

Product Version 06.40 August 2005

Updated September 2005

© 1990-2005 Cadence Design Systems, Inc. All rights reserved. Printed in the United States of America.

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

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

**Restricted Print Permission:** This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

- 1. The publication may be used solely for personal, informational, and noncommercial purposes;
- 2. The publication may not be modified in any way;
- 3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
- 4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

# **Contents**

<u>Preface</u> 19
About the SKILL Language
Quick Look at All the Chapters
<u>What's New</u>
SKILL Development Helpful Hints
SKILL API Documentation
Document Conventions
Syntax Conventions
<u>Data Types</u>
<u>1</u>
Getting Started
SKILL's Relationship to Lisp
Programming Notation
<u>Data Manipulation</u> 28
<u>Characters</u>
Cadence SKILL Language at a Glance
Terms and Definitions
Invoking a SKILL Function
Return Value of a Function
Simplest SKILL Data
Calling a Function
Operators Are SKILL Functions
<u>Using Variables</u>
Alternative Ways to Invoke a Function
Solving Some Common Problems
<u>SKILL Lists</u>
Building Lists
Accessing Lists
Modifying Lists
File Input/Output

Displaying Data	11
Writing Data to a File	
Reading Data from a File	
Flow of Control	
Relational Operators	
Logical Operators	
<u>The if Function</u>	
The when and unless Functions	
The case Function	
The for Function	
The foreach Function	
Developing a SKILL Function	
Grouping SKILL Statements	
Declaring a SKILL Function	
Defining Function Parameters	
Selecting Prefixes for Your Functions	
Maintaining SKILL Source Code	
Loading Your SKILL Source Code	
Redefining a SKILL Function	
2	
Language Characteristics	-7
Naming Conventions	
Names of Functions	
<u>Cadence-Private Functions</u>	
Names of Variables 5	
Function Calls	
SKILL Syntax	
Special Characters6	31
Comments	
White Space6	3
White Space Characters 6	34
Parentheses6	34
Super Right Bracket	35
Backquote, Comma, and Comma-At	35

Line Continuation	66
Length of Input Lists	
Data Characteristics	
 <u>Data Types</u>	
Numbers	
 Strings	
Atoms	
Escape Sequences	
<u> </u>	
Characters	
<u>3</u>	
Creating Functions in SKILL	7
Terms and Definitions	
Kinds of Functions	
Syntax Functions for Defining Functions	
procedure	
<u>lambda</u>	
nprocedure	
<u>defmacro</u>	
mprocedures	
Summary of Syntax Functions	
<u>Defining Parameters</u>	
@rest Option	
@optional Option	81
@key Option	
Combining Arguments	82
Type Checking	
<u>Local Variables</u>	
Defining Local Variables Using the let Function	83
Defining Local Variables Using the prog Function	
Initializing Local Variables to Non-nil Values	
Global Variables	
Testing Global Variables	85
Avoiding Name Clashes	85

Naming Scheme
Reducing the Number of Global Variables86
Redefining Existing Functions86
Physical Limits for Functions87
<u>4</u>
Data Structures89
Access Operators90
<u> Symbols</u> 90
Creating Symbols9
The Print Name of a Symbol9
The Value of a Symbol92
The Function Binding of a Symbol93
The Property List of a Symbol93
Important Symbol Property List Considerations95
<u>Disembodied Property Lists</u> 95
Important Considerations97
Additional Property List Functions97
<u>Strings</u> 99
Concatenating Strings99
Comparing Strings100
Getting Character Information in Strings10
Indexing with Character Pointers10
Creating Substrings102
Converting Case
Pattern Matching of Regular Expressions
Pattern Matching Functions105
<u>Defstructs</u>
Behavior Is Similar to Disembodied Property Lists
Additional Defstruct Functions108
Accessing Named Slots in SKILL Structures109
Extended defstruct Example110
<u> Arrays</u> 11´
Allocating an Array of a Given Size112
Accessing Arrays

Association Tables
 Initializing Tables
Manipulating Table Data
Association Table Functions
Traversing Association Tables
Implementing Sparse Arrays
Association Lists
User-Defined Types
<del></del>
<u>5</u>
Arithmetic and Logical Expressions
·
Creating Arithmetic and Logical Expressions
Role of Parentheses
Quoting to Prevent Evaluation
Arithmetic and Logical Operators
Predefined Arithmetic Functions
Bitwise Logical Operators
Bit Field Operators
Mixed-Mode Arithmetic
Function Overloading
Integer-Only Arithmetic
True (non-nil) and False (nil) Conditions
Controlling the Order of Evaluation132
Testing Arithmetic Conditions133
Differences Between SKILL and C Syntax
SKILL Predicates
The atom Function
The boundp Function
Using Predicates Efficiently135
The eq Function
The equal Function
The neg Function
The nequal Function
The member and memq Functions
The tailp Function

Type Predicates	137
6	
<u>Control Structures</u>	120
Control Functions	
Conditional Functions	
<u>Iteration Functions</u>	
Selection Functions	
Declaring Local Variables with prog	
The prog Function	
The return Function	
Grouping Functions	
Using prog, return, and let	
<u>Using the progn Function</u>	
Using the prog1 and prog2 Functions	147
<b>7</b>	
<u>7</u>	
I/O and File Handling	149
File System Interface	150
<u>Files</u>	150
<u>Directories</u>	150
Directory Paths	150
The SKILL Path	151
Working with the SKILL Path	152
Working with the Installation Path	153
Checking File Status	154
Working with Directories	156
<u>Ports</u>	159
Predefined Ports	160
Opening and Closing Ports	160
<u>Output</u>	162
Unformatted Output	162
Formatted Output	
Pretty Printing	
Input	

Reading and Evaluating SKILL Formats
Reading but Not Evaluating SKILL Formats
Reading Application-Specific Formats
Reading Application-Specific Formats from Strings
System-Related Functions
Executing UNIX Commands
System Environment
8
Conceptual Background
How Lists Are Stored in Virtual Memory
Destructive versus Non-Destructive Operations
Summary of List Operations
Altering List Cells
The rplaca Function
The rplacd Function
Accessing Lists
Selecting an Indexed Element from a List (nthelem)
Applying cdr to a List a Given Number of Times (nthcdr)
Getting the Last List Cell in a List (last)18
Building Lists Efficiently
Adding Elements to the Front of a List (cons, xcons)
Building a List with a Given Element (ncons)
Adding Elements to the End of a List (tconc)
Appending Lists
Reorganizing a List
Reversing a List
Sorting Lists
Searching Lists
The member Function
The memq Function
The exists Function 18
Copying Lists
The copy Function

Copying a List Hierarchically187
Filtering Lists
Removing Elements from a List
Non-Destructive Operations
Destructive Operations
Substituting Elements
Transforming Elements of a Filtered List
Validating Lists
The forall Function
The exists Function
Using Mapping Functions to Traverse Lists
Using lambda with the map* Functions
Using the map* Functions with the foreach Function
The maps Function
The map Function
The map ranction
The maplist Function
The map can Function
Summarizing the List Traversal Operations
List Traversal Case Studies
Handling a List of Strings
Making Every List Element into a Sublist
Using mapcan for List Flattening
Flattening a List with Many Levels
Manipulating an Association List
Using the exists Function to Avoid Explicit List Traversal
Commenting List Traversal Code
<u>9</u>
<u> Advanced Topics</u>
Cadence SKILL Language Architecture and Implementation
<u>Evaluation</u> 205
Evaluating an Expression (eval)
Getting the Value of a Symbol (symeval)
Applying a Function to an Argument List (apply)

<u>Function Objects</u>	207
Retrieving the Function Object for a Symbol (getd)	
Assigning a New Function Binding (putd)	
Declaring a Function Object (lambda)	
Evaluating a Function Object	
Efficiently Storing Programs as Data	
<u>Macros</u>	
Benefits of Macros	210
Macro Expansion	210
Redefining Macros	211
<u>defmacro</u>	211
mprocedure	211
Using the Backquote (`) Operator with defmacro	211
Using an @rest Argument with defmacro	212
Using @key Arguments with defmacro	213
<u>Variables</u>	214
Lexical Scoping	214
Dynamic Scoping	214
Dynamic Globals	214
Error Handling	215
The errset Function	215
Using err and errset Together	216
The error Function	217
The warn Function	217
The getWarn Function	217
<u>Top Levels</u>	218
Memory Management (Garbage Collection)	219
How to Work with Garbage Collection	219
Printing Summary Statistics	220
Allocating Space Manually	221
Exiting SKILL	221
40	
<u>10</u>	
<u>Delivering Products</u>	223
<u>Contexts</u>	224

Deciding When to Use Contexts	25
Creating Contexts	
Creating Utility Functions	
Building the Contexts	
Initializing Contexts	
Loading Contexts	31
Customizing External Contexts	33
Potential Problems	33
Context Building Functions23	35
Autoloading Your Functions	37
Encrypting and Compressing Files	38
Protecting Functions and Variables	38
Explicitly Protecting Functions	38
Protecting Variables23	39
Global Function Protection24	<b>4</b> C
<u>11</u> <u>Writing Style</u> 24	41
Code Layout	42
Comments and Documentation	
Function Calls and Brackets24	44
<u>Commas</u>	46
Using Globals	46
Coding Style Mistakes	48
Inefficient Use of Conditionals24	48
Misusing prog and Conditionals24	49
Red Flags	5C
Any Use of eval or evalstring25	50
Excessive Use of reverse and append25	51
Excessive Use of gensym and concat	
Overuse of the Functions Combining car and cdr	
Use of eval Inside Macros	51
Misuse of prog and return in SKILL++ mode	51

12
 Optimizing SKILL253
Optimizing Techniques
<u>Macros</u>
<u>Caching</u>
Mapping and Qualifying
Write Protection
Minimizing Memory
General Optimizing Tips
Element Comparison
List Accessing
<u>List Building</u>
List Searching
List Sorting
Element Removal and Replacing
Alternatives to Lists
Miscellaneous Comparative Timings
Element Comparison
List Building
Mapping Functions
<u>Data Structures</u>
13
About SKILL++ and SKILL
Background Information about SKILL and Scheme
Relating SKILL++ to IEEE and CFI Standard Scheme
Semantic Differences
Syntax Options
Compliance Disclaimer
References
Extension Language Environment
Contrasting Variable Scoping
SKILL++ Uses Lexical Scoping

SKILL Uses Dynamic Scoping	. 278
Lexical versus Dynamic Scoping	
Example 1: Sometimes the Scoping Rules Agree	. 279
Example 2: When Dynamic and Lexical Scoping Disagree	
Example 3: Calling Sequence Effects on Memory Location	
Contrasting Symbol Usage	
How SKILL Uses Symbols	. 280
How SKILL++ Uses Symbols	. 281
Contrasting the Use of Functions as Data	. 282
Assigning a Function Object to a Variable	. 282
Passing a Function as an Argument	. 282
SKILL++ Closures	. 283
Relationship to Free Variables	. 283
How SKILL++ Closures Behave	. 284
SKILL++ Environments	. 286
The Active Environment	. 286
The Top-Level Environment	. 287
Creating Environments	. 287
Functions and Environments	. 289
Persistent Environments	. 290
14	
Using SKILL++	. 293
Declaring Local Variables in SKILL++	
Using let	
Using letseq	
Using letrec	
Using procedure to Declare Local Functions	
Sequencing and Iteration	
Using begin	
Using do	
Using a Named let	
Software Engineering with SKILL++	
SKILL++ Packages	
The Stack Package	
<u>1110 Otaok Faokago</u>	. 500

Retrofitting a SKILL API as a SKILL++ Package	
SKILL++ Modules	
Stack Module Example	
The Container Module	309
4-	
<u>15</u>	
Using SKILL and SKILL++ Together	313
Selecting an Interactive Language	315
Starting an Interactive Loop (toplevel)	
Exiting the Interactive Loop (resume)	
Partitioning Your Source Code	
Cross-Calling Guidelines	316
Avoid Calling SKILL Functions That Call eval, symeval, or evalstring	316
Avoid Calling nlambda Functions	
Use the set Function with Care	
Redefining Functions	
Sharing Global Variables	318
Using importSkillVar	
How importSkillVar Works	319
Evaluating an Expression with SKILL Semantics	320
Debugging SKILL++ Applications	
Retrieving a Function Object (funobj)	320
Examining the Source Code for a Function Object	
Pretty-Printing Package Functions	
Inspecting Environments	
Retrieving the Active Environment	322
Testing Variables in an Environment (boundp)	323
Using the -> Operator with Environments	
Using the ->?? Operator with Environments	323
Evaluating an Expression in an Environment (eval)	
Retrieving an Environment (envobj )	
Examining Closures	
General SKILL Debugger Commands	

16	
SKILL++ Object System32	27
Basic Concepts	
Classes and Instances	
Generic Functions and Methods	
Subclasses and Superclasses	
Defining a Class (defclass)	
Instantiating a Class (makeInstance)	
Reading and Writing Instance Slots	
Defining a Generic Function (defgeneric)	
Defining a Method (defmethod)	
Class Hierarchy	34
Browsing the Class Hierarchy	35
Getting the Class Object from the Class Name	36
Getting the Class Name from the Class Object	36
Getting the Class of an Instance	36
Getting the Superclasses of an Instance	36
Checking if an Object Is an Instance of a Class	37
Checking if One Class Is a Subclass of Another	37
Advanced Concepts	38
Method Argument Restrictions	38
Applying a Generic Function	36
Incremental Development34	40
Methods versus Slots 34	41
Sharing Private Functions and Data Between Methods	41
<u>17</u>	
Programming Examples34	43
<u> List Manipulation</u>	44
Symbol Manipulation	44
Sorting a List of Points	45
Computing the Center of a Bounding Box	
Computing the Area of a Bounding Box	
Computing a Bounding Box Centered at a Point	

Computing the Union of Several Bounding Boxes
Computing the Intersection of Bounding Boxes
Prime Factorizations
Evaluating a Prime Factorization
Computing the Prime Factorization
Multiplying Two Prime Factorizations
Using Prime Factorizations to Compute the GCD
<u>Fibonacci Function</u>
Factorial Function
Exponential Function
Counting Values in a List
Counting Characters in a String
Regular Expression Pattern Matching
Geometric Constructions
Application Domain
Implementation
<u>Classes</u> 36
Generic Functions
Describing the Methods by Class
<u>Source Code</u>
Extending the Implementation
Index37

August 2005 18 Product Version 06.40

# **Preface**

The SKILL Language User Guide introduces the SKILL language to new users and

- Introduces advanced topics
- Encourages sound SKILL programming methods
- Introduces the Cadence<sup>®</sup> SKILL++ Language, the second-generation extension language for CAD software from Cadence

This manual is intended for the following users:

- People learning to program
- Software users who want to know some of the basics of SKILL
- Programmers beginning to program in SKILL
- CAD developers (internal users and customers) who have experience programming in SKILL
- CAD integrators

You can find companion information in the <u>SKILL Language Functions Reference</u>.

