Introduction to SKILL® Programming

Version 6.1

Lecture Manual   **April 30, 2007**

cādence

Table of Contents
Introduction to SKILL Programming

Module 1   About This Course

Module 2   SKILL Programming Fundamentals

Module 3   Lists

Module 4   Windows

Module 5   Database Queries

Module 6   Developing a SKILL Function

Module 7   Flow of Control

Module 8   List Construction

Appendix A Menus

Appendix B Customization

Appendix C File I/O

About This Course

Module 1

April 30, 2007

Course Objectives

In this course you will

- Learn the role of the SKILL® programming language in the Virtuoso® Design Environment.

- Master SKILL syntax, loop constructs and conditional statements.

- Build and manipulate lists.

- Examine the design database model and implement SKILL queries and functions to create, access, and update the data.

- Define, develop, and debug SKILL functions and programs.

Terms and Definitions

SKILL	The SKILL programming language is a LISP-like language with I/O structures based on those in the C language. It extends and enhances the functionality of Cadence® tools.

Course Agenda

Day 1	Day 2
About This Course	Queries
↓	↓
Fundamentals	SKILL Functions
↓	↓
Lunch	Lunch
↓	↓
Lists	Flow of Control
↓	↓
Windows	List Construction
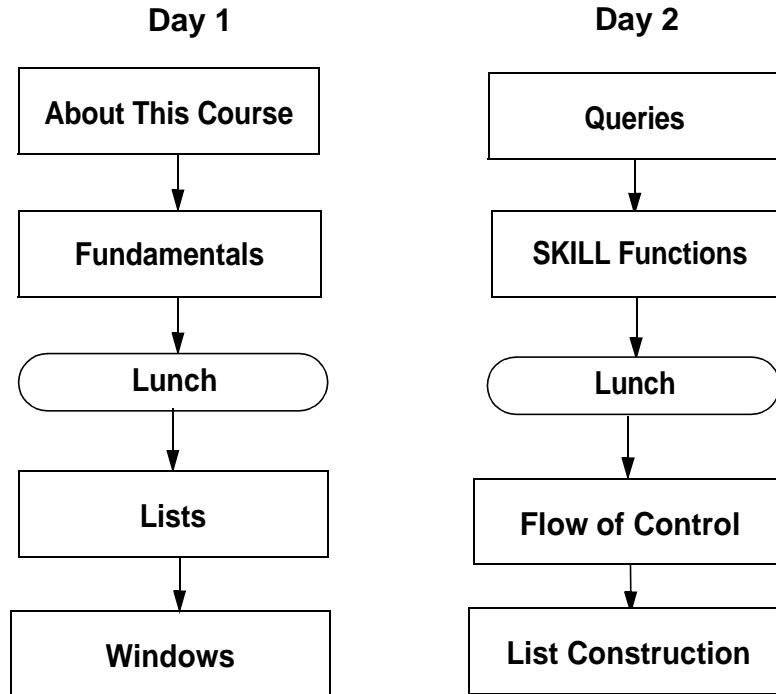
#	Module	Purpose or Objective
1	About This Course	Orientation to the course and SKILL documentation.
2	SKILL Programming Fundamentals	Survey the role of the SKILL Evaluator in the Virtuoso Design Environment. Survey SKILL data types and syntax.
3	SKILL Lists	Build SKILL lists. Retrieve data from SKILL lists.
4	Windows	Manipulate Virtuoso Design Environment windows. Open Virtuoso Design Environment application windows. Define application bindkeys.
5	Database Queries	Query Virtuoso Design Environment cellviews.
6	Developing a SKILL Function	Learn to define a SKILL function. Define global and local variables.
7	Flow of Control	Use the branching capabilities of the SKILL language.
8	List Construction	Use advanced techniques to build SKILL lists.

Curriculum Planning

Cadence offers courses that are closely related to this one. You can use this course map to plan your future curriculum. The courses listed are currently offered or planned for the near future.
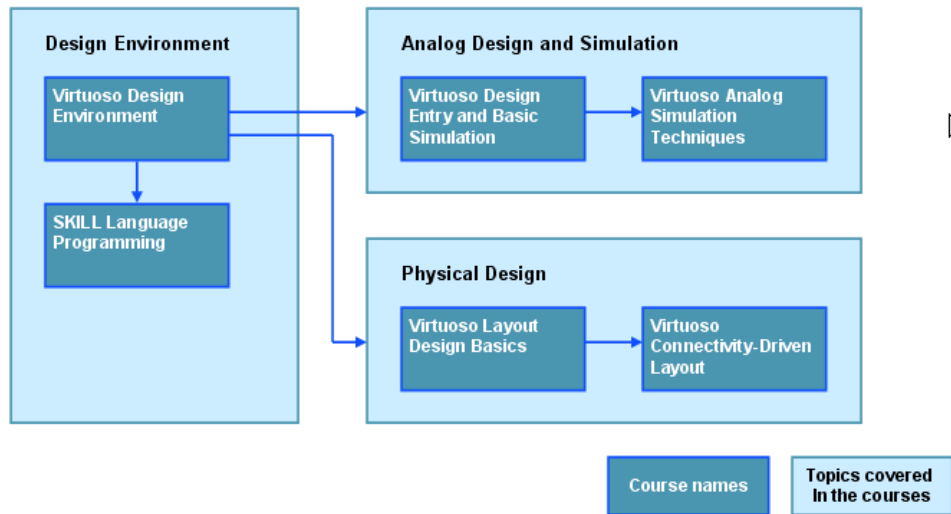
For more information about Cadence courses:

