the code at once. Contexts, as snapshots of the interpreter's internal state, can be saved into separate files, even though code that goes into one context might depend on code in another.

The dependencies are resolved at load time. For example, the SKILL code for the schematics editor relies, among other things, on having code for the graphics editor present, but the context for the schematics editor does not contain any of the graphic editor's code or structures.

Create Contexts at Integration Time

The process of creating contexts must always be a separate step done at integration time, the time when all C code is compiled and linked and SKILL files are digested to produce context files. During integration and when contexts are being created, the interpreter enters a state that renders all normal use inefficient.

For example, during context creation, the memory management subsystem works in a special mode such that incremental snapshots of the memory can be made and saved into contexts. Context creation and code that is used to create contexts should not be part of the normal function of any product.

Creating Contexts

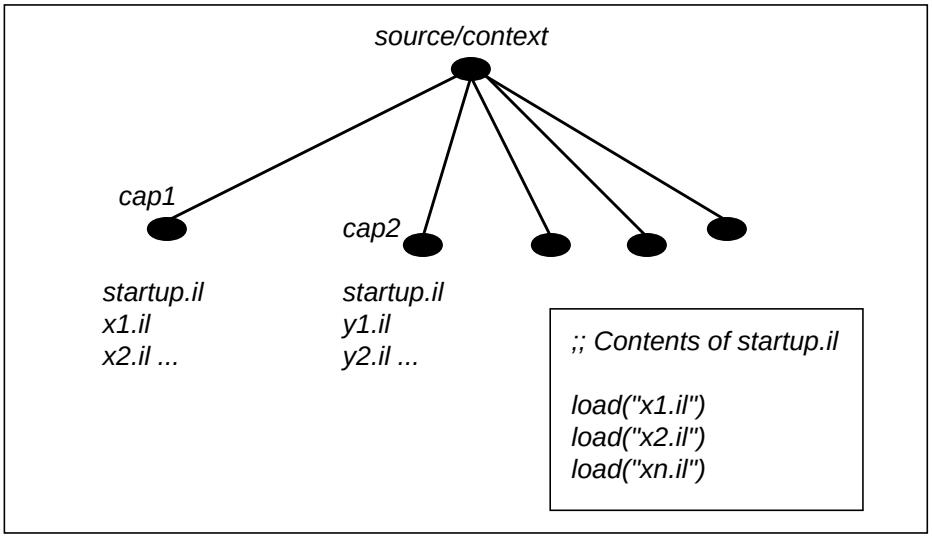
When you use SKILL contexts for delivering a product, you must develop an organization for those contexts.

Creating the Directory Structure

First, group the SKILL code on a product/capability basis. This is best done by designating a special directory, for example, <*source/context*> and then using make files or simple scripts to copy the code into separate directories under <*source/context*>.

If the capability names are *cap1* and *cap2,* create two directories under <*source/context*> named <*source/context/cap1*> and <*source/context/cap2*>. Copy the source code for each capability into the respective directory.

For each capability context directory, create a single file named *startup.il* that uses the SKILL *load* command to load all the files in that directory in the necessary order. The figure below shows a layout of the context directories.



Caution

The first 12 characters in a context file name must be unique.

How the Process of Building Contexts Works

Using the directory structure described above, the process of building contexts starts by loading code from each directory and generating a binary context corresponding to the state of the interpreter when the files are loaded (in the sequence specified in the *startup.il* file).

The binary context file can go into one of several directories. If auto-loading of the context is necessary, the file can be placed in the */your_install_dir/tools/dfII/local/context* directory. The autoload mechanism looks in this directory. If direct commands are issued to load the context, it can be placed anywhere system administrators deem appropriate.

Creating Utility Functions

Given the directory structure above, the code for generating the contexts can now be written. First let us define a few utility functions.

Assumptions

Source code is stored under the */user/source/context* directory. The created contexts are saved under */user/etc/context*, where *user* is any installation path you choose for keeping sources and contexts. These contexts are hard-coded in the sample code below, but you can pass them as arguments to the functions.

Create *myContextBuild.il*

Put the following code in a separate file. Call it *myContextBuild.il.*

```
procedure( getContext(cxt)
;;------------------------
;; Given a context name load the context into the session
(let ((ff (strcat "/<user>/etc/context/" cxt ".cxt")))
      (cond ((null (isFile ff)) nil)
            ((null (loadContext ff))
                  (printf "Failed to load context %s\n" cxt))
            ((null (callInitProc cxt))
                  (printf "Failed to initialize context %s\n"
                        cxt))
                  (t (printf "Loading Context %s\n" cxt))
      )
      )
)
procedure( makeContext(cxt)
;;-------------------------
;; Given a context name create and save the context
;; under "/<user>/etc/context". Assumes user source
;; code is located under /<user>/source/<context>
```

```
(let ( (newPath (strcat "/<user>/source/" cxt))
      (oldPath (getSkillPath))
             (fileName (strcat "/<user>/etc/context/" cxt ".cxt"))
             (oldStatus (status writeProtect))
            )

            (printf "Building context for %s\n" cxt)
      (setSkillPath newPath)
            (sstatus writeProtect t)
      ;; setContext is a function that takes the name of
      ;; a context and indicates to the system that
      ;; whatever is loaded or evaluated from
      ;; the point of this call to the time context
      ;; is saved belongs to the named context.

      (setContext cxt)

      ;; Load all the SKILL files corresponding to the
      ;; given context. Relies on skill path being set earlier

      (loadi "startup.il")

      ;; After the load save the files into a context
      ;; file containing all the necessary data to load
      ;; the context

      (saveContext fileName)

            ;; It is important to call the initialization function
            ;; AFTER the context file is saved. This procedure
            ;; may create structures that can't be saved into
            ;; a context file but a subsequent load of SKILL files
            ;; for another context may need the call to have taken
            ;; place.

      (callInitProc cxt)

      ;; Set the SKILL path back to what it was.

      (setSkillPath oldPath)
            (unless oldStatus (sstatus writeProtect nil))
))

procedure( buildContext(cxt)
;;-------------------------
;; Deletes existing context and prepares to create the context

(progn
      (deleteFile (strcat "/<user>/etc/context/" cxt ".cxt"))
      (cond ((isDir cxt "/<user>/source/")
                  (makeContext cxt))
            (t (printf "Can't find context directory %s\n" cxt))
      )
))

;; Using getContext, load all the contexts that the code is
;; likely to need. For instance, the most basic
;; of contexts from Cadence are loaded first.

getContext("skillCore")
```

```
getContext("dbRead")
getContext ....

;; After loading all the dependencies, build all local
;; contexts

buildContext("cap1")
buildContext("cap2")
buildContext("capn")

exit
;; end of file myContextBuild.il
```

Building the Contexts

At this point everything is ready to build the contexts in a special call to the Cadence executable. Let us assume that the executable is called *cds*. The following line can be typed at the UNIX command line or can be made part of a make file to be called during the normal integration cycle.

```
cds -ilLoadIL myContextBuild.il -nograph
```

The output from the *cds* call can be piped into a file if a record of the build is needed.

ilLoadIL Option

The *-ilLoadIL* option causes the executable to switch into a special mode for context building and to allow itself to first read and evaluate the SKILL file before doing any of the normal operations for initializing the executable. That is why a call to the *exit* command is the last thing the file *myContextBuild.il* performs. It is meaningless to do more with the system after that.

nograph Option

The second option *-nograph* causes the executable to run with all graphics turned off. This option is useful if the integration is done on a machine that does not have X running. This option is not necessary for context building to work.

Initializing Contexts

Certain process-dependent constructs cannot be saved into a context. In the following example, a file port needs to be opened as part of the initialization phase of the code to be loaded. Use

```
myFile = outfile("/tmp/data")
```

The contents of the *myFile* symbol are not saved into a context because they would be meaningless when loaded into a different session. You need to define a special initialization function to initialize variables or structures that can only be initialized in the current session.

You can use the *defInitProc* function to define an initialization for each context. Let us assume a capability called *cap1* that opens a file and starts a child process. If the code for the capability needs to go into a binary context file, you need an initialization function to set up the file and the child process in the current session. The arrangement is as follows:

```
procedure(cap1InitProc()
;;---------------------
;; This procedure initializes two global variables: myFile
;; and myChild

      myFile = outfile("/tmp/data")
      myChild = hiBeginProcess(..)
)

;; The next line of code designates the above function
;; to be the initialization code for the context cap1.

(defInitProc "cap1" 'cap1InitProc)

;; end sample code
```

The call to *defInitProc* must be executed when the code is loaded, that is, it should not be included in a procedure but rather it should be placed at the top-level of a file to be executed during the loading of the file.

To summarize the work so far, you have seen how to

- Organize the source code
- Set up code to initialize the context
- Write the SKILL code to drive the building of contexts

Loading Contexts

Context loading can be done in two ways, depending on the need for a particular context.

Loading Contexts at the Start of a Session

If the code in the context is needed at the start of a session, use the *.cdsinit* file to force the loading of the required contexts by adding the following lines to the *.cdsinit* file.

```
loadContext(strcat(prependInstallPath("etc/context/")
                   "cap1.cxt")
callInitProc("cap1")
```

These two lines are needed for every context that is force-loaded by a call to *loadContext*. The call to *callInitProc* is needed to cause the initialization function for the context to be called. The basic *loadContext* function does not automatically do that.

Loading Contexts on Demand

Another option for contexts is to use the auto-load mechanism. This mechanism forces the context to be loaded on demand. There are two steps to achieving this.

- Generate an auto-load file.
- Load the auto-load file in the *.cdsinit* file.

The context is loaded automatically whenever one of its functions is called. To generate the auto-load file, first isolate all the entry point functions of a certain capability/context. When these functions are known, the contents of the auto-load file should look as follows.

```
;; This is the auto-load file for cap1. func1 through funcN
;; are the entry point functions for the capability cap1.
;; Call this file cap1.al

foreach(x '(func1 func2 func3 ..funcN)
           (putprop x "/<user>/etc/context/cap1.cxt" 'autoload))
;; end of auto-load file
```

The call to the *putprop* function causes the symbol name of the function to have the *autoload* property assigned a string value corresponding to the name of the context, with full path, to which the function belongs. This is how the evaluator makes the connection between function name and context file. The autoload mechanism always looks for a context file to autoload under */your_install_dir/tools/dfII/local/context* directory. If you choose to place the file under this directory, you don't need to give the property a full path; just the name of the context file is enough.

When the function is called during a normal session, and it does not have a function definition, the evaluator force-loads and initializes the context at that point. It is important to stress that the auto-load mechanism automatically calls the initialization function associated with the loaded context. After the context is loaded and the symbol gets a function definition, the evaluator calls the function and continues with the session.

Now that the auto-load file has been created, it can go anywhere the *.cdsinit* file can find it. You can place this file anywhere you choose, provided it is loaded at startup. The entry in the *.cdsinit* file needed to load the auto-load file is as follows.

```
load( "/your_install_dir/tools/dfII/local/context/cap1.al")
```

Working with a Menu-Driven User Interface

Some applications or capabilities have a menu-driven user interface. This implies that potentially the context for a capability does not have to be loaded, but the menus from which the capability is driven have to be put in place. In this case, the auto-load file (*cap.al*) must be augmented with the necessary code to create and insert menus in the appropriate places, such as banners.

When the auto-load file is loaded during the initialization phase of the executable, the desired menus appear and the callbacks for menu entries have the auto-load property set as explained above. The effect is that when menu commands are selected, the callback forces the corresponding context to load and execute the selected function. It is always safer to create and insert menus before adding the auto-load property on the function symbols.

Customizing External Contexts

You might want to customize the loading of externally supplied contexts. Consider the following example. Whenever the schematics context is loaded in an instance, a block of customer code needs to execute to set up special menus or load extra functionality. This can be accomplished using the *defUserInitProc* function. This function takes the following arguments.

- A string denoting the name of the capability and context to customize
- A symbol denoting the user-defined callback function to trigger whenever the named context is loaded.

```
defUserInitProc("schematic", 'mySchematicCustomFunc)
```

The *defUserInitProc* call causes the *mySchematicCustomFunc* function to be called *after* the context for schematic is loaded and initialized. Only a single customizing callback function can be added for each capability context. You can centralize your customizing for each context in a single predefined function and associate the function with a context using *defUserInitProc*.

The *defUserInitProc* call is similar to the *defInitProc* function that initializes a context. The context loading mechanism calls the function defined by *defInitProc* first and then calls the function defined by *defUserInitProc*.

Potential Problems

Binary context files are built incrementally. This can introduce inter-context errors, which means that references are made in one context to values outside its own space. Therefore, when the contexts are loaded independently, those values become meaningless. Because

the SKILL language is dynamically scoped, excessive use of global data structures and vague boundaries around the program and data spaces of applications can result in this type of error. SKILL programmers can practice caution by modularizing their code and by using strict naming conventions for their symbol names.

When inter-context errors are detected during context building, they are flagged but the context continues to build, leaving the values indicated as crossing context boundaries to be *nil*. The following sample cases of code cause inter-context errors.

Sharing Lists Across Contexts

Consider the following sample code involving two contexts.

```
;; Code in context 1. This code builds list structures by
;; sharing sub-lists

field = list('nil '_item "hello world" 'item2 22)
form1 = list('xxx field)
form2 = list('yyy field)

;; Code in context 2. Also building list structures but
;; sharing sub-lists from context1

form3 = list('zzz field)

;; end sample code
```

In the three form variables, the *field* structure is shared. However when contexts are saved separately, the list structure for the *field* part of *form3* is not saved with the data in context 2, because it is outside the context's bounds. If the two contexts were created in the sequence given, an error would be flagged indicating the part of the list in context 2 that is crossing boundaries with context 1. Both contexts would be named and the lists involved displayed.

If each context maintained the lists it needs within the context boundary, this would solve the problem. In this case, if the contents of the variable *field* are needed, a *copy* should be performed in context 2 to get a copy of the list locally to that context.

The Conditional eq Failure

The context saving algorithms attempt to smooth out the inter-context problems by copying atoms across context boundaries. For instance:

```
;; Code in context 1. Here a function is defined such that
;; when called it constructs and returns a list.

procedure( foo()
      list(42 "hello world" 42)
)
var1 = foo()
```

```
;; Code in context 2. A call to foo is made and the result is
;; stored in var2

var2 = foo()

(eq (cadr var1) (cadr var2))     ;; Normally returns t
(equal (cadr var1) (cadr var2))  ;; Normally returns t

;; end sample code
```

When separate contexts are generated incrementally for the code above and you load both contexts in a session, the call to *eq* returns *nil* indicating the two strings “hello world” in the two separate lists pointed to by *var1* and *var2* are not the same instance or pointer. That is because the string “hello world” was copied into context 2 when the binary context for context 2 was saved. This allows the list pointed to by *var2* to remain complete. Such copying is only done on atomic structures, such as integers or strings.

This condition is not flagged by any errors at context build time.

Execution During Load

The SKILL code executed during the loading of *startup.il* executes all top-level statements (that is how procedure definitions are done). It was explained earlier that certain data types cannot be saved into a context and have to be regenerated using a context-specific initialization function. There are process-related executions (that is, not necessarily data-related) that have to occur during the context initialization phase. For example, consider the following.

```
;; Assume the function isMorning is a boolean that calls
;; time related internal functions and returns t if the
;; current time is AM.

if ( isMorning()
then greeting = "good morning"
else greeting = "good afternoon")
form->title->value = greeting

;; end of sample
```

When the context is built using the code above, the *isMorning* function is executed and the *greeting* is set correctly. The value of *greeting* and the value of *form->title->value* are “frozen” in the context file. On subsequent loads of the context, the *isMorning* function will not execute so the greeting will take the value given to it at the time the context was built. To remedy this situation, the code above should be made part of the initialization function for a context.

Context Building Functions

Saving the Current State of the SKILL Language Interpreter as a Binary File (saveContext)

saveContext saves the current state of the SKILL language interpreter as a binary file. This function is best used in conjunction with *setContext*. *saveContext* saves all function and variable definitions that occur, usually due to file loading, between the calls to *setContext* and *saveContext*. Those definitions can then be loaded into a future session much faster in the form of a context using the *loadContext* function.

By default all functions defined in a context are read and write protected unless the *writeProtect* system switch was turned off when the function was defined between the calls to *setContext* and *saveContext*. If the full path is not specified, this function uses the *SKILL path* to determine where to store the context file name.

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

Saving Contexts Incrementally (setContext)

setContext allows contexts to be saved incrementally, creating micro contexts from a session's SKILL context. To understand this, think of the SKILL interpreter space as linear; the function call *setContext* sets markers along the linear path. Any SKILL files loaded between a *setContext* and a *saveContext* are saved in the file named in the *saveContext* call. This function can be used more than once during a session.

Loading the Context File into the Current Session (loadContext)

loadContext loads the context file into the current session. You should always fully qualify the file name. The default directories will be searched for the file if the name is not fully qualified. The context file must have been created using the function *saveContext*. For example:

```
loadContext( "/usr/mnt/charb/myContext.cxt" )
```

Loads the *myContext.cxt* context.

It is advisable to not load context files supplied by Cadence using *loadContext*. It is better to rely on the autoload mechanism for context files you do not own.

Calling Initialization Functions Associated with a Context (callInitProc)

callInitProc takes the same argument as *loadContext* and causes the initialization functions associated with the context to be called. This function need not be used if the loading of the context is happening through the autoload mechanism. Use this function only when calling *loadContext* manually. For example

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

Functions defined through *defInitProc* and *defUserInitProc* are called.

Registering an Initialization Function for a Context (defInitProc)

defInitProc registers an initialization function associated with a context. The initialization function is called when the context is loaded.

defInitProc always returns *t* when set up. The initialization function is not actually called at this point, but is called when the associated context is loaded.

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

Registering a Function for Contexts You Don't Own (defUserInitProc)

defUserInitProc registers a user defined function that the system calls immediately after loading and initializing a context. For instance, this function lets you customize Cadence supplied contexts. Generally, most Cadence-supplied contexts have internally defined an initialization function through the *defInitProc* function. *defUserInitProc* defines a second initialization function, called after the internal initialization function, thereby allowing you to customize on top of Cadence-supplied contexts. The call to *defUserInitProc* is best done in the *.cdsinit* file.

defUserInitProc always returns *t* when set up. Note that the initialization function is not actually called at this point, but is called when the associated context is loaded.

```
defUserInitProc( "someContext" 'myInitSomeContext) => t
```

Autoloading Your Functions

Autoloading is the facility through which SKILL code can be loaded dynamically. That is, the code supporting a capability is not loaded until that capability is needed. The load is usually triggered by a call to a function that is undefined in the current session. The evaluator calls on the loader to locate and load the code for the function. You should use autoloading to tune the amount of code loaded in a session to that only needed for that session.

The autoloader follows these rules:

- If there is a property on the function being called with the symbol "autoload" and the value of the property is a string denoting the name of a context (with the extension *.cxt*), the loader looks for the file under */your_install_dir/tools/dfII/etc/context* (this is where Cadence-supplied contexts are stored) and */your_install_dir/tools/dfII/local/context*.
- If the value of the autoload property is a string denoting the name of a file with a full path and the file is a context file (*.cxt* extension), the context is loaded. If the extension is anything other than *.cxt*, *loadi* is called with the file name as its argument.
- If the value of the autoload property is an expression, the expression is evaluated. The expression is responsible for taking the necessary steps to define the function triggering the autoload.

Encrypting and Compressing Files

Encrypting and compressing files allows distribution of SKILL code in a manner that an end user cannot read the code. This is an alternative method to contexts and is intended for small sets of SKILL code that need protection.

Encrypting a File (encrypt)

You can encrypt SKILL programs and data files. These can subsequently be reloaded using the *load*, *loadi*, or *include* functions. *encrypt* encrypts a file and places the output into another file. If a password is supplied, the same password must be given to the command used to reload the encrypted file.

```
encrypt( "triadb.il" "triadb_enc.il" "option") => t
```

Encrypts the *triadb.il* file into the *triadb_enc.il* file with *option* as the password. Returns *t* if successful.

Reducing the Size of a File (compress)

You can compress SKILL files to remove unnecessary blank spaces and comments from the file. *compress* reduces the size of a source file and places the output into a destination file.

Compression renders the data less readable because indentation and comments are lost. It is not the same as encrypting the file because the representation of destination file is still in ASCII format.

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

Protecting Functions and Variables

The following functions get and set the write-protect status bit on functions and variables. These functions can be used to secure the definitions of functions and the contents of variables when delivering products.

The easy way to write protect functions is to use the status flag *writeProtect*. Once turned on, all subsequent function definitions will be protected.

Explicitly Protecting Functions

Protecting functions on a per-function basis is an alternative to *sstatus* which protects functions on a per-context basis.

Setting the Write-Protect Bit on a Function (setFnWriteProtect)

setFnWriteProtect sets the write-protect bit on a function.

- If the function has a function value, it can no longer be changed.
- If the function does not have a function value but does have an autoload property, the autoload is still allowed. This is treated as a special case so that all the desired functions can be write-protected first and autoloaded as needed.

This example defines a function and sets its write protection so it cannot be redefined.

```
procedure( test() println( "Called function test" ))
setFnWriteProtect( 'test ) => t

procedure( test() println( "Redefine function test" ))

*Error* def: function name already in use and cannot be
      redefined - test

setFnWriteProtect( 'plus ) => nil
```

Returns *nil* because the *plus* function is already write protected.

Finding the Value of a Function's Write-Protect Bit (getFnWriteProtect)

getFnWriteProtect returns the value of the write-protect bit on a function. The value is *t* if the function is write-protected or *nil* otherwise.

```
getFnWriteProtect( 'strlen ) => t
```

Protecting Variables

Setting the Write-Protect Bit on a Variable (setVarWriteProtect)

setVarWriteProtect sets the write-protect on a variable. Use this function only when the variable and its contents are to remain constant.

- If the variable has a value, it can no longer be changed
- If the variable does not have a value, it cannot be used
- If the variable holds a list as its value, that list can no longer be changed. For example, if the list was a disembodied property list, attempting to modify the value of properties will fail

```
y = 5                                   ; Initialize the variable y.
setVarWriteProtect( 'y )=> t            ; Set y to be write protected.
setVarWriteProtect( 'y )=> nil          ; Already write protected.

y = 10                                  ; y is write protected.
*Error* setq: Variable is protected and cannot be
                          assigned to - y
```

getVarWriteProtect

Returns the value of the write-protect on a variable.

```
x = 5
getVarWriteProtect( 'x )=> nil
```

Returns *nil* if the variable x is not write protected.

Global Function Protection

Write-protecting code has two benefits. First, it renders the code secure from tampering. Second, write-protected code is saved in memory segments that incur no garbage collection costs.

To turn on global protection for functions saved in a context, be sure the following line is in your *makeContext* utility function described earlier:

```
sstatus( writeProtect t)
```

However, if certain pieces of code need to have write-protection turned off (for example, when a function is user definable), use the following method.

```
;; Sample code showing how to turn off write-protection
;; The following let statement should surround all
;; that is to have write-protection turned off.
```

```
let( ((priorStatus (status writeProtect)))

      ;; Turn off write-protection
      (sstatus writeProtect nil)

      ;; Body of code to have no write-protection
      procedure( proc1() ...)
      procedure( proc2() ...)
      ...etc.

      ;; Turn write-protection status back to what it was
           (sstatus writeProtect priorStatus)
) ;; end-of-let
;; end of sample
```

The call to *sstatus* takes only the values *t/nil* as its second argument.

11

Writing Style

Overview information:

Introduction

Good style in any language can lead to programs that are understandable, reusable, extensible, efficient, and easy to maintain. These attributes are even more desirable to have in programs developed in an extension language like the Cadence® SKILL language because the programs tend to change a lot and the number of contributors are many.

Useful SKILL programs, or pieces of them, get copied and passed around via e-mail or bulletin boards. Hence, it is important to consider the following when writing SKILL programs.