#### See also:

- About the SKILL Language on page 20
- Quick Look at All the Chapters on page 21
- What's New on page 23
- SKILL Development Helpful Hints on page 23
- SKILL API Documentation on page 24
- Document Conventions on page 24

Preface

### 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.

Preface

### **Quick Look at All the Chapters**

<u>Chapter 1, "Getting Started,"</u> 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.

<u>Chapter 2, "Language Characteristics,"</u> 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.

<u>Chapter 3, "Creating Functions in SKILL,"</u> 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.

<u>Chapter 4, "Data Structures,"</u> 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.

<u>Chapter 5, "Arithmetic and Logical Expressions,"</u> 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.

<u>Chapter 6, "Control Structures,"</u> 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.

<u>Chapter 7, "I/O and File Handling,"</u> 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.

<u>Chapter 8, "Advanced List Operations,"</u> 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.

<u>Chapter 9, "Advanced Topics,"</u> 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.

Preface

<u>Chapter 10, "Delivering Products,"</u> 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.

<u>Chapter 11, "Writing Style,"</u> shows ways to make your SKILL programs understandable, reusable, extensible, efficient, and easy to maintain.

<u>Chapter 12, "Optimizing SKILL,"</u> 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.

<u>Chapter 13, "About SKILL++ and SKILL,"</u> introduces the Cadence-supplied Scheme called SKILL++, which is the second generation extension language for the CAD tools from Cadence.

<u>Chapter 14, "Using SKILL++,"</u> focuses in detail on the key areas in which SKILL++ semantics differ from SKILL semantics.

<u>Chapter 15, "Using SKILL and SKILL++ Together,"</u> 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.

<u>Chapter 16, "SKILL++ Object System,"</u> 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.

Preface

### What's New

The following changes have been made for this release:

- <u>"Naming Conventions"</u> on page 58, which 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 <u>"Scaling Factors"</u> on page 69 has been updated.

### **SKILL Development Helpful Hints**

Here are some helpful hints:

- You can click *Help* in the SKILL Development Toolbox to access <u>SKILL Development Help</u> for information about utilities available in the toolbox. The <u>Walkthrough</u> can guide you through the tasks you perform when you develop SKILL programs using the SKILL Development Toolbox. You can also find information about <u>SKILL lint messages</u>, and <u>message groups</u>.
- You can use the <u>SKILL Finder</u> to access syntax and abstracts for SKILL language functions and application procedural interfaces (APIs).
- You can copy examples from windows and paste the code directly into the Command Interprete Window (CIW) or use the code in nongraphics SKILL mode. To select text
  - ☐ Use Control+drag to select a text segment of any size.
  - ☐ Use Control+double-click to select a word.
  - ☐ Use Control+triple-click to select an entire section.

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

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

Note: The <u>Cadence Installation Guide</u> tells you how to install the product.

### **SKILL API Documentation**

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

- <u>Cadence Design Framework II SKILL Functions Reference</u> 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.
- <u>Cadence User Interface SKILL Functions Reference</u> contains APIs for management of windows and forms.

### **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 <sup>®</sup> SKILL language function.
	Indicates that you can repeat the previous argument.

Preface

Precedes the values returned by a Cadence SKILL language function.textIndicates names of manuals, menu commands, form buttons, and form fields.



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)
е	envobj	environment
f	flonum	floating-point number
F	opffile	OPF file ID
g	general	any data type
G	gdmSpecIIUserType	gdm spec
h	hdbobject	hierarchical database configuration object
1	list	linked list
m	nmpIIUserType	nmpll user type
Μ	cdsEvalObject	_

# SKILL Language User Guide Preface

# **Data Types by Type**, continued

Prefix	Internal Name	Data Type
n	number	integer or floating-point number
0	userType	user-defined type (other)
p	port	I/O port
q	gdmspecListIIUserType	gdm spec list
r	defstruct	defstruct
R	rodObj	relative object design (ROD) object
S	symbol	symbol
${\mathcal S}$	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	<del>_</del>
W	wtype	window type
X	integer	integer number
Y	binary	binary function
&	pointer	pointer type

1

# **Getting Started**

Cadence<sup>®</sup> SKILL is a high-level, interactive programming language based on the popular artificial intelligence language, <u>Lisp</u>. Because SKILL supports a more conventional C-like syntax, novice users can learn to use it quickly, and expert programmers can access the full power of the Lisp language. SKILL is as easy to use as a calculator as well as being a powerful programming language whose applications are virtually unlimited.

SKILL brings you 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.

- SKILL's Relationship to Lisp on page 28
- Cadence SKILL Language at a Glance on page 29
- SKILL Lists on page 35
- File Input/Output on page 40
- Flow of Control on page 45
- Developing a SKILL Function on page 51

**Getting Started** 

### SKILL's Relationship to Lisp

**Note:** If you are new either to SKILL or to programming in general, start with <u>"Cadence SKILL Language at a Glance"</u> on page 29.

### **Programming Notation**

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

- Algebraic notation used by most programming languages: func( arg1 arg2 ...)
- Prefix notation used by the Lisp programming language: (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.

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

### **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

**Getting Started** 

evaluation eliminates the long sequences of calls to list and append. See <u>Backquote</u>, <u>Comma</u>, and <u>Comma-At</u> on page 65.

#### **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.

### Cadence SKILL Language at a Glance

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.

**Getting Started** 

#### **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 Command Interpreter Window: 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

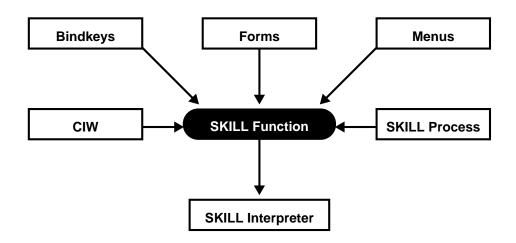
and providing appropriate parameters.

SKILL procedure This term is used interchangeably with SKILL function.

**Getting Started** 

### **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.

**Getting Started** 

#### **Return Value of a Function**

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"

### **Calling 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.

**Getting Started** 

### **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 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.

**Getting Started** 

### **Alternative Ways to Invoke 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.

**Getting Started** 

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	х
floating point number	f
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 67.

### **SKILL Lists**

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.

**Getting Started** 

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.

**Getting Started** 

### 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.

```
'(123)
a=1
b=2
list(ab3)
=> (123)
=> 1
=> 2
=> (123)
```

### 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.

### Merging two lists (append)

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

#### **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 )?	${\tt nil}$ is a list containing nothing. Its length is 0. '( ${\tt nil}$ ) 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.

**Getting Started** 

### **Typical Questions**

Question	Answer
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.

### 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.

### Retrieving an element given an index (nth)

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

**Getting Started** 

### 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.

### Counting the elements in a list (length)

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

### **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.

**Getting Started** 

### **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.

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

```
bBox = '((300400)(500450))
```

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	II = car( bBox)
cadr	car( cdr( ) )	upper right corner	ur = cadr( bBox)
caar	car( car( ) )	x-coord of lower left corner	IIx = caar( bBox)
cadar	car( cdr( car( ) ) )	y-coord of lower left corner	Ily = cadar( bBox)
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

**Getting Started** 

- 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	(123)

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

#### The printf Function

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

```
printf(
     "\n%-15s %-15s %-10d %-10d %-10d"
     layerName purpose
```

**Getting Started** 

```
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 <u>Formatted Output</u> on page 163.

```
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.

### 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
```

**Getting Started** 

```
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.

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

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.

**Getting Started** 

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.
```

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

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

**Getting Started** 

For common output format specifications, see Formatted Output on page 163.

### 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

**Getting Started** 

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
	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
```

**Getting Started** 

### 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.

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.

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

**Getting Started** 

```
++miscCount
)
Message: *Error* if: too many arguments ...
```

#### The when and unless Functions

Use the when function whenever you have only then expressions.

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.

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:

**Getting Started** 

- 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.

### 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.

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

```
1
3
6
10
15
```

**Getting Started** 

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.

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:

**Getting Started** 

### **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 11y 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 11, ur, ury, and 11y 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.

**Getting Started** 

### **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.

### **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.

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

**Getting Started** 

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.

**Getting Started** 

#### **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 <u>Directory Paths</u> on page 150.

### **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.

**Getting Started** 

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.

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
```

August 2005 55 Product Version 06.40

# SKILL Language User Guide Getting Started

2

# **Language Characteristics**

This chapter explains the basic structure and syntax of the Cadence<sup>®</sup> 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.

For more information, see the following topics:

- Naming Conventions on page 58
- Function Calls on page 61
- SKILL Syntax on page 61
- Data Characteristics on page 66

Language Characteristics

## **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 <u>Cadence-Private Functions</u> on page 59.
- 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	С	Enumerated constant
Errors	е	Name of a structure describing an error

Language Characteristics

Name Type	Character	Meaning
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 Cadenceprivate, 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.

Language Characteristics

Variable	Meaning
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.
echoInput	Controls the echoing of expressions or all contents of files being loaded (via load()/loadi()), for example sstatus(echoInput t)

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

Language Characteristics

### **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.

Language Characteristics

**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
1	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
í	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

Language Characteristics

#### 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:

```
\mathbf{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 <u>Solving Some Common Problems on page 34</u>.

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.

Language Characteristics

### White Space Characters

The white space characters in SKILL are generally the same as the C standard white space characters: \t \f \r \n \v. Respectively, they are space, tab, form feed, carriage return, new line, and vertical tab.

The default break characters used by the SKILL language function parseString(), when the optional second argument is not specified, are the white space characters: /t /f /r /n /v.

#### **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.

Language Characteristics

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

Language Characteristics

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

Language Characteristics

- Strings
- Atoms
- Escape Sequences
- Symbols
- Characters

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

- Lists see <u>"Advanced List Operations"</u> on page 175
- Property Lists, Defstructs, and Arrays see "Data Structures" on page 89
- Type Predicates see <u>"Arithmetic and Logical Expressions"</u> on page 119

### **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	а
Cadence database object	dbobject	d
floating-point number	flonum	f
any data type	general	g
linked list	list	1
integer or floating point number		n
user-defined type		0
I/O port	port	р

Language Characteristics

### **Data Types Supported by SKILL**

Data Type	Internal Name	Single Character Mnemonic
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
binary function	binary	у

#### **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 ]

# Language Characteristics

### **Prefixes for Binary/Octal/Hexadecimal Integers**

Radix	Prefix	Examples [value in decimal]
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:

### **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 <sup>24</sup>	10Y [ 10e+25 ]	
Z	Zetta	10 <sup>21</sup>	10Z [ 10e+22 ]	
Е	Exa	10 <sup>18</sup>	10E [ 10e+19 ]	
Р	Peta	10 <sup>15</sup>	10P [ 10e+16 ]	

Language Characteristics

### **Scaling Factors**

Character	Name	Multiplier	Examples
Т	Tera	10 <sup>12</sup>	10T [ 10e+13 ]
G	Giga	10 <sup>9</sup>	10G [ 10,000,000,000 ]
М	Mega	10 <sup>6</sup>	10M [ 10,000,000 ]
k or K	kilo	10 <sup>3</sup>	10k [ 10,000 ]
%	percent	10 <sup>-2</sup>	5% [ 0.05 ]
m	milli	10 <sup>-3</sup>	5m [ 5.0e-3 ]
u	micro	10 <sup>-6</sup>	1.2u [ 1.2e-6 ]
n	nano	10 <sup>-9</sup>	1.2n [ 1.2e-9 ]
p	pico	10 <sup>-12</sup>	1.2p [ 1.2e-12 ]
f	femto	10 <sup>-15</sup>	1.2f [ 1.2e-15 ]
а	atto	10 <sup>-18</sup>	1.2a [ 1.2e-18 ]
z	zepto	10 <sup>-21</sup>	1.2z [ 1.2e-21 ]
у	yocto	10 <sup>-24</sup>	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

Language Characteristics

#### 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

Language Characteristics

### **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 ( <u>The Property List of a Symbol on page 93</u>.)

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(xy) 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".

Language Characteristics

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

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

This function prints the following:

```
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,

As an alternative to ASCII octal values, you can refer to the unprintable characters listed in the <u>Escape Sequences on page 71</u>. 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>	\ <space> (backslash followed by a space)</space>
! . ; : ,	\! \. \; \:
()[]{}	\( \) \[ \] \{

# SKILL Language User Guide Language Characteristics

# **Symbols for ASCII Characters**

Character(s)	SKILL Symbol Name(s)
" # % & + - *	\" \# \%
< = > @ / \ ^ ~	\< \= \>
DEL	\177

# **Creating Functions in SKILL**

<u>"Getting Started"</u> on page 27 introduces you to developing a Cadence<sup>®</sup> SKILL language function. This chapter introduces you to constructs for defining a function and defining local and global variables.

This chapter covers the following topics:

- <u>Terms and Definitions</u> on page 76
- Kinds of Functions on page 77
- Syntax Functions for Defining Functions on page 77
- <u>Defining Parameters</u> on page 80
- Type Checking on page 82
- Local Variables on page 83
- Global Variables on page 84
- Redefining Existing Functions on page 86
- Physical Limits for Functions on page 87

See also <u>"Advanced Topics"</u> on page 203 for more information.

Creating Functions in SKILL

# **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.

Creating Functions in SKILL

# **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:

Creating Functions in SKILL

#### 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 208 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

Creating Functions in SKILL

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

Refer to "Macros" on page 210 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 210 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.

Creating Functions in SKILL

#### **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.

Creating Functions in SKILL

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.

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

<code>@optional</code> relies on order to determine what actual arguments are assigned to each formal argument. The <code>@key</code> 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:

## Creating Functions in SKILL

# **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:

# **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 <u>"Data Characteristics"</u> on page 66. For type checking purposes, you can use several composite type characters (shown in the table below) representing a union of data types.

# Composite Characters for Type Checking

Character	Meaning
S	Symbol or string
n	Number: fixnum, flonum

Creating Functions in SKILL

#### Composite Characters for Type Checking

Character	Meaning
u	Function: Either the name of a function (symbol) or a lambda function body (list)
a	Any data type

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. You can define local variables using the let and prog functions:

- <u>Defining Local Variables Using the let Function</u> on page 83
- <u>Defining Local Variables Using the prog Function</u> on page 84

See also "Initializing Local Variables to Non-nil Values" on page 84.

# **Defining Local Variables Using the let Function**

You can use the let function to establish temporary values for local variables.

- You can 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 )
```

Creating Functions in SKILL

- The local variables are 11, ur, 11y, and ury.
- They are initialized to nil.
- The return value is ury 11y.

# **Defining Local Variables Using the prog Function**

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

```
prog( ( localVariables ) yourSKILLstatements )
```

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 (see "Defining Local Variables Using the let Function" on page 83).