1. Point your web browser to cadence.com.

2. Click **Education**.

3. Click the **Course catalog** link near the top of the center column.

4. Click a Cadence technology platform (such as **Custom IC Design**).

5. Click a course name.

The browser displays a course description and gives you an opportunity to enroll.

Getting Information

There are three ways to get information about the SKILL language:

■ From the CDSDoc online documentation system

■ From SourceLink® online customer support

■ From the Finder

CDSDoc Online Documentation

CDSDoc is the Cadence online product documentation system.

Documentation for each product installs automatically when you install the product. Documents are available in both HTML and PDF format.

The Library window lets you access documents by product family, product name, or type of document.

You can access CDSDoc from:

- The graphical user interface, by using the Help menu in windows and the Help button on forms

- The command line

- SourceLink online customer support (if you have a license agreement)

```
┌─────────────────────────────────────────┐
│ ─        CDSDoc: Library          ▪ □     │
│ File   Edit   View                  Help  │
│                                           │
│  Active Document Hierarchy:               │
│  USIM31  /net/twain/usr4/test/web/USIM31 ▽│
│                                           │
│  Docs by Product        ▽                 │
│  Docs by Family                        △  │
│  Docs by Product                          │
│  Docs by Type        ▽                    │
│  ─── NC-Verilog                           │
│  ─ ☐ UltraSim                             │
│     + ☐ UltraSim C-Macromodel Interface   │
│     + ☐ UltraSim Open Model Interface     │
│     + ☐ UltraSim Reliability Interface    │
│     + ☐ UltraSim Tutorial                 │
│     + ☐ UltraSim User's Manual            │
│     + ☐ UltraSimVerilog                   │
│     + ☐ UltraSimWaveform Interface        │
│  + ─ Verilog-XL Turbo                  ▽  │
│                                           │
│   Search      Open       Exit     Help    │
│                                           │
│  ┌────┬──────────────────────────────┐    │
└─────────────────────────────────────────┘
```
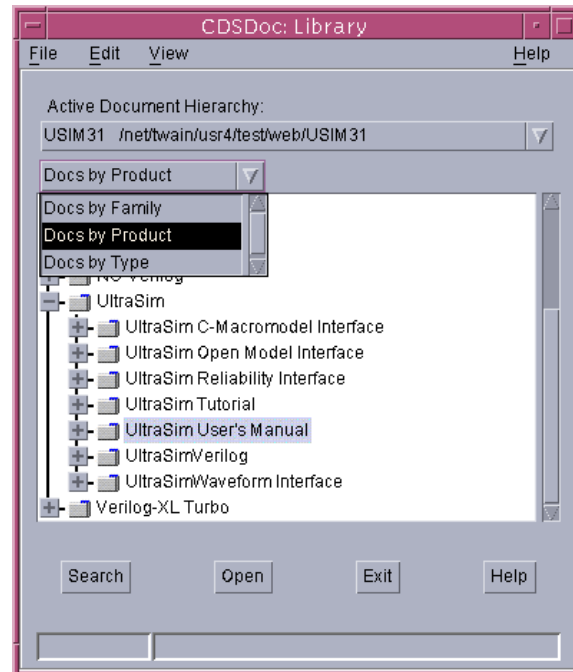
To access CDSDoc from a Cadence software application or the Command Interpreter Window (CIW), click **Help**. The document page is loaded into your browser. After the document page opens, click the **Library** button to open the CDSDoc Library window.

To access CDSDoc from the UNIX command line, enter *cdsdoc &* in a terminal window. When the Library window appears, navigate to the manual you want, then click **Open**. The manual appears in your browser.

To access CDSDoc from the Windows environment, navigate to the *<release>\tools\bin* directory and double-click **cdsdoc.exe**, or select **Start — Programs — Cadence <release> — Online Documentation**. When the Library window appears, navigate to the manual you want, then click **Open**. The manual appears in your browser.