- Be specific, concise, and most importantly consistent.
- Anticipate a novice reader's questions and document the code accordingly.
- Be conventional. Using fancy techniques that are generally possible in a Lisp-based language, yet generally not well understood, might not be a good idea (unless you are doing it to gain in performance or reduce memory use).
- Avoid too many global states and direct access to them. Use abstractions.
- Make functional interfaces non-modal. That is, try and not predicate your interfaces on a global state or global mode such that a second user of the interface does not experience unpredictable behavior. For example, it is bad style to have a procedure that puts an editor in "insert" mode and uses subsequent calls to insert objects into the design. Better to have one call that takes a list of objects to be inserted.

BAD	**GOOD**
EDstartInsert(database)	EDinsert(database '(obj1 obj2 obj3))
EDinsert(obj1)	
EDinsert(obj2)	
EDinsert(obj3)	
EDendInsert(database)	

As in all programming languages, the layout of code within a file can greatly affect the readability, and therefore maintainability, of the code. The code examples in this manual use a layout that is both intuitive and easy to understand, but layout is really a matter of taste.

The rest of this chapter describes specific situations where coding style issues are important.

Code Layout

There are no absolute rules about layout of code in SKILL or in any other programming language, but there are certain guides which can help the programmer to write more readable code. The readability of the code is the most important aspect of the code layout.

Comments and Documentation

There are two methods of commenting available in SKILL. You can use either the ';' or the '/*...*/' structure, providing they are used consistently. Because '/*...*/' comments cannot be nested, some programmers find them easier to see.

The more comments within a piece of code, the easier it is to understand and maintain. However, the more comments there are, the more that have to be kept consistent with the code. Comments which are misleading because they have not been maintained along with the code are possibly worse than not having any comments at all.

Document your data structures. Don't just give details; try and describe the motivation and effect of the structure on the algorithm. Well designed and documented data structures can tell a lot about the nature of the program.

If you are reading someone else's code and find it inadequately documented, write down your questions as comments. They may get answered or will encourage other readers to persevere.

As you develop a program, always maintain a "todo" list. This list will be a great help for future reference. For example, a "todo" comment around a dubious looking piece of code explaining your misgivings may help another developer track a bug in that code.

Generally, if you find that you need to write convoluted comments about a convoluted algorithm, rewrite the algorithm. A lot of times good, strong code is self documenting.

Things to Comment and Document

As a guideline, it is suggested that the following are commented:

- **Code Modules.** Each code module should have a header containing at least the author, creation date, and change history of the module, plus a general description of the contents.
- **Procedure Definitions.** Precede procedure definitions with a block comment detailing the functionality of and interfaces to the procedure. Also add a help string inside the procedure (see definition of *procedure*). Help systems will extract this information for

display. As much as possible add type templates for procedure arguments. This helps catch erroneous use, and type information can be extracted by help systems.

- **Data structures.** Describe contents and impact on algorithms.
- **Complex Conditionals.** Comment any test within a conditional function that is non-trivial to indicate the pass/fail conditions.
- **Mapping Functions.** Comment complex mapping functions to state what the return values are.
- **Terminating Parentheses.** Where a function call extends over several lines, label the closing parenthesis with the function name.

Things Not to Comment and Document

As a guideline, it is suggested that the following are *not* commented in production code:

- **Long change details.** Full details of changes made should be maintained by the source code control system, and only a brief outline included in the module header. They should not be present in the body of the code.

  It is the function of source code control systems, such as RCS or SCCS, to maintain details of changes to the code. Such details should not be maintained in the code itself, because they can make maintenance of the code more rather than less difficult.
- **PCR details.** Product change request information should be updated on the PCR itself.

Inclusion and number of comments in no way affects the performance of the code once it has been read into the SKILL interpreter.

Function Calls and Brackets

Function calls in SKILL can be written in two distinct ways, with the opening parenthesis either before the function name, as in Lisp, or after it, as in C. Once again, the method chosen is not as important as ensuring that it is chosen consistently. However, putting the parenthesis after the function name does make it easier for a non-Lisp programmer to read the code. It is also easier to distinguish between function names and arguments, and between function calls and other lists.

When a function call extends over more than one line in a file, it is recommended that the closing parenthesis is aligned, on a separate line, with the beginning of the function name.

This makes it is easier to see where a particular function call finishes and has the added advantage that missing parentheses are easier to see.

Having extra newlines, to allow alignment of function arguments or parenthesis, will in no way affect the performance of the code once it has been read into the SKILL interpreter.

Avoid Using a Super Right Bracket

Using the super right bracket is strongly discouraged, except when using the interactive interpreter, because

- Missing parenthesis become difficult to locate, which can lead to great confusion.
- Inserting another function call around the code containing the super right bracket (]) is difficult.
- Bracket-matching procedures within editors cannot cope with this.

One method of helping to ensure that brackets are not missed out when writing SKILL code is to always insert the closing parenthesis immediately after the opening parenthesis of a function call, and then to go back and fill in the arguments.

Brackets in SKILL Are Always Significant

- In C it is possible to insert brackets where they are not really needed but not affect the functionality of the program.
- In SKILL, the insertion of extra brackets can be, and usually is, incorrect.

The problem usually occurs with infix operators. It is important to remember that in SKILL, as in Lisp, every function evaluates to a list whose head is the function name. Thus code such as *a + b* is held internally as the list *(plus a b)*. It is therefore important for the SKILL programmer to understand the relative precedence of the in-built SKILL functions. As an example, consider the following code:

```
a = b && c
```

This is in fact held as the list:

```
(setq a (and b c))
```

rather than:

```
(and (setq a b) c)
```

This is exactly what would be expected in any language, but is not necessarily what the user wants. Consider the following:

```
if(res = func1(arg) && res != no_val then ...)
```

Clearly, what the writer meant to do was to call the function *func1*, storing the result in the variable *res*, and checking that the result was not *no_val*. In fact, the value assigned to *res* is the result of *func1(arg) && (res != no_val*). On the other hand, using too many parentheses can cause the code to fail. The code should be:

```
if((res = func1(arg)) && res != no_val then ...)
```

For example, the following statement has too many levels of parentheses around the function calls:

```
a = ((func1(arg1)) && (func2(arg2)))
```

Parentheses in function calls are only optional for the in-built unary and infix operators, such as '!' and '+'. For example, the following are equivalent:

```
!a
!(a)
(!a)
```

Commas

Commas between function arguments, and list elements, are optional in SKILL. Programmers from a C programming background will probably want to insert commas, those from a Lisp background probably will not. The general recommendation is that commas should not be used.

Using Globals

The use of global variables in SKILL, as with any language, should be kept to a minimum. The problems are greater in SKILL however because of its dynamic scoping of variables.

Misusing Globals

The problem with global variables is that different programmers can be using a variable with the same name. This type of "name clash" can cause problems that are even more difficult to isolate than the "Error Global" problem, because programs can fail because of the order in which they have been run. However, because this problem can usually be easily avoided by adopting a standard naming scheme, SKILL Lint will report this type of variable as a "Warning Global". To illustrate the problem, consider the example code below:

```
/*******************************************************
 * myShowForm()
*******************************************************/

procedure( myShowForm()
      /* If we don't already have the form, then create it. */
      unless(boundp('theForm) && theForm
            myBuildForm()
      )
      /* Display the form. */
      hiDisplayForm(theForm)
) /* end myShowForm */
/*******************************************************
 * yourShowForm()
*******************************************************/

procedure( yourShowForm()
      /* If we don't already have the form, then create it. */
      unless(boundp('theForm) && theForm
            yourBuildForm()
      )
      /* Display the form. */
      hiDisplayForm(theForm)
) /* end yourShowForm */
/*******************************************************
 * myBuildForm()
*******************************************************/

procedure( myBuildForm()
      hiCreateForm('theForm,
             "My Form"
             "myFormCallback()"
             list( hiCreateStringField(?name 'string1,
                                ?prompt "my field")
             ) /* end list */
 )       /* end hiCreateForm */
) /* end myBuildForm */
/*******************************************************
 * yourBuildForm()
 *
 * Build the form.
*******************************************************/

procedure( yourBuildForm()
      hiCreateForm('theForm,
             "Your Form"
             "yourFormCallback()"
             list( hiCreateStringField(?name 'string1,
                                ?prompt "your field")
             ) /* end list */
      ) /* end hiCreateForm */
) /* end myBuildForm */
```

If *myShowForm* is called before *yourShowForm*, the global variable *theForm* will be set to a different value than if *yourShowForm* is called before *myShowForm*.

Common Coding Style Mistakes

This section describes mistakes commonly made by C programmers in SKILL. They fall more in the coding style category although some of them have a minor impact on performance.

Inefficient Use of Conditionals

A common mistake with conditional checks is to use multiple inversions and boolean checks when using De Morgan's Law would result in a simpler and clearer test. For example, the code below shows a common form of check.

```
if( !template || !templateDir
      warn("Invalid templates\n")
) /* end if */
```

This check can be optimized using De Morgan's Law to be:

```
if( !(template && templateDir)
      warn("Invalid templates\n")
) /* end if */
```

Another common mistake is made by many programmers used to having only the standard *if* and *case* conditionals. SKILL provides a rich variety of conditional functions, and appropriate use of these functions can lead to much clearer and faster code. For example, an *unless* could be used in the above check to yield the clearer and more efficient:

```
unless( template && templateDir
      warn("Invalid templates\n")
) /* end unless */
```

Another example is where multiple if-then-else checks are used, such as:

```
if(stringp(layer) then
      layerName = layer
else
      if(fixp(layer) then
            layerNum = layer
      else
            if(listp(layer) then
                  layerPurpose = cadr(layer)
            ) /* end if */
      ) /* end if */
) /* end if */
```

This can be more clearly and efficiently implemented using a *cond*:

```
cond(
      (stringp(layer)  layerName = layer)
      (fixp(layer)     layerNum = layer)
      (listp(layer)    layerPurpose = cadr(layer) )
) /* end cond */
```

When applying multiple tests, either in a nested *if* function or a *cond* function, it is important to consider the order in which the tests will be carried out.

- If one test is more likely to be true than the others then it should go first.
- If all tests are equally likely to be true, then the test that involves the most work should go last.

Misusing prog and Conditionals

Quite often, *prog* statements are used to return from a procedure when an error condition occurs. In the example below, *template* and *templateDir* are both verified, and only if both are correct is the rest of the procedure executed:

```
procedure( EditCallback()
      prog( ( templateFile templateDir fullName )
            templateFile  = ReportForm->Template->value
            templateDir   = ReportForm->TemplateDir->value
            /* Check the template directory. */
            if( ((templateDir == "") || (!templateDir)) then
                  return(warn("Invalid template directory.\n"))
            )
            /* Check the template file. */
            if( ((templateFile == "") || (!templateFile)) then
                  return(warn("Invalid template file name.\n"))
            )
            /* If both are correct then act.*/
            sprintf(fullName "%s/%s" templateDir templateFile)
            if(isFile(fullName) then
                  LoadCallback()
            else
                  SetUpEnviron()
            ) /* end if */
            return(hiDisplayForm(EditTemplateForm ))
      ) /* end prog */
) /* end EditCallback */
```

Using the fact that a *cond* returns the value of the last statement executed allows us to more efficiently implement this example using a *let*:

```
procedure( EditCallback()
      let(((templateFile ReportForm->Template->value)
             (templateDir  ReportForm->TemplateDir->value)
              fullName)
            cond(
                  /* Check the template directory. */
                  ( (templateDir == "") || (!templateDir)
                        warn("Invalid template directory.\n")
                  )
                  /* Check the template file. */
```

```
                ( (templateFile == "") || (!templateFile)
                      warn("Invalid template file name.\n")
                )
                /* If both are correct then act. */
                (t
                      sprintf(fullName "%s/%s" templateDir
                                        templateFile)
                      if(isFile(fullName) then
                            LoadCallback()
                      else
                            SetUpEnviron()
                      ) /* if */
                      hiDisplayForm(EditTemplateForm )
                    )
          ) /* end cond */
      ) /* end let */
) /* end of EditCallback */
```

Red Flags

The following are situations or functions to watch out for. Often their use is symptomatic of problems with the way interfaces or algorithms are designed. In some cases their use is legitimate and these comments do not apply.

Any Use of *eval* or *evalstring*

Strictly speaking these calls are inefficient and any code using them is either suffering through using a bad interface from another application or the code itself is badly designed.

Excessive Use of *reverse* and *append*

Excessive use of *reverse* and *append* is an indication that algorithms using list structures are badly designed. They are acceptable when prototyping but production code should not suffer their consequences. Both these functions are capable of generating a lot of memory. There are alternatives to these functions and the recommendation is that code using these functions should be rewritten using *tconc*.

Excessive Use of *gensym* and/or *concat*

Symbols are large structures and applications that have to generate symbols at run time may not be designed to use the right data structure. Many times applications use symbols in place of small strings to save on memory because symbols are unique in the system. However, SKILL caches strings so this optimization might not always yield the desired effect.

Overuse of the Functions Combining *car* and *cdr*

For example, using *cdaddr*. These are not as intuitive as calls to *nthelem* and *nthcdr* and can lead to programmer and reader errors.

Using *eval* Inside Macros

Macros should in general massage expressions and return expressions. Calling an *eval* inside the macro means that you are determining the value of an entity at compile time as opposed to evaluating it at run time. This may be inadvertent and may yield to undesirable behavior. See "Macros" on page 212.

12

Optimizing SKILL

Overview information:

Introduction

The need to optimize arises when there is a perception that code is not running as fast as it should. As a programmer, you should always be aware of when, what, and how optimizing is carried out.

Focus Your Efforts

Generally, you should make the program work before you think about optimizing it. Optimizing for performance when the program has not met the stated functional requirements is a waste of time. This chapter

- Only touches upon techniques for tweaking your programs to run faster
- Does not teach you how to create better algorithms on the whole (that kind of performance is designed into the algorithms from the outset and no amount of tweaking can help)

Before you begin to think about optimizing, find out what needs to be optimized. A general rule of thumb is that 80% of the time is spent in 20% of the code. To be effective, optimizing should be done on that small portion of the program that is the performance bottle-neck.

Always measure the benefit gained after you tweak the code. It is better to leave well written and well structured code untouched if the changes did not yield significant speed up.

Use Profiling Tools

Use tools provided to measure performance. Programs written in the Cadence® SKILL language can be optimized on two fronts: time and memory.

Where Is Program Time Spent?

To find out where most of the time is spent use the profiling tool described in SKILL development environment. This is a good way to perform global statistical gathering of time spent in functions called in a given session. The profiler takes a sample of the runtime stack at pre-specified intervals and presents timing information as call graphs identifying the critical paths. Use this information to direct your effort.

You can use *measureTime* to evaluate specific expressions and get timing results. You can also add your own more deterministic instrumentation to the code to collect relevant information about your algorithms and structures.

How Is Memory Used?

Memory usage can be tracked by the same profiling tool described above but in a memory profiling mode. The information is presented graphically. Remember that SKILL has an automatic memory manager and programs can be written to generate excessive amounts of memory. Before you even optimize for time make sure to profile memory usage to see if that is where you need to spend your effort. For instance a good indicator that memory usage needs tweaking is if the function *gc* (garbage collector) appears high among functions profiled for time

Profiling and performance data gathering are operations intrusive in nature, so expect reduced execution speed. Be sure the measurements you gather take this into account. It is advisable to think of performance of code in percentages as opposed to absolute values when using profiling tools. Refer to "Printing Summary Statistics" on page 222 for information on the how to obtain and interpret a summary of memory allocation.

Optimizing Techniques

This section will describe specific techniques you can use to optimize your code. It is recommended that you experiment and measure performance gained as not all the techniques described are effective in all circumstances.

Macros

Use macros to in-line function calls see "Macros" on page 212. In most situations replacing function calls by macros can improve the performance of code. However, because macros are in-line expanded, they can grow the size of the code at runtime. So there is a balance as to what kind of functions can be written as macros.

For example, functions small in size that are likely to be used a lot are good candidates. Expressions that can be reduced at compile time leaving smaller subexpressions to be evaluated at run time are also good candidates.

Caching

Caching is a technique used to save the results of costly computations in a fast access cache structure. You have to balance the benefit of time saved versus amount of memory used for the cache structure.

A good data structure to use for caches is the association table. For example, if you called a compute-intensive function, say factorial or fibonacci, many times in a session, you might

consider caching the results so a second call to the function with same argument will run faster. To do this, you first need to create an assoc table and store it as a property on the symbol for the function, for example:

The fibonacci function described in "Fibonacci Function" on page 358.

```
myFib.cache = makeTable("fibonacciTable" nil)
procedure( myFib(num)
(let ((res myFib.cache[num]))
      cond( !res myFib.cache[num]= fibonacci(num))
            ( t res)
))
```

You could write the function *myFib* as a macro:

```
defmacro( myFib (num)
      'or(myFib.cache[,num]
            myFib.cache[,num] = fibonacci(,num)
))
```

For example, compare the following timing numbers (in seconds).

```
            fibonacci(25)myFib(25)
1st call 23 23
2nd call 23 0.009
```

Mapping and Qualifying

Mapping functions (*map, mapcar, maplist*, and so forth) have been described in detail in "Advanced List Operations" on page 177 When manipulating lists, you can achieve significant performance gains if you use map* functions instead of loops that directly manipulate the lists. The same argument applies to qualifiers like *foreach*, *setof* and *exists*. The qualifiers are generic in nature in that they apply to lists as well as to association tables.

Consider the following two examples performing the same operation of adding consecutive numbers in two equal length lists and returning a list of the results. The example using *mapcar* is at least twice as fast.

```
L1 = '(1 2 3 4 5 ... 100)
L2 = '(100 99 98 .. 1)
mapcar(lambda((x y) x+y) L1 L2)

(let (res)
      while(      car(L1)
                  res=cons(car(L1)+car(L2) res)
                  L1 = cdr(L1)
                  L2 = cdr(L2)
      ) res
)
```

Write Protection

Ensuring that all of your functions and data structures that are static in nature are write protected reduces the amount of work the garbage collector has to perform whenever it is triggered. Generally, all functions can be write protected because they are not likely to be over-written at run time. Some data structures, like lookup tables, whose contents are not likely to be modified at run time, can also be write protected.

It is highly recommended that SKILL code destined for production be packaged in contexts and that all contexts are built with the status flag *writeProtect* set to *t* (see "Protecting Functions and Variables" on page 241). To set write protection on global variables use *setVarWriteProtect*.

When SKILL memory is write protected, the garbage collector does not touch it, thus considerably reducing the amount paging and work done at run time.

Minimizing Memory

The way memory is used in SKILL is by *cons*ing (the basic list building operation), creating strings, arrays, and so forth, or by generating instances of defstructs and user types. The goal of memory optimizing is to reduce the overall amount of memory used. This reduction

- Saves on run-time page swapping that an operation has to perform when physical memory resources are scarce
- Reduces the work load on the garbage collector

Run the SKILL Profiler in Memory Mode

To start optimizing memory usage, use the SKILL profiler in memory mode to discover the functions responsible for the largest amount of memory used. From there start tweaking the functions. You should be aware of the nature of the utilities and library calls you use.

For example, removing elements from lists using *remove* makes a copy of the original list minus the element removed. You should check to see if that is the desired behavior. If you can use a destructive remove instead, such as *remd*, you can cut down memory use considerably on this particular operation. Experiment.

Ways to Minimize

When generating large data structures in memory from information on disk, you should ask yourself whether you need to generate the whole image in memory if all of it is not likely to be

used. Use the technique known as lazy evaluation to expand your structures on demand rather than at start-up.

For example, you can embed *lambda* constructs in your data structures to retrieve information on demand (see "Declaring a Function Object (lambda)" on page 210 for a description of *lambda* constructs). Experiment.

If you know from the outset the amount of memory your application is likely to need at run time, you can preallocate that memory to reduce the number of times the garbage collector is triggered. There is a delicate balance you have to make here between total memory allocated and garbage collection.

Remember, preallocating memory is **not** a substitute for fine tuning memory use in your program.

Only preallocate memory when you have tuned the program and determined the minimum amount of memory needed to run efficiently. Read "Memory Management (Garbage Collection)" on page 220 to better understand the nature of automatic memory management.

General Tips

This section describes situations when optimizing can be applied.

Element Comparison

In SKILL, there are two basic functions used to compare values. These functions are *eq* and *equal* (also known as the infix operator '==').

It is important to understand the difference between these two functions, because both are useful in particular circumstances. There are several functions in SKILL which have alternative implementations depending on whether the user wants to compare using the *eq* or *equal* function. For example, the two functions *memq* and *member* are used to search a list for an object. *memq* uses the *eq* function for comparison and *member* uses the *equal* function.

The *eq* function is far stricter in its comparison than the *equal* function. There are objects which *equal* would consider to be the same, but which *eq* considers to be different.

The objects which can be compared successfully using *eq* are

- SKILL symbols
- Small integers (-256 <= i <= 1024)

- List objects (NOT their contents)
- Any pointers
- Characters (NOT strings)
- Ports

The important things that cannot be reliably compared by *eq* are strings and lists, unless they are identical objects referenced by the same pointer. In many situations SKILL tries to optimize memory use by caching certain objects and reusing them. For example, there is a string caching mechanism that saves SKILL from generating the same string multiple times. Code and data segments in static (write-protected) memory are also cached so they are reused within the static space.

The following are some examples of the more unexpected differences between the comparison operations:

```
x = '(1 2 3)
y = '(1 2 3)
eq(x y)              => nil
equal(x y)           => t

s1 = "string"
s2 = "string"
s3 = s2
eq(s1 s2)            => t    ; unreliable
equal(s1 s2)         => t    ; reliable
eq(s3 s2)            => t    ; reliable
equal(s3 s2)         => t    ; reliable

eq(12345 12345)      => nil
l = '(12345)
eq(car(l) car(l))    => t
l = '("string")
eq(car(l) car(l))    => t
```

To understand more about *equal* and *eq*, keep in mind how they are implemented. The following are pseudo-code definitions of the two functions (they are actually implemented in C):

```
eq(A B)
 if A is the same object as B
 then t
 else nil
end eq

equal(A B)
 if eq(A B)
 then t
 else
       if the contents of A and B are 'equal'
       then t
       else nil
end equal
```

Suppose the two functions are used to compare two SKILL objects, A and B. If A and B are in fact the same object then *eq* will immediately return *t*. Because the first thing that *equal* does is call *eq*, it too will immediately return *t* in this case. Now suppose that A and B represent distinct objects. In this case, *eq* will immediately return *nil*. *equal*, however, goes on to try to establish if the *contents* of the two objects are the same, (for example, if the objects are lists, *equal* compares each element of the two lists for equality) and this process involves a large overhead. To summarize this behavior:

```
eq('a 'a)            => t        (fast)
equal('a 'a)         => t        (fast)
eq('a 'b)            => nil      (fast)
equal('a 'b)         => nil      (SLOW)
```

If in doubt about which of *eq* and *equal* to use, observe the following rules:

- If the objects are simple (symbols, small integers, pointers, characters), use *eq* because *eq* is faster than *equal*.
- If the objects are compound or complex (lists or strings, for example), consider what functionality is needed. To test whether two strings contain the same characters, use *equal*; To test whether two strings are in fact the same *object*, use *eq*.

Some common tests can be more efficiently implemented by using the in-built SKILL functions, or simply by using the fact that in SKILL, any non-*nil* object is true, not just *t*. Examples of some simple transformations are:

Original Test	**Improved Test**
a == 0	zerop(a)
a == 1	onep(a)
strcmp(a,"teststring") == 0	a == "teststring"
a != nil	a
a == nil	!a
!null(a)	a

For *a != nil*, providing a Boolean value is all that is required, for example,

```
if( a != nil ) can be coded as if( a )
```

For *!null(a),* providing a Boolean value is all that is required, for example,

```
if( !null(a)) can be coded as if( a )
```

List Accessing

The basic list accessing operations of SKILL (*car*, *cdr* and so forth) are very fast, and their performance is very predictable. The *nth* and *nthcdr* functions are significantly faster than the equivalent number of basic operations (because they avoid procedure call overhead) and should be used if lists are long. The operation that should be used with the most care is the *last* operation. This function must traverse the entire list to find the last element, so a large overhead is incurred for long lists.