# Initializing Local Variables to Non-nil Values

You can use let to initialize local variables to non-nil values by making a two element list with the local variable and its initial value. You cannot refer to any other local variable in the initialization expression. For example:

# **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 <u>"Naming Conventions"</u> on page 58.

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

Creating Functions in SKILL

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.

Creating Functions in SKILL

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.

Creating Functions in SKILL

- 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.

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

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

# SKILL Language User Guide Creating Functions in SKILL

# **Data Structures**

For information on data structures and related topics, see the following sections:

- Access Operators on page 90
- Symbols on page 90
- <u>Disembodied Property Lists</u> on page 95
- Strings on page 99
- <u>Defstructs</u> on page 107
- Arrays on page 111
- Association Tables on page 113
- Association Lists on page 117
- <u>User-Defined Types</u> on page 117

**Data Structures** 

# **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<sup>®</sup> 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

**Data Structures** 

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"
```

**Data Structures** 

# 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

**Data Structures** 

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')
```

**Data Structures** 

#### **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 to Retrieve 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 )
```

**Data Structures** 

## 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.

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
```

**Data Structures** 

```
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.

**Data Structures** 

## **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.

**Data Structures** 

#### **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)
```

**Data Structures** 

# **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( "1" "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"
```

**Data Structures** 

# **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 nonnil 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.

**Data Structures** 

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

#### **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 <u>"Pattern Matching of Regular Expressions" on page 103</u> 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:

**Data Structures** 

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*.

*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*.

# **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

**Data Structures** 

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.

**Data Structures** 

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
С	Any ordinary character (not a special character listed below) matches itself.
	A dot matches any character.
\	A backslash when followed by a special character matches that character literally. When followed by one of <, >, (, ), and 1,,9, it has a special meaning as described below.
[c]	A nonempty string of characters enclosed in square brackets (called a set) matches one of the characters in the set. If the first character in the set is ^, it matches a character not in the set. A shorthand S-E is used to specify a set of characters S up to E, inclusive. The special characters ] and - have no special meaning if they appear as the first character in a set.
*	A regular expression 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.
l <l></l>	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.

**Data Structures** 

#### **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.

**Data Structures** 

#### 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.

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.

#### 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.

**Data Structures** 

```
(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.

#### 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

# **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
```

**Data Structures** 

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( q 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
```

**Data Structures** 

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 111.

```
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
```

**Data Structures** 

#### 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.

## **Extended defstruct Example**

1. Define a structure.

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

**2.** Define another structure.

```
defstruct(bbox 11 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 - > 11 = p2 => array[4]:xxxx
```

**7.** Set the other field.

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

**8.** Look inside and note the recursive printing.

**Data Structures** 

```
Structure of type point:
                   x:0
             y:0
ur:
      Structure of type point:
             x: 100
      y: 200
b1 -> 11 -> x = 12
=> 12
printstruct( p2 )
Structure of type point :
      x: 12
      y: 0
p1->??
=> (y 200 \times 100)
b1 -> ??
=> (ur array[4]:xxx ll array[4]:xxx)
```

**9.** Add a previously undefined slot.

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.

**Data Structures** 

■ 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.

- 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*.
- $\blacksquare$  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.

**Data Structures** 

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.

**Data Structures** 

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.

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 <i>unbound</i> .

**Data Structures** 

#### **List-Oriented Functions for Tables**

Use this	To do this
foreach function  To execute a collection of SKILL expressions for each I 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
```

**Data Structures** 

#### 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.

You can also use the functions *forall*, *exists* and *setof* to traverse association tables. (These functions are described in detail in <u>"Advanced List Operations"</u> on page 175)

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.

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

- To write the entire contents of a table to a file, use write Table.
- 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

**Data Structures** 

a one-dimensional array. Use integers as the keys. To implement a two-dimensional sparse array, use lists of index pairs as keys.

## **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

**Data Structures** 

-> 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*.

August 2005 118 Product Version 06.40

5

## **Arithmetic and Logical Expressions**

Expressions are Cadence<sup>®</sup> 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. The three types of primitive expressions in SKILL are constants, variables, and function calls. You can combine constants, variables, and function calls with *operators* to form arithmetic and logical expressions (see "Creating Arithmetic and Logical Expressions" on page 120).

A *constant* is an expression that evaluates to itself. That is, evaluating a constant returns the constant itself. For example:

```
123
10.5
"abc"
```

A *variable* stores values used during the computation and returns its value when evaluated. For example:

```
a
x
init var
```

When SKILL creates a variable, it gives the variable an initial value of unbound (that-value-which-represents-no-value). If the interpreter encounters an unbound variable, you get an error message. You must initialize all variables. For example:

```
myVariable = 5
```

**Note:** You will encounter the unbound variable error message if you misspell a variable because the misspelling creates a new variable.

A *function call* applies the named function to the list of arguments and returns the result when called. For example:

```
f(a b c d)
abs(-123)
exit()
```

#### Arithmetic and Logical Expressions

See the following sections for more information:

- Creating Arithmetic and Logical Expressions on page 120
- <u>Differences Between SKILL and C Syntax</u> on page 133
- SKILL Predicates on page 134
- Type Predicates on page 137

## **Creating Arithmetic and Logical Expressions**

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

```
1+2
a*b+c
x>y
```

You can form arbitrarily complicated expressions by combining any number of primitive expressions (constants, variables, and function calls) and operators.

For more information about creating arithmetic and logical expressions, see the following:

- Role of Parentheses on page 121
- Quoting to Prevent Evaluation on page 121
- Arithmetic and Logical Operators on page 121
- Predefined Arithmetic Functions on page 125
- Bitwise Logical Operators on page 126
- Bit Field Operators on page 126
- <u>Mixed-Mode Arithmetic</u> on page 128
- Function Overloading on page 131
- Integer-Only Arithmetic on page 131
- True (non-nil) and False (nil) Conditions on page 132
- Controlling the Order of Evaluation on page 132
- Testing Arithmetic Conditions on page 133

Arithmetic and Logical Expressions

#### **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 <u>Data Types</u> on page 67 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></bit>	<>
setqbitfield1	x <bit>=expr</bit>	
setqbitfield	x <msb:lsb>=expr</msb:lsb>	
quote	'expr	1
getqq	g.s	
getq	g->s	->
putpropqq	g.s = expr, g->s = expr	~>
putpropq	$d \sim s$ , $d \sim s = expr$	
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	1
plus	n1 + n2	+

# **SKILL Language User Guide** Arithmetic and Logical Expressions

## **Arithmetic and Logical Operators**

Name of Function(s)	Synopsis	Operator
difference	n1 - n2	-
leftshift	x1 << x2	<<
rightshift	x1 >> x2	>>
lessp	n1 <n2< td=""><td>&lt;</td></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	٨
bxnor	x1 ~^ x2	~^
bor	x1   x2	
bnor	x1 ~  x2	, ~
and	rel. expr && rel. expr	&&
or	rel. expr    rel. expr	
range	g1 : g2	:
setq	s = expr	=

Arithmetic and Logical Expressions

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</w-1></w-1:0>	
zxtd(x w)	Zero-extends the rightmost w bits of x, executes faster than doing x <w-1:0></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

## Arithmetic and Logical Expressions

#### **Predefined Arithmetic Functions**

Synopsis	Result	
Random Number Generator		
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	1
bitwise exclusive OR	۸
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.

### Arithmetic and Logical Expressions

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.

$$x<1> => 0$$

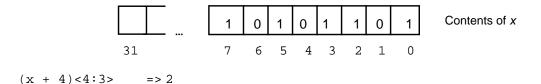
The contents of bit one of x is 0.

$$x < 2:0 > = 5$$

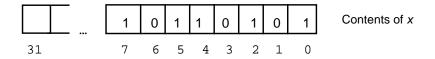
Extracts the contents of the rightmost three bits of x. These are 101 or 5 in decimal.

$$x < 7:4 > = 0b1010 = > 173$$

Stores the bit pattern into the bits 4 through 7.



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

#### Arithmetic and Logical Expressions

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 )
setabitfield1(
      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 uses 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.

See the following topics for more information:

- Floating-Point Issues on page 128
- Integer vs. Floating-Point Division on page 129
- Type Conversion Functions (fix and float) on page 129
- Comparing Floating-Point Numbers on page 130

#### Floating-Point Issues

Because IEEE floating-point numbers use what are essentially binary (base 2) fractions to represent real numbers, some functions that operate on floating-point numbers yield unexpected (and incorrect) results.

#### Arithmetic and Logical Expressions

There are some floating-point numbers that the computer cannot represent exactly as binary fractions. In these cases, the computer uses a binary (base 2) floating-point number to approximate your decimal (base 10) floating-point number. These approximations can lead to incorrect results for some functions that operate on floating-point numbers. Consider the following:

<b>Decimal fraction</b>	Equivalent	Binary fraction	Result
0.125	1/10 + 2/100 + 5/1000	0.001	Exact representation
0.3333	3/10 + 3/100 + 3/1000	0.010101010101	Approximation
.1	1/10	0.0001100110011	Approximation

Binary representation can result in a loss of precision. Also, when a program does not save or retrieve the least significant bits of a number, that number loses precision. See also "Comparing Floating-Point Numbers" on page 130 and "Type Conversion Functions (fix and float)" on page 129.

**Note:** Some applications use single precision to represent floating-point numbers. SKILL uses double precision.

#### Integer vs. Floating-Point Division

The division operator requires special attention because

- Integer division truncates its results
- Floating-point division computes an exact number

#### Type Conversion Functions (fix and float)

Before you can compare an integer to a floating-point number, you must convert both numbers to the same type (that is, integer or float).

**Note:** See also <u>"Comparing Floating-Point Numbers"</u> on page 130 for more information.

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. See "SKILL Language Functions" in the SKILL Language Reference for more information about these functions.

Arithmetic and Logical Expressions

## / Important

Because of the <u>"Floating-Point Issues"</u> on page 128, you can pass a floating-point argument to the fix function that yields an incorrect result. For example:

```
Skill > fix(4.1 * 100)
409
Skill > fix((30 - 18.1) * 10)
118
```



You can load the following SKILL file (real\_fix.il) into your SKILL development environment to force correct results in cases like those shown above.

```
procedure(real_fix(number)
let( (num_string fix_num)
sprintf( num_string, "%f" number)
fix_num = atoi(car(parseString(num_string)))
fix_num
)
);
```

#### **Comparing Floating-Point Numbers**

**Note:** Two numbers can only be equal if they are of the same type (either integer or float) and have identical values. Two floating-point numbers can appear the same when printed while differing internally in their least significant bits.

Simple comparison rarely works for floating-point numbers. For example:

```
if( (a == b) println("same"))
```

You can compare against an acceptable tolerance range with more success as follows (assuming a is not zero):

```
if( (abs(a - b)/a < 1e-6) println("same"))
```



Unless two floating-point numbers are assigned identical constants or are generated by exactly the same sequence of computations, it is unlikely SKILL will treat them as equal when compared.

You can think of a loop (such as for) as a special case of comparing floating-point numbers. Each time a program increments a floating-point loop variable, the number can lose precision resulting in a cummulative error. Under these circumstances, your loop might not execute the expected number of times.

Arithmetic and Logical Expressions

### **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 complexnumber 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 integer mode and returns t if integer mode 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

August 2005 131 Product Version 06.40

Arithmetic and Logical Expressions

can only be set by *status*, rather than as SKILL variables you can set directly. Refer to <u>"Names of Variables"</u> on page 59 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.

## Arithmetic and Logical Expressions

#### 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:

- SKILL supports the following exponentiation operator: \*\* (two asterisks).
- The function mod replaces the modulo operator "%" so that you do not have to use "%" for this infrequently-used function: Use mod(i j) instead of i % j.

#### Arithmetic and Logical Expressions

- The more general if/then/else control construct in SKILL makes the conditional expression operators? and: obsolete.
- The indirection operator \* and the address operator & do not have any meaning in SKILL and are not supported.
- The set of bitwise operators is augmented by  $\sim \&$  (nand),  $\sim |$  (nor), and  $\sim ^{\land}$  (xnor).
- Logical expressions that evaulate to false return the special atom nil and those that evaluate to true return any non-nil value (usually, the special atom t).

## **SKILL Predicates**

The following predicate functions test for a condition:

- The atom Function on page 134
- The boundp Function on page 134

For a list of predicate functions for testing the type of a data object, see <u>"Type Predicates"</u> on page 137.

#### 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 (has an assigned value).

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

## Arithmetic and Logical Expressions

## 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, memg, and caseg 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

eg checks addresses when testing for equality. The eg 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 = '(123)
                            => ( 1 2 3 )
list2 = '(123)
                            => ( 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
eq( cdr( list3 ) list1 )
aList = '(abc)
                            => a b c
eq( 'a car( aList ) )
```

## 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.

Arithmetic and Logical Expressions

## 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.

### The nequal Function

nequal checks if two arguments are *not* logically equivalent and returns t if they are not.

## 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.

## Arithmetic and Logical Expressions

### 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 on the name of a function usually indicates that it is a predicate function. Type predicates appear in the table below.

**Note:** g (general) can be any data type.

### **Type Predicates**

Fur	nction	Value Returned
arı	cayp(g)	t if g is an array, nil otherwise
bcc	dp(g)	t if g is a binary function, nil otherwise
dtr	pr(g)	t if g is a non-empty list, nil otherwise (note that dtpr (nil) returns nil)
fix	kp(g)	t if g is a fixnum, nil otherwise
flo	patp(g)	t if g is a flonum, nil otherwise
lis	stp(g)	t if g is a list, nil otherwise (note that listp(nil) returns t)
nul	ll(g)	t if g is nil, nil otherwise
nun	mberp(g)	t if g is a number (that is, fixnum or flonum), nil otherwise
otł	nerp(g)	t if g is a foreign data pointer, nil otherwise
por	ctp(g)	t if g is an I/O port, nil otherwise
stı	cingp(g)	t if g is a string, nil otherwise
syn	mbolp(g)	t if g is a symbol, nil otherwise

**SKILL Language User Guide**Arithmetic and Logical Expressions

## **Type Predicates**

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**

You can read about control structures in the following sections:

- Control Functions on page 140
- Selection Functions on page 142
- <u>Declaring Local Variables with prog</u> on page 143
- Grouping Functions on page 145

Control Structures

## **Control Functions**

The Cadence<sup>®</sup> 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 27.

Control Structures

#### 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 expNlexpN2 ....
```

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 144.

#### **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.

Control Structures

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:

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:

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  $\leq$  i  $\leq$  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.

Control Structures

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

But this function will not work because the t case is the default and always matches. The correct way to write this test is

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

August 2005 143 Product Version 06.40

**Control Structures** 

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.

Control Structures

## **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:

Control Structures

```
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:

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:

When setting initial values within the declaration, reference cannot be made to other local variables. For example, the following is wrong:

Control Structures

```
)
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.

## The prog2 Function

prog2 evaluates expressions from left to right and returns the value of the second expression.

```
prog2( x = 4
```

**Control Structures** 

```
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:

August 2005 148 Product Version 06.40

7

## I/O and File Handling

You can read about I/O and file handling topics in the following sections:

- File System Interface on page 150
- Ports on page 159
- Output on page 162
- <u>Input</u> on page 166
- System-Related Functions on page 172

I/O and File Handling

## File System Interface

All input and output in the Cadence<sup>®</sup> 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.

I/O and File Handling

If the path begins with a tilde followed by a slash  $(\sim /)$ , 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  $\sim$  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 <code>setSkillPath</code> and <code>getSkillPath</code> 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.

August 2005 151 Product Version 06.40

I/O and File Handling

#### 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.

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")
```

I/O and File Handling

## 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.

## 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"
```

I/O and File Handling

## **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
```

I/O and File Handling

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.

I/O and File Handling

```
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

- 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 \text{ now}
fileSeek(p 6 1)
                      => t
fscanf(p "%s" s)
                      => 1
                             s = "xyz" now
fileSeek(p -12 2)
                      => t
                     fscanf(p "%d" x)
```

## **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.

August 2005 156 Product Version 06.40

I/O and File Handling

```
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.)

August 2005 157 Product Version 06.40

I/O and File Handling

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")

```
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.

=> "/tmp/12345678a08717"

## 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"
```

I/O and File Handling

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.

August 2005 159 Product Version 06.40

I/O and File Handling

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
woport	Output port for printing warning messages; analogous to and initialized to stdout 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.

I/O and File Handling

#### 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("out.il" "w") => port:"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.

I/O and File Handling

```
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.

I/O and File Handling

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

I/O and File Handling

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
%0	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.

I/O and File Handling

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.

August 2005 165 Product Version 06.40

I/O and File Handling

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.

## 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.

I/O and File Handling

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 <u>"Top Levels"</u> on page 218 for a discussion of the top level.

Signals that car is an unbound variable.

I/O and File Handling

#### 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 => 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 223

I/O and File Handling

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.

## 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.

In the last example, only the first form is read in.

I/O and File Handling

## **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:

## 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

I/O and File Handling

character in the string. gets returns nil if EOF is encountered and the variable maintains its last value. Assume the testl.data file has the following first two lines:

## 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:

## 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.

I/O and File Handling



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 <code>junk</code> file from the <code>/tmp</code> directory and returns <code>t</code> 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.

I/O and File Handling

## 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)

 ${\tt getShellEnvVar}\ \textbf{returns the value of a UNIX environment variable, if it has been set.}$ 

```
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.

# **SKILL Language User Guide** I/O and File Handling

August 2005 174 Product Version 06.40

8

## **Advanced List Operations**

<u>"Getting Started"</u> on page 27 introduces you to building Cadence<sup>®</sup> SKILL language lists. Information in this chapter helps you perform more advanced list operations.

It is helpful to understand how lists are stored in virtual memory so that you can understand how SKILL functions such as car and cdr manipulate lists and the issues behind building long lists efficiently (see "Conceptual Background" on page 176).

You can read more about the following topics:

- Summary of List Operations on page 178
- Altering List Cells on page 179
- Accessing Lists on page 180
- Building Lists Efficiently on page 181
- Reorganizing a List on page 184
- Searching Lists on page 185
- Copying Lists on page 186
- <u>Filtering Lists</u> on page 187
- Removing Elements from a List on page 188
- Substituting Elements on page 189
- <u>Transforming Elements of a Filtered List</u> on page 190
- <u>Validating Lists</u> on page 190
- <u>Using Mapping Functions to Traverse Lists</u> on page 191
- List Traversal Case Studies on page 197

## **Advanced List Operations**

## **Conceptual Background**

## **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 list and symbol data types. Instead of displaying memory addresses, SKILL displays

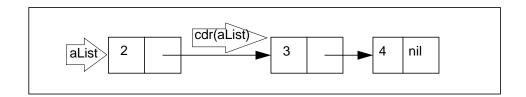
- The elements of a list surrounded by parentheses
- The name of a symbol

#### 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
```

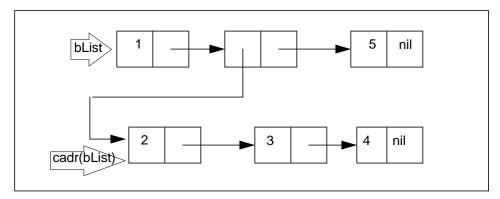
The cdr function returns the contents of the second location of a list cell.

$$cdr(aList) => (3 4)$$

Advanced List Operations

## **Lists Containing Sublists**

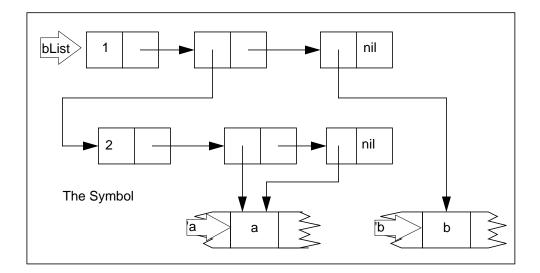
The expression bList = (1 (2 3 4) 5) allocates the following list cells.



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		Х
Accessing a List	nthelem, nthcdr, last	X	

**Advanced List Operations** 

#### **List Operations**

_	_	_	
Operation	Function	Non-destructive	Destructive
Building a List	cons, ncons, xcons, append1	Х	
	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

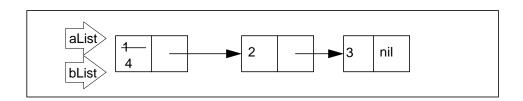
## **Altering List Cells**

The most fundamental destructive operations concern altering a list cell. You can change either the car cell or the cdr cell. rplaca and rplacd are destructive operations.

## The rplaca Function

Use the rplaca function to replace the first element of a list.

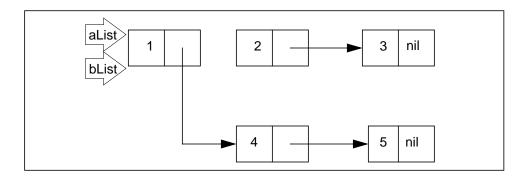
```
aList = '(123) => (123)
bList = rplaca(aList 4) => (423)
aList => (423)
eq(aList bList) => t
```



## The rplacd Function

Use the rplacd function to replace the tail of a list.

```
aList = '(123) => (123)
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 27.

## 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'(abc)) => a
z = '(123)
nthelem(2z)
                    => 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 = '(123)
nthcdr(2z)
                       => (3)
```

# Advanced List Operations

## **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)

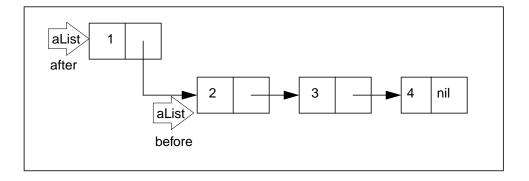
If order is unimportant, the easiest way to build a list is to repeatedly call cons to add new elements at the head of the list.

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.

# **Advanced List Operations**

```
xcons('(bc)'a) => (abc)
```

## Building a List with a Given Element (ncons)

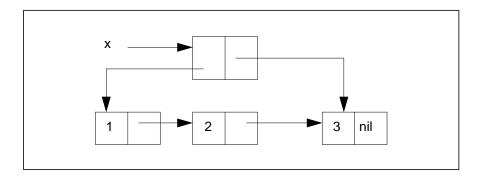
ncons builds a list by adding the element you supply to the beginning of an empty list. It is equivalent to cons ( g\_element nil ).

```
ncons( 'a )
                => ( a )
                                 ;;; equivalent to cons( 'a nil )
z = '(123)
               => ( ( 1 2 3 ) ) ;;; equivalent to cons( z nil )
ncons(z)
```

## 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:

 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

#### **Advanced List Operations**

the value returned by tconc to a variable because tconc modifies the tconc structure. For example:

```
tconc(x 2), tconc(x 3) ...
```

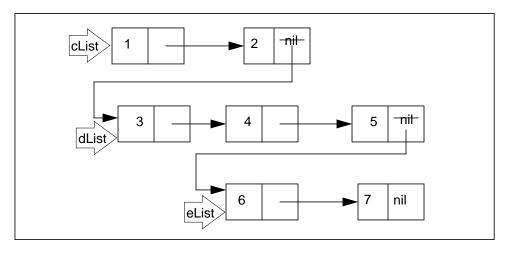
3. After the list has been built, take the car of the tconc structure to get the actual value of the list being built. For example:

```
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 => ( 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 )
```

**Advanced List Operations** 

#### The Iconc 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.

# 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

**Advanced List Operations** 

```
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 27. 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

**Advanced List Operations** 

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 <u>"Substituting Elements"</u> on page 189.

## 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
```

**Advanced List Operations** 

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.

# **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.

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:

The more efficient way is as follows:

**Advanced List Operations** 

```
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.

**Advanced List Operations** 

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.

```
remg( 'x '( a b x d f x q ) ) => ( a b d f q )
```

The remg 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.

```
remdg('x'(abxdfxg)) \Rightarrow (abdfg)
```

# **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.

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'(123456)oddp(x)) => (135)
```

#### 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.

# **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 134 and "Type Predicates" on page 137.) Predicates return to 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.

**Advanced List Operations** 

- $\blacksquare$  for all is represented mathematically as  $\forall$ .
- $\blacksquare$  exists is represented mathematically as  $\exists$ .

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 113 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.

**Advanced List Operations** 

#### 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 <u>"Syntax Functions for Defining Functions"</u> on page 77 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:

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

#### **Advanced List Operations**

- 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:

#### 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:

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
```

#### **Advanced List Operations**

```
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

**Advanced List Operations** 

- 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 )
                        else
                         ) ; 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

**Advanced List Operations** 

- 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 ).

## **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

**Advanced List Operations** 

#### **Summary of the Mapping Functions**

Mapping Function Return Value	Function is Passed Successive Elements	Function is Passed Successive List Cells
Collects the result of each iteration. Uses mapcan nconc to append the result of each iteration.		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:

Using the mapcar function allows this code to be written as

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:

#### **Advanced List Operations**

## **Making Every List Element into a Sublist**

You can perform this operation with either version of the trMakeSublists function.

## **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 = ((123)(4)(5678)()(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 <code>copied</code> within the <code>foreach</code> function to produce a new sublist that can be harmlessly modified:

## Advanced List Operations

## 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:

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 oneelement 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:

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)
```

**Advanced List Operations** 

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:

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)
```

**Advanced List Operations** 

Another approach is to jump out of the while loop:

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:

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:

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

**Advanced List Operations** 

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 '(1234) evenp( x ) ) => (234)
```

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.

# **Advanced Topics**

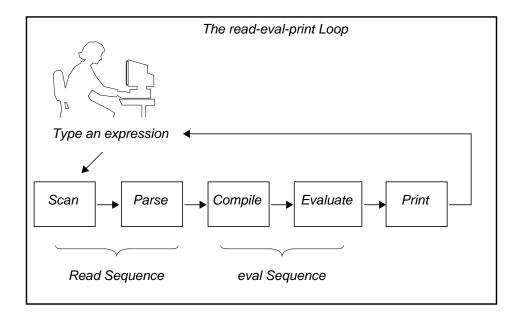
You can find information about advanced topics in the following sections:

- Cadence SKILL Language Architecture and Implementation on page 204
- Evaluation on page 205
- Function Objects on page 207
- Macros on page 210
- Variables on page 214
- Error Handling on page 215
- Top Levels on page 218
- Memory Management (Garbage Collection) on page 219
- Exiting SKILL on page 221

# Cadence SKILL Language Architecture and Implementation

When you first encounter the Cadence<sup>®</sup> 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

**Advanced Topics** 

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 <u>Delivering Products</u> on page 223) 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.

**Advanced Topics** 

```
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 a List 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 208.) 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).

**Advanced Topics** 

```
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:

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.

**Advanced Topics** 

#### The getd Function

Function	Return Value	Explanation
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( 'mySgrt getd( 'sgrt ))=> lambda:0x108b8
```

Assigns the function mySqrt the same definition as sqrt.

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 rightangled triangle.

**Advanced Topics** 

## **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.

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
```

**Advanced Topics** 

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.

**Advanced Topics** 

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.

**Advanced Topics** 

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 inline 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:

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

**Advanced Topics** 

is defined recursively. For each variable declaration, it nests let statements, thereby guaranteeing that all bindings are performed sequentially. For example:

## 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

```
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.

**Advanced Topics** 

## **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 <code>lexical scope</code> 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:

**Advanced Topics** 

```
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.

**Advanced Topics** 

```
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 construct after determining that errset returned nil.

## **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.

**Advanced Topics** 

#### The error Function

error prints any error messages and then calls err flagging the error. The first argument can be a format string that causes the rest of the arguments to print using that format. Here are some examples:

This function:	Prints the following:	
error( "myFunc" "Bad List" )	*Error* myFunc: Bad List	
error( "bad args - %s %d %L" "name"	100 '(1 2 3) )	
	*Error* bad args - name 100 (1 2 3)	
errset( error( "test" ) t)=> nil	*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.

The testWarn function prints the warning if t is passed in and gets the warning if nil is given as an argument.

**Advanced Topics** 

```
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.

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.

**Advanced Topics** 

# **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

**Advanced Topics** 

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
Туре	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.

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

**Advanced Topics** 

User Type (ID)	A	llocated	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
strings strings(perm) IL stack (Internal) TOTAL GC COUNT 9 Summary of Sy	= 68708 = 49140 = 12288 mbol Tabl		cs	
Total Number of Sym				
Hash Buckets Occupi	ed = 4116	out of 44	99	

# **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.

**Advanced Topics** 

To give you even more control, an exit-before function can return the atom <code>ignoreExit</code> to abort the exit call totally. When <code>exit</code> 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 <code>ignoreExit</code>, the exit request is aborted and it returns <code>nil</code> 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 <code>exitbefore</code> and <code>exitafter</code> 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 on page 224
- Autoloading Your Functions on page 237
- Encrypting and Compressing Files on page 238
- Protecting Functions and Variables on page 238

**Delivering Products** 

### **Contexts**

**Note:** This information is for users who are writing a large volume of using the Cadence<sup>®</sup> SKILL language code and are interested in modularizing the code and possibly autoloading it at run time.

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.