List Building

There are two main methods of building lists: iteratively, either as part of a program's operation or when all the elements are the same type and non-iteratively when the format and number of elements are known. List building is an area that is open to abuse, and it is important that the processes involved are clearly understood.

Iterative List Creation

The standard function for adding an element to the start of a list is the *cons* function, which is very efficient and has predictable performance. New users often find *cons* difficult to use because elements are added to the *front* of the list, giving a result that is "back to front". There are several methods of producing a list in the "right" order, which vary in efficiency:

Use *append1* to add each element to the end of the list rather than to the start.

This is the most inefficient method, and lists should never be iteratively created in this way. As each element is added, the *append1* function makes a copy of the original list, with the new element on the end. If a list of n elements is built using this method, then on the order of n^2 list cells are created, and most of them are promptly discarded again.

Use *nconc* to add each element to the end of the list rather than to the start.

This method is also quite inefficient, and lists should never be iteratively created in this way. Because *nconc* is a "destructive" append, only *n* list cells need to be created to form the list. However, on the order of n^2 list cells must be traversed to build the list.

Use *cons* to build the list backwards, and then use *reverse* to turn the list around.

This is a much more efficient, and easily understood method. To create a list of *n* elements, *2n* list cells are created, but half of them are immediately discarded.

Use the *tconc* structure and function to build the list in the right order.

This is the most efficient method in terms of storage requirements. To create a list of *n* elements, only *n+1* list cells are created. Because creation of list cells is relatively time consuming, this means that using *tconc* to build a long list is faster than using *cons* and *reverse*. A slight disadvantage is that the code is less intuitive.

In general, if the code is not time critical, it might be better to build the list backwards using *cons* and then apply *reverse*. If the code is in a time critical part of the program, then the *tconc* method should be used, along with some detailed commenting. From a memory usage point of view, if the list being built is long, then it is better to use the *tconc* method to prevent the garbage collector being called unnecessarily.

To build lists that are derived from existing lists, it is far better to use the mapping functions, possibly coupled with the *foreach* function. This is discussed in detail later.

Programmers coming from a Prolog or compiled Lisp background might be tempted to build a list in the right order using a tail recursive algorithm. SKILL has no tail recursion optimization so this method yields no performance gain.

Non-Iterative List Creation

To build lists of known format, use one of the *list*, *quote*, or *backquote* functions as follows:

- Use *list* when all elements of the result must be evaluated (in other words, none of the list members are known constants).
- Use *backquote* when the list contains a mixture of known constants and evaluated entries.
- Use *quote* when the list consists entirely of known constants.

For example, suppose we have the three variables, a, b, and c such that:

```
a = 1
b = "a string"
c = '(A B C D E)
```

If we wanted to make a disembodied property list (dpl) of symbol-value pairs using these variables then we could use *list* or a mixture of *quote* and *backquote*, as follows:

```
/* These are identical in function. */
dpList1 = list('a a 'b b 'c c)
dpList2 = `(a ,a b ,b c ,c)
```

In the second case, some items are preceded with commas (with no space after the comma) and some are not. Those *not* preceded with commas are treated as literals, as if this was a normal quoted list. Those preceded by commas are evaluated, as if the list was declared using *list*.

If this was the only use of *backquote*, it would not be very useful. However, there are two useful extra features. The first is that you can splice in entire *lists* by using the *,@* (comma-at) specifier, as follows:

```
‘(a ,a b ,b c ,@c) => (a 1 b "a string" c A B C D E)
```

Here, the five elements *A*, *B*, *C*, *D*, and *E* have been used in place of the placeholder *,@c*. This cannot be done easily using *list*. The second useful feature is that the expansion can descend hierarchically. Suppose that instead of a dpl we wanted to create an assoc list. To do this using just *list*, would require the following:

```
list( list('a a) list('b b) list('c c) )
=> ((a 1) (b "a string") (c (A B C D E)))
```

Using *backquote* simplifies this to:

```
‘((a ,a) (b ,b) (c ,c))
=> ((a 1) (b "a string") (c (A B C D E)))
```

The last element can still be flattened using ,@ if required:

```
‘((a ,a) (b ,b) (c ,@c))
=> ((a 1) (b "a string") (c A B C D E))
```

Note: Care should be taken with lists defined using *quote*. These lists form part of the program code, and if edited using the destructive list operations the actual program code will be changed. This is particularly important when building *tconc* lists. It is very tempting to initialize a *tconc* structure to *'(nil nil)*, but this is wrong. If this is done, then as each element is added the program itself is being modified.

Consider the following naive list copy:

```
procedure(copylist(inputList)
      let( ((tc '(nil nil)))
            foreach(element inputList
                  tconc(tc element)
            )
            car(tc)
      ) /* let */
) /* copylist */
```

This works the first time it is called, but the list assigned to *tc* in the variable declaration part is being modified as part of the program, so the next time the function is called, the copied list will actually be appended to the list that was copied in the first call:

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

The list should be initialized by using either *list(nil nil)* or *tconc(nil nil)*. The second method makes it more obvious that the variable is being initialized as a *tconc* structure, but in this case the return value would be *cdar(tc)* rather than *car(tc)*.

List Searching

There are two methods for searching a list, depending on its structure. For a simple list, the functions *memq* and *member* can be used to search the list. The *memq* function is faster because it uses the *eq* function for comparison and is therefore preferred whenever the list contains elements for which the *eq* function is suitable.

If the list is an assoc list, that is, it is a list of key-value pairs, then the functions *assoc* and *assq* should be used to do the searching. Again, *assq* is faster because it uses the *eq* function for the comparison. It is therefore worthwhile, when building assoc lists, trying to ensure that the key elements are suitable for use with the *eq* function. In particular, when building an assoc list that would normally have keys that are strings, it may be worthwhile using the *concat* function to turn these strings into symbols, and then using those symbols as the keys in the list. This will then allow the *assq* rather than the *assoc* function to be used.

There are, however, two disadvantages with this method. The first disadvantage is that symbols in SKILL use memory and are not garbage collected (they are persistent), so creating many symbols uses up memory. Garbage collection is also slowed because the speed of this is directly related to the number of symbols. The second disadvantage is that the *concat* function is itself quite slow, so the overhead of this might outweigh any gains from using *assq* instead of *assoc*.

List Sorting

The *sort* and *sortcar* functions for lists are based on a recursive merge sort and are thus reasonably efficient. The list is sorted in-place, with no new list elements created, thus the list returned replaces the one passed as argument, and the one passed as argument should no longer be used.

Element Removal and Replacing

Two non-destructive functions are provided for removal of elements from a list, *remq* and *remove*. The equivalent but destructive functions are *remd* and *remdq* These functions remove all elements from the list which match a given value, using the *eq* and *equal* function respectively. The *remq* function is faster than the *remove* function.

The function *subst* is provided for replacement of all elements of a list matching a particular value with another value. This function uses *equal* and so should be used sparingly.

It should be noted that the non-destructive functions return a *copy* of the original list with the matching elements removed. This means that these functions should not be used within a loop in order to remove a large number of elements. If a number of different elements must

be removed from a list, then it is more efficient to generate a new list by traversing the old one just once, selecting only the required elements for the new list.

Alternatives to Lists

In many cases, there are faster and more compact alternatives to list structures. For example, if you need a property list that is likely to remain small in contents and most of the properties are known, consider using a *defstruct*.

If you need an assoc or property list whose contents are likely to be large (in the order of tens at least), then consider using assoc tables. Assoc tables offer much faster access time and for a large set of key-value pairs memory usage is more efficient. Assoc tables are not ordered (they are implemented as hash tables).

Miscellaneous Comparative Timings

This section gives comparative timings for various pieces of SKILL code to further demonstrate and reinforce the comments made in the previous sections. The examples are listed in an order that matches the structure of the preceding sections.

The timings were ascertained using the SKILL *profile* command and are expressed in ratios of the first example. In producing the timings, every effort has been made to compare like with like.

Element Comparison

The difference in speed between the *eq* and *equal* functions can be demonstrated using the following functions:

```
procedure(equal_test()
      equal('fred 'joe)
      equal('fred 'fred)
) /* end equal_test */

procedure(eq_test()
      eq('fred 'joe)
      eq('fred 'fred)
) /* end eq_test */
```

Note that each procedure has two comparisons, one failing and one succeeding. The comparative times for these were:

```
equal_test 1.00
eq_test    0.92
```

Further tests demonstrated that it is when the symbols are *not* equal that the *eq* function gains over the *equal* function. This means that if the test is expected to succeed on most occasions, there is little difference between the two functions.

List Building

As noted, there are several methods for building a list. The following examples attempt to build a list of the first 50 integers. The last example builds the list in descending order; the others build the list in ascending order.

```
procedure(list1()
 let( (returnList)
            for(i 1 50
                  returnList = append1(returnList i)
            )
            /* return */
            returnList
 ) /* end let */
) /* end list1 */

procedure(list2()
 let( ((returnList list(nil nil))
            )
            for(i 1 50
                  tconc(returnList i)
            )
            car(returnList)
 ) /* end let */
) /* end list2 */

procedure(list3()
 let( (returnList)
            for(i 1 50
                  returnList = cons(i returnList)
            )
            reverse(returnList)
      ) /* end let */
) /* end list3 */

procedure(list4()
 let( (returnList)
            for(i 1 50
                  returnList = cons(i returnList)
            )
            returnList
 ) /* end let */
) /* end list4 */
```

The outcome of this test depends on the length of the list being built. These examples use a medium length list, and the results of running these examples are:

```
list1 1.00
list2 0.14
list3 0.11
list4 0.10
```

These results demonstrate that with this size of list there is little difference between using the *tconc* method and the *cons* and *reverse* method. In fact, there will be little difference between these methods for any length of list because they both have to carry out the same basic functions. The only difference is that the *reverse* method must find twice as many list elements as the *tconc* method. This gives a greater chance of the garbage collector being called, which might cause the program to slow down, especially for large lists.

Mapping Functions

To demonstrate the relative speeds of the mapping functions, consider the following implementations of a function that picks every even integer out of a list of integers, returning the list of even integers in the same order as the originals.

```
procedure(map1(intList)
      let( (res)
            while(intList
                  when(evenp(car(intList))
                              res = cons(car(intList) res)
                  )
                  intList = cdr(intList)
            ) /* end while */
            reverse(res)
      ) /* end let */
) /* end map1 */

procedure(map2(intList)
      let( (res)
            foreach(i intList
                  when(evenp(i)
                        res = cons(i res)
                  )
            ) /* end foreach */
            reverse(res)
      ) /* end let */
) /* end map2 */

procedure(map3(intList)
      foreach(mapcan i intList
            when(evenp(i)
                  ncons(i)
            )
      ) /* end foreach */
) /* end map3 */

procedure(map4(intList)
      mapcan((lambda (i) when(evenp(i) ncons(i)))
            intList
      )
) /* end map4 */
```

The relative timings for these are:

```
map1    1.00
map2    0.68
map3    0.64
map4    0.29
```

This shows that the version using a *lambda* function along with the basic mapping function is the fastest. However, this is the least readable of these functions and should only be used with a great deal of caution, and a large number of comments.

Data Structures

It is difficult to give meaningful comparisons between the data structure functions because they are all suitable for different tasks. The following two examples attempt to compare the time taken to access and change one element of a data structure stored as an array, simple list, assoc list, defstruct and property list.

```
elem_number = 9
elem_symbol = 'j

procedure(access1(array)
      array[elem_number]
) /* end access1 */

procedure(access2(list)
 nth(elem_number list)
) /* end access2 */

procedure(access3(assoc_list)
 cadr(assq(elem_symbol assoc_list))
) /* end access3 */

procedure(access4(dstruct)
 get(dstruct elem_symbol)
) /* end access4 */

procedure(access5(plist)
 get(plist elem_symbol)
) /* end access5 */

procedure(access6(assocTable)
 assocTable[elem_symbol]
) /* end access */
```

The comparative timings for these are:

```
access1    1.00
access2    1.3
access3    1.47
access4    1.08
access5    1.55
access6    1.11

procedure(set1(array val)
 array[elem_number] = val
) /* end set1 */

procedure(set2(list val)
 rplaca(nthcdr(elem_number list) val)
) /* end set2 */

procedure(set3(assoc_list val)
 rplaca(cdr(assq(elem_symbol assoc_list)) val)
) /* end set3 */
```

```
procedure(set4(dstruct val)
 putprop(dstruct val elem_symbol)
) /* end set4 */

procedure(set5(plist val)
 putprop(plist val elem_symbol)
) /* end set5 */

procedure(set6(assocTable val)
 assocTable[elem_symbol] = val
) /* end set6 */
```

The comparative timings for these are:

```
set1    1.00
set2    1.14
set3    1.09
set4    0.93
set5    1.71
set6    1.2
```

No comment will be made about the readability of these functions because access procedures should be made available for all data structure access anyway. It is clear from these results that there is little to be gained, in terms of speed from the choice of data structure, although arrays do seem to be fastest overall. When choosing data structures it is more important to consider the other factors mentioned in the data structures section of this document.

Because the structures used contained a small number of elements, the experiment is naturally biased. You can repeat this experiment using *measureTime* to find out how effective your data structure accesses are for a given volume of data. For example, for sets of hundreds of elements, association tables will be significantly faster to access than property lists and assoc lists. For large sets it would be impractical to use arrays or defstructs.

13

About SKILL++ and SKILL

Overview information:

Introducing the Cadence SKILL++ Language

The Cadence-supplied Scheme is referred to as theCadence® SKILL++ language . Two main forces drive Cadence's decision to supply support for SKILL++ within the Cadence SKILL environment: software engineering and standards.

What Is SKILL++?

SKILL++ is the name of the second generation extension language for the CAD tools from Cadence. It combines the ease-of-use of the well-received SKILL environment with the tremendous power of the highly-acclaimed programming language Scheme, to give the users a more capable customization and extension-development platform.

Lexical Scoping and the Power of Closures

The major power brought in from Scheme is its use of “lexical scoping” and functions with lexically closed environments called *closures*. Lexical scoping makes reliable and modular programming more easily achievable, because you have total control over the use and reference of variables for any code without worrying about accidental corruptions caused by the use of the same variable name in a remote place.

Closures are powerful entities that only exist in the more advanced programming languages. They encapsulate the code and related data into a single unit, with total control on the exported interface. Many modern programming idioms and paradigms, such as message-passing and objects with inheritance, can be implemented elegantly using closures.

Environments as First-Class Objects

In addition, SKILL++ provides “environments” as first-class objects. With closures and first-class environments, users can create their own module or package systems easily. In other words, with first-class environments, users are no longer restricted to the single flat name space model used by SKILL, one can now easily organize the code into a hierarchy of name spaces.

SKILL++ Object Layer

Besides Scheme semantics, SKILL++ includes an object layer that makes explicit object-oriented style programming possible. The object layer supports classes, generic functions, methods, and single inheritance.

SKILL and SKILL++ Work in Harmony

Because SKILL++ and SKILL can coexist harmoniously in the same environment, backward compatibility and interoperability are not an issue. All existing SKILL code can still run without any changes, and all or part of any SKILL package can be migrated to SKILL++. Code developed in SKILL++ and SKILL can call each other and share the same data structures transparently.

Background of SKILL

SKILL was originally based on a flavor of Lisp called “Franz Lisp.” Franz Lisp and all other flavors of Lisp were eventually superceded by an ANSI standard for Lisp called “Common Lisp.”

The semantics and nature of SKILL make it ideal for scripting and fast prototyping. But the lack of modularity and good data abstraction, or its general openness, make it harder to apply modern software engineering principles especially for large endeavors. Since its inception, SKILL has been used for writing very large systems within the Cadence tools environments. To this end Cadence chose to offer Scheme within the SKILL environment.

Origins of Scheme

Scheme is a Lisp-like language developed originally for teaching computer science at the Massachusetts Institute of Technology and is now a popular language in computer science and sometimes EE curriculums. Scheme is a modern language whose semantics empower engineers to develop sound software systems. There is an IEEE standard for Scheme. Scheme was also the choice of the CAD Framework Initiative (CFI) for an extension language base.

Scheme Is Supplied within the SKILL Environment

Cadence is supplying major Scheme functionality as part of the SKILL environment. This has many advantages:

- New programs written in Scheme can co-exist and call procedures in existing programs written in SKILL without paying penalties in performance or functionality.
- Scheme and SKILL will share the run-time environment so structures allocated in Scheme programs can be passed without modification to SKILL programs and visa-versa.

- Suppliers and consumers can choose to migrate to Scheme independently of each other. For example, a Cadence-supplied layer can choose to remain written in SKILL while users of that layer can switch to using Scheme and visa-versa.

Relating SKILL++ to IEEE and CFI Standard Scheme

CFI has chosen IEEE standard Scheme as the base of their proposed CAD Framework extension language. Because the intended use was as a CAD tool extension language, which must be embeddable within large applications in a mixed language environment, CFI relaxed the requirement for the support of full “call-with-current-continuation” and the full numeric tower (only numbers equivalent to C's long and double are required).

For the same reason, CFI added a few extensions such as exception handling and functions for evaluating Scheme code, as well as a specification on the foreign function interface APIs, to their proposal.

SKILL++ is designed with IEEE Scheme and CFI Scheme compliance in mind, but due to its SKILL heritage and compatibility, it is not fully compliant with either standard.

The following sections describe the differences between SKILL++ and the standard Scheme language.

Syntax Differences

SKILL++ uses the same familiar SKILL syntax with the following restrictions and extensions.

Restrictions

- Because most of the special characters are used as infix symbols in SKILL++ and SKILL, they cannot be used as regular name constituents. However, many standard Scheme functions and syntax forms have been systematically renamed for ease of use under SKILL++'s syntax, for example

```
pair?           ==> pairp
list->vector    ==> listToVector
make-vector     ==> makeVector
set!            ==> setq
let*            ==> letseq (for “sequential let”)
```

- Except for vector literals (e.g. #(1 2 3)), the “#...” syntax is not supported, so use 't' for #t, 'nil' for #f, and use single character symbols for character literals and so forth.

Extensions

- Like SKILL, but unlike standard Scheme, SKILL++ symbols are case- sensitive.
- SKILL++ inherited all the SKILL syntactic features, such as infix notation, optional/keyword arguments with default values, and many powerful looping special forms (such as *for, foreach, setof*).
- SKILL++ code can define macros using the *mprocedure/defmacro* as well as use existing macros defined in SKILL.

Semantic Differences

SKILL++ adopts the standard Scheme semantics with the following restrictions and extensions.

Restrictions

- The atom *nil* is the same as the empty list '(), as well as the *false* value. Standard Scheme uses #f for the only false value and treats the empty list as a true value.
- The *cons* cells in SKILL++ are like SKILL's, that is, their *cdr* slot can only be either *nil* or another *cons* cell. In standard Scheme, the *cdr* slot of a cons cell can hold any value.
- The SKILL++ *map* function and the SKILL *map* function share the same name and implementation, but behave differently from standard Scheme's *map* function. To get the behavior of Scheme's *map*, use *mapcar* in SKILL++.
- Strings in SKILL++ and SKILL are immutable, so there is no support for functions like Scheme's *string-set!*.
- The *character* type is not supported yet. As in SKILL, symbols of one character can be used as characters.
- No "eof" object. The *lineread* function returns *nil* on end-of-file.
- Tail-call optimization is normally turned-off (for better debugging and stack tracing support). Because there are many looping constructs inherited from SKILL, this is normally not a problem for programming in SKILL++.

Extensions

- Environments are treated as first class objects. The *theEnvironment* form can be used to get the enclosing lexical environment, and bindings in an environment can be accessed easily. This provides a powerful encapsulation tool.

- SKILL++ inherits the SKILL set of powerful data structures, such as *defstruct* and association tables, as well as all the functions and many special forms of SKILL.
- Support for transparent cross-language (SKILL <-> SKILL++) mixed programming.

Syntax Options

SKILL++ adopts the same SKILL syntax, which means SKILL++ programs can be written in the familiar infix syntax and the general Lisp syntax.

If syntax is not an issue for you and you are comfortable with the infix notation, continue to use that.

If you are concerned that the knowledge of SKILL++ you build by programming in the infix syntax will not be useful if you were to program in a Scheme environment (without SKILL), then use the Lisp syntax for programming in SKILL++. The syntactic differences between SKILL++ using Lisp syntax and standard Scheme are:

- In SKILL++ you are not allowed to use any special characters (such as +, -, /, *, %, !, $, &, and so forth) in identifiers because most of these characters are used as infix operators.
- As a general convention, Scheme functions ending with an exclamation point (!) are provided either without the exclamation point or as the equivalent SKILL function. For example, Scheme *set!* is SKILL++ *setq*. Scheme functions using “->” are provided using “To” as part of the function name. For example “list->vector” becomes *listToVector*. See “Scheme/SKILL++ Equivalents” in the *SKILL Language Reference* for a complete list of name mappings.
- Scheme's dotted pairs are not available in SKILL++. Use simple lists instead.
- Using “=>” and “...” as identifiers is supported.

Compliance Disclaimer

Scheme as supplied by Cadence will not be fully IEEE compliant for various reasons.

Scheme was not originally designed as an extension language. So the features in Scheme that cannot be safely used in conjunction with a system written in C/C++ are omitted, such as, “call/cc” and non-null terminated strings.

Cadence puts a high value on making the system fully backward compatible for SKILL programs and procedural interfaces. As a result, the empty list *nil* is a Boolean *true* in the Scheme standard, whereas Cadence's SKILL and SKILL++ treat *nil* as a Boolean *false*.

Without this treatment of *nil*, the migration of existing SKILL programs to Scheme would require many existing procedural interfaces written in SKILL to change.

In general, SKILL++ is implemented to support both the Lisp syntax and the more familiar SKILL infix syntax for writing Scheme programs, as well as to provide a smooth path for migrating SKILL code to Scheme.

References

For further readings on Scheme:

Structure and Interpretation of Computer Programs, Harold Abelson, Gerald Sussman, Julie Sussman, McGraw Hill, 1985.

Scheme and the Art of Programming, G. Springer & D. Friedman, McGraw Hill, 1989.

An Introduction to Scheme, J. Smith, Prentice Hall, 1988.

"Draft Standard for the Scheme Programming Language," P1178/D5, October 1, 1990. IEEE working paper.

Extension Language Environment

Cadence's extension language environment supports two integrated extension languages, SKILL and SKILL++. Programs in either language can freely

- Share data
- Call each other's functions

Your application can consist of source code written in either language. This chapter helps you decide which language to use in your software development.

Maintaining Existing SKILL Programs

You can maintain existing SKILL applications *with no change whatsoever*.

Hiding Private Functions and Private Data with SKILL++

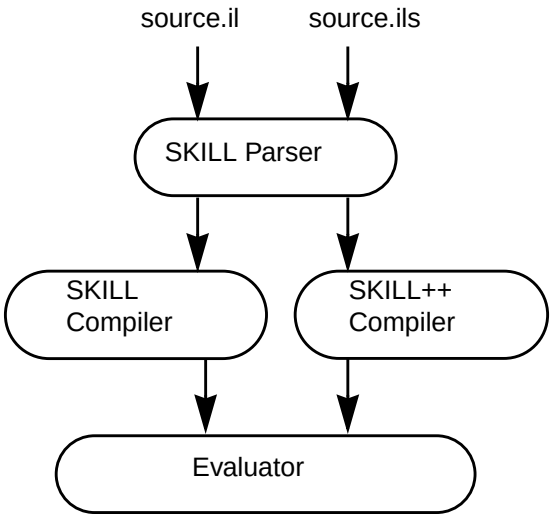
Using SKILL++ in new applications lets you hide private functions and private data. You can therefore design and implement your application with reusable components. If you are

developing a new application, you should consider using the SKILL++ language in conjunction with various SKILL procedural interfaces when necessary.

Specifying the Language

Specifying Languages for Source Code

For source code files, the file extension indicates the language.