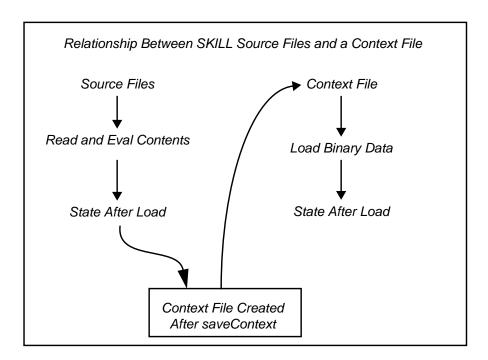
SKILL contexts contain source code and data. Their main purpose is to allow for fast loading and initializing of product code. One way of looking at contexts is to view them as snapshots of the internal state of the interpreter, much like a core file in UNIX is an image of the running process.

Contexts cannot store at save time and retrieve at load time process-dependent structures. 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: Lists, numbers, strings, arrays, and so forth can be saved into and retrieved from a context.

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,

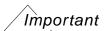
**Delivering Products** 

contain binary data that is loaded directly into memory. The binary data stored into a context has been parsed and evaluated before being saved.



# **Deciding When to Use Contexts**

You must decide when it is better to use contexts versus straight or encrypted SKILL code. Some considerations include the following:



Context size 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. 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 autoloaded during the physical manipulation of graphics, the impact of the load and initialization should be minimized. Contexts can help in this case.

**Delivering Products** 

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.

Do you need to modularize your 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.

Can you 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.

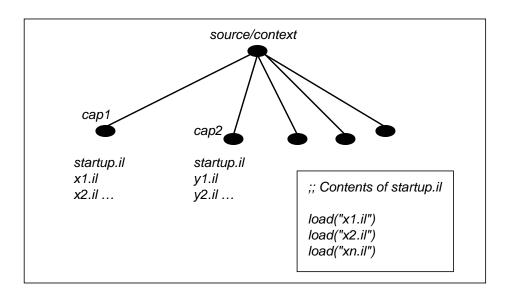
#### Creating the Directory Structure

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.

**Delivering Products** 

If the capability names are <code>cap1</code> and <code>cap2</code>, <code>create</code> two directories under <code>source/context</code> named <code>source/context/cap1</code> and <code>source/context/cap2</code>. 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.





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.

**Delivering Products** 

### **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")))
      (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)
```

#### **Delivering Products**

```
;; 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")
;; end of file myContextBuild.il
```

**Delivering Products** 

# **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 <code>-ilLoadIL</code> 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 <code>exit</code> command is the last thing the file <code>myContextBuild.il</code> 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 <code>cap1</code> 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
```

**Delivering Products** 

```
;; 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 capl.
(defInitProc "capl" 'caplInitProc)
;; end sample code
```

The call to *deflnitProc* 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.

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.

**Delivering Products** 

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.

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.

**Delivering Products** 

### **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
```

**Delivering Products** 

```
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:

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 varl and varl 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 varl to remain complete. Such copying is only done on atomic structures, such as integers or strings.

**Delivering Products** 

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 <code>isMorning</code> function is executed and the <code>greeting</code> is set correctly. The value of <code>greeting</code> and the value of <code>form->title->value</code> are "frozen" in the context file. On subsequent loads of the context, the <code>isMorning</code> 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.

#### **Delivering Products**

#### 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 <u>setContext</u> and a <u>saveContext</u> 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 <code>saveContext</code>. 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.

**Delivering Products** 

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.

You can write protect a function by setting the writeProtect status flag. Once turned on, all subsequent function definitions are write protected.

# **Explicitly Protecting Functions**

Protecting functions on a per-function basis is an alternative to sstatus which protects functions on a per-context basis.

**Delivering Products** 

#### **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 is a disembodied property list, attempting to modify the value of properties will fail:

```
y = 5
setVarWriteProtect( 'y )=> t
setVarWriteProtect( 'y )=> nil
; Initialize the variable y.
; Set y to be write protected.
; Already write protected.
```

**Delivering Products** 

#### 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.

The call to sstatus takes only the values t/nil as its second argument.

11

# **Writing Style**

Good style in any language can lead to programs that are understandable, reusable, extensible, efficient, and easy to maintain. These attributes are particularly important to have in programs developed using an extension language like the Cadence<sup>®</sup> SKILL language because the programs tend to change a lot and the number of contributors can be 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. It is better to have one call that takes a list of objects to be inserted:

GOOD	BAD
EDinsert(database '(obj1 obj2 obj3))	EDstartInsert(database) 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 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.

Writing Style

The rest of this chapter describes specific situations where coding style issues are important.

- Code Layout on page 242
- Using Globals on page 246
- Coding Style Mistakes on page 248
- Red Flags on page 250

# **Code Layout**

The readability of the code is the most important aspect of the code layout. The following guidelines can help you to write more readable code.

#### **Comments and Documentation**

You can use either of the following methods for commenting in SKILL:

- ; at the beginning of a line
- /\*...\*/ surrounding your comments

**Note:** Because /\*...\*/ comments cannot be nested, some programmers find them easier to see.

While it is typically easier to understand and maintain code that is well-commented, you must also keep your comments consistent with your code. Comments that are misleading because they have not been maintained along with the code can be worse than not having any comments at all.

You should document your data structures by describing the motivation behind them and the effect of each 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, you should maintain a "to-do" list. For example, you can put a "to-do" comment around a dubious looking piece of code to explain your misgivings so that another developer can track a bug in that code.

Generally, if you find that you need to write convoluted comments about a convoluted algorithm, you should rewrite the algorithm. Strong code can be self-documenting.

Writing Style

### **Things to Comment and Document**

Cadence suggests that you comment the following items:

Item to comment	What to comment
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. Add a help string inside the procedure (see <u>"procedure"</u> on page 77). Help systems extract this information for display. As much as possible, add type templates for procedure arguments to help catch erroneous use; type information can be extracted by help systems.
Data structures	Describe contents of data structures and impact on algorithms.
Complex conditionals	Comment any test within a conditional function that is nontrivial 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**

While including any number of comments in your code does not affect performance once it has been read into the SKILL interpreter, Cadence suggests that you do not comment the following in production code:

Item not to comment	Explanation
Long change details	You should use your source code control system (such as RCS or SCCS) to maintain full details of any changes you make and include only a brief outline of your changes in the module header (rather than in the body of the code). Including details in the body of the code can make maintenance more difficult.
PCR details	You should maintain product change request information in the PCR itself.

Writing Style

#### **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.

**Note:** Having extra newlines to allow alignment of function arguments or parentheses does not 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 parentheses can become difficult to locate.
- Inserting another function call around the code containing the super right bracket can be difficult.
- Bracket-matching procedures in editors neither manage nor account for super right bracket.

One method you can use to make sure that all your parentheses are matching when writing SKILL code is to 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 and not affect the functionality of the program.
- In SKILL, the insertion of extra brackets can be, and usually is, incorrect.

Writing Style

The problem usually occurs with infix operators. 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)
```

You must understand the relative precedence of the built-in SKILL functions. For example, consider the following code:

```
a = b \&\& c
```

The line of code above is held as the following list:

```
(setq a (and b c))
```

#### rather than

```
(and (setq a b) c)
```

While this precedence is exactly what you would expect from any language, it might not necessarily be what you want. Consider the following:

```
if(res = func1(arg) && res != no_val then ...)
```

#### What the programmer meant to do:

#### What actually happens:

- 1. Call func1 and store the result in res.
- **2.** Check that res is not equal to no\_val.
- The interpreter evaluates the expression func1(arg) && (res != no\_val) and assigns the result to res.

The programmer needs to write the code as follows to perform the desired function:

```
if((res = func1(arg)) && res != no val then ...)
```



Using too many parentheses can cause the code to fail. For example, the following statement has too many levels of parentheses:

```
a = ((func1(arg1)) && (func2(arg2)))
```

**Note:** Parentheses in function calls are only optional for the built-in unary and infix operators, such as ! and +. The following functions are equivalent:

```
!a
!(a)
```

(!a)

Writing Style

#### **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.

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,
```

Writing Style

```
?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.

Writing Style

# **Coding Style Mistakes**

C programmers sometimes make the following mistakes when programming in SKILL. Even though these mistakes have mostly to do with coding style, some of them can have an impact on performance.

- Inefficient Use of Conditionals
- Misusing prog and Conditionals on page 249

#### 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.

This check can be optimized using De Morgan's Law to be:

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:

Another example is where multiple if-then-else checks are used, such as:

This can be more clearly and efficiently implemented using a cond:

```
cond(
     (stringp(layer) layerName = layer)
```

Writing Style

```
(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
                          = ReportForm->TemplateDir->value
            templateDir
            /* 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:

Writing Style

```
( (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 that require special attention. Their use is often 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 on page 250
- Excessive Use of reverse and append on page 251
- Excessive Use of gensym and concat on page 251
- Overuse of the Functions Combining car and cdr on page 251
- Use of eval Inside Macros on page 251
- Misuse of prog and return in SKILL++ mode on page 251

# 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.

Writing Style

#### **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 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

One function that combines car and cdr is cdaddr. Using such functions, which are not as intuitive as calls to nthelem and nthcdr, can lead to programmer and reader errors.

#### **Use of eval Inside Macros**

Calling eval inside a macro means you are determining the value of an entity at compile time as opposed to evaluating it at run time, which may result in undesirable behavior. In general, macros should massage expressions and return expressions (see "Macros" on page 210).

# Misuse of prog and return in SKILL++ mode

Misuse of prog and return in SKILL++ might corrupt SKILL++ environment frames and can lead to a fatal programming error. The following example demonstrates misuse that you should avoid (using prog and return in a SKILL++ .ils file). Immediately following this example is an example of what you should do instead.

```
procedure( myFunc(namelist keys "ll")
  let( (selectedlist)
    foreach( name namelist
    prog(()
        foreach( key keys
        when( rexMatchp(key name)
            return(t)
        )
        return(nil)
    }
}
```

Writing Style

```
selectedlist = cons(name selectedlist)
)
)

Instead, do this:
procedure( myFunc(namelist keys "ll")
  let( ()
    setof( name namelist
        setof(key keys rexMatchp(key name))
    )
)
)
```

**12** 

# **Optimizing SKILL**

Before you embark on optimizing your SKILL code, your SKILL program should work. You might waste time if you try to optimize the performance of a program that has not yet met its functional requirements.

**Note:** In this chapter, you can find information about techniques for modifying your programs to run faster. You will not learn how to create better algorithms: You need to design algorithmic performance into your programs from the outset.

When optimizing your SKILL code, you should do the following:

### Focus your efforts

Before optimizing, you should determine what you need to optimize. If 80% of the time is spent in 20% of the code, you should focus your efforts on optimizing that small portion of the program where there is a performance bottle-neck.

Always measure the benefit gained after you modify your code. You should leave well-written and well-structured code untouched if the changes do not yield significant performance improvements.

■ Use <u>profiling tools</u> to measure code performance

# /Important

You should expect reduced execution speed when profiling and gathering performance data because these operations are necessarily intrusive in nature. The measurements you gather should take this into account. When profiling, you should think of code performance in terms of percentages rather than absolute values. See "Printing Summary Statistics" on page 220 for information on the how to obtain and interpret a summary of memory allocation.

You can optimize your SKILL code for time spent and memory used.

Time: You can use the <u>SKILL Profiler</u> to find out where most of the time is spent in your program. You can gather global statistical information about time spent in functions called in a given session. The profiler takes a sample of the runtime stack

Optimizing SKILL

at pre-specified intervals and presents timing information as call graphs identifying the critical paths. You can 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.

Memory: You can use the <u>SKILL Profiler</u> to track memory usage in your program. SKILL has an automatic memory manager and programs can be written to generate excessive amounts of memory. Before you optimize for time, make sure to profile memory usage to see if that is where you need to spend your effort. A good indicator that memory usage needs optimizing is if the function gc (garbage collection) appears high among functions profiled for time.

See the following sections for more information:

- Optimizing Techniques on page 255
- General Optimizing Tips on page 258
- Miscellaneous Comparative Timings on page 265

Optimizing SKILL

# **Optimizing Techniques**

You can try the following techniques to optimize your code. Not all techniques are effective in all circumstances. You should experiment and measure performance gain.

- Macros
- Caching on page 255
- Mapping and Qualifying on page 256
- Write Protection on page 256
- Minimizing Memory on page 257

#### **Macros**

In many situations, you can improve the performance of your code by replacing function calls with <u>macros</u>. 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 354.

Optimizing SKILL

You could write the function myFib as a macro:

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 175 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.

### 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")

Optimizing SKILL

<u>Functions and Variables</u>" on page 238). 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 consing (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 208 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.

Optimizing SKILL

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 <u>"Memory Management (Garbage Collection)"</u> on page 219 to better understand the nature of automatic memory management.

# **General Optimizing Tips**

See the following sections for general optimizing tips:

- Element Comparison
- <u>List Accessing</u> on page 261
- List Building on page 261
- List Searching on page 264
- List Sorting on page 264
- Element Removal and Replacing on page 264
- Alternatives to Lists on page 265

# **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.

You can compare the following objects using eq:

- SKILL symbols
- Small integers (-2\*\*29 <= i <= 2\*\*28)</p>

Optimizing SKILL

- 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
                    => t
equal(x y)
s1 = "string"
s2 = "string"
s3 = s2
                   => t ; unreliable
eq(s1 s2)
                   => t ; reliable
equal(s1 s2)
eq(s3 s2)
                   => t ; reliable
equal(s3 s2)
                   => t
                          ; reliable
eq(12345 12345)
                    => nil
1 = '(12345)
eq(car(1) car(1))
                    => t
1 = '("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
```

Optimizing SKILL

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 built-in 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)	а

```
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)

Optimizing SKILL

# **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 n² list cells are created, and most of them are promptly discarded again.

Use ngong 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.

Optimizing SKILL

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.

Optimizing SKILL

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:

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)) \Rightarrow (1 2 3)
copylist('(a b c)) \Rightarrow (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).

Optimizing SKILL

# **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

Optimizing SKILL

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:

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
```

Optimizing SKILL

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
```

Optimizing SKILL

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
```

Optimizing SKILL

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:

```
1.00
access1
           1.3
access2
           1.47
access3
           1.08
access4
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 */
```