Specifying Languages in Interactive Sessions

By default the session accepts SKILL expressions. Interactively, you can invoke a top level for either SKILL or SKILL++.

Language	Command
SKILL	toplevel 'il
SKILL++	toplevel 'ils

Specifying Languages under Program Control

For advanced programmers, the *eval* function evaluates an expression with either SKILL semantics or SKILL++ semantics.

Contrasting Variable Scoping

Variables associate an identifier with a memory location. A referencing environment is the collection of identifiers and their associated memory locations. The *scope* of a variable refers to *the part of your program* within which the variable refers to the same location.

During your program's execution, the referencing environment changes according to certain scoping rules. Even though the syntax of both languages is identical, the SKILL language and the SKILL++ language

- Use different scoping rules
- Partition a program into pieces differently

SKILL++ Uses Lexical Scoping

The lexical scoping rule is only concerned with the source code of your program. The phrase *part of your program* means *a block of text* associated with a SKILL++ expression, such as *let*, *letrec*, *letseq*, and *lambda* expressions. You can nest blocks of text in the source code.

SKILL Uses Dynamic Scoping

The dynamic scoping rule is only concerned with the flow of control of your program. In SKILL, the phrase *a part of your program* means *a period of time* during execution of an expression. Usually, the dynamic scoping and lexical scoping rules agree. In these cases, identical expressions in both SKILL and SKILL++ return the same value.

Example 1: Sometimes the Scoping Rules Agree

The following example has line numbers added for reference only.

```
1: let( ( x )
2:      x = 3
3:      x
4:      )
```

In both SKILL and SKILL++, the *let* expression establishes a scope for *x* and the expression returns 3.

- In SKILL++, the scope of *x* is the *block of text* comprising line 2 and line 3. The *x* in line 2 and the x in line 3 refer to the same memory location.
- In SKILL, the scope of *x* begins when the flow of control enters the *let* expression and ends when the flow of control exits the *let* expression.

Example 2: When Dynamic and Lexical Scoping Disagree

In example 1, the two scoping rules agree and the *let* expression returns the same value in both SKILL and SKILL++. Example 2 illustrates a case in which dynamic and lexical scoping disagree. Notice that the following extends the first example by merely inserting a function call to the *A* function between two references to *x*. However, the *A* function assigns a value to *x*.

```
1: procedure( A() x = 5 )
2: let( ( x )
3:      x = 3
4:      A()
5:      x
6:      )
```

Consider the *x* in line 1.

- In SKILL++, the *x* in line 1 and the *x* in line 5 refer to different locations because the *x* in line 1 is outside the block of text determined by the *let* expression. The *let* expression returns 3.
- In SKILL, because line 1 executes during the execution of the *let* expression, the *x* in line 1 and the *x* in line 5 refer to the same location. The *let* expression returns 5.

Example 3: Calling Sequence Effects on Memory Location

In SKILL, dynamic scoping dictates that the memory location affected depends on the function calling sequence. In the code below, function *B* updates the global variable *x*. Yet when called from the function A, function B *alters* function *A*'s local variable *x* instead. Function *B* actually updates the *x* that is local to the *let* expression in *A*.

- In SKILL, function *A* returns 6.
- In SKILL++, function *A* returns 5.

```
procedure( A()
      let( ( x )
           x = 5        ;;; set the value of A's local variable x
           B()
           x            ;;; return the value of A's local variable x
```

```
            ) ;let
      ) ; procedure
procedure( B()
      let( ( y z )
            x = 6
            z
            )
      ) ; procedure
```

Refer to “Dynamic Scoping” on page 216 for guidelines concerning the use of dynamic scoping.

Why Lexical Scoping Is Better for Reusable Code

It is important that the scope of variables not be unintentionally disrupted when the programmer modifies or otherwise reuses a program. Often subtle bugs result when modification or reuse in another setting extends the scope of a variable.

The programmer should be able to inspect the source code to determine the effect on the scope of variables. Because dynamic scoping relies on the execution history of your program, it can prevent confidently reusing existing code and therefore dynamic scoping impacts writing reusable, modular code.

Summary

SKILL++ uses lexical scoping. Because lexical scoping relies solely on the static layout of the source code, the programmer can confidently determine how reusing a program affects the scope of variables.

SKILL uses dynamic scoping. Modifying a SKILL program can sometimes unintentionally disrupt the scope of a variable. The probability of introducing subtle bugs is higher.

Contrasting Symbol Usage

SKILL and SKILL++ share the same symbol table. Each symbol in the symbol table is visible to both languages.

How SKILL Uses Symbols

SKILL has a data structure called a symbol. A symbol has a name which uniquely identifies it and three associated memory locations. For more information, refer to “Symbols” on page 94.

The Value Slot

SKILL uses symbols for variables. A variable is bound to the value slot of the symbol with the same name. For example, *x = 5* stores the value 5 in the value slot of the symbol x. The *symeval* function returns the contents of the value slot. For example, *symeval('x)* returns the value 5.

The Function Slot

SKILL uses the function slot of a symbol to store function objects. SKILL evaluates a function call, such as

```
fun( 1 2 3 )
```

by fetching the function object stored in the function slot of the symbol *fun*. Dynamic scoping does not affect the function slot at all.

The Property List

Dynamic scoping does not affect the property list at all. Refer to "Important Symbol Property List Considerations" on page 99.

Summary

By calling the SKILL functions *set*, *symeval*, *getd*, *putd*, *get*, and *putprop*, you can access the three slots of a symbol. The following table summarizes the SKILL operations that affect the three slots of a symbol.

SKILL Construct	Value Slot	Function Slot	Property List
The assignment operator (=)	x		
set, symeval	x		
let and prog constructs	x		
procedure declaration		x	
getd, *putd*		x	
Function Call		x	
get, *putprop*			x

How SKILL++ Uses Symbols

Normally, each SKILL++ global variable is bound to the function slot of the symbol with the same name. This allows SKILL and SKILL++ to share functions transparently.

Sometimes, your SKILL++ program needs to access a SKILL global variable. By using the *importSkillVar* function, you can change the binding of the SKILL++ global from the function slot of a symbol to the value slot of the symbol.

Contrasting the Use of Functions as Data

In SKILL++ it is much easier to treat functions as data than it is in SKILL.

Assigning a Function Object to a Variable

In SKILL++, function objects are stored in variables just like other data values. You can use the familiar SKILL algebraic or conventional function call syntax to invoke a function object indirectly. For example, in SKILL++:

```
addFun = lambda( ( x y ) x+y ) => funobj:0x1e65c8
addFun( 5 6 ) => 11
```

In SKILL, the same example can be done two different ways, both of them less convenient.

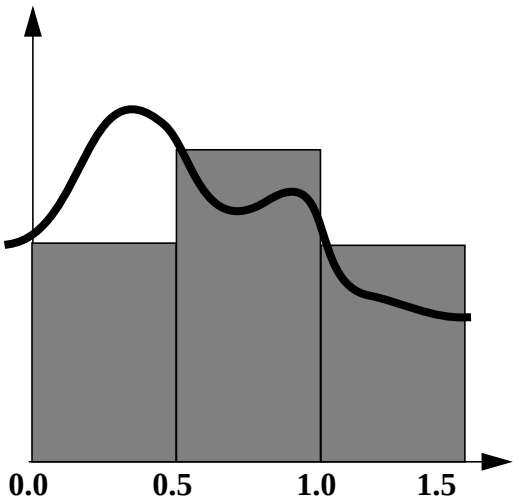
```
addFun = lambda( ( x y ) x+y )
apply( addFun list( 5 6 )) => 11
```

or

```
putd( 'addFun lambda( ( x y ) x+y ))
addFun( 5 6 ) => 11
```

Passing a Function as an Argument

To pass a function as an argument in SKILL++ does not require special syntax. In SKILL the caller must quote the function name and the callee must use the *apply* function to invoke the passed function.



The *areaApproximation* function in the following example computes an approximation to the area under the curve defined by the *fun* function over the interval (0 1). The approximation consists of adding the areas of three rectangles, each with a width of 0.5 and with heights of fun(0.0), fun(0.5), and fun(1.0).

In SKILL++

```
procedure( areaApproximation( fun )
      0.5*( fun( 0.0 ) + fun( 0.5 ) + fun( 1.0 ) )
      ) => areaApproximation

areaApproximation( sin ) => 0.6604483
areaApproximation( cos ) => 1.208942
```

In SKILL

```
procedure( areaApproximation( fun )
      0.5*(
            apply( fun '( 0.0 )) +
            apply( fun '( 0.5 )) +
            apply( fun '( 1.0 ))
            )
      ) => areaApproximation

areaApproximation( 'sin ) => 0.6604483
areaApproximation( 'cos ) => 1.208942
```

SKILL++ Closures

A SKILL++ *closure* is a function object containing one or more *free* variables (defined below) bound to data. Lexical scoping makes closures possible. In SKILL, dynamic scoping prevents effective use of closures, but a SKILL++ application can use closures as software building blocks.

Relationship to Free Variables

Within a segment of source code, a *free* variable is a variable whose binding you cannot determine by examining the source code. For example, consider the following source code fragment.

```
procedure( Sample( x y )
           x+y+z
           )
```

By examination, *x* and *y* are not free variables because they are arguments. *z* is a free variable. In SKILL++ lexical scoping implies that the bindings of all of a function's free variables is determined at the time the function object (closure) is created. In SKILL, dynamic scoping implies that all references to free variables are determined at run time.

For example, you can embed the above definition in a *let* expression that binds *z* to *1*. Only code in the same lexical scope of *z* can affect z's value.

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            )
      ;;; code that invokes Sample goes here.
      )
```

How SKILL++ Closures Behave

A SKILL++ application can use closures as software building blocks. The following examples increase in complexity to illustrate how SKILL++ closures behave.

Example 1

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            )
      ;;; code that invokes Sample goes here.
      Sample( 1 2 )
      )
=> 4
```

Example 2

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            )
      ;;; code that invokes Sample goes here.
      z = 100
      Sample( 1 2 )
      )
=> 103
```

This example assigns *100* to the binding of *z*. Consequently, the *Sample* function returns *103*. In this case, dynamic scoping and lexical scoping agree.

Example 3

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            ) ; procedure
      ;;; code that invokes Sample goes here.
      let( (( z 100 ))
            Sample( 1 2 )
            ) ; let
      ) ; let
=> 4
```

This example invokes *Sample* from within a nested *let* expression. Although dynamic scoping would dictate that *z* be bound to *100*, lexically it is bound to *1*.

Example 4

```
procedure( CallThisFunction( fun )
      let( (( z 100 ))
                  fun( 1 2 )
            )
      )
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            ) ; procedure
      CallThisFunction( Sample )
      )
=> 4
```

In this SKILL++ example, the *let* expression binds *z* to *1*, creates the *Sample* function and then passes it to the *CallThisFunction* function. Whenever the *Sample* function runs, *z* is bound to *1*. In particular, when the *CallThisFunction* function invokes *Sample*, *z* is bound to *1* even though *CallThisFunction* binds *z* to a different value prior to calling *Sample*.

Therefore, *Sample* has encapsulated a value for its free variable *z*. To do this in SKILL is impossible, because dynamic scoping dictates that *Sample* would see the binding of *z* to *100*.

Therefore, SKILL++ allows you to build a function object that you can pass an argument, certain that its behavior will be independent of specifics of how it is ultimately called.

Example 5

```
W = let( (( z 1 ))
            lambda( ( x y )
                  x+y+z
                  )
            )
=> funobj:0x1bc828
W( 1 2 ) => 4
```

In this example, the name *Sample* is insignificant because the *let* expression itself does not contain a call to *Sample*. Instead, the *let* expression returns the function object. This function object is a closure. The code returns a distinct closure each time the code is executed. In SKILL++, there is no way to affect the binding of *z*. The function object has effectively encapsulated the binding of *z*.

Example 6

The *makeAdder* function below creates a function object which adds its argument *x* to the variable *delta*. Each call to *makeAdder* returns a distinct closure.

```
procedure( makeAdder( delta )
      lambda( ( x ) x + delta )
      )
=> makeAdder
```

In SKILL++, you can pass 5 to *makeAdder* and assign the result to the variable *add5*. No matter how you invoke the *add5* function, its local variable *delta* is bound to 5.

```
add5 = makeAdder( 5 )=> funobj:0x1e3628
add5( 3 ) => 8
```

```
let( ( ( delta 1 ) )
      add5( 3 )
      ) => 8
```

```
let( ( ( delta 6 ) )
      add5( 3 )
      ) => 8
```

SKILL++ Environments

This section introduces the run-time data structures called *environments* that SKILL++ uses to support lexical scoping. This section covers how SKILL++ manages environments during run time. Understanding this material is important if your application use closures.

For more information, "Using SKILL and SKILL++ Together" on page 317 covers how inspecting environments can help you debug SKILL++ programs.

The Active Environment

During the execution of your SKILL++ program, the set of all the variable bindings is called an *environment*. To accommodate the sequence of nested lexical scopes which contain the current SKILL++ statement being executed, an environment is a list of environment frames such that

- SKILL++ stores all the variables with the same lexical scope in a single environment frame
- The sequence of nested lexical scopes correspond to a list of environment frames

Consequently, each environment frame is equivalent to a two column table. The first column contains the variable names and the second column contains the current values.

The Top-Level Environment

When a SKILL++ session starts, the active environment contains only one environment frame. There are no other environment frames. All the built-in functions and global variables are in this environment. This environment is called the top-level environment. The *toplevel('ils)* function call uses the SKILL++ top-level environment by default. However, it is possible to call the *toplevel('ils)* function and pass a non-top-level environment. Consider an expression such as

```
let( (( x 3 )) x )  => 3
```

in which we insert a call to *toplevel('ils)*. During the interaction we attempt to retrieve the value of *x* and then set it. References to *x* affect the SKILL++ top-level.

```
ILS-<2> let( (( x 3 )) toplevel( 'ils ) x )
ILS-<3> x
*Error* eval: unbound variable - x
ILS-<3> x = 5
5
ILS-<3> resume()
3
ILS-<2>
```

Compare it with the following in which we call *toplevel* passing in the lexically enclosing (active) environment. Thus the *toplevel* function can be made to access a non-top-level environment!

```
ILS-<2> let( (( x 3 )) toplevel( 'ils theEnvironment() ) x )
ILS-<3> theEnvironment()->??
(((x 3)))
ILS-<3> x
3
ILS-<3> x = 5
5
ILS-<3> resume()
5
ILS-<2> x
*Error* eval: unbound variable - x
ILS-<2>
```

Creating Environments

During the execution of your program, when SKILL++ evaluates certain expressions that affect lexical scoping, SKILL++ allocates a new environment frame and adds it to the front of the active environment. When the construct exits, the environment frame is removed from the active environment.

Example 1

```
let( (( x 2 ) ( y 3 ))
      x+y
      )
```

When SKILL++ encounters a *let* expression, it allocates an environment frame and adds it to the front of the active environment.

An Environment Frame

Variable	Value
x	2
y	3

To evaluate the expression *x+y*, SKILL++ looks up *x* and *y* in the list of environment frames, starting at the front of the list. When the expression terminates, SKILL++ removes it from the active environment. The environment frame remains in memory as long as there are references to the environment frame.

In this simple case, there are none, so the frame is discarded, which means it’s garbage and therefore liable to be garbage collected.

Example 2

```
let( (( x 2 ) ( y 3 ))
      let( (( u 4 ) ( v 5 )( x 6 ))
            u*v+x*y
            )
      )
```

At the time SKILL++ is ready to evaluate the expression *u*v+x*y*, there are two environment frames at the front of the active environment.

Environment Frame for the Outermost *let*

Variable	Value
x	2
y	3

Environment Frame for the Innermost *let*

Variable	Value
u	4
v	5
x	6

To determine a variable's location is a straight-forward look up through the list of environment frames. Notice that *x* occurs in both environment frames. The value 6 is the first found at the time the expression *u*v+x*y* is evaluated.

Functions and Environments

When You Create a Function

- SKILL++ allocates a function object with a link to the environment that was active at the time the function object was created.

When You Call a Function

- SKILL++ makes the function object's environment active (again) and allocates a new environment frame to hold the arguments. For example,

```
procedure( example( u v )
      let( (( x 2 ) ( y 3 ))
            x*y + u*v
```

```
            )
        )
    example( 4 5 )
```

allocates the following:

An Environment Frame

Variable	Value
u	4
v	5

and adds to the front of the active environment, which is the environment saved when the *example* function was first created.

When the Function Returns

- SKILL++ removes the environment frame holding the arguments from the active environment and, in this case, the environment frame becomes garbage. It then restores the environment that was active before the function call.

Persistent Environments

The *makeAdder* example below shows a function which allocates a function object and then returns it. The returned function object contains a reference to the environment that was active at the time the function object was created. This environment contains an environment frame that holds the actual argument to the original function call.

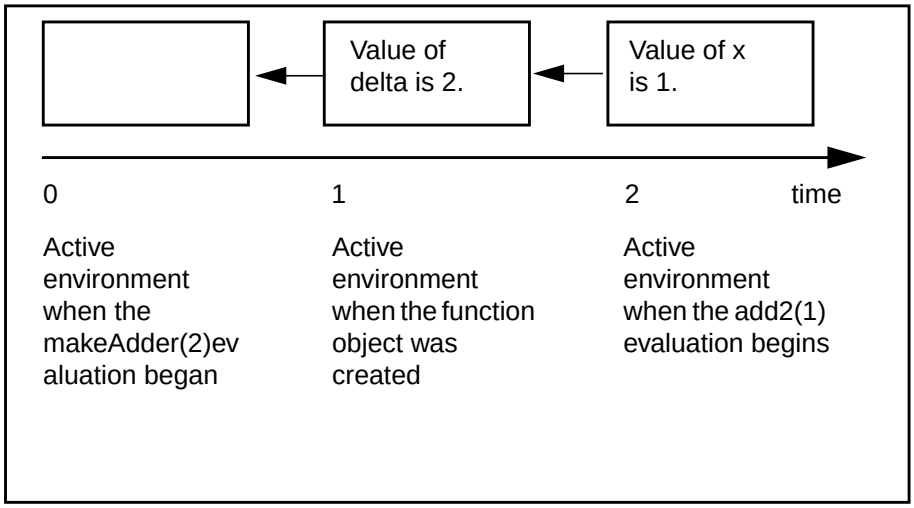
Therefore, whenever you subsequently call the returned function object, it can refer the local variables and arguments of the original function even though the original function has returned.

This capability gives SKILL++ the power to build robust software components that can be reused. Full understanding of this capability is the basis for advanced SKILL++ programming.

```
procedure( makeAdder( delta)
      lambda( ( x ) x + delta )
      )
=> makeAdder

add2 = makeAdder(2) => funobj:0x1e6628
add2( 1 ) => 3
```

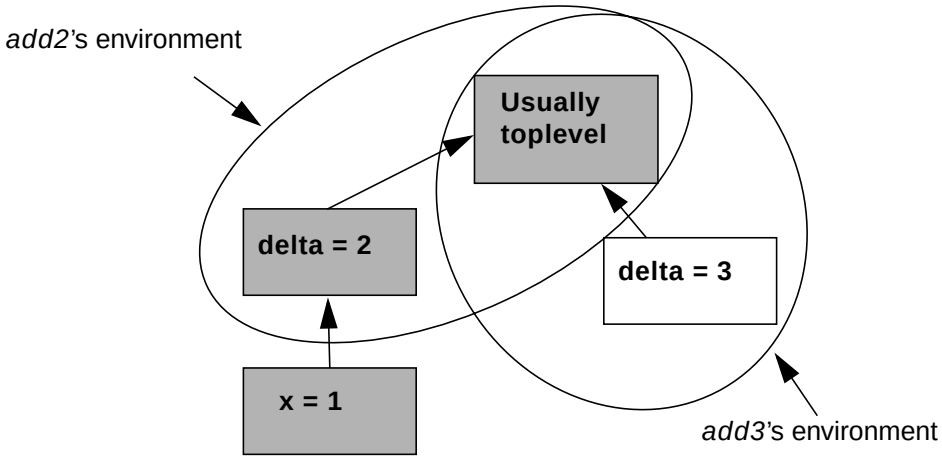
The function object that *makeAdder* returns is within the lexical scope of the *delta* argument.



Calling *makeAdder* again returns another function object.

```
add3 = makeAdder(3) => funobj:0x1e6638
```

The figure below shows several environments. The encircled environments belong to the *add2* and *add3* functions. The gray environment is the active environment at the time *add2(1)* at its entry point. The other environment belongs to the *add3* function, which becomes active only if *add3* is called.



14

Using SKILL++

Overview information:

Introduction

This chapter deals with the pragmatics of writing programs in the Cadence® SKILL++ language.

"About SKILL++ and SKILL" on page 275 provides an overview of the differences between the Cadence SKILL language and SKILL++.

"Using SKILL and SKILL++ Together" on page 317 focuses in detail on the key areas in which SKILL++ semantics differ from SKILL semantics.

"SKILL++ Object System" on page 331 describes a system that allows for object-oriented interfaces based on classes and generic functions composed of methods specialized on those classes.

Declaring Local Variables in SKILL++

SKILL++ provides three binding constructs to declare local variables together with initialization expressions. The syntax for *let, letseq* and *letrec* is identical but they differ in the order of evaluation of the initialization expressions and in the scope of the local variables.

Syntax Template for let, letseq and letrec

```
let(
      ( ( s_var1 g_initExp1 ) ( s_var2 g_initExp2 )... )
      g_bodyExp1
      g_bodyExp2
      ...
      )
```

Using let

Each local variable has the body of the *let* expression as its lexical scope. The order of evaluation of the initialization expressions and the binding sequence is unspecified. Hence, observe the following caution.

The initialization expression bound to one local variable should not refer to any of the other local variables.

Example 1

```
let( ( ( x 2 ) ( y 3 ) )
      x*y
      ) => 6
```

Example 2

```
let( ( ( x 2 ) ( y 3 ) )
      let( (( z 4 ))
            x + y + z
            ) ; let
      ) ; let
=> 9
```

Avoid Cross-References Between Variables

These examples demonstrate the wisdom of avoiding cross-references between variables in a *let* expression.

Example 3

```
let( ( ( x 2 ) ( y 3 ) )
      let( (( x 7 ) ( z x+y ) )
            z*x
            )
      ) => 35
```

Because the initialization expressions are outside of the scope of the *let*, *z* is bound to *2+3*, instead of *7+3*.

Example 4: Initialization Expressions Are Outside the Scope of *let*

```
let( ( ( x 2 ) ( y 3 ) )
      let( (( x 7 ) ( foo lambda( ( z ) x + y + z ) ) )
            foo( 5 )
            )
      )
=> 10
```

This example shows that the initialization expressions are also outside the scope of the *let*. Specifically, the occurrence of *x* in the body of *foo* is in the scope of the outer *let*.

Using letseq

Use *letseq* to control the order of evaluation of the initialization expressions and the binding sequence of the local variables. Evaluation proceeds from left to right. The scope of each

variable includes the remaining initialization expressions and the body of *letseq*. It is equivalent to a corresponding sequence of nested *let* expressions.

Example 1

```
letseq( ( ( x 1 ) ( y x+1 ) )
      y
      )
=> 2
```

The code above is a more convenient equivalent to the code below in which you control the sequence explicitly by the nesting.

```
let( ( ( x 1 ) )
      let( ( ( y x+1 ) )
            y
            )
      )
```

Example 2

```
let( ( ( x 2 ) ( y 3 ) )
      letseq( (( x 7 ) ( z x+y ) )
            z*x
            )
      ) => 70
```

This example is identical to Example 3 on page 299 except that the inner *let* is replaced with *letsec*.

Example 3

```
let( ( ( x 2 ) ( y 3 ) )
      letseq( (( x 7 ) ( foo lambda( ( z ) x + y + z ) ) )
            foo( 5 )
            )
      )
=> 15
```

This example is identical to “Example 4: Initialization Expressions Are Outside the Scope of let” on page 299 except that the inner *let* is replaced with *letseq*.