Optimizing SKILL

```
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.

# SKILL Language User Guide Optimizing SKILL

13

# About SKILL++ and SKILL

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

- 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.

Other advantages of SKILL++ include the following:

- SKILL++ provides environments as first-class objects. With closures and first-class environments, you can create your own module or package systems. You are no longer restricted to SKILL's single flat name space model. Instead, you can organize the code into a hierarchy of name spaces.
- In addition to 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.
- 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.

About SKILL++ and SKILL

See the following sections for more information:

- Background Information about SKILL and Scheme
- Relating SKILL++ to IEEE and CFI Standard Scheme on page 273
- Extension Language Environment on page 276
- Contrasting Variable Scoping on page 277
- Contrasting Symbol Usage on page 280
- Contrasting the Use of Functions as Data on page 282
- SKILL++ Closures on page 283
- SKILL++ Environments on page 286

# **Background Information about SKILL and Scheme**

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.

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. Cadence supplies major Scheme functionality as part of the SKILL environment. Benefits include the following:

- New programs written in Scheme can coexist 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. The converse is also true.

About SKILL++ and SKILL

# 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

Except for vector literals (such as #(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.

About SKILL++ and SKILL

#### **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.

About SKILL++ and SKILL

- 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 may not 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.
- You can use => and ... as identifiers.

# **Compliance Disclaimer**

Cadence-supplied Scheme is not fully IEEE compliant for the following reason:

- Scheme was not originally designed as an extension language. Features in Scheme that cannot be used safely 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 while Cadence's SKILL and SKILL++ treat nil as a Boolean

About SKILL++ and SKILL

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

You can read the following for more information about 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**

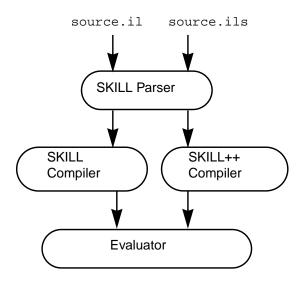
Cadence's extension language environment supports two integrated extension languages, SKILL and SKILL++. Your application can consist of source code written in either language. Programs in either language can call each other's functions and share data.

- You can maintain existing SKILL applications with no change whatsoever.
- Using SKILL++, you can hide private functions and private data so that you can design and implement your application with reusable components.

You should consider developing new applications using the SKILL++ language, in conjunction with SKILL procedural interfaces as necessary.

About SKILL++ and SKILL

For source code files, the file extension indicates the language: .il for SKILL, .ils for SKILL++.



By default, the session accepts SKILL expressions. Interactively, you can invoke a top level for either SKILL or SKILL++ as follows:

Language	Command	
SKILL	toplevel 'il	
SKILL++	toplevel 'ils	

Advanced programmers can use the eval function to evaluate an expression using 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, SKILL and SKILL++ use different scoping rules and partition a program into pieces differently.

About SKILL++ and SKILL

See the following sections for more information:

- SKILL++ Uses Lexical Scoping
- SKILL Uses Dynamic Scoping on page 278
- Lexical versus Dynamic Scoping on page 278
- Example 1: Sometimes the Scoping Rules Agree on page 279
- Example 2: When Dynamic and Lexical Scoping Disagree on page 279
- Example 3: Calling Sequence Effects on Memory Location on page 280

# **SKILL++ Uses Lexical Scoping**

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.

The lexical scoping rule is only concerned with the source code of your program. The phrase 'a 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**

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.

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.

# **Lexical versus Dynamic Scoping**

It is important that the scope of variables not be unintentionally disrupted when a programmer modifies or otherwise reuses a program. Subtle bugs can result when modification or reuse in another setting changes the scope of a variable.

Lexical scoping makes it possible for you to inspect the source code to determine the effect on the scope of variables. Dynamic scoping—which relies on the execution history of your

About SKILL++ and SKILL

program—can make it difficult to write reusable, modular code by preventing confident reuse of existing code.

# **Example 1: Sometimes the Scoping Rules Agree**

The following example has line numbers added for reference only.

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  $\mathbb A$  function between two references to  $\mathbb X$ . However, the  $\mathbb A$  function assigns a value to  $\mathbb X$ .

```
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  $\mathtt{B}$  updates the global variable  $\mathtt{x}$ . Yet when called from the function A, function B alters function A's local variable  $\mathtt{x}$  instead. Function B actually updates the  $\mathtt{x}$  that is local to the let expression in A.

- In SKILL, function A returns 6.
- In SKILL++, function A returns 5.

See "Dynamic Scoping" on page 214 for guidelines concerning the use of dynamic scoping.

# **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, see "Symbols" on page 90.

#### 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.

About SKILL++ and SKILL

#### **The Function Slot**

SKILL uses the function slot of a symbol to store function objects. SKILL evaluates a function call, such as

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. See <u>"Important Symbol Property List Considerations"</u> on page 95.

### **Summary**

You can call the set, symeval, getd, putd, get, and putprop SKILL functions to 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
assignment operator (=)	Х		
set, symeval	X		
let and prog constructs	Х		
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. In this way, SKILL and SKILL++ can share functions transparently.

Sometimes your SKILL++ program needs to access a SKILL global variable. You can use the importSkillvar function to change the binding of the SKILL++ global from the function slot of a symbol to the value slot of the symbol.

About SKILL++ and SKILL

# 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
```

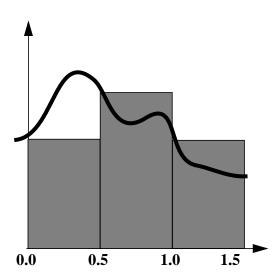
In SKILL, the same example can be done two different ways, both of them less convenient.

```
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.



About SKILL++ and SKILL

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.

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.

About SKILL++ and SKILL

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.

### **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

# Example 2

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**

About SKILL++ and SKILL

```
) ; 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

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.

About SKILL++ and SKILL

### **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 313 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.

### About SKILL++ and SKILL

# 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.

About SKILL++ and SKILL

### Example 1

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
у	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
у	3

#### About SKILL++ and SKILL

#### **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,

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.

About SKILL++ and SKILL

#### **Persistent Environments**

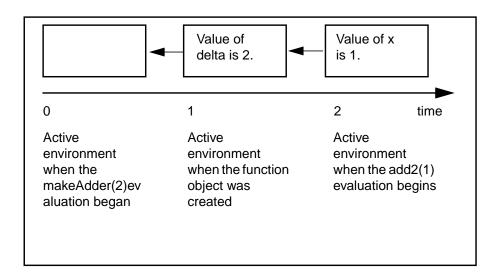
The makeAdder example below shows a function which allocates a function object and then returns it. The returned function object contains a reference to the environment that was active at the time the function object was created. This environment contains an environment frame that holds the actual argument 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:0xle6628
add2( 1 ) => 3
```

The function object that makeAdder returns is within the lexical scope of the delta argument.



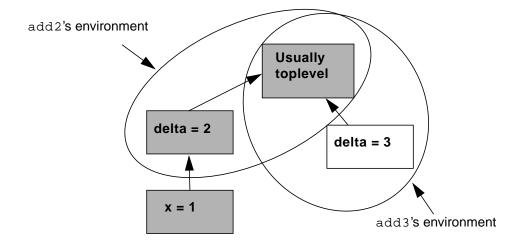
Calling makeAdder again returns another function object.

```
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

About SKILL++ and SKILL

add2(1) at its entry point. The other environment belongs to the add3 function, which becomes active only if add3 is called.



# SKILL Language User Guide About SKILL++ and SKILL

14

# **Using SKILL++**

This chapter deals with the pragmatics of writing programs in the Cadence<sup>®</sup> SKILL++ language.

<u>"About SKILL++ and SKILL"</u> on page 271 provides an overview of the differences between the Cadence SKILL language and SKILL++.

"Using SKILL and SKILL++ Together" on page 313 focuses on the key areas in which SKILL++ semantics differ from SKILL semantics.

<u>"SKILL++ Object System"</u> on page 327 describes a system that allows for object-oriented interfaces based on classes and generic functions composed of methods specialized on those classes.

For more information, see the following sections:

- <u>Declaring Local Variables in SKILL++</u> on page 293
- Sequencing and Iteration on page 298
- Software Engineering with SKILL++ on page 304
- SKILL++ Packages on page 304

## **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. You should avoid cross-references between variables in a let expression.



The initialization expression bound to one local variable should not refer to any of the other local variables.

#### Example 1

#### **Example 2**

#### 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

Using SKILL++

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

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

This example is identical to <u>"Example 3"</u> on page 294 except that the inner let is replaced with letsec.

#### **Example 3**

This example is identical to <u>"Example 4"</u> on page 294 except that the inner let is replaced with letseq.

Using SKILL++

## **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

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

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 is Even and isOdd.

Using SKILL++

#### **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.

#### Example 2

```
procedure( makeGauge( tolerance )
    let( ( ( iteration 0 ) ( previous 0.0 ) test )
        procedure( performTest( value )
             ++iteration
             test = ( abs( value - previous ) <= tolerance )</pre>
            previous = value
             when( test list( iteration value ))
             ) ; procedure
        performTest
        ) ; let
    ) ; procedure
G = makeGauge( .1 )
=> funobj:0x322b28
G(2) \Rightarrow nil
                               ;; first iteration
G(3) \Rightarrow 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 <u>"Example 4: Using a Gauge When Computing the Area Under a Curve"</u> on page 301.

Using SKILL++

#### Example 3

```
procedure( trPartition( nList )
   procedure( loop( numbers nonneg neg )
        cond(
            (!numbers list( nonneg neg ))
            (car(numbers) > 0
                1000(
                    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('(3-2165)) => ((5613)(-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.

Using SKILL++

#### 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
```

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**

A do expression evaluates in two phases: the initialization phase and the iteration phase.

Using SKILL++

The initialization expressions <code>g\_initExp1</code>, <code>g\_initExp2</code>, ... are evaluated in an unspecified order and the results bound to the local variables <code>var1</code>, <code>var2</code>, ...

The iteration phase is a sequence of steps going around the loop zero or more times with the exit determined by the termination condition.

- 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, terminationExp2, ... 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 <code>g\_loopExp1</code>, <code>g\_loopExp2</code>, ... are evaluated in order.
- 6. The step expressions <code>g\_stepExp1</code>, <code>g\_stepExp2</code>, ..., 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

#### 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.

Using SKILL++

#### **Example 3**

```
procedure( approximateArea( dx fun lower upper )
   do(
      ( ; loop variables
          ( sum
                       ;;; initial value
              0.0
              sum+fun(x) ;; update
          ( x
              lower ;;; initial value
              x+dx ;;; update expression
          ) ; end loop vars
      ( x \ge 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**

Using SKILL++

```
( ; 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)01.00001) => (160.4999924)
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\*(upperlower)/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 297 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
```

Using SKILL++

This example computes the factorial of 5 with a named let expression. Compare the example above with the following

#### and with the following

## 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
```

Using SKILL++

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 298 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 <u>Chapter 16, "SKILL++ Object System"</u>)

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.

Using SKILL++

## 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 ))
            ) ; 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,

Using SKILL++

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 ))
          ) ; 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.

Using SKILL++

- 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.

Using SKILL++

#### 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(S5) => (534)
```

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 ) ))
```

Using SKILL++

#### 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 the Stack.

Notice the syntactic convenience of calling the stack object indirectly through the fun variable.

#### **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.

Using SKILL++

#### 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 )
   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
           )
```

Using SKILL++

```
); let
); procedure
```

#### **Allocating Container Instances**

This example allocates a container instance with initial value 0.

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.

# SKILL Language User Guide Using SKILL++

This chapter discusses the pragmatics of developing programs in the Cadence<sup>®</sup> SKILL++ language. You should be familiar with both SKILL and SKILL++, specifically the material in <u>"About SKILL++"</u> on page 271 and <u>"Using SKILL++"</u> on page 293.

Developing a SKILL++ application involves the same basic tasks as for developing SKILL applications. Because most viable applications will involve tightly integrated SKILL and SKILL++ components, there are several more factors to consider:

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.

Communicating 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.

Using SKILL and SKILL++ Together

The phrase 'from within a SKILL program' means

- 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'.

For more information, see the following sections:

- Selecting an Interactive Language on page 315
- Partitioning Your Source Code on page 316
- Cross-Calling Guidelines on page 316
- Redefining Functions on page 318
- Sharing Global Variables on page 318
- <u>Debugging SKILL++ Applications</u> on page 320

# Selecting an Interactive Language

You can use SKILL or SKILL++ for interactive work. Both languages support a read-eval-print loop in which you repeat the following steps:

- 1. You type 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)

You can call 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 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 may call SKILL application procedural interface functions from a SKILL++ program. Most applications 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++, including the following:

- 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

Using SKILL and SKILL++ Together

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.

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
```

#### SetMyArg Called from SKILL++

```
> toplevel 'ils
ILS-<2> let( (( x 3 ))
   SetMyArg( 'x 5 )
   x
   ) ; let
```

# **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 )
```

**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  $f \circ o$  is bound to the function slot of the symbol  $f \circ o$ . 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.

Using SKILL and SKILL++ Together

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

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

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 \rightarrow ??? \Rightarrow (((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 \Rightarrow envobi: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)))
```

See the makeStack example in "Implementing the makeStack Function" on page 308

```
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.

# Using SKILL and SKILL++ Together

### 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)
funcall((MathPackage->add) 3 4)
toplevel('ils)
ILS-<3>
```

**SKILL Language User Guide**Using SKILL and SKILL++ Together

# SKILL++ Object System

To gain benefits from object-oriented programming, the Cadence<sup>®</sup> 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 SKILL++ Object System is modelled after a subset of the Common Lisp Object System with the following restrictions:

- In the SKILL++ Object System, a class can have only a single superclass. (CLOS allows a class to have more than one superclass.)
- In the SKILL++ Object System, a method can only be applied to the class of the method's first argument.
- In the SKILL++ Object System, all methods are primary. (CLOS allows for ancillary methods known as *before* and *after* methods which are combined with the primary method.)

For more information, see the following sections:

- Basic Concepts on page 328
- Class Hierarchy on page 334
- Browsing the Class Hierarchy on page 335
- Advanced Concepts on page 338

SKILL++ Object System

# **Basic Concepts**

The following items are central concepts of the SKILL++ Object System:

- Classes and Instances on page 328
- Generic Functions and Methods on page 328
- Subclasses and Superclasses on page 329

For more information, see the following sections:

- Defining a Class (defclass) on page 329
- Instantiating a Class (makeInstance) on page 331
- Reading and Writing Instance Slots on page 331
- Defining a Generic Function (defgeneric) on page 332
- <u>Defining a Method (defmethod)</u> on page 332

#### 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

SKILL++ Object System

### **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.

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

SKILL++ Object System

### **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

### Example 2

# 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.

The print representation for a SKILL++ Object System instance consists of stdobj: followed by a hexadecimal number.

# **Reading and Writing Instance Slots**

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.

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.

SKILL++ Object System

```
@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.

This example indicates that relevant subclasses of the <code>geometricObject</code> class, such the <code>polygon</code> class, should have a <code>Perimeter</code> 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 338). 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.

SKILL++ Object System

- 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 following syntax:

```
( s_arg1 s_class )
```

### **Example 1**

This example defines a method named Perimeter. It is specialized on the Triangle class.

### Example 2

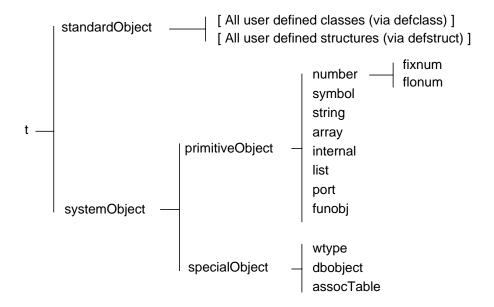
```
defmethod( Perimeter (( c Circle ))
    2*c->r*3.1415
    ); defmethod
```

This example defines a Circle class and defines the Perimeter method for the Circle class.

### SKILL++ Object System

# **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 hierarchy is at any given time.

```
procedure( getDirectSubclasses( classObject )
    foreach( mapcar c subclassesOf( classObject )
        className( c )
        ) ; foreach
    ) ; procedure
```

### SKILL++ Object System

```
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:

- Getting the Class Object from the Class Name on page 336
- Getting the Class Name from the Class Object on page 336
- Getting the Class of an Instance on page 336
- Getting the Superclasses of an Instance on page 336
- Checking if an Object Is an Instance of a Class on page 337
- Checking if One Class Is a Subclass of Another on page 337

SKILL++ Object System

Examples in these 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)
```

SKILL++ Object System

### 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.

SKILL++ Object System

### Example 3

```
subclassp(
    findClass( 'Triangle )
    findClass( 'fixnum )
    ) => nil
```

Triangle is not a subclass of fixnum.

# **Advanced Concepts**

This section covers the following advanced aspects of the SKILL++ Object System:

- Method Argument Restrictions
- Applying a Generic Function on page 339
- Incremental Development on page 340
- Methods versus Slots on page 341
- Sharing Private Functions and Data Between Methods on page 341

# **Method Argument Restrictions**

Method argument lists have the following restrictions:

Aspect	Restriction
Number of arguments	All methods of a generic function must have the same number of required arguments and @optional arguments.
@rest <b>arguments</b>	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
	<ul> <li>Allow a superset of the keyword arguments specified in the defgeneric declaration</li> </ul>
	@rest picks up all 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.

SKILL++ Object System

### **Applying a Generic Function**

When you apply a generic function to some arguments, the SKILL++ Object System performs the following actions to complete the function call. This process is called *method dispatching*. 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.

You can invoke the callNextMethod function from within a method to access the next applicable method in the ordering. For example:

SKILL++ Object System

```
) ; 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

In the SKILL++ environment, you can redefine SKILL++ functions incrementally. You should 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

SKILL++ Object System

- 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.

SKILL++ Object System

```
defmethod( Fun1 (( obj Example) .... )
defmethod( Fun2 (( obj Example ) ... )
...
) ; let
```

# **Programming Examples**

See the following sections for programming examples:

- <u>List Manipulation</u> on page 344
- Symbol Manipulation on page 344
- Sorting a List of Points on page 345
- Computing the Center of a Bounding Box on page 346
- Computing the Area of a Bounding Box on page 347
- Computing a Bounding Box Centered at a Point on page 347
- Computing the Union of Several Bounding Boxes on page 348
- Computing the Intersection of Bounding Boxes on page 348
- Prime Factorizations on page 349
- Fibonacci Function on page 354
- Factorial Function on page 354
- Exponential Function on page 355
- Counting Values in a List on page 355
- Counting Characters in a String on page 357
- Regular Expression Pattern Matching on page 357
- Geometric Constructions on page 358

# Programming Examples

# **List Manipulation**

A list is a linear sequence of Cadence<sup>®</sup> 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")
```

Programming Examples

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 symbol 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.

# **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 <u>xCoord</u> and <u>yCoord</u> functions are aliases for the <u>car</u> and <u>cadr</u> functions.

■ Using the cond function

```
procedure( trPointLowerLeftp( pt1 pt2 )
   let( ( pt1x pt2x pt1y pt2y )
     pt1x = xCoord( pt1 )
   pt2x = xCoord( pt2 )
   cond(
```

### Programming Examples

The trSortPointList function returns a list of points sorted destructively and illustrates

- Documenting at the function level
- Using the sort function

# 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 )
```

**Programming Examples** 

```
); 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

Programming Examples

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 )
                list( maxllx:maxlly minurx:minury ))
      ) ; 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:

### **Programming Examples**

- 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

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
```

**Programming Examples** 

```
); 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

The trpf function returns the prime factorization a number. For example:

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

### **Programming Examples**

```
procedure( trPF( aNumber )
                   ; locals
    let(
              (
        ( 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, pfl and pfl. 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

**Programming Examples** 

```
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
                ;; destructively appends these lists 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 the greatest common denominator (GCD) of two prime factorizations. This function illustrates

Using foreach( mapcan ... ) to merge two association lists.

Primes found in both association lists are given an exponent equal to the minimum of the exponents in both the association lists.

Using the backquote (`) and comma (,) operators.

Programming Examples

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( numl num2 )
          trEvalPF( trPFGCD( trPF( numl ) 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:

# **Factorial Function**

This is the recursive implementation of the factorial function

This is an iterative implementation

```
procedure( factorial( n )
    let( ( ( f 1 ))
        for( i 1 n
```

### Programming Examples

```
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.

To get a sense of the accuracy of this implementation of the e function, observe

```
e(log(10)) \Rightarrow 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

■ The second implementation of trCountValues produces

### Programming Examples

```
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
```

# Programming Examples

```
); 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.

Programming Examples

```
("?" (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")
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**

Here is an extensive example of a SKILL++ Object Layer application.

### **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

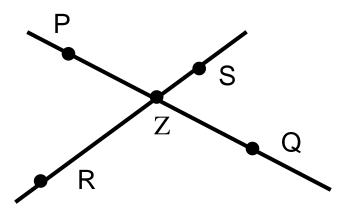
When you move any one of the free points, the application propagates the change to all the constrained points and lines

#### 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.

### **Programming Examples**

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.

### **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

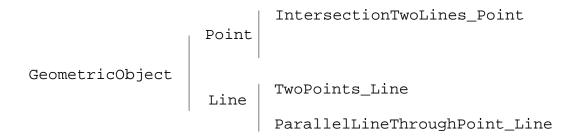
To focus on SKILL++ language issues, the implementation does not address graphics. Instead, you non-graphically

- 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.

**Programming Examples** 

#### **Classes**

The implementation uses the SKILL++ Object System to define several classes in the following class hierarchy.



### **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 notified when the object updates.

#### **Point Class**

The Point class represents a point with slots x and y.

#### **Line Class**

The Line class represents a line with the slots A, B, and C. These are the coefficients in the line's equation

$$Ax+By+C = 0$$

#### 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.

**Programming Examples** 

#### **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.

#### 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.

#### **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.

**Programming Examples** 

#### **Describing the Methods by Class**

The following tables describe, for each generic function, all the methods by class. In the following tables,

- 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.

#### **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.	

#### **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.	

**Programming Examples** 

#### **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.	

#### **Connect Methods**

#### **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.	

#### **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
   ) ; 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 \rightarrow L2 \rightarrow 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( obi )
   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" ))
        ) ; 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
```

#### Programming Examples

```
LC = obj -> L -> C
            = obj->A
        Α
        В
            = obj->B
        С
           = obj->C
        when( A*LB-LA*B != 0.0 | A*x1+B*y1+C != 0
            error( "Invalid ParallelLineThroughPoint_Line\n" ))
        ) ; 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
        ob\bar{j} -> 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

**Programming Examples** 

■ 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
R = makeInstance( 'Point ?x 3 ?y 0 )
stdobj:0x36f024
Describe(R)
Point at 3:0
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
P->x = 1
1
Update( P )
Updating constraints (stdobj:0x36f030) for stdobj:0x36f018
Updating constraints nil for stdobj:0x36f030
Describe( L )
TwoPoints_Line 2.000000x+1y+-6.000000=0
...containing
Point at 1:4
Point at 3:0
```

#### **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
S->y = 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
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
```

**Programming Examples** 