Using letrec

Unlike *let* and *letseq*, each variable’s scope is the entire *letrec* expression. In particular, each variable’s scope includes all of the initialization expressions. Each initialization expression can refer to the other local variables with the following restriction: each initialization expression must be executable without accessing the other variables. This

restriction is met when each initialization expression is a *lambda* expression. Therefore, use *letrec* to declare mutually recursive local functions.

Example 1

```
letrec(
      ( ;;; variable list
            ( f
                  lambda( ( n )
                        if( n > 0 then n*f(n-1) else 1
                              ) ; if
                        ) ; lambda
                  ) ; f
            ) ; variable list
      f( 5 )
      ) ; letrec
=> 120
```

This example declares a single recursive local function. The *f* function computes the factorial of its argument. The *letrec* expression returns the factorial of 5.

Example 2

```
procedure( trParity( x )
      letrec(
            (
                  ( isEven
                        lambda( (x)
                              x == 0 || isOdd( x-1 )
                              ) ; lambda
                        ) ; isEven
                  ( isOdd
                        lambda( (x)
                              x > 0 && isEven(x-1)) ; lambda
                        ) ; isOdd
                  ) ;
            if( isEven( x ) then 'even else 'odd )
            ) ; letrec
      ) ; procedure
```

The *trParity* function returns the symbol *even,* if its argument is even, and returns the symbol *odd* otherwise. *trParity* relies on two mutually recursive local functions *isEven* and *isOdd*.

Using procedure to Declare Local Functions

As an alternative to using *letrec* to define local functions, you can use the *procedure* syntax.

Example 1

This example uses the *procedure* construct to declare a local function instead of using *letrec*.

```
procedure( trParity( x )
      procedure( isEven(x)
            x == 0 || isOdd( x-1 )
            )
      procedure( isOdd(x)
            x > 0 && isEven(x-1)
            )
      if( isEven( x ) then 'even else 'odd )
      ) ; procedure
```

Example 2

```
procedure( makeGauge( tolerance )
      let( ( ( iteration 0 ) ( previous 0.0 ) test )
            procedure( performTest( value )
                  ++iteration
                  test = ( abs( value - previous ) <= tolerance )
                  previous = value
                  when( test list( iteration value ))
                  ) ; procedure
            performTest
            ) ; let
      ) ; procedure

G = makeGauge( .1 )
=> funobj:0x322b28
G(2) => nil                  ;; first iteration
G(3)=> nil                   ;; second
G(3.01) = >                  ;; third iteration
      ( 3 3.01 )
```

The *makeGauge* function declares the local *performTest* function and returns it. This function object is the gauge. Passing a value to the gauge compares it to the previous value passed then updates the previous value. The gauge returns a list of the iteration count and the value, or *nil*. The function object has access to the local variables *iteration*, *previous*, and *test,* as well as access to the argument *tolerance*. Notice that using a gauge object can simplify your code by isolating variables used only for the tracking of successive values.

For another *makeGauge* example, see “Example 4: Using a Gauge When Computing the Area Under a Curve” on page 306.

Example 3

```
procedure( trPartition( nList )
      procedure( loop( numbers nonneg neg )
            cond(
                  ( !numbers list( nonneg neg ))
                  ( car( numbers ) > 0
```

```
                        loop(
                              cdr( numbers )
                              cons( car( numbers ) nonneg ) ; ppush on nonneg
                              neg
                              ) ; loop
                        )
                  ( car( numbers ) < 0
                        loop(
                              cdr( numbers )
                              nonneg
                              cons( car( numbers ) neg ) ; ppush on neg
                              ) ; loop
                        )
                  ) ; cond
            ) ; procedure
      loop( nList nil nil )
      ) ; procedure

trPartition( '( 3 -2 1 6 5 )) => ((5 6 1 3) (-2))
```

In this example, the *trPartition* function separates a list of integers into a list of non-negative elements and negative elements. The local *loop* function is recursive.

Sequencing and Iteration

The following sequencing and iteration functions are provided in SKILL++:

- Use *begin* to construct a single expression from one or more expressions.
- Use *do* to iteratively execute one or more expressions.
- Use a *named let* construct to extend the *let* construct with a recursive iteration capability.

Using begin

Use *begin* to construct a single expression from one or more expressions. The expressions are evaluated from left to right. The return value of the *begin* expression is the return value of the last expression in the sequence.

The *begin* function is equivalent to the *progn* function. The *progn* function is used to implement the { } syntax. Use the *begin* function to write SKILL++-compliant code.

Example 1

```
ILS-1> begin(
      x = 0
      printf( "Value of x: %d\n" ++x )
      printf( "Value of x: %d\n" ++x )
```

```
      x
      ) ; begin
Value of x: 1
Value of x: 2
2
```

This example shows a transcript using the *begin* function.

Example 2

```
ILS-1> { x = 0
      printf( "Value of x: %d\n" ++x )
      printf( "Value of x: %d\n" ++x )
      x }
Value of x: 1
Value of x: 2
2
```

This example uses the { } braces to group the same expressions.

Using do

Use *do* to iteratively execute one or more expressions. The *do* expression allows multiple loop variables with arbitrary variable initializations and step expressions. You can specify

- One or more loop variables, including an initialization expression and a step expression for each variable.
- A termination condition that is evaluated before the body expressions are executed.
- One or more termination expressions that are evaluated upon termination to determine a return value.

Syntax Template for *do* Expressions

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

A *do* expression evaluates in two phases.

Initialization Phase

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

Iteration Phase

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

More formally stated:

1. Each iteration begins by evaluating the termination condition.
2. If the termination condition evaluates to a non-nil value, the *do* expression exits with a return value computed as follows:
3. The termination expressions *terminationExp1, terminationEx*p2, ... are evaluated in order. The value of the last termination condition is returned as the value of the *do* expression.
4. Otherwise, the *do* expression continues with the next iteration as follows.
5. The loop body expressions *g_loopExp1, g_loopExp2, …* are evaluated in order.
6. The step expressions *g_stepExp1, g_stepExp2, …*, if given, are evaluated in an unspecified order.
7. The local variables *var1, var2, …* are bound to the above results. Reiterate from step one.

Example 1

```
procedure( sumList( L )
      do(
            (
                  ( tail L cdr( tail ))
                  ( sum 0 sum + car( tail ))
            )
            ( !tail sum )
            )
      ) ; procedure

sumList( nil ) => 0
sumList( '( 1 2 3 4 5 6 7 )) => 28
```

Example 2

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

```
procedure( trTriangularNumber( limit )
      do(
            (                       ;;; start loop variables
                  ( i 0 i+1 )
                  ( sum 0 )         ;;; no update expression
```

```
                              ;;; same as ( sum 0 sum )
            )                    ;;; end loop variables
            ( sum > limit        ;;; test
                  sum              ;;; return result
                  )
            sum = sum+i ;;; body
            ) ; do
      ) ; procedure

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

Example 3

```
procedure( approximateArea( dx fun lower upper )
      do(
            ( ; loop variables
                  ( sum
                        0.0       ;;; initial value
                        sum+fun(x)   ;; update
                        )
                  ( x
                        lower  ;;; initial value
                        x+dx ;;; update expression
                        )
                  ) ; end loop vars
            ( x >= upper ;;; exit test
                  dx*sum ;;; return value
                  )
            ;;; no loop expressions
            ;;; all work is in the update expression for sum
            ) ; do
      ) ; procedure

approximateArea( .001 lambda( ( x ) 1 ) 0.0 1.0 ) => 1
approximateArea( .001 lambda( ( x ) x ) 0.0 1.0 ) => .4995
```

The function *approximateArea* computes an approximation to the area under the graph of the *fun* function over the interval from *lower* to *upper*. It sums the values *fun(x)*, *fun(x+dx)*, *fun(x+dx+dx)* …

Example 4: Using a Gauge When Computing the Area Under a Curve

```
procedure( makeGauge( tolerance )
      let( ( ( iteration 0 ) ( previous 0.0 ) test )
            lambda( ( value )
                  ++iteration
                  test = ( abs( value - previous ) <= tolerance )
                  previous = value
                  when( test list( iteration value ))
                  ) ; lambda
            ) ; let
      ) ; procedure

procedure( computeArea( fun lower upper tolerance )
      let( ((gauge makeGauge( tolerance )) result )
```

```
            do(
                  ( ; loop variables
                        (
                              dx
                              1.0*(upper-lower)/2 ;;; initial value
                              dx/2                ;;; update expression
                              )
                        ) ; end loop variables
                  ( result =
                        gauge( approximateArea( dx fun lower upper ))
                        result
                        )
                  nil ;;; empty body
                  ) ; do
            ) ; let
      ) ; procedure

computeArea( lambda( ( x ) 1 ) 0 1 .00001 ) => ( 2 1.0 )
computeArea( lambda( ( x ) x ) 0 1 .00001 ) => ( 16 0.4999924)
pi = 3.1415
computeArea( sin 0 pi/2 .00001 ) =>(18 0.9999507)
```

The *computeArea* function invokes *approximateArea* iteratively until two successive results fall within the given tolerance. The *dx* loop variable is initialized to *1.0*(upper-lower)/2* and updated to *dx/2*. This example uses a gauge to hide the details of comparing successive results. The source for the *makeGauge* function is replicated for your convenience. See Example 2 on page 302 in the "Using procedure to Declare Local Functions" section for a discussion of *makeGauge* and gauges in general.

Using a Named let

The *named let* construct extends the *let* construct with a recursive iteration capability. Besides the name you provide in front of the list of the local variables, the *named let* has the same syntax and semantics as the ordinary *let* except you can recursively invoke the named *let* expression from within its own body, passing new values for the local variables.

Syntax Template for Named let

```
let(
      s_name
      ( ( s_var1 g_initExp1 ) ( s_var2 g_initExp2 )... )
      g_bodyExp1
      g_bodyExp2
      ...
      )
```

Example 1

```
let(
      factorial
      (( n 5 ))
```

```
      if( n > 1
            then
                  factorial( n-1)*n
            else
                  1
            )
      ) ; let => 120
```

This example computes the factorial of 5 with a named *let* expression. Compare the example above with the following

```
let( (( n 5 ))
      procedure( factorial( n )
            if( n> 1
                  then
                        n*factorial( n-1)
                  else
                        1
                  ) ; if
            ) ; procedure
      factorial( n )
      ) ; let => 120
```

and with the following

```
letrec(
      ( ;;; variable list
            ( n 5 )
            ( factorial
                  lambda( ( n )
                        if( n > 0 then n*factorial(n-1) else 1 ) ; if
                        ) ; lambda
                  ) ; f
            ) ; variable list
      factorial( n )
      ) ; letrec => 120
```

Example 2

```
let( loop                                 ;;; name for the let
      (              ;;; start let variables
            ( numbers '( 3 -2 1 6 -5 ))
            ( nonneg nil )
            ( neg nil )
            )       ;;; end of let variables
      cond(
            ( !numbers ;;; loop termination test and return result
                  list( nonneg neg )
                  )
            ( car( numbers ) > 0 ;;; found a non-negative number
                  loop( ;; recurse
                        cdr( numbers )
                        cons( car( numbers ) nonneg )
                        neg
                        ) ; loop
                  )
            ( car( numbers ) < 0 ;;; found a negative number
                  loop( ;; recurse
                        cdr( numbers )
```

```
                    nonneg
                    cons( car( numbers ) neg )
                    ) ; loop
                )
            ) ; cond
        ) ;;; loop let
=> ((613) (-5 -2))
```

This example separates an initial list of integers into a list of the negative integers and a list of the non-negative integers. Compare this example with the *trPartition* function in "Example 3" on page 302 which explicitly relies on a local recursive function.

Software Engineering with SKILL++

SKILL++ supports several modern software engineering methodologies, such as

- Procedural packages
- Modules
- Object-oriented programming with classes (see Chapter 16, SKILL++ Object System)

SKILL++ also facilitates information hiding. *Information hiding* refers to using private functions and private data which are not accessible to other parts of your application. Information hiding promotes reusability and robustness because your implementation is easier to change with no adverse effect on the clients of the module.

SKILL++ Packages

A *package* is a collection of functions and data. Functions within a package can share private data and private functions that are not accessible outside the package. Packages are a hallmark of modern software engineering.

SKILL++ facilitates two approaches to packages.

- You can explicitly represent the package as an collection of function objects and data. Clients of the package use the arrow (->) operator to retrieve the package functions. Different packages can have functions with the same name.
- You might want to reimplement a collection of SKILL functions as a SKILL++ package. Informal SKILL packages have no opportunity to hide private functions or data. Reimplementing a SKILL package in SKILL++ provides the opportunity. Usually, you want to do this in a way that clients do not need to change their calling syntax. In this case, you do not need to represent the collection as a data structure. Instead, the package exports some of its function objects and hides the remainder.

The Stack Package

```
Stack = let( ()
      procedure( getContents( aStack )
            aStack->contents
            ) ; procedure
      procedure( setContents( aStack aList )
            aStack->contents = aList
            ) ; procedure
      procedure( ppush( aStack aValue )
            setContents(
                  aStack
                  cons(
                        aValue
                        getContents( aStack )
                        ) ; cons
                  )
            ) ; procedure
      procedure( ppop( aStack )
            letseq( (
                        ( contents getContents( aStack ))
                        ( v car( contents ))
                        )
                  setContents( aStack cdr( contents ))
                  v
                  ) ; letseq
            ) ; procedure
      procedure( new( initialContents )
                        list( nil 'contents initialContents )
                        ) ; procedure
      list( nil 'ppop ppop 'ppush ppush 'new new )
      ) ; let

=> ( nil
            ppop funobj:0x1c9b38
            ppush funobj:0x1c9b28
            new funobj:0x1c9b48 )
```

Using the Stack Package

```
S = Stack->new( '( 1 2 3 4 )) => (nil contents ( 1 2 3 4 ))
Stack->ppop( S ) => 1
Stack->ppush( S 1 ) => (1 2 3 4)
```

Comments

The Stack package is represented by a disembodied property list. Alternate representations such as a defstruct are possible. The only requirement is that the package data structure obey the -> protocol.

Only the *ppush*, *ppop*, and *new* function are visible to the clients of the package.

The *ppush* and *ppop* functions use the *getContents* and *setContents* functions. If you choose a different representation for a stack, you only need to change the *new*, *getContents*,

and *setContents* functions. The *getContents* and *setContents* functions are hidden to protect the abstract behavior of a stack.

Retrofitting a SKILL API as a SKILL++ Package

```
define( stackPush nil )
define( stackPop nil )
define( stackNew nil )
let( ()
      procedure( getContents( aStack )
            aStack->contents
            ) ; procedure
      procedure( setContents( aStack aList )
            aStack->contents = aList
            ) ; procedure
      procedure( ppush( aStack aValue )
            setContents(
                  aStack
                  cons(
                        aValue
                        getContents( aStack )
                        ) ; cons
                  )
            ) ; procedure
      procedure( ppop( aStack )
            letseq( (
                        ( contents getContents( aStack ))
                        ( v car( contents ))
                        )
                  setContents( aStack cdr( contents ))
                  v
                  ) ; letseq
            ) ; procedure
      procedure( new( initialContents )
                        list( nil 'contents initialContents )
                        ) ; procedure
      stackPush = ppush
      stackPop = ppop
      stackNew = new
nil
) ; let
```

Using the stackNew, stackPop, and stackPush Functions

```
S = stackNew( '( 1 2 3 4 )) => (nil contents (1 2 3 4))
stackPop( S ) => 1
stackPush( S 5 ) => (5 2 3 4)
```

Comments

This example assumes *stackNew*, *stackPop*, and *stackPush* are the names of the functions to be exported from the stack package. As is customary, the package prefix *stack* informally indicates the functions that compose a package.

- The *getContents* and *setContents* functions are local functions invisible to clients.
- Using the *define* forms for *stackNew*, *stackPop*, and *stackPush* is not strictly necessary. Using the *define* form alerts the reader to those functions which the ensuing *let* expression assigns a value to an exported API.

SKILL++ Modules

You can structure a SKILL++ module around a creation function, which the client invokes to allocate one or more instances of the module. The client passes an instance to a procedural interface.

The *makeStack* and *makeContainer* functions in the following examples are creation functions in the following sense: when you call *makeStack* it “creates” a stack instance. The stack instance is a function object whose internals can only be manipulated (outside of the debugger) by the *ppushStack* and *popStack* functions.

The creation function has

- Arguments
- Local variables
- Several local functions that can access the arguments to the creation function and can communicate between themselves through the local variables

The creation function returns one of the following, depending on the implementation:

- A single local function object
- A data structure containing several of the local function objects
- A single function object which dispatches control to the appropriate local functions

Stack Module Example

A *stack* is a well-known data structure that allows the client to push a data value onto it and to pop a data value from it.

The Procedural Interface

The following table summarizes the procedural interface functions to the sample stack module.

Action	Function Call	Return Value
Allocate a stack.	makeStack(aList)	Function object
Push a value onto the stack.	pushStack(aStack aValue)	A list of the stack contents
Pop a value from the stack.	popStack(aStack)	A popped value

The variable *aStack* is assumed to contain a stack object allocated by calling the *makeStack* function.

Allocating a Stack

```
S = makeStack( '( 1 2 3 4 )) => funobj:0x1e36d8
```

Popping a Value

```
popStack( S ) => 1
popStack( S ) => 2
```

Pushing a Value on the Stack

```
pushStack( S 5 ) => (5 3 4)
```

Returns a list of stack contents at this point.

Implementing the makeStack Function

The *makeStack* function returns a function object. This function object is an instance of the stack module. In turn, this function object returns one of several functions local to the *makeStack* function.

```
procedure( makeStack( initialContents )
      let( (( theStack initialContents ))

            procedure( ppush( value )
                  theStack = cons( value theStack )
                  )

            procedure( ppop( )
                  let( (( v car( theStack ) ))
```

```
                        theStack = cdr( theStack )
                        v
                        )
                  )

            lambda( ( msg)           ;;;; return a function object
                  case( msg
                        ( ( ppush ) ppush )
                        ( ( ppop ) ppop )
                        ( t nil )
                        )
                  )
            ) ; let
      ) ; procedure
```

Implementing the pushStack and popStack Functions

The variable *aStack* contains a function object.

- When *aStack* is called, it returns the appropriate local function *ppush* and *ppop*.
- The *ppush* and *ppop* functions are within the lexical scope of the local variable *theStack*.

Notice the syntactic convenience of calling the stack object indirectly through the *fun* variable.

```
procedure( pushStack( aStack aValue )
      let( ( ( fun aStack( 'ppush ) ) )
                              ;;; retrieve local ppush function
            fun( aValue )         ;;; call it
            )
      )

procedure( popStack( aStack )
      let( ( ( fun aStack( 'ppop ) ) )
                              ;;; retrieve local ppop function
            fun()                 ;;; call it
            )
      )
```

The Container Module

Containers are like variables with an important difference. You can reset a container to the original value that you provided when you created the container.

The Procedural Interface

The following table summarizes the procedural interface to the sample container module. This interface relies on the availability of several function objects in the container instance's data structure. The arrow (->) operator is used to retrieve the interface functions.

Action	Function Call	Return Value
Allocate a container with an initial value.	aContainer = makeContainer(aValue)	A disembodied property list representing the container instance.
Return the container's current value.	aContainer->get()	Current value in *aContainer*
Store a new value in the container.	aContainer->set(bValue)	The container's new value.
Reset the container to the initial value	aContainer->reset()	The container's original value.

Implementing the makeContainer Function

- The *makeContainer* function returns a disembodied property list containing the local functions as property values.
- The three functions *resetValue*, *setValue*, and *getValue* are local but are accessible through *makeContainer'*s return value.
- Unlike the stack module example, there are no global functions in the procedural interface.

```
procedure( makeContainer( initialValue )
      let( ( (value initialValue))
                        ;;; initialize the container
            procedure( resetValue()              ;;; reset value
                  value = initialValue
                  )
            procedure( setValue( newValue ) ;;; store new value
                  value = newValue
                  )
            procedure( getValue( ) ;;; return current value
                  value
                  )

            resetValue()
            list( nil               ;;; the return value
                  'set setValue
                  'get getValue
                  'reset resetValue
                  )
```

```
            ) ; let
        ) ; procedure
```

Allocating Container Instances

```
x = makeContainer( 0 )
      => (nil
                  set funobj:0x1e38f8
                  get funobj:0x1e3908
                  reset funobj:0x1e38e8 )
```

This example allocates a container instance with initial value 0.

```
y = makeContainer( 2 )
      => (nil
                  set funobj:0x1e3928
                  get funobj:0x1e3938
                  reset funobj:0x1e3918 )
```

This example allocates a container instance with an initial value of 2. Notice that the returned value contains different function objects.

Retrieving Container Values

```
x->get() + y->get() => 2
```

This example retrieves the values of the two containers and adds them. Notice the conventional function call syntax accepts an arrow (->) operator expression in place of a function name to access member functions.

15

Using SKILL and SKILL++ Together

Overview information:

Introduction

This chapter discusses the pragmatics of developing programs in the Cadence® SKILL++ language. It assumes you are familiar with both the Cadence SKILL language and the SKILL++ language, specifically the material in About SKILL++ and SKILL on page 275 and Using SKILL++ on page 297

Developing a SKILL++ application involves the same basic tasks with which you are familiar from developing SKILL applications. Because most viable applications will involve tightly integrated SKILL and SKILL++ components, there are several more factors to consider. This chapter covers

Selecting an Interactive Language

When entering a language expression into the command interpreter, you need to choose the appropriate language “mode.”

Partitioning an Application Into a SKILL Portion and a SKILL++ Portion

You are free to implement your application as a heterogenous collection of source code files. You need to choose a file extension accordingly.

Cross Calling Between SKILL and SKILL++

In general, SKILL++ functions and SKILL functions can transparently call one another. However, a few families of SKILL functions can operate differently when called from SKILL than when called from SKILL++. You need to be able to identify such SKILL functions, adjust your expectations, and exercise caution, when calling them from SKILL++.

Debugging a SKILL++ Program

In a hybrid application, errors can occur in either SKILL functions or SKILL++ functions. Displaying the SKILL stack will reveal SKILL++ environments and SKILL++ function objects which you will want to examine.

Terminology

This chapter uses the following terminology

from within a SKILL program

to mean

- from a SKILL interactive loop
- from within SKILL source code, outside of a function definition
- from within a SKILL function

Similar definitions apply *from within a SKILL++ program*.

Communication Between SKILL and SKILL++

Data allocated with one language is accessible from the other language. For example, you can allocate a list in a SKILL++ function and retrieve data from it in a SKILL function. Both languages use the same print representations for data.

Selecting an Interactive Language

You can use SKILL or SKILL++ for interactive work. Both languages support a read-eval-print loop in which, repeatedly,

1. You enter a language expression.
2. The system parses, compiles, and evaluates the expression in accordance with either SKILL or SKILL++ languages syntax and semantics.
3. The system displays the result using the print representation appropriate to the result's data type.

Starting an Interactive Loop (toplevel)

Use the *toplevel* function to start an interactive loop with either SKILL or SKILL++. SKILL is the default.

To select the SKILL language, type

```
toplevel( 'il )
```

To select the SKILL++ language and the SKILL++ top-level environment, type

```
toplevel( 'ils )
```

To select the SKILL++ language and also the environment to be made active during the interactive loop, pass the environment object as the second argument.

```
toplevel( 'ils envobj( 0x1e00b4 ))
```

In this example, the environment object is retrieved from the print representation.

Exiting the Interactive Loop (resume)

Use the *resume* function to exit the interactive loop, returning a specific value. This value is the return value of the *toplevel* function. The following example is a transcript of a brief session, including prompts.

```
> R = toplevel( 'ils )
ILS-<2> resume( 1 )
1
> R
1               ;;;return value of the toplevel function.
```

Partitioning Your Source Code

You are free to implement your application as a heterogenous collection of source code files.The *load* and *loadi* functions select the language to apply to the source code based on the file extension.