```
...containing
Point at -3:4
Point at 3:0
t
```

#### **Extending the Implementation**

Consider the following extensions:

Protecting the implementation from inconsistent access

You can use the -> operator to change coordinates of any point, even a constrained point. You can extend the implementation to allow the user to change only coordinates of free points.

Adding graphics

You can extend the implementation to associate a database object with each point or line and update the database object at appropriate times. You should have to affect only the Point and Line classes.

# SKILL Language User Guide Programming Examples

## Index

Symbols

#f 274 #t 273 && (and) operator 46 -> operator 323 @key option 81, 347 @optional option 81 @rest option 78, 80    (or) operator 46	arrow operator 94, 310, 323 assignment operator (=) 72, 281 assoc function 117, 193, 356 association lists 113, 117, 199 association tables caching with 255 definition 113 functions for 114 initializing 113 scanning the contents 114 traversing 116 atom function 134 autoload feature 168 autoload property 232
absolute path 150 access operators array access syntax [] 90 arrow (->) 90 dot operator 94 slot access (->?) 110 slot access (->??) 110 squiggle arrow (~>) 90 algebraic notation 61 alphalessp function 100 alphaNumCmp function 100 alphanumeric characters 72, 91 append function 37, 114 append1 function 261 apply function 80, 183, 195, 206, 348 argument definition 76 type template 83 arithmetic	backquote (') 65, 262 begin function 298 binary minus operator 63 bindkeys 31 bitwise operators 126, 134 bounding boxes 40, 346 boundp function 85, 134, 323 bracket, super right 65, 218 break function 325 building a list 36 buildString function 99 byte-code 76, 204
integer-only 131 mixed-mode 128 operators 121 precision 129 predefined functions 125 predicate functions 133 arrays accessing 90, 112 declaring 112 definition 111 pointers to 112 sparse 116 syntax statement 113	C language comparison arithmetic and logical syntax 133 brackets affecting evaluation 244 commas 246 common mistakes 248 defstructs 107 draining ports 161 escape characters 70 getc and getchar 171 getchar 102 handling variables 214 macros 210

parentheses <u>64</u> string functions <u>99</u>	concat function 91 cond function 141, 345
strings 70	conditional operators 134
symbol property list 93	cons cells $\frac{274}{27}$
symbols <u>90, 92</u>	cons function <u>37</u> , <u>38</u> , <u>181</u> , <u>261</u> , <u>267</u>
true and false 132	constants 119
caching results 255	containers, definition of 309
cadr function 350	contexts
callInitProc function 236	autoloading <u>237</u>
car function 38, 350	creating directory structure for 226
case function 48, 142	creating utility functions for 228
caseq function 142	customizing external 233
cdr function 38	definition <u>224</u>
cdsGetInstPath function 153	difference from source files 224
CFI Scheme, relationship to SKILL++ 273	functions for building 235
changeWorkingDir function 159	global function protection 240
characters, counting in a string 357	how the process works 227
CIW 30	how to build 230
class data structure 328	initializing <u>230</u>
class hierarchy	loading <u>231</u>
browsing 335	potential problems 233
definition of 334	restrictions <u>224</u>
className function 336	when to use <u>225</u>
classOf function 336	control functions 140
classp function 337	conventions
close function <u>161</u>	naming <u>58</u>
closures 271	converting case 103
behavior of 284	coordinates 39
examining <u>324</u>	copy function <u>186</u>
examples <u>284</u>	copy_ <name> function 109</name>
in SKILL++ <u>283</u>	copyDefstructDeep function 109
relationship to free variables 283	copying and pasting examples 23
coding <u>248</u>	createDir function <u>156</u>
commenting <u>242</u>	cross-calling guidelines 316
comparing execution times 265	csh function <u>172</u>
layout <u>242</u>	
optimizing <u>253</u>	<b>D</b>
serious errors 2 <u>50</u>	ט
style <u>242</u>	
style mistakes 248	data sharing across languages 313
comma (,) <u>65</u>	data structures 268
comma-at (,@) <u>65</u>	data types
Command Interpreter Window(CIW) 30	supported <u>67</u>
commenting code 63, 242	user-defined <u>117</u>
common questions <u>37</u>	declare function 112
compareTime function <u>173</u>	defclass function 328, 329, 341
compilation <u>76, 204</u>	defgeneric function 328, 332
compile time 76	defining
complex numbers <u>131</u>	functions <u>77</u>
compress function 238	parameters <u>80</u>
compressing files 238	definitProc function 230, 236

1 - (	
defmacro	using arrow operator with 323
function <u>78, 211, 274</u>	envobj function 323
with @key <u>213</u>	eof object 274
with @rest <u>212</u>	eq function 135, 258, 265
with backquote 211	equal (==) operator <u>132</u>
defmethod function 328, 332	equal function 123, 135, 258, 265
defprop function 98	
	equals sign (=) operator 72 err function 216
defetruete	
defstructs	error function <u>217</u>
alternatives to lists 265	error handling 215
definition 108	errset function 215
example 110	errsetstring function 168
defUserInitProc function 237	escape sequences 71
deleteDir function 157	eval function <u>205</u> , <u>277</u> , <u>323</u>
deleteFile function <u>157</u>	evalstring function 167
deleting	evaluation <u>30</u>
directories <u>157</u>	advanced <u>205</u>
files <u>157</u>	order of 132
disembodied property lists 95, 113	preventing <u>121</u>
displaying data 41	process <u>204</u>
division, integer vs float 129	evenp function <u>133</u>
do function 299	exists function <u>116, 186, 191, 200</u>
documenting a function 83	exiting SKILL <u>221</u>
dot (.) operator 94	exponentiation operator (**) 133
drain function 161	expressions, nested 121
draining	extension language environment 276
output buffer <u>161</u>	<b>0 0</b> —
ports <u>161</u>	_
dynamic	
W 1 1 1 W 1 1 1 1 V	F
	Г
globals <u>214</u>	
	false condition 132
globals <u>214</u>	false condition 132 Fibonacci function 354
globals <u>214</u> scoping <u>214, 278</u>	false condition 132 Fibonacci function 354 fileLength function 155
globals <u>214</u>	false condition 132 Fibonacci function 354 fileLength function 155 files
globals 214 scoping 214, 278	false condition 132 Fibonacci function 354 fileLength function 155 files compressing 238
globals 214 scoping 214, 278  E  encrypt function 238	false condition 132 Fibonacci function 354 fileLength function 155 files compressing 238 encrypting 238
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238	false condition 132 Fibonacci function 354 fileLength function 155 files compressing 238 encrypting 238 fileSeek function 156
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238 fileSeek function 156 fileTell function 156
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments creating 287	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments creating 287 definition 286	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments creating 287 definition 286 evaluating an expression in 323	false condition 132 Fibonacci function 354 fileLength function 155 files compressing 238 encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments creating 287 definition 286 evaluating an expression in 323 first-class 271	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238     ileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point     numbers 69
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286 inspecting 322	false condition 132 Fibonacci function 354 fileLength function 155 files     compressing 238     encrypting 238     fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point     numbers 69     precision 129
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286 inspecting 322 persistent 290	false condition 132 Fibonacci function 354 fileLength function 155 files  compressing 238 encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 float function 129 floating-point numbers 69 precision 129 for function 49, 142
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286 inspecting 322 persistent 290 retrieving 323	false condition 132 Fibonacci function 354 fileLength function 155 files  compressing 238 encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point numbers 69 precision 129 for function 49, 142 forall function 116, 191
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286 inspecting 322 persistent 290 retrieving 323 retrieving active environment 322	false condition 132 Fibonacci function 354 fileLength function 155 files  compressing 238 encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point numbers 69 precision 129 for function 49, 142 forall function 116, 191 foreach function 50, 116, 192
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286 inspecting 322 persistent 290 retrieving 323 retrieving active environment 322 testing variables in 323	false condition 132 Fibonacci function 354 fileLength function 155 files  compressing 238 encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point numbers 69 precision 129 for function 49, 142 forall function 116, 191 foreach function 50, 116, 192 form 166
globals 214 scoping 214, 278  E  encrypt function 238 encrypting files 238 environments    creating 287 definition 286 evaluating an expression in 323 first-class 271 for extension languages 276 in SKILL++ 286 inspecting 322 persistent 290 retrieving 323 retrieving active environment 322	false condition 132 Fibonacci function 354 fileLength function 155 files  compressing 238 encrypting 238 fileSeek function 156 fileTell function 156 findClass function 336 Finder quick reference tool 23 fix function 129 float function 129 floating-point numbers 69 precision 129 for function 49, 142 forall function 116, 191 foreach function 50, 116, 192

free variable 283 fscanf function 44, 170	I
function body <u>76</u> function call <u>119</u>	IEEE Scheme, relationship to SKILL++ 273
function objects 76, 207 functions. See SKILL functions funobj function 320	if function <u>47, 140</u> ilLoadIL option <u>230</u> importSkillVar function <u>319</u>
G	include function <u>238</u> index function <u>102</u> indirection operator <u>134</u>
garbage collection 219	infile function 43, 161
manual allocation 221 process 219	infix arithmetic operators 61 infix operators 64, 120 information hiding 304
summary statistics 220 write protection effect on 256	input form 166
GC. See garbage collection gcsummary function 220	functions 167
generic functions 328	lines <u>66</u> inSkill function <u>320</u>
gensym function 91 get function 98, 281	installation instructions 23 instring function 171
get_pname function 91 getc function 171	integers <u>68</u> interactive loop
getCurrentTime function 173 getd function 207, 281	exiting <u>315</u> starting <u>315</u>
getDirFiles function <u>158</u> getFnWriteProtect function <u>239</u>	interpreter managing symbols 144
getInstallPath function 153 getqq function 94	top level 218 isDir function 154
gets function 170 getShellEnvVar function 173	isExecutable function 155
getSkillPath function 53, 152 getVarWriteProtect function 240	isFile function 154 isFileName function 154
getVersion function 173 getWarn function 217	isReadable function <u>154</u> isWritable function <u>155</u>
getWorkingDir function 159	•
global variables misusing 246	L
naming <u>85</u> reducing <u>86</u>	lambda function 77, 78, 191, 208, 268 languages
sharing <u>318</u> testing <u>84</u>	data sharing 313 interactive selection 315
globals, dynamic 214	partitioning source code 313, 316 redefining restriction 318
Н	last function 181
hash table. See association tables	length function <u>39</u> let function <u>83, 281, 294, 351</u>
hiding private functions 276, 304	letrec function 296 letseq function 295 lexical scoping 214, 271, 278

line continuation 66 lineread function 169, 274 linereadstring function 169 Lisp language comparison brackets affecting evaluation 245 building lists 262 commas 246 data manipulation comparison 28 data type difference 29 lambda calculus 208 programming notation 28	validating 190 literal characters 24 load function 168, 238 loadContext function 236 loadi function 168, 238 loadstring function 168 local variables 83 logical operators 46, 121, 132 lowerCase function 103 lowerLeft function 346
to SKILL++ Object System 327 list cells 176	M
list function <u>37, 262</u>	•••
list function 37, 262 listp function 345 lists  accessing 38, 180, 261 altering 179 alternatives to 265 appending 183 association lists 199 building 36, 181, 261, 266 cells 176 commenting traversal code 202 containing sublists 177 containing symbols 177 copying hierarchically 187 copying the top-level 186 definition 35 destructive modification 178 element removal 264 filtering 187 flattening many levels 199 flattening with mapcan 198 how they are stored 176 input length 66 modifying 39 non-destructive modification 178 preventing evaluation of 121 removing elements from 188 reorganizing 184 searching 185, 264 sorting 264 substituting elements 189	macro function 77 macros benefits 210 defining 211 expansion 210 optimizing usage 255 redefining 211 make_ <name> function 109 makeContext function 240 makeInstance function 113 makeTable function 113 makeTempFileName function 157 map function 193 map function 192 mapcan function 194 mapc function 194 mapping function 194 mapping function 267 mapping, performance gains 256 max function 348 measureTime function 254, 269 member function 39, 136, 185, 344 memory allocation 219 memory management. See garbage collection memory, optimizing usage 254, 257 memq function 136, 186 method dispatching 339 min function 348</name>
summary of operations 178	minusp function 133
transforming elements 190	misusing conditionals 248
traversing	modulo operator 133
checking part of a list 200 examples <u>197</u> summary <u>196</u> with mapping functions <u>191</u>	mprocedure function <u>79, 82, 211, 274</u>

N	backquote (') 65, 262
••	binary minus <u>63</u>
named let 302	bitwise <u>126, 134</u>
naming conventions	bnot (~) <u>122</u>
Cadence-private functions 59	brackets [ ] <u>122</u>
functions <u>58</u>	colon (:) <u>346</u>
variables <u>59</u>	comma (,) <u>65</u>
nconc function 183, 196, 261, 344	comma-at (,@) <u>65</u>
ncons function 182	conditional 134
needNCells function 221	difference (-) <u>123</u>
neq function 136	dot (.) 94, 122
nequal (!=) operator 132	equal (==) <u>123, 132</u>
nequal function 123, 136	equals sign (=) <u>72, 281</u>
nesting	exponentiation $(**)$ 122, 133
expressions <u>121</u>	greater than (>) 123
function calls 61	greater than or equal (>=) 123
newline function <u>162</u>	indirection 134
nindex function 102	infix 61, 120, 125
nlambda function <u>77</u>	introduction <u>33</u>
nograph option 230	leftshift (<<) <u>123</u>
notation	less than (<) 123
algebraic <u>61</u>	less than or equal (<=) 123
prefix <u>61</u>	logical <u>46, 132</u>
nprocedure function <u>78</u>	minus (–) <u>122</u>
nth function 38	modulo <u>133</u>
nthcdr function 180	nand (~&) 123
nthelem function 180	nequal (!=) <u>132</u>
numbers	nor (~ ) <u>123</u> not equal (!=) <u>123</u>
floating-point <u>69</u>	
integers <u>68</u>	null (!) <u>122</u> order of evaluation <u>132</u>
scaling factors 69	plus (+) 122
numOpenFiles function <u>155</u>	postdecrement (s) 124
	postincrement (s++) 122, 124, 351
0	precedence 121, 124
O	predecrement (s) 122, 124
ship at a right of a regression 227	preincrement (++s) 122, 124, 351
object-oriented programming 327	quote (') <u>262</u>
oddp function 133	quotient (/) 122
onep function 133	range (:) 123
operators — 123	relational 45, 132
<u> </u>	rightshift (>>) 123
<> 122	slot access (->?) 110
^ 123	slot access (->??) 110, 323
123	squiggle arrow (~>) <u>90, 122</u>
123   123, 133	times (*) 122
and (&&) 123, 133	unary minus 63
arithmetic 124	xnor (~^) <u>123</u>
arrow (->) 90, 94, 122, 310, 323	optimizing tips 258
assignment (=) $\frac{72}{72}$ , $\frac{281}{281}$	order of evaluation 64, 132
, <u></u> , <u></u> ,	outfile function 42, 161

output	printstruct function 108, 116
formatted 163	private functions, hiding 276, 304
unformatted <u>162</u>	problems
	common questions 37
n	data type mismatches 35
P	inappropriate space characters 35
	most common 34
package, definition of 304	system doesn't respond 34
parameters	procedure function <u>77</u> , <u>281</u> , <u>297</u>
defining <u>80</u>	procedures
definition 76	definition <u>76</u>
parentheses <u>64, 121</u>	returning from 249
parser, character limits 66	See also SKILL functions
parseString function 102, 357	profiling <u>253</u>
partitioning source code 313, 316	prog function 84, 144, 145, 249, 281
Pascal language comparison	prog1 function 147
handling variables 214 p-code 204	prog2 function <u>147</u> Prolog language comparison <u>262</u>
symbol property list 93	property name/value pairs 93
symbols <u>92</u>	putd function 208, 281
paths	putprop function <u>97, 232, 281</u>
absolute 150	putpropg function 94, 122
relative 150	putpropgq function 122
SKILL path <u>151</u>	F = 14
pattern matching	
example <u>357</u>	Q
functions 105	
plist function 93	querying system status 172
plusp function <u>133</u>	quick reference for chapters 21
ports	quick reference tool 23
closing <u>160</u>	quoting <u>121,</u> <u>262</u>
draining <u>161</u>	
opening 160	D
predefined 160	R
pp function <u>165, 321, 324</u>	and and aniation 004
pprint function 166	read-eval-print loop 204
precision, floating point 129	reading UNIX text files 43
predicates 125	readTable function <u>115</u> regExitAfter function 221
comparing functions 135 functions for testing 134	regExitAfter function <u>221</u> regExitBefore function <u>221</u>
testing the data type of 134, 137	relational operators 45, 132
prefix notation 61	relative path 150
prependInstallPath function 153	remd function 189
pretty printing 165, 321, 324	remdq function 189
preventing evaluation 121	remove function 188
prime factorization examples 349	remprop function 98
primitives <u>119</u>	remg function 188
print function 41, 162	reserved names 84
printf function 41, 43, 164	resume function 315
printlev function 162	return function 144, 146
println function 41, 162	return value (=>) 32

returning from a procedure 249 reverse function 184, 261, 267 rexCompile function 105 rexExecute function 106 rexMagic function 107 rexMatchAssocList function 106 rexMatchList function 106, 357 rexMatchp function 105, 357 rexReplace function 107 rindex function 102 rplaca function 117, 179, 193, 356 rplacd function 180 run time 76	compiler 30 evaluator 204, 205 exiting 221 expression 30 interpreter 30 Lisp background 272 path 54, 151 primitives 119 top level 218 white space characters in code 64 white space in code 63 SKILL Development Help 23 SKILL functions arguments 76
S	calling syntax 32 conditional functions 140 constructs for defining 77
saveContext function 224, 235 scaling factors 69 Scheme bibliography 276 coexistence with SKILL 272 compliance disclaimer 275 functionality in SKILL 272 lexical scoping in 271 migration to 272 origins 272 run-time environment 272 scope of a variable 277, 278 scoping dynamic 214, 278 examples 279 lexical 214 lexical in Scheme 271 lexical in SKILL++ 278	declaring 52 defining 77 defining parameters 52 definition 30, 76 developing 51 documenting 83 global protection 240 grouping statements 51, 145 hiding private functions 304 iteration functions 141 kinds of 76 making calls 61 overloading 131 parameters 76 physical limits 87 protecting 238 redefining 54, 86 redefining restriction 318
selecting a language 315 set function 92, 281, 318	selecting prefixes for 52 selection functions 142
setContext function 236 setFnWriteProtect function 239 setof function 116, 187, 344	terminology <u>76</u> testing predicates <u>134</u> ways to submit <u>31</u> SKILL++
setplist function 93 setq function 123 setShellEnvVar function 173 setSkillPath function 53, 152 setVarWriteProtect function 239 sh function 172 shell function 172 simplifyFilename function 158 single quote operator 37 SKILL commenting code 63	backward compatibility 271, 275 calling a function in 289 creating a function 289 creating environments 287 debugger commands, general 324 debugging applications 320 declaring local variables in 293 environments 286 function objects in variables 282 functions as arguments 282

functions as data 282 inspecting environments 322 introducing 271 iteration functions 298 modules 307 no Scheme dotted pairs 275 packages 304 returning function behavior 289 semantic differences from Scheme 274 sequencing functions 298 special character restrictions 275 syntax differences from Scheme 273 syntax options 275 SKILL++ Object System advanced concepts 338 browsing class hierarchy 334 classes 328 defining a class 329 defining a method 32 generic functions 328, 332 getting subclasses 337 getting superclasses 336 getting superclasses 336 incremental development 340 instance slots 331 instance slots 331 instance slots 331 introduction 327 method dispatching 339 methods vs. slots 341 relationship to Lisp 327 sharing data 341 sharing private functions 328 subclasses 329 superclasses 329 superclasses 329 superclasses 329 sort function 185, 346	stack definition 307 object 308 stacktrace function 325 status function 131 storing programs as data 209 strcat function 99 strcmp function 100 strings characters in 101 comparing 100 concatenating 99 converting case 103 creating substrings 102 definition 70 functions for 99 indexing 101 pattern matching 103 tail of 102 stringToFunction function 209 strlen function 101 strncat function 99 strncmp function 101 subclassp function 101 subclassp function 102 substring function 189 substring function 102 superclassesOf function 336 sxtd function 127 symbol names 72, 91 symbols assigning values to 92 components of 90 creating 91 disembodied property lists 95 function slot 281 in SKILL 280 in SKILL++ 281 print name 91
sortcar function 185	property list 281
source code	property lists 95
loading <u>53</u>	retrieving <u>92</u>
maintaining <u>53</u>	using quote operator <u>92</u> value slots 2 <u>80</u>
partitioning 313, 316 specifying extension language	symeval function <u>92, 205, 280, 281, 345</u>
by file extension 277	syntax <u>61</u>
interactively <u>277</u>	functions 77
using eval function 277	syntax conventions 24
using toplevel function 277	· —
sprintf function 165	
sstatus function 131	

tablep function 115 tableToList function 115 tailp function 137 tconc function 182, 184, 261, 267	protecting <u>238</u> reserved names <u>84</u> sharing globals <u>318</u> unbound <u>71, 72</u> virtual machine <u>76</u>
text files reading from 43 writing to 42 theEnvironment function 274, 322 toolbox for SKILL development 23 top level of SKILL 218 specifying a language for 277 toplevel function 315 tracef function 324 true condition 132 type characters, composite 82 type template 83	warn function 217 What's New 23 when function 48, 140, 351 while function 141, 200, 351 white space 63 white space characters 64 writeTable function 115 writing UNIX text files 42
U	xcons function $\frac{181}{346}$
unary minus operator 63 unbound variables 71, 72 UNIX device 150 directory 150 directory paths 150 file system 150 reading text files 43 writing text files 42 unless function 48, 140 upperCase function 103 upperRight function 346	yCoord function 346  Z zerop function 133
V	

variables

advanced concepts 214

global 84
internal system 59
introduction 33
misusing globals 246

definition 119

declaring locals with prog defining local 83

preventing evaluation of 121