```
FileA.il
FileB.il
FileC.ils
FileD.ils
FileE.ils
```

Functions defined in each file can call functions defined in the other files without regard to the language in which the functions are written. The syntax for function calls is the same regardless of whether the function called is a SKILL function or a SKILL++ function. Specifically,

- SKILL++ functions are visible to the SKILL portions of your application
- SKILL functions are automatically visible to the SKILL++ portions of your application

You are free to call SKILL application procedural interface functions from a SKILL++ program. In fact, most applications will continue to rely heavily on SKILL functions.

Cross-Calling Guidelines

Several key semantic differences between SKILL and SKILL++ dictate certain guidelines you should follow when calling SKILL functions from SKILL++. These semantic differences include

- All SKILL++ environments, other than the top-level environment, are invisible to SKILL

- All SKILL++ local variables are invisible to SKILL

You should avoid calling SKILL functions that call

- The *eval*, *symeval*, or *evalstring* functions
- The *set* function

You should avoid calling *nlambda* SKILL functions.

Avoid Calling SKILL Functions That Call eval, symeval, or evalstring

When you call the one-argument version of *eval*, *symeval*, or *evalstring* functions from a SKILL function, you are using dynamic scoping. Any symbol or expression which you pass to a SKILL function will probably evaluate to a different result than it would have in the SKILL++ caller.

In general, to determine whether a SKILL function calls any of these functions, you should consult the reference documentation.

Avoid Calling nlambda Functions

The nlambda category of SKILL functions are highly likely to call the *eval* or *symeval* functions.

A SKILL *nlambda* function receives all of its argument expressions unevaluated in a list. Such a function usually evaluates one or more of the arguments. The *addVars* SKILL function adds the values of its arguments.

```
nprocedure( addVars( args )
      let( (( sum 0 ))
            foreach( arg args
                  sum = sum+eval(arg)
                  ) ; foreach
            sum
            )
      )

let( ((x 1) (y 2) (z 3 ))
      addVars( x y z )
      )
=> 6
```

When called from SKILL++, the *eval* function uses dynamic scoping to resolve the variable references. In this case, the variable *x* was unbound.

```
ILS-<2> let( ((x 1) (y 2) (z 3 ))
      addVars( x y z )
      )
```

```
*WARNING* (addVars): calling NLambda from Scheme code -
      addVars(x y z)
*Error* eval: unbound variable - x
ILS-<2>
```

If necessary, reimplement the SKILL *nlambda* function as a SKILL or SKILL++ macro, using *defmacro*. For example,

```
defmacro( addVars ( @rest args )
      `let( (( sum 0 ))
            foreach( arg list( ,@args )
                  sum = sum + arg
                  ) ; foreach
            sum
            )
      )
```

Use the set Function with Care

Avoid calling SKILL functions that in turn call the *set* function. Usually such SKILL functions store values in other SKILL variables. If you call such a function from SKILL++ and pass a quoted local variable, the SKILL function will not store the value in the SKILL++ local variable. Instead, the value goes into the SKILL variable of the same name.

The following *SetMyArg* SKILL function behaves differently when called from SKILL++ than when called from SKILL.

SetMyArg Called from SKILL

```
> procedure( SetMyArg( aSymbol aValue )
      set( aSymbol aValue )
      ) ; procedure
SetMyArg
> let( ( ( x 3 ))
            SetMyArg( 'x 5 )
            x
            ) ; let
5
```

SetMyArg Called from SKILL++

```
> toplevel 'ils
ILS-<2> let( (( x 3 ))
      SetMyArg( 'x 5 )
      x
      ) ; let
3
```

Redefining Functions

During a single session, you are warned when you redefine a SKILL function to be a SKILL++ function, or visa versa.

You are only likely to encounter this when doing interactive work and are confused about which language “owns” the interaction.

Sharing Global Variables

It is generally desirable to avoid relying on global variables. However, it is sometimes necessary or expedient for the SKILL++ and SKILL portions of your application to communicate through global variables.

Using importSkillVar

Before the SKILL and SKILL++ portions of your application can share a global variable, you must first call the *importSkillVar* function. The SKILL++ global variable and the SKILL global variable will then be bound to the same location.

For example, consider the following interaction with a SKILL top level.

```
> delta = 2
2
> procedure( adder( y )
      delta+y
      ) ;
adder
>
```

To set the value of the *delta* variable from within a SKILL++ program, you should first

```
importSkillVar( delta )
```

as the following sample interaction with a SKILL++ top level shows.

```
> toplevel 'ils
ILS-<2>delta
*Error* eval: unbound variable - delta
ILS-<2> importSkillVar( delta )
ILS-<2> delta
2
ILS-<2> adder( 4 )
6
```

Note: You do not need to import *delta* just to call *adder* from within SKILL++ code.

How importSkillVar Works

Although understanding this level of detail is not necessary to effectively use *importSkillVar*, this section is provided for expert users.

In SKILL++, a variable is bound to a memory location called the variable's binding. The familiar operation of “storing a value in a variable” actually stores the value in the variable's binding. SKILL++ variable bindings are organized into environment frames. The SKILL++ top-level environment contains all the variable bindings initially available at system start up.

Normally, all global (top-level) SKILL++ variables are bound to the function slot of the SKILL symbol with the same name as the variable. For example, the variable *foo* is bound to the function slot of the symbol *foo*. Consequently, in SKILL++, when you retrieve the value of a SKILL variable, you are getting the contents of the symbol’s function slot.

The *importSkillVar* function directs the compiler to instead bind a SKILL++ global variable in the top-level environment to the value slot of the symbol with the same name. Informally, you can use *importSkillVar* to enable access to a SKILL symbol's current value binding from within SKILL++.

Evaluating an Expression with SKILL Semantics

As an advanced programmer, you might find that separating closely related SKILL code and SKILL++ code into different files is distracting or otherwise not convenient. For example, suppose that in the middle of a SKILL++ source code file you want to declare a SKILL function that refers to a SKILL variable.

Using the previous example

```
delta = 5
procedure( adder( y )
      delta+y )
```

The *inSkill* macro below allows you to splice SKILL language source code into a SKILL++ source code file. You can use the *inSkill* macro as shown.

```
inSkill(
      delta = 5
      procedure( adder( y )
            delta+y ) ; procedure
      )
```

Debugging SKILL++ Applications

This section addresses common tasks that arise when debugging hybrid SKILL and SKILL++ applications.

Retrieving a Function Object (funobj)

You can retrieve a function object from its print representation. This capability gives you full use of the information displayed in stack traces and environment objects. The argument to the *funobj* function should be the hexadecimal number displayed in the print representation. The following example confirms that the function object is indeed retrieved by applying it to some arguments.

```
lambda( ( x y ) x + y ) => funobj:0x1e3768
apply( funobj( 0x1e3768 ) list( 5 6 ) ) => 11
```

Examining the Source Code for a Function Object

Use the *pp* SKILL function to display the source code for a global function. The *pp* function expects that its argument is a symbol. It retrieves the function object stored in the function slot of the symbol you pass. To use *pp* to display the source code for a function object, store the function object in an unused global.

In SKILL++

```
G4 = funobj( 0x1e3628 )
pp( G4 )
```

In SKILL

```
putd( 'G4 ) = funobj( 0x1e3628 )
pp( G4 )
```

The *pp* function pretty-prints the function object stored in the function slot of the symbol. The *pp* function uses the global symbol to name the function object. This is only seriously misleading if the function object is recursive.

Pretty-Printing Package Functions

Use the *pp* function as explained above to pretty-print package functions.

```
MathPackage = let( ()
      procedure( add( x y ) x+y )
      procedure( mult( x y ) x*y )
      list( nil 'add add 'mult mult )
      )
      => (nil add funobj:0x1c9c48 nult funobj:0x1c9c58)
```

```
ILS-1> Q = MathPackage->add
funobj:0x1c9c48
ILS-1> pp( Q )
procedure( Q(x y)
      (x + y)
      )
```

Inspecting Environments

A significant SKILL++ application is likely to include many function objects, each with its own separate environment. While debugging, you may need to interactively examine or set a local variable in an environment other than the active environment.

You can

- Retrieve the active environment
- Inspect an environment with the -> operator
- Retrieve the environment of a function object

Retrieving the Active Environment

The *theEnvironment* function returns the enclosing lexical environment when you call it from within SKILL++ code.

Example 1

```
Z = let( (( x 3 ))
      theEnvironment()
      ) ; let
      => envobj:0x1e0060
```

This example returns the environment that the *let* expression establishes. The value of *Z* is an environment in which *x* is bound to *3*. Each time you execute the above expression, it returns a different environment object.

Example 2

```
W = let( (( r 3 ) ( y 4 ))
            let( (( z 5 ) ( v 6 ))
                  theEnvironment()
                  )
            )
```

This example returns the environment that the nested *let* expressions establish.

Testing Variables in an Environment (boundp)

Use the *boundp* function to determine whether a variable is bound in an environment. The optional second argument should be a SKILL++ environment.

```
boundp( 'b W ) => nil
boundp( 'r W ) => t
```

Using the -> Operator with Environments

You can use the -> operator against an environment to read and write variables bound in the environment.

```
W->z => 5
W->v = 100
```

Alternatively, you can use the *symeval* function to retrieve the value of a variable relative to an environment.

```
symeval( 'r W ) => 3
```

Alternatively, you can use the *set* function to set the value of a variable in an environment.

```
set( 'r 200 W ) => 200
```

Using the ->?? Operator with Environments

Use the ->?? operator to dump out the environment as a list of association lists with one association list for each environment frame.

```
W->?? => ( ((z 5) (v 6)) ((r 3) (y 4)) )
```

Evaluating an Expression in an Environment (eval)

Use the *eval* function to evaluate an expression in a given lexical environment.

```
eval( '( z+v ) W ) => 11
eval( '( z = 100 ) W ) => 100
eval( '( z+v ) W ) => 106
```

Retrieving an Environment (envobj)

You can retrieve an environment object from its print representation. The argument to the *envobj* function should be the hexadecimal number displayed in the print representation. This capability gives you full use of the information displayed in stack traces.

```
E => envobj:0x1e00b4
envobj( 0x1e00b4 ) => envobj:0x1e00b4
```

Examining Closures

As function objects, closures have both source code and data. For example, consider the following closure generated when the *makeAdder* function is called.

```
procedure( makeAdder( delta )
      lambda( ( x ) x + delta )
      )
=> makeAdder
add5 = makeAdder( 5 )
=> funobj:0x1fe668
```

Examining the Source Code

Use the *pp* function as explained above to examine the source code for the closure.

```
ILS-1> pp( add5 )
procedure( add5(x)
      (x + delta)
)
nil
```

Examining the Environment

Install the SKILL Debugger and use the *theEnvironment* function to retrieve the environment for the function object. Use the *->??* operator to examine the environment.

```
theEnvironment( funobj( 0x1fe668 ) )->??
=> (((delta 5)))
```

Refer to the *makeStack* example in “Implementing the makeStack Function” on page 313

```
S = makeStack( '( 1 2 3 )) => funobj:0x1e3758
E = theEnvironment( S ) => envobj:0x1e00b4
E->push => funobj:0x1e3738
E->initialContents => (1 2 3)
```

General SKILL Debugger Commands

Tracing (tracef)

You can only use the *tracef* function to trace SKILL functions or SKILL++ functions defined in the top-level SKILL++ environment.

Setting Breakpoints

You can only set breakpoints at SKILL functions or SKILL++ functions defined in the top-level SKILL++ environment.

Calling the break Function

You can insert a call to the *break* function but that requires redefining the function that calls the *break* functions. If called from SKILL++, the enclosing lexical environment is the active environment during the debugger session.

```
ILS-<2> MathPackage = let( ()
      procedure( add( x y ) break() x+y )
      procedure( mult( x y ) x*y )
      list( nil 'add add 'mult mult )
      )
(nil add funobj:0x1c9ca8 mult funobj:0x1c9cb8)
ILS-<2> MathPackage->add( 3 4 )
<<< Break >>> on explicit 'break' request
SKILL Debugger: type 'help debug' for
      a list of commands or debugQuit to leave.
ILS-<3> theEnvironment()->??
(((x 3)
            (y 4)
      )
      ((add funobj:0x1c9ca8)
            (mult funobj:0x1c9cb8)
      )
)
ILS-<3>
```

Examining the Stack (stacktrace)

During the execution of both SKILL and SKILL++ function calls, use the *stacktrace* function to examine the SKILL stack. The stack will probably contain several function object and environment object references. You can use the techniques discussed above to display source code for a function object and to examine an environment.

For example, at the break point in the previous example notice that the *break* function passes the active environment to the break handler.

```
ILS-<3> stacktrace()
<<< Stack Trace >>>
breakHandler(envobj:0x1bb108)
break()
funcall((MathPackage->add) 3 4)
toplevel('ils)
4
ILS-<3>
```

16

SKILL++ Object System

Overview information:

Introduction

To gain benefits from object-oriented programming, the Cadence® SKILL language requires extensions beyond lexical scoping and persistent environments.

The Cadence SKILL++ Object System allows for object-oriented interfaces based on classes and generic functions composed of methods specialized on those classes. A class can inherit attributes and functionality from another class known as its superclass. SKILL++ class hierarchies result from this single inheritance relationship.

To attain the maximum benefit from the SKILL++ Object System, you should only use it with lexical scoping, because lexical scoping magnifies the power of the interfaces you can develop with the SKILL++ Object System.

You do not need to be familiar with another object-oriented programming language or system to understand or use the SKILL++ Object System. However, if you are familiar with the Common Lisp Object System (CLOS), the following comments will help you apply your experience to learning the SKILL++ Object System.

The Common Lisp Object System

The SKILL++ Object System is modelled after a subset of the Common Lisp Object System with three restrictions. In the SKILL++ Object System

- A class can have only a single superclass. CLOS allows a class to have more than one superclass.
- A method can only be applied to the class of the method's first argument.
- All methods are primary. CLOS allows for ancillary methods known as *before* and *after* methods which are combined with the primary method.

Basic Concepts

The central concepts of the SKILL++ Object System are class, instance, generic functions, and methods.

Classes and Instances

A *class* is a data structure template. A specific application of the template is termed an *instance*. All instances of a class have the same slots. SKILL++ Object System provides the following functions.

- *defclass* function to create a class
- *makeInstance* function to create an instance of a class

Generic Functions and Methods

A *generic function* is a collection of function objects. Each element in the collection is called a *method*. Each method corresponds to a class. When you call a generic function, you pass an instance as the first argument. The SKILL++ Object System uses the class of the first argument to determine which methods to evaluate.

To distinguish them from SKILL++ Object System generic functions, SKILL functions are called *simple functions*. The SKILL++ Object System provides the following functions.

- *defgeneric* function to declare a generic function
- *defmethod* function to declare a method

Subclasses and Superclasses

SKILL++ Object System provides for one class B to inherit structure slots and methods from another class A. You can describe the relationship between the class A and class B as follows:

- B is a subclass of A
- A is a superclass of B

Defining a Class (defclass)

The domain of geometric objects provides good examples for using object oriented programming. Use the *defclass* function to define a class. You specify the superclass, if any, and all the slots of the class.

```
defclass( GeometricObject
      ()              ;;; superclass
      ()              ;;; list of slot descriptions
      ) ; defclass
```

This example defines the *GeometricObject* class. Defining the *GeometricObject* class allows the subsequent definition of default behavior of all geometric objects. It has no slots. Because no superclass is specified, the superclass is the *standardObject* class.

```
defclass( Triangle
      ( GeometricObject ) ;;; superclass
      (
```

```
            ( x )       ;;; x slot description
            ( y )       ;;; y slot description
            ( z )       ;;; z slot description
            )
      ) ; defClass
```

This example defines the *Triangle* class. It declares that

- The *Triangle* class is a subclass of the *GeometricObject* class
- Each instance shall have three slots named *x*, *y*, and z

Slot Options

Slot options, also known as slot specifiers, govern how you initialize the slot as well as your access to the slot.

Slot Option	Value	Meaning
@initarg	symbol	Defines a keyword argument for the *makeInstance* function.
@initform	expression	Defines an expression which initializes the slot.
@reader	symbol	Defines a generic function with this name. The function returns the value of the slot.
@writer	symbol	Defines a generic function with this name. The function accepts a single argument which becomes the new slot value.

Example 1

```
defclass( Triangle
      ( GeometricObject )
      (
            ( x
                  @initarg x
                  )
            ( y
                  @initarg y
                  )
            ( z
                  @initarg z
                  )
            )
      ) ; defClass
```

Example 2

```
defclass( Circle
      ( GeometricObject )
      (
            ( r @initarg r )
            )
      ) ; defClass
```

Instantiating a Class (makeInstance)

Use the *makeInstance* function to instantiate a class. The first argument designates the class you are instantiating. Subsequent keyword arguments initialize the instance's slots. The *makeInstance* function returns the newly allocated instance of the class.

```
procedure( makeTriangle( x y z )
      if( x<y+z && y<z+x && z<x+y
            then
                  makeInstance( 'Triangle
                        ?x 1.0*x
                        ?y 1.0*y
                        ?z 1.0*z
                        )
            else
                  error(
                        "%n %n %n fail triangle inequality test\n"
                        x y z
                        )
            ) ; if
      ) ; procedure

exampleTriangle = makeTriangle( 3 4 5 ) => stdobj:0x1e6030
```

The print representation for a SKILL++ Object System instance consists of *stdobj:* followed by a hexadecimal number.

Reading and Writing Instance Slots

The Arrow (->) Operator

You can use the arrow operator to read a slot's value.

```
exampleTriangle->x => 3.0
exampleTriangle->y => 4.0
exampleTriangle->z => 5.0
```

You can use the -> operator on the left-side of an assignment statement.

```
exampleTriangle->x = 3.5
```

The ->?? expression returns a list of the slots and their values.

Generic Functions for Accessing Slots

Another approach is to use the *@reader* and *@writer* slot options to define generic functions for reading and writing slots when you define the class.

```
defclass( Triangle
      ( GeometricObject )
      (
            ( x
                  @initarg          x
                  @reader           get_x
                  @writer           set_x
                  )
            ( y
                  @initarg          y
                  @reader           get_y
                  @writer           set_y
                  )
            ( z
                  @initarg          z
                  @reader           get_z
                  @writer           set_z
                  )
            )
      ) ; defClass

exampleTriangle = makeTriangle( 3 4 5 ) => stdobj:0x1e603c
get_y( exampleTriangle ) => 4.0
set_x( exampleTriangle 3.5 ) => 3.5
```

Defining a Generic Function (defgeneric)

Use the *defgeneric* function to define a generic function. The body of the generic function defines the default method for the generic function.

```
defgeneric( Perimeter ( geometricObject )
      error( "Subclass responsibility\n" )
      ) ; defgeneric
```

This example indicates that relevant subclasses of the *geometricObject* class, such the *polygon* class, should have a *Perimeter* method. Although not strictly necessary to do so, defining a generic function before defining any methods for it has two advantages.

You Can Specify a Default Method

Using the *defgeneric* function gives you control over the default method. When you invoke a generic function that has no default method, the SKILL++ Object System raises an error. The following example illustrates calling a generic function which does not have a method defined for the specific argument.

```
Perimeter( 3 )
*Error* (Default-method) generic:Perimeter class:fixnum
```

You Can Document the Template Argument List

All methods for a generic function must have congruent argument lists. See "Method Argument Restrictions" on page 342.

In the absence of a generic function, the first method you define automatically declares the generic function. The method's argument list becomes the template argument list.

Defining a Method (defmethod)

Use the *defmethod* function to define a method. You do not need to define the generic function before you define a method for it. When you invoke a generic function, the SKILL++ Object System chooses the method to run based on the class of the first argument you pass to the function.

- If you use conventional syntax *defmethod(…)* in place of the LISP syntax *(defmethod …)*, use white space to separate the method name from the argument list.
- You must specify the class of the method's first argument to the *defmethod* function. The first argument to *defmethod* uses the syntax:

```
( s_arg1 s_class )
```

Example 1

```
defmethod( Perimeter (( triangle Triangle ))
      let( (
            (x triangle->x)
            (y triangle->y)
            (z triangle->z)
            )
            x+y+z
            ) ; let
      ) ; defmethod

Perimeter( exampleTriangle ) => 12.0
```

This example defines a method named *Perimeter*. It is specialized on the *Triangle* class.

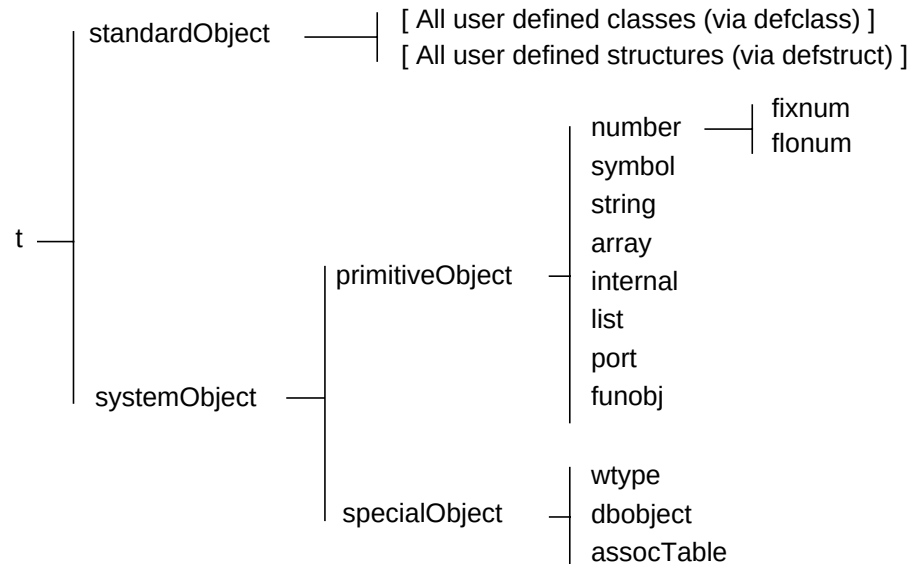
Example 2

```
defmethod( Perimeter (( c Circle ))
      2*c->r*3.1415
      ) ; defmethod
```

This example defines a *Circle* class and defines the *Perimeter* method for the *Circle* class.

Class Hierarchy

The diagram below is a horizontal view of the SKILL++ Object System class hierarchy.



- *t* is the superclass of all classes. Class *t* has two immediate subclasses, *standardObject* and *systemObject*.
- *standardObject* is the superclass of all classes you define with *defclass* function. This is the primary portion of the class hierarchy that you can extend.
- *systemObject* is the superclass of *primitiveObject* and *specialObject*. No subclasses of *systemObject* can be used with *defclass*. However, you can add a subclass of *specialObject* by passing a *defstruct* to the *addDefstructClass* function.
- *primitiveObject* is the superclass of all SKILL built-in classes.
- *specialObject* is the superclass of all classes corresponding to the C-level registrable “user-types.” It is also the superclass of all classes you define with the *addDefstructClass* function.

This example shows how you can list all of the subclasses. Run this program to see what the class hierachy is at any given time.

```
procedure( getDirectSubclasses( classObject )
      foreach( mapcar c subclassesOf( classObject )
            className( c )
            ) ; foreach
      ) ; procedure
```

```
procedure( getAllSubclasses( classObject )
      let( ( direct )
            direct = getDirectSubclasses( classObject )
            cons(
                  className( classObject )
                  direct && foreach( mapcar c direct
                        getAllSubclasses( findClass( c ))
                        ) ; foreach
                  ) ; cons
            ) ; let
      ) ; procedure
getAllSubclasses( findClass( 't )) =>

(t
      (standardObject
            (GeometricObject
                  (Triangle)
                        (Point)
                        )
                  )
      (systemObject
            (primitiveObject
                  list()
                  (port)
                  (funobj)
                  (array)
                  (string)
                  (symbol)
                  (number
                        (fixnum)
                        (flonum)
                        )
                  )
      ( specialObject
            (other)
            (assocTable)
            )
      )
)
```

Browsing the Class Hierarchy

The SKILL++ Object System provides a number of functions for browsing the class hierarchy. Examples in the following sections refer to the following code.

```
defclass( GeometricObject
      ()      ;;; superclass
      ()      ;;; list of slot descriptions
      ) ; defclass
defclass( Triangle
      ( GeometricObject ) ;;; superclass
      (
            ( x @initarg x )  ;; x slot description
            ( y @initarg y )  ;; y slot description
            ( z @initarg z )  ;; z slot description
            )
      ) ; defClass
```

```
exampleTriangle = makeTriangle( 3 4 5 ) => stdobj:0x1e6030
```

Getting the Class Object from the Class Name

Use the *findClass* function to get the class object from its name. Use a SKILL symbol to represent the class name.

```
findClass( 'Triangle ) => funobj:0x1cb2d8
```

Getting the Class Name from the Class Object

Use the *className* function to get the class symbol. The term *class symbol* refers to the symbol used to represent the class name. The SKILL++ Object System uses a SKILL symbol to represent the class name.

```
className( findClass( 'Triangle )) => Triangle
```

Getting the Class of an Instance

Use the *classOf* function to get the class of an instance.

```
className( classOf( exampleTriangle ) ) => Triangle
```

Getting the Superclasses of an Instance

Use the *superclassesOf* function to get the superclasses of a class. The function returns a list of class objects.

```
L = superclassesOf( classOf( exampleTriangle ) )

foreach( mapcar classObject L
      className( classObject )
      ) ; foreach
=> (Triangle GeometricObject standardObject t)
```

Checking if an Object Is an Instance of a Class

Use the *classp* function to check if an object is an instance of a class. You can pass either the class symbol or the class object as the second argument.

Example 1

```
classp( exampleTriangle 'Triangle ) => t
classp( 5 'fixnum ) => t
classp( 5 'Triangle ) => nil
```

5 is a *fixnum*. 5 is not an instance of *Triangle*.

Example 2

```
classp( exampleTriangle 'GeometricObject ) => t
classp( exampleTriangle 'standardObject ) => t
classp( exampleTriangle t ) => t
```

This example illustrates that *classp* returns *t* for all superclasses of the class of an instance. *Triangle* is a subclass of *GeometricObject*. *GeometricObject* is a subclass of *standardObject*. *standardObject* is a subclass of *t*.

Checking if One Class Is a Subclass of Another

Use the *subclassp* function to determine whether one class is a subclass of another.

Example 1

```
subclassp(
      findClass( 'Triangle )
      findClass( 'GeometricObject )
      ) => t
```

Triangle is a subclass of *GeometricObject*.

Example 2

```
subclassp(
      findClass( 'Triangle )
      findClass( t )
      ) => t
```

Triangle is a subclass of *t*.

Example 3

```
subclassp(
      findClass( 'Triangle )
      findClass( 'fixnum )
      ) => nil
```

Triangle is not a subclass of *fixnum.*

Advanced Concepts

This section covers more advanced aspects of the SKILL++ Object System.

Method Argument Restrictions

There are several restrictions concerning method argument lists.

Number of Arguments

All methods of a generic function must have the same number of required arguments and *@optional* arguments.

@rest Arguments

All methods of a generic function must take *@rest* arguments if any of the methods take *@rest* arguments.

Keyword Arguments

Each method of a generic function must

- Take *@rest* arguments if any of the methods take @rest arguments
- Allow a superset of the keyword arguments specified in the *defgeneric* declaration

@rest argument will pick up all the keyword arguments that have no matching keyword in the formal argument list. Different methods may have different default forms for the optional arguments and may accept different set of keywords, however.

Applying a Generic Function

When you apply a generic function to some arguments, the SKILL++ Object System performs the following actions, called *method dispatching*, to complete the function call. The SKILL++ Object System

1. Retrieves the methods of the generic function
2. Determines the class of the first argument to the generic function

   Based on the class of the first argument passed to the generic function, the SKILL++ Object System finds

   - No applicable methods

     SKILL++ Object System calls the default method for the generic function if one exists. Otherwise it signals an error.

- Exactly one method
- More than one applicable method

  This situation occurs when you have methods specialized on one or more superclasses of the first argument's class.

3. Determines applicable methods by examining the method's class specializer. A method is applicable if it specialized on the class of the first argument or a superclass of the class of the first argument.
4. Sorts the applicable methods according to the chain of superclasses of the first argument's class.
   - The first method in the ordering is the *most specific* method.
   - The last method in the ordering is the *least specific* method.
5. Calls the first method. Invoke the *callNextMethod* function from within a method to access the next applicable method in the ordering.

Using the callNextMethod Function

```
defgeneric( describe (obj) ())
defclass( GeometricObject () () ) ; no slots or superclasses

defclass( Point
      ( GeometricObject )
      (
            ( name      @initarg name )
            ( x         @initarg x );;; x-coordinate
            ( y         @initarg y );;; y-coordinate
            )
      ) ; defclass

defmethod( describe (( object GeometricObject ))
      className( classOf( object ))
      ) ; defmethod

defmethod( describe (( p Point ))
      sprintf( nil "%s %s @ %n:%n"
            callNextMethod( p )
            p->name
            p->x
            p->y
            )
      ) ; the most specific method

aPoint = makeInstance( 'Point ?name "A" ?x 1 ?y 0 )
describe( aPoint )
      => "Point A @ 1:0"
```

In the example, the *describe* generic function has two methods that are applicable to the argument *aPoint*:

- The method specialized on the *Point* class
- The method specialized on the *GeometricObject* class

The method specializing on the *Point* class is the more specific method, therefore the SKILL++ Object System applies the most specific method to the argument.

Incremental Development

One of the hallmarks of the SKILL++ environment is your ability to incrementally redefine SKILL++ functions. Observe the following guidelines when redefining SKILL++ Object System elements of your application.

Redefining Methods

During development, you can expect to redefine methods about as frequently as you redefine procedures. You can redefine a method as long as the redefined method's argument list continues to conform to the generic function.

Redefining Generic Functions

You need to redefine a generic function to change the generic function's default method or argument list. Such need occurs infrequently. When you redefine a generic function, the SKILL++ Object System discards all existing methods for the generic function.

Redefining Classes

You need to redefine a class when you want to

- Change the superclass
- Add or remove a slot
- Add or remove a slot option

If you need to redefine a class, you should exit the SKILL++ environment and reload you application. A frequent need to redefine classes probably indicates that you should analyze your application before further programming.

Methods versus Slots

Methods are generally more expensive to use compared to slots but they offer data hiding and safety. Consider whether the *Triangle*'s *Area* method should access a slot containing the (precomputed) area or whether the area should be computed on the fly. The nature of your application dictates your final decision.

Computing the area on the fly may be costly if, for example, the area of triangles is used often. In such a situation it would be more advantageous to add a slot for area to the triangle class. But then we would have to add @writer methods for the sides of a triangle to recalculate the area when the length of a side changes.

Sharing Private Functions and Data Between Methods

Using lexical scoping with the SKILL++ Object System allows all methods specialized on a class to share private functions and data.

The methods for a class might need access to data, such as an association table, that is shared between all instances of the class. Slots you specify in the *defclass* declaration are allocated within each instance of the class.

The methods for a class might all rely on certain helper functions which you need to make private.

Using the following template as a guide achieves both goals.

```
defgeneric( Fun1 ( obj ... ) ... )
defgeneric( Fun2 ( obj ... ) ... )
defclass( Example ( ... ) ( ... ))
let(
      (
            ( classVar1 ... )              ;data shared between all
            ( classVar2 ... ) .... )       ;instances of the class
            )
      procedure( HelpFun1( .... ) ;private helper functions
      procedure( HelpFun2( .... )
      ...
      defmethod( Fun1 (( obj Example) .... )
      defmethod( Fun2 (( obj Example ) ... )
      ....
      ) ; let
```

17

Programming Examples

Overview information:

List Manipulation

A list is a linear sequence of Cadence® SKILL language data objects. The elements of a list can have any data type, including symbols or other lists. The printed presentation for a SKILL list uses a matching pair of parentheses to enclose the printed representations of the list elements. The *trListIntersection* and *trListUnion* functions illustrate

- Documenting at the function level
- Using the *setof* function
- Using the *member* function

The *trListUnion* function also illustrates the *nconc* function, which destroys all but its last argument. In this case, the first argument is a new, anonymous list created by the *setof* function.

```
procedure( trListIntersection( list1 list2 )
      setof( element list1
            member( element list2 )
      ) ; setof
) ; procedure

procedure( trListUnion( list1 list2 )
      nconc(
            setof( element list2
                  !member( element list1)
            ) ; setof
            list1
      ) ; nconc
) ; procedure

trListIntersection( ‘(a b c) ‘(b c d)) => (b c)
trListUnion( list(1 2 3) list(3 4 5 6)) => ( 4 5 6 1 2 3)
```

Symbol Manipulation

A symbol is the primary data structure within SKILL. A SKILL symbol has four data slots: the name, the value, the function definition, and the property list. Except for the name slot, all slots can be empty.

The *trReplaceSymbolsWithValues* function makes a copy of an arbitrary SKILL expression, in which all references to a symbol are replaced by the symbol’s value.

```
a = "one" b = "two" c = "three"
testCase = '( 1 2 ( a b ) )
trReplaceSymbolsWithValues( testCase ) => (1 2 ("one" "two"))

testCase = '(1 ( a ( c )) b )
trReplaceSymbolsWithValues( testCase ) =>
      (1 ("one" ("three")) "two")
```

The *trReplaceSymbolsWithValues* illustrates

- Using recursion to process an arbitrary SKILL expression, making a copy of the expression in which all references to a symbol are replaced by the symbol's value.
- How the *cond* function handles several possibilities.

The *listp* function determines whether the expression is a list.

The *symbolp* function determines whether the expression is a list.

The *symeval* function retrieves the value of the expression, provided it is a symbol.

In the general case, the *cond* function recursively descends into the *car* of the expression and the *cdr* of the expression, and builds a list from the results.

```
procedure( trReplaceSymbolsWithValues( expression )
            values for symbols."
      cond(
            ( null( expression ) nil )
            ( symbolp( expression ) symeval( expression ) )
            ( listp( expression )
                  cons(
                        trReplaceSymbolsWithValues( car( expression ))
                        trReplaceSymbolsWithValues( cdr( expression ))
                        )
                  )
            ( t expression )
      ) ; cond
) ; procedure

x = 5
a = 1
trReplaceSymbolsWithValues( '(x a))
=> (5 1)
```

Sorting a List of Points

The *trPointLowerLeftp* function indicates whether *pt1* is located to the lower left of *pt2*. This function illustrates

- Documenting at the function level
- Using the *xCoord* and *yCoord* functions

  Note: The *xCoord* and *yCoord* functions are aliases for the *car* and *cadr* functions.
- Using the *cond* function

```
procedure( trPointLowerLeftp( pt1 pt2 )
      let( ( pt1x pt2x pt1y pt2y )
            pt1x = xCoord( pt1 )
            pt2x = xCoord( pt2 )
```

```
            cond(
                  ( pt1x < pt2x t )
                  ( pt1x == pt2x
                        pt1y = yCoord( pt1 )
                        pt2y = yCoord( pt2 )
                        pt1y < pt2y )
                  ( t nil )
                  ) ; cond
      ) ; let
) ; procedure

trPointLowerLeftp( 0:0 100:100)
=> t

trPointLowerLeftp( 100:100 0:0)
=> nil
```

The *trSortPointList* function returns a list of points sorted destructively and illustrates

- Documenting at the function level
- Using the *sort* functio

```
procedure( trSortPointList( aPointList )
      sort( aPointList 'trPointLowerLeftp )
); procedure

x = list(100:0 100:100 0:0 50:50 0:100)
trSortPointList( x )
=>
(( 0 0)
(0 100)
(50 50)
(100 0)
(100 100))
```

Computing the Center of a Bounding Box

The *trBBoxCenter* function returns the point at the center of a bounding box and illustrates

- Documenting at the function level
- Using the *xCoord* and *yCoord* function
- Using the *lowerLeft* and *upperRight* function
- Using the colon (:) operator to build a point

```
procedure( trBBoxCenter( bBox )
      let( ( llx lly urx ury )
            ury = yCoord( upperRight( bBox ))
            urx = xCoord( upperRight( bBox ))

            llx = xCoord( lowerLeft( bBox ))
            lly = yCoord( lowerLeft( bBox ))
            ( urx + llx )/2 : ( ury + lly )/2
      ) ; let
) ; procedure
```

```
trBBoxCenter( list(0:0 100:100))
=> (50 50)
```

Computing the Area of a Bounding Box

The *trBBoxArea* function returns the area of a bounding box and illustrates

- Documenting at the function level
- Using the *xCoord* and *yCoord* functions
- Using the *lowerLeft* and *upperRight* functions
- Using parentheses to change priority of arithmetic operations

```
procedure( trBBoxArea( bBox )
      let( ( llx lly urx ury )
            urx = xCoord( upperRight( bBox ))
            ury = yCoord( upperRight( bBox ))
            llx = xCoord( lowerLeft( bBox ))
            lly = yCoord( lowerLeft( bBox ))
            ( ury - lly ) * ( urx - llx )
      ) ; let
) ; procedure
```

Computing a Bounding Box Centered at a Point

The *trDot* function returns bounding box coordinates with a given point as its center and illustrates

- Using @*key* to declare keyword arguments
- Establishing default values for a keyword argument
- Documenting at the function level
- Using the *let* function
- Using the *xCoord* and *yCoord* functions
- Building a bounding box with the *list* function and colon (:) operator

```
procedure( trDot( aPoint @key ( deltaX 1 ) ( deltaY 1 ) )
      let( ( llx lly urx ury aPointX aPointY )
            aPointX = xCoord( aPoint )
            aPointY = yCoord( aPoint )
            llx = aPointX - deltaX
            urx = aPointX + deltaX
            lly = aPointY - deltaY
            ury = aPointY + deltaY
            list( llx:lly urx:ury )
```

```
      ) ; let
) ; procedure

trDot( 100:100 ?deltaX 50 ?deltaY 50)
=> ((50 50) (150 150))
```

Computing the Union of Several Bounding Boxes

The *trBBoxUnion* function returns the smallest bounding box coordinates containing all the boxes in a given list and illustrates

- Using *foreach(mapcar ...)*
- Using the *apply* function with the *min* and *max* functions
- Using the *list* function and the colon (:) operator to construct a bounding box
- Documenting at the function level

```
procedure( trBBoxUnion( bBoxList )
      let( ( llxList llyList
            urxList     uryList
            minllx      minlly
            maxurx      maxury
            )
            llxList = foreach( mapcar bBox bBoxList
                  xCoord( lowerLeft( bBox )))
            llyList = foreach( mapcar bBox bBoxList
                  yCoord( lowerLeft( bBox )))
            urxList = foreach( mapcar bBox bBoxList
                  xCoord( upperRight( bBox )))
            uryList = foreach( mapcar bBox bBoxList
                  yCoord( upperRight( bBox )))
            minllx = apply( 'min llxList )
            minlly = apply( 'min llyList )
            maxurx = apply( 'max urxList )
            maxury = apply( 'max uryList )
            list( minllx:minlly maxurx:maxury )
      ) ; let
) ; procedure

trBBoxUnion( list( list(0:0 100:100) list(50:50 150:150)))
=> ((0 0) (150 150))
```

Computing the Intersection of Bounding Boxes

The *trBBoxIntersection* function illustrates

- Using *foreach(mapcar ...)*
- Using the *apply* function with the *min* and *max* functions
- Using the *cond* function

- Using the *list* function and colon (:) operator to construct a bounding box

```
procedure( trBBoxIntersection( bBoxList )
      let( ( llxList llyList
            urxList     uryList
            maxllx      maxlly
            minurx      minury      )
            llxList = foreach( mapcar bBox bBoxList
                  xCoord( lowerLeft( bBox )))
            llyList = foreach( mapcar bBox bBoxList
                  yCoord( lowerLeft( bBox )))
            urxList = foreach( mapcar bBox bBoxList
                  xCoord( upperRight( bBox )))
            uryList = foreach( mapcar bBox bBoxList
                  yCoord( upperRight( bBox )))
            minurx = apply( 'min urxList )
            minury = apply( 'min uryList )
            maxllx = apply( 'max llxList )
            maxlly = apply( 'max llyList )
            cond(
                  ( maxllx >= minurx nil )
                  ( maxlly >= minury nil )
                  ( t
                        list( maxllx:maxlly minurx:minury ))
                  ) ; cond
      ) ; let
) ; procedure

trBBoxIntersection( list( list(0:0 100:100) list(50:50 150:150))) =>
( (50 50) (100 100))
```

Prime Factorizations

A prime factorization of an integer is a list of pairs and is an example of an association list. The first element of each pair is a prime number that divides the number and the second element is the exponent to which the prime is to be raised. Each such pair is termed a prime-exponent pair.

```
pf1 = '( ( 2 3 ) ( 3 5 ))
pf2 = '( ( 3 2 ) ( 5 2 ) ( 7 3 ))

pf3 = '( ( 3 2 ) ( 5 4 ) ( 7 2 ))
pf4 = '( ( 3 6 ) ( 7 3 ) ( 11 2 ) ( 5 1 ))
```

The *assoc* function is used to determine whether a prime number occurs in a prime factorization. It returns either the prime-exponent pair or *nil*. For example:

```
assoc( 2 pf1 ) => ( 2 3 )
assoc( 7 pf2 ) => ( 7 3 )
```

Evaluating a Prime Factorization

To evaluate the prime factorization means to perform the arithmetic operations implied:

- For each prime-exponent pair, raise the prime to the corresponding exponent
- Multiply the resulting list of integers together

For example, evaluating the prime factorization

```
( ( 3 2 ) ( 5 2 ) ( 7 3 ))
```

is equivalent to evaluating

```
3**2 * 5**2 * 7**3
```

The *trTimes* functions multiplies a list of numbers together. It handles two cases that the *times* function does not handle.

The *trTimes* function illustrates

- Using an @rest argument to collect an arbitrary number of arguments into a list.
- Using the *apply* function. In the general case, we use the *apply* function to invoke the normal times function
- Using the *cond* function
- The *null* function tests for an empty list. The *onep* function tests whether a number is one

```
procedure( trTimes( @rest args )
      cond(
            ( null( args ) 1 )
            ( onep( length( args ) ) car( args ))
            ( t apply( 'times args ))
      ) ; cond
) ; procedure
```

The *trEvalPF* function evaluates the prime factorizations. For example:

```
pf1 = '( ( 2 3 ) ( 3 5 ))
pf2 = '( ( 3 2 ) ( 5 2 ) ( 7 3 ))
trEvalPF( pf1 ) => 1944
trEvalPF( pf2 ) => 77175
```

The *trEvalPF* function illustrates

- Using the *apply* function with the *trTimes* function above
- Using the *foreach(mapcar …)*
- Using the *car* and *cadr* functions

```
procedure( trEvalPF( pf )
      apply( 'trTimes
            foreach( mapcar pePair pf
                  car( pePair )**cadr( pePair )
            ) ; foreach
```

```
        ) ; apply
    ) ; procedure
```

Computing the Prime Factorization

The *trLargestExp* function returns the largest *x* such that divisor ** x <= number and illustrates

- Documenting at the function level
- Using the *preincrement* operator
- Using the *let* expression to initialize a local variable to a non-nil value
- Using the *while* function

```
procedure( trLargestExp( number divisor )
      let( ( ( exp 0 ) )
            while( zerop( mod( number divisor ) )
                  ++exp
                  number = number / divisor
                  ) ; while
            exp
            ) ; let
      ) ; procedure
```

The *trPF* function returns the prime factorization a number. For example:

```
trPF( 1003 )     => ((59 1) (17 1))
trPF( 10003 )    => ((1429 1) (7 1))
trPF( 100003 )   => ((100003 1))
trPF( 123456 )   => ((643 1) (3 1) (2 6))
```

The *trPF* function illustrates

- Documenting at the function level
- Using the *postincrement* operator
- Using the *let* expression to initialize a local variable to a non-*nil* value
- Using the *while* function
- Using parentheses to control operator precedence
- Using *let* to initialize a local variables to non-nil values
- Using a *while* loop
- Using *when* with an embedded assignment expression
- Using *trLargestExp*

```
procedure( trPF( aNumber )
      let(       (       ; locals
            ( divisor 2 )
            result
            exp
            ( num aNumber )
            )
            while( num > 1
                  when( ( exp = trLargestExp( num divisor )) > 0
                        result = cons( list( divisor exp ) result )
                        num = num / divisor ** exp
                        ) ; when
                  divisor++        ;;; try next divisor
                  ) ; while
            result
      ) ; let
) ; procedure
```

Multiplying Two Prime Factorizations

The *trPFMult* function returns the prime factorization of the product of two prime factorizations*, pf1* and *pf2*. The *trPFMult* function uses the following algorithm to construct the resultant prime factorization.

1. Those prime-exponent pairs whose primes occur in only one of the prime factorizations lists are carried across unaltered into the resultant prime factorization.
2. For those primes that have entries in both prime factorizations, a prime-exponent pair using the prime is included with an exponent equal to the sum of prime's exponent in either prime factorization.

The *trPFMult* function illustrates

- Using the *setof* function and the *assoc* function to build two prime factorizations

  The list *pf1Notpf2* contains all the prime-exponent pairs in *pf1* whose prime does not occur in *pf2*.

  The list *pf2Notpf1* contains all the prime-exponent pairs in *pf2* whose prime does not occur in *pf1*.

- Using *foreach(mapcan …)* to manage the lists of primes found in both lists

  The list *bothpf1Andpf2* contains prime-exponent pairs whose primes are found in both *pf1* and *pf2*. The exponent is the sum of the exponent for the prime in *pf1* and *pf2*.

- Using the backquote (‘) and comma (,) operators

- Using the *nconc* function to destructively append the three lists *pf1Notpf2, pf2Notpf1*, and *bothpf1Andpf2*

```
trPFMult( pf1 pf2 ) => ( ( 2 3 ) ( 5 2 ) ( 7 3 ) ( 3 7 ))
```

```
procedure( trPFMult( pf1 pf2 )
      let( ( pf1Notpf2 pf2Notpf1 bothpf1Andpf2 pePair2 )
            pf1Notpf2 = setof( pePair1 pf1
                  !assoc( car( pePair1 ) pf2 )
                  )
            pf2Notpf1 = setof( pePair2 pf2
                  !assoc( car( pePair2 ) pf1 )
                  )
            bothpf1Andpf2 = foreach( mapcan pePair1 pf1
                  when( pePair2 = assoc( car( pePair1 ) pf2 )
                        ;; build a list containing
                        ;; a single prime-exponent pair.
                        ;; The mapcan option to the foreach function
                        ;; will destructively append these list together.
                        `( (
                                    ,car( pePair1 )
                                    ,(cadr( pePair1 )+cadr( pePair2 ))
                              ) )
                        ) ; when
                  ) ; foreach
            ;; destructively append the three lists together.
            nconc( pf1Notpf2 pf2Notpf1 bothpf1Andpf2 )
      ) ; let
) ; procedure
```

Using Prime Factorizations to Compute the GCD

The *trPFGCD* function returns the prime factorization of GCD of two prime factorizations. This function illustrates

- Using *foreach(mapcan …)* to merge two association lists. Primes found in both associations lists are given an exponent equal to the minimum of the exponents in both the association lists
- Using the backquote (`) and comma (,) operators

```
procedure( trPFGCD( pf1 pf2 )
      let( ( pePair2 )
            foreach( mapcan pePair1 pf1
                  pePair2 = assoc( car( pePair1 ) pf2 )
                  when( pePair2
                        ;; build a list containing
                        ;; a single prime-exponenet pair
                        ;; the mapcan option to the foreach function
                        ;; will destructively append
                        ;; these lists
                        `( (
                                    ,car( pePair1 )
                                    ,min( cadr( pePair1 ) cadr( pePair2 ) )
                              ) )
                  ) ; when
            ) ; foreach
      ) ; let
) ; procedure
```

The *trGCD* function illustrates finding the greatest common denominator (GCD) of two numbers by

- Finding their prime factorizations
- Manipulating the prime factorizations to find the prime factorization of the GCD
- Evaluating the prime factorization

```
procedure( trGCD( num1 num2 )
      trEvalPF( trPFGCD( trPF( num1 ) trPF( num2 )) )
      ) ; procedure
```

Fibonacci Function

This example illustrates a recursive implementation of the Fibonacci function, implemented directly from the mathematical definition.

```
procedure( fibonacci(n)
if( (n == 1 || n == 2) then 1
else fibonacci(n-1) + fibonacci(n-2)
))
fibonacci(3)         => 2
fibonacci(6)         => 8
```

The same example implemented in SKILL using LISP syntax looks like the following:

```
(defun fibonacci (n)
      (cond
            ((or (equal n 1) (equal n 2)) 1)
            (t (plus (fibonacci (difference n 1))
                        (fibonacci (difference n 2))))
      )
)
```

Factorial Function

This is the recursive implementation of the factorial function

```
procedure( factorial( n )
      if( zerop( n ) then
            1
      else
            n*factorial( n-1)
      ) ; if
) ; procedure
```

This is an iterative implementation

```
procedure( factorial( n )
      let( ( ( f 1 ))
            for( i 1 n
```

```
                f = f*i
                ) ; for
            f ;;; return the value of f
            ) ; let
      ) ; procedure
```

Exponential Function

This function computes e to the power x by summing terms of the power series expansion of the mathematical function. It uses the factorial function.

```
procedure( e( x )
      let( ((sum 1.0))
            for( n 1 10
                  sum = sum + (1.0/factorial(n)) * x**n
                  ) ; for
            sum         ;;; return the value of sum.
            ) ; let
      ) ; procedure
```

To get a sense of the accuracy of this implementation of the e function, observe

```
e( log( 10 ) ) => 9.999702 ;; should be 10.0
```

Counting Values in a List

The *trCountValues* function tallies the number of times each distinct value occurs as a top-level element of a list. It prints a report and returns an association list that pairs each unique value with it's count. Two implementations are presented. The results are equivalent except for ordering.

- The first implementation of *trCountValues* produces

  ```
  trCountValues( '( 1 2 a b 2 b 3 ) ) =>
        ((a 1) (b 2) (3 1) (2 2) (1 1))
  ```

  and prints

  ```
  1 occurred 1 times.
  2 occurred 2 times.
  3 occurred 1 times.
  b occurred 2 times.
  a occurred 1 times.
  ```

- The second implementation of *trCountValues* produces

  ```
  trCountValues( '( 1 2 a b 2 b 3 ) ) =>
        ((3 1) (b 2) (a 1) (2 2) (1 1))
  ```

  and prints

  ```
  3 occurred 1 times.
  b occurred 2 times.
  a occurred 1 times.
  ```

```
2 occurred 2 times.
1 occurred 1 times.
```

The first implementation of the *trCountValues* function illustrates

- Using an association table to maintain the counts

  Each element in the list is used as an index into the table.

- Using the *tableToList* function to build an association list for an association table

```
procedure( trCountValues( aList )
      let( ( ( countTable makeTable( "ValueTable" 0 ) ) )

            foreach( element aList
                  countTable[ element ] = countTable[ element ] + 1
                  ) ; foreach

            foreach( key countTable
                  printf( " %L occurred %d times.\n"
                        key countTable[ key ] )
                  ) ; foreach

            tableToList( countTable ) ;; convert to assoc list
            ) ; let
      ) ; procedure
```

The second implementation of the *trCountValues* function illustrates

- Using an association list to maintain the counts
- Using the *assoc* function to retrieve an entry, if any, for an element
- Using the *rplaca* function to update the count for an entry

```
procedure( trCountValues( aList )
      let( ( countAssocList countEntry count )

            foreach( element aList
                  countEntry = assoc( element countAssocList )
                  if( countEntry
                        then ;; update the count for this element
                              count = cadr( countEntry )
                              rplaca( cdr( countEntry ) ++count )
                        else ;; add an new entry for this element
                              countAssocList = cons(
                                          list( element 1 ) ;; new entry
                                          countAssocList )
                        ) ; if
                  ) ; foreach

            foreach( entry countAssocList
                  printf( " %L occurred %d times.\n"
                        car( entry )         ;; the element
                        cadr( entry )   ;; the count
                        )
                  ) ; foreach

            countAssocList          ;; return value
```

```
        ) ; let
) ; procedure
```

Counting Characters in a String

The *trCountCharacters* function counts the occurrences of characters in a string.

The *trCountCharacters* illustrates

- Using the *parseString* function to construct a list of characters that occur in a string
- Using either implementation of the *trCountValues* function to count the occurrences of the single-character strings

```
procedure( trCountCharacters( aString )
      trCountValues( parseString( aString "" ))
      ) ; procedure
```

Regular Expression Pattern Matching

The following functions take the regular expression pattern matching functions *rexMatchp* and *rexMatchList* provided by SKILL and build two new functions *shMatchp* and *shMatchList*, which provide a simple shell-filename-like pattern matching facility.

The rules:

- Target strings should be single words
- A pattern is to match the entire target words
- Special characters in a pattern:

*	Matches any string, including the null string
?	Matches any single character
[...]	Same as in the original "rex" package
.	Matches "." literally

The function *sh2ed* is used to build a regular expression that is passed to the *rex* functions.

```
(defun sh2ed (s)
      (let ((sh_chars (parseString s ""))
            (ed_chars (tconc nil "^")))
            (while sh_chars
                  (case (car sh_chars)
                        ("*" (tconc ed_chars ".") (tconc ed_chars "*"))
```

```
                        ("?" (tconc ed_chars "."))
                        ("." (tconc ed_chars "\\") (tconc ed_chars "."))
                        (t (tconc ed_chars (car sh_chars))))
                  (setq sh_chars (cdr sh_chars)))
                  (tconc ed_chars "$")
                  (buildString (car ed_chars) "")))

(defun shMatchp (pattern target)
      (rexMatchp (sh2ed pattern) target))

(defun shMatchList (pattern targets)
      (rexMatchList (sh2ed pattern) targets))

shMatchp("*.out" "a.out")        => t
shMatchp("test.??" "test.il")    => t
shMatchp("*.??" "test.out")      => nil

shMatchList("*test.?" '("ALUtest.1" "data.in" "MEMtest.13" "test.5"))=>
("ALUtest.1" "test.5")
```

Geometric Constructions

This is an extensive example of a SKILL++ Object Layer application.

The Application Domain

A geometric construction is a collection of points and lines you build up from an initial collection of points. The initial collection of points are called *free* points. You can add points and lines to the collection through various familiar constraints. You can constrain

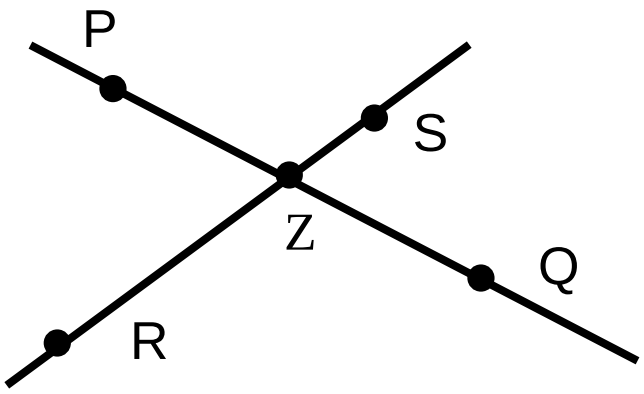
- A point to lie on two interesecting lines
- A line to pass through two points
- A line to pass through a point and to be parallel to another line
- A line to pass through a point and to be perpendicular to another line

When you move any one of the free points, the application propagates the change to all the constrained points and lines

Example

1. You specify the free points P and Q.
2. You construct the line PQ passing through P and Q.
3. You specify the free points R and S.
4. You construct the line RS passing through R and S.

5. You construct the intersection point Z of the line PQ and the line RS.



When you move any of the points P, Q, R, or S the lines PQ and RS and the point Z move accordingly.

The Implementation

The implementation uses the SKILL++ Object System to define several classes and generic functions. The following sections discuss

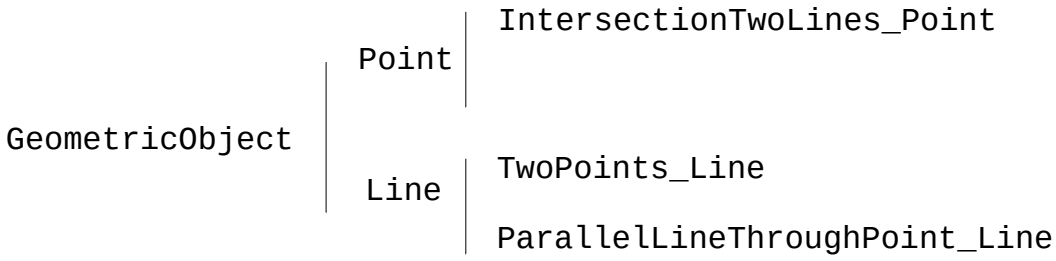
- The class hierarchy
- The generic functions
- The source code
- Several examples

To focus on SKILL++ language issues, the implementation does not address graphics. Instead, you non-graphically

1. Call a SKILL function repeatedly to specify several free points.
2. Call other SKILL functions to construct the dependent points and lines.
3. Enter a SKILL expression to change the coordinates of one of the free points.
4. Call a SKILL function to propagate the change through the constrainted points and lines.
5. Call a SKILL function to describe one of the constrained points or lines.

The Classes

The implementation uses the SKILL++ Object System to define several classes in the following class hierarchy.



The GeometricObject Class

The *GeometricObject* class represents all the objects in the construction. It defines the *constraints* slot. This slot lists all the other objects which need to be notfied when the object updates.

The Point Class

The *Point* class represents a point with slots x and y.

The Line Class

The *Line* class represents a line with the slots A, B, and C. These are the coefficients in the line's equation

```
Ax+By+C = 0
```

The IntersectionTwoLines_Point Class

The *IntersectionTwoLines_Point* class is a subclass of the *Point* class and represents a point that lies on two intersecting lines. It includes two slots that store the lines.

The TwoPoints_Line Class

The *TwoPoints_Line* class is a subclass of the *Line* class and represents a line passing through two points. The class defines two slots to store the points.

The ParallelLineThroughPoint_Line Class

The *ParallelLineThroughPoint_Line* class is a subclass of the *Line* class and represents a line that passes through a point parallel to another line.

To specify a free point, you instantiate the *Point* class. To specify a constrained point or line, you instantiate the associated subclass.

The Generic Functions

The implementation uses several generic functions.

- The *Update* function propagates changes. It updates the coordinates or equations of an object and call itself recursively for each dependent object.
- The *Describe* function prints a description of an object and calls itself recursively for each dependent object.
- The *Validate* function verifies that a point or line satisfies its constraints.
- The *Connect* function adds an object to another object's list of constraints.

The *defgeneric* declarations for these generic functions each a declare default method that raises an error.

Describing the Methods by Class

The following tables describe, for each generic function, all the methods by class. In each table,

- Earlier rows are at the same level or higher in the class hierarchy

- The *etc.* rows refer to other constraint subclasses of *Point* or *Line* that you can add to the implementation to extend its functionality

The Update Methods

Class	Action
GeometricObject	Call *Update* for each of the dependent objects.
Point	No method for this class.
Line	No method for this class.
IntersectionTwoLines_Point	Based on the two lines, recompute the coordinate of the point. Call the next *Update* method.
TwoPoints_Line	Based on the two points, recompute the equation of the line. Call the next *Update* method.
etc.	

The Describe Methods

Class	Method description
GeometricObject	Raise an error since there is no meaningful description at level.
Point	Print a description of the point's coordinates.
Line	Print a description of the line's equation.
IntersectionTwoLines_Point	Call the next *Describe* method to display the coordinates. Then call *Describe* for each of the two lines that define this point.
TwoPoints_Line	Call the next *Describe* method to display the equation. Then call *Describe* for each of the two point that define this line.
etc.	

The Validate Methods

Class	Method description
GeometricObject	Raise an error since there is no meaningful validation to perfom at this level of an object.
Point	No method for this class
Line	No method for this class
IntersectionTwoLines_Point	Verify that the point's coordinates satisfy the equations of the two lines.
TwoPoints_Line	Verify that the two point's coordinates satisfy the line's equation.
etc.	

The Connect Methods

The Connect generic Function

Class	Method description
GeometricObject	Add a dependent object to the list of dependent objects.
Point	No method for this class.
Line	No method for this class.
IntersectionTwoLines_Point	No method for this class.
TwoPoints_Line	No method for this class.
etc.	

The Source Code

```
;;; toplevel( 'ils )

defgeneric( Connect ( obj constraint )
      error( "Connect is a subclass responsibility\n" )
      ) ; defgeneric

defgeneric( Update ( obj )
      error( "Update is a subclass responsibility\n" )
      ) ; defgeneric
```

```
defgeneric( Validate ( obj )
      error( "Validate is a subclass responsibility\n" )
      ) ; defgeneric

defgeneric( Describe ( obj )
      error( "Describe is a subclass responsibility\n" )
      ) ; defgeneric



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; GeometricObject
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

defclass( GeometricObject
      ()
      (
            ( constraints
                  @initform nil
                  )
            )
      ) ; defclass

defmethod( Connect (( obj GeometricObject ) constraint )
      when( !member( constraint obj->constraints )
            obj->constraints = cons( constraint obj->constraints )
            ) ; when
      ) ; defmethod

defmethod( Update (( obj GeometricObject ))
      printf( "Updating constraints %L for %L\n"
            obj->constraints obj )
      Validate( obj )
      foreach( constraint obj->constraints
            Update( constraint )
            ) ; foreach
      t
      ) ; defmethod



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;; Point
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

defclass( Point ( GeometricObject )
      (
            ( name @initarg name )
            ( x @initarg x );;; x-coordinate
            ( y @initarg y );;; y-coordinate
            )
      ) ; defclass

defmethod( Describe (( obj Point ))
      printf( "%s at %n:%n\n"
            className( classOf( obj )) obj->x obj->y )
      ) ;defmethod

defmethod( Validate ((obj Point))
      t
      )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;; IntersectionTwoLines_Point
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
defclass( IntersectionTwoLines_Point ( Point )
      (
            ( L1 @initarg L1 )
            ( L2 @initarg L2 )
            )
      ) ; defclass

defmethod( Describe (( obj IntersectionTwoLines_Point ))
      callNextMethod( obj );;; generic point description
      printf( "...intersection of\n")
      Describe( obj->L1 )
      Describe( obj->L2 )
      ) ;defmethod


procedure( make_IntersectionTwoLines_Point( line1 line2 )
      let( ( point  )
            point = makeInstance( 'IntersectionTwoLines_Point
                  ?L1 line1
                  ?L2 line2
                  )
            Update( point )
            Connect( line1 point )
            Connect( line2 point )
            point
            ) ; let
      ) ; procedure

defmethod( Validate (( obj IntersectionTwoLines_Point ))
      let( ( A1 B1 C1 A2 B2 C2 x y )
            A1 = obj->L1->A
            B1 = obj->L1->B
            C1 = obj->L1->C
            A2 = obj->L2->A
            B2 = obj->L2->B
            C2 = obj->L2->C
            x = obj->x
            y = obj->y
            when( A1*x+B1*y+C1 != 0.0 || A2*x+B2*y+C2 != 0.0
                  error( "Invalid IntersectionTwoLines_Point\n" )
                  ) ; when
            t
            ) ; let
      ) ; defmethod


defmethod( Update (( obj IntersectionTwoLines_Point ))
      ;; check to see if my two lines have values ...
      printf( "Figure out my x & y from lines %L %L\n"
            obj->L1 obj->L2 )
      let( ( A1 B1 C1 A2 B2 C2 det )
            A1 = obj->L1->A
            B1 = obj->L1->B
            C1 = obj->L1->C
```

```
            A2 = obj->L2->A
            B2 = obj->L2->B
            C2 = obj->L2->C
            det = A1*B2-A2*B1
            when( det == 0
                  error( "Can not intersect two parallel lines\n" )
                  )
            obj->x = ((-C1)*B2-(-C2)*B1)*1.0/det
            obj->y = (A1*(-C2)-A2*(-C1))*1.0/det
            ) ; let
      callNextMethod( obj )
      ) ; defmethod


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;; Line
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

defclass( Line ( GeometricObject )
      (             ;;;; Ax+By+C = 0
            ( A )
            ( B )
            ( C )
            )
      ) ; defclass

defmethod( Describe (( obj Line ))
      printf( "%s %nx+%ny+%n=0\n"
            className( classOf( obj ))
            obj->A obj->B obj->C
            )
      ) ; defmethod


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;; TwoPoints_Line
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

defclass( TwoPoints_Line ( Line )
      (
            ( P1 @initarg P1 )
            ( P2 @initarg P2 )
            )
      ) ; defclass

defmethod( Describe (( obj TwoPoints_Line ))
      callNextMethod( obj )
      printf( "...containing\n" )
      Describe( obj->P1 )
      Describe( obj->P2 )
      ) ; defmethod

procedure( make_TwoPoints_Line( p1 p2 )
      let( ( line  )
            line = makeInstance( 'TwoPoints_Line
                  ?P1 p1 ?P2 p2 )
            Update( line )
            Connect( p1 line )
            Connect( p2 line )
            line
```

```
            ) ; let
      ) ; procedure


defmethod( Validate (( obj TwoPoints_Line ))
      let( (x1 y1 x2 y2 A B C)
            x1  = obj->P1->x
            x2  = obj->P2->x
            y1  = obj->P1->y
            y2  = obj->P2->y
            A   = obj->A
            B  = obj->B
            C  = obj->C
            if( A*x1+B*y1+C != 0.0 then
                  error( "Invalid TwoPoints_Line\n" ))
            if( A*x2+B*y2+C != 0.0 then
                  error( "Invalid TwoPoints_Line\n" ))
            t
            ) ; let
      ) ; defmethod

defmethod( Update (( obj TwoPoints_Line ))
      let( (x1 y1 x2 y2 m b)
            x1 = obj->P1->x
            x2 = obj->P2->x
            y1 = obj->P1->y
            y2 = obj->P2->y
            if( x2-x1 != 0
                  then
                        m = (y2-y1)*1.0/(x2-x1)
                        b = y2-m*x2
                        obj->A = -m
                        obj->B = 1
                        obj->C = -b
                  else
                        obj->A = 1.0
                        obj->B = 0.0
                        obj->C = -x1
                  ) ; if
            ) ; let
            callNextMethod( obj )
      ) ; defmethod


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;; ParallelLineThroughPoint_Line
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

defclass( ParallelLineThroughPoint_Line ( Line )
      (
            ( P @initarg P)
            ( L @initarg L)
            )
      ) ; defclass

defmethod( Validate (( obj ParallelLineThroughPoint_Line ))
      let( (x1 y1 A B C LA LB LC)
            x1  = obj->P->x
            y1  = obj->P->y
```

```
            LA  = obj->L->A
            LB  = obj->L->B
            LC  = obj->L->C
            A   = obj->A
            B   = obj->B
            C   = obj->C
            when( A*LB-LA*B != 0.0 || A*x1+B*y1+C != 0
                  error( "Invalid ParallelLineThroughPoint_Line\n" ))
            t
            ) ; let
      ) ; defmethod

defmethod( Describe (( obj ParallelLineThroughPoint_Line ))
      callNextMethod( obj )
      printf( "...Containing\n" )
      Describe( obj->P )
      printf( "...Parallel to\n" )
      Describe( obj->L )
      ) ; defmethod


procedure( make_ParallelLineThroughPoint_Line( point line )
      let( ( parallel_line  )
            parallel_line = makeInstance(
                  'ParallelLineThroughPoint_Line
                  ?P point
                  ?L line
                  )
            Update( parallel_line )
            Connect( point parallel_line )
            Connect( line  parallel_line )
            parallel_line
            ) ; let
      ) ; procedure

defmethod( Update (( obj ParallelLineThroughPoint_Line ))
      let( ( A B C x1 y1 )
            A   = obj->L->A
            B   = obj->L->B
            C   = obj->L->C
            x1  = obj->P->x
            y1  = obj->P->y
            obj->A = A
            obj->B = B
            obj->C = -(A*x1+B*y1)
            ) ; let
      callNextMethod( obj )
      ) ; defmethod
```

Example 1

This example

- Creates a point P and describes it
- Creates another point R and describes it

- Constructs the line L that passes through the points P and R and describe it
- Changes the x coordinate of P to 1 and calls the *Update* function to propagate the changed coordinates of P

  In this version of the application, it is your responsibility to call the *Update* function after changing the coordinates of a free point. You should not use the -> operator to change the coordinates of a non-free point.
- Describes the line L to verify that it has been updated

```
P = makeInstance( 'Point ?x 0 ?y 4 )
stdobj:0x36f018

Describe( P )
Point at 0:4
t

R = makeInstance( 'Point ?x 3 ?y 0 )
stdobj:0x36f024

Describe( R )
Point at 3:0
t

L = make_TwoPoints_Line( P R )
Updating constraints nil for stdobj:0x36f030
stdobj:0x36f030
Describe( L )
TwoPoints_Line 1.333333x+1y+-4.000000=0
...containing
Point at 0:4
Point at 3:0
t

P->x = 1
1

Update( P )
Updating constraints (stdobj:0x36f030) for stdobj:0x36f018
Updating constraints nil for stdobj:0x36f030
t

Describe( L )
TwoPoints_Line 2.000000x+1y+-6.000000=0
...containing
Point at 1:4
Point at 3:0
t
```

Example 2

This example

- Creates four points P, Q, S, and R

- Constructs the line PQ that passes through the points P and Q
- Constructs the line SR that passes through the points S and R
- Constructs the point Z that is on both the lines PQ and SR and describes the point Z
- Changes the y coordinate of the point S to 4 and updates the point S
- Describes the point Z to verify that it has been updated

```
P = makeInstance( 'Point ?x 0 ?y 4 )
stdobj:0x36f090

Q = makeInstance( 'Point ?x 0 ?y -4 )
stdobj:0x36f09c

S = makeInstance( 'Point ?x -3 ?y 0 )
stdobj:0x36f0a8

R = makeInstance( 'Point ?x 3 ?y 0 )
stdobj:0x36f0b4

PQ = make_TwoPoints_Line( P Q )
Updating constraints nil for stdobj:0x36f0c0
stdobj:0x36f0c0

SR = make_TwoPoints_Line( S R )
Updating constraints nil for stdobj:0x36f0cc
stdobj:0x36f0cc

Z = make_IntersectionTwoLines_Point( PQ SR )
Figure out my x & y from lines stdobj:0x36f0c0 stdobj:0x36f0cc
Updating constraints nil for stdobj:0x36f0d8
stdobj:0x36f0d8

Describe( Z )
IntersectionTwoLines_Point at 0.000000:0.000000
TwoPoints_Line 1.000000x+0.000000y+0=0
...containing
Point at 0:4
Point at 0:-4
TwoPoints_Line -0.000000x+1y+-0.000000=0
...containing
Point at -3:0
Point at 3:0
t

S->y = 4
4

Update( S)
Updating constraints (stdobj:0x36f0cc) for stdobj:0x36f0a8
Updating constraints (stdobj:0x36f0d8) for stdobj:0x36f0cc
Figure out my x & y from lines stdobj:0x36f0c0 stdobj:0x36f0cc
Updating constraints nil for stdobj:0x36f0d8
t

Describe( Z )
IntersectionTwoLines_Point at 0.000000:2.000000
TwoPoints_Line 1.000000x+0.000000y+0=0
...containing
Point at 0:4
```

```
Point at 0:-4
TwoPoints_Line 0.666667x+1y+-2.000000=0
...containing
Point at -3:4
Point at 3:0
t
```

Extending the Implementation

There are two extensions to consider.

Protecting the Implementation from Inconsistent Access

You currently can use the -> operator to change coordinates of any point, even a constrainted point. You extend the implementation to allow you to only change coordinates of free points.

Adding Graphics

You extend the implementation to associate a database object with each point or line and update the database object at appropriate times. You should only have to affect the Point and Line classes.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z