The SKILL Documentation Set

The following documents cover the SKILL language independently of the design environment:

SKILL Language Reference
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/sklangref/sklangref06.70/sklangrefTOC.html

SKILL Language User Guide
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/sklanguser/sklanguser06.70/sklanguserTOC.html

SKILL Quick Reference Guide

These manuals cover SKILL programming interfaces to the Virtuoso Design Environment:

Cadence User Interface SKILL Functions Reference
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/skuiref/skuiref6.1/skuirefTOC.html

Virtuoso Design Environment SKILL Functions Reference
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/skdfref/skdfref6.7/skdfrefTOC.html

These manuals cover SKILL programming interfaces to design tools:

Custom Layout/Physical Verification SKILL Functions

Virtuoso Analog Design Environment

For a list of what is new, examine the following documents:

Virtuoso Design Environment Adoption: What's New – IC 6.1
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/dfiiOAWN/dfiiOAWN6.1/dfiiOAWN.pdf

Cadence User Interface SKILL: What's New in IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/skuirefWN/skuirefWN6.1/skuirefWNTOC.html

Virtuoso Design Environment SKILL: What's New, IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/skdfrefWN/skdfrefWN6.7/skdfrefWNTOC.html

Virtuoso Design Environment: What's New, IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/wincfgWN/wincfgWN6.1/wincfgWNTOC.html

Virtuoso Design Environment Known Problems and Solutions, IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/wincfgKPNS/wincfgKPNS6.1/wincfgKPNSTOC.html

Cadence User Interface on Qt Compatibility Guide, IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/skuirefCompat/skuirefCompat6.1/hiqtcompat.html#1040431

Virtuoso Parameterized Cell Known Problems and Solutions, IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/pcellKPNS/pcellKPNS6.1/pcellKPNSTOC.html

Cadence Application Infrastructure: What's New in IC 6.1:
http://sourcelink.cadence.com/docs/files/Release_Info/Docs/caiuserWN/caiuserWN6.01/caiuserWN.html#1012204

Searching Cadence Documents

In CDSDoc you can:

- Search all Cadence documents

- Search selected documents

- Search one document

- Narrow you search with a larger variety of operators

- Search with a combination of operators

- Search for special characters

About This Course   1-15

Search Operators

Type this:	Example:	Result:
myword	place	Finds place, placing, places, placed.
all words in a phrase	layer colors	Finds the phrase layer colors. Does not find layer and colors
"myword"	"place"	Finds only place
? for 1 character * for many characters	??Create*	Finds *syCreatePin*, *hiCreateForm*, *leCreateCell*, and other functions that begin with two characters followed by "Create" and another string

About This Course   1-17

Search Operators (continued)

Type this:	Example:	Result:
word AND word	printer AND CalComp	Finds only documents with both printer and CalComp
word OR word	printer OR plotter	Finds either the words printer or plotter
word NOT word	plotter NOT CalComp	Finds all documents with the word plotter that do not have the word CalComp
word <NEAR> word	place <NEAR> route	Finds place, placing, places near route, routing, routes
<CASE> word	<CASE> SUBMIT	Finds SUBMIT but not submit
word "and" word	place "and" route	Finds the phrase place and route
word, word, word	command, block	Finds documents with command and block, ranking those with both words higher than others

SourceLink Online Customer Support

SourceLink Online Customer Support

sourcelink.cadence.com

- Search the solutions database and the entire site.
- Access all documentation.
- Find answers 24x7.

If you don't find a solution on the SourceLink site...

Submit a service request online.

Online Form

From the SourceLink web site, fill out the Service Request Creation form.

Customer Support

Service Request

PCR → R&D

If you have a Cadence software support service agreement, you can get help from SourceLink online customer support.

The web site gives you access to application notes, frequently asked questions (FAQ), installation information, known problems and solutions (KPNS), product manuals, product notes, software rollup information, and solutions information.

If your problem requires more than customer support, then a product change request (PCR) is initiated.

About This Course   1-21

To view information in SourceLink:

1. Point your web browser to sourcelink.cadence.com.

2. Log in.

3. Enter search criteria.

   You can search by product, release, document type, or keyword. You can also browse by product, release, or document type.

SourceLink SKILL Information

SourceLink online customer support provides a top-level branch to specific SKILL information:

- All service request solutions that contain SKILL code

- A collection of documented example SKILL programs

The Finder

The Finder provides quick online help for the core SKILL language functions. It displays the syntax and a brief description for each SKILL function.

Example description of the *strcat* function:

```
strcat( t_string1 [t_string2 t_string3 ...] )  => t_result
Takes input strings and concatenates them.
```

You can access the Finder in three ways:

- In an xterm window, enter
    ```
    cdsFinder &
    ```

- In the SKILL Development window, click the Finder button.

- In the CIW, enter
    ```
    startFinder()
    ```

The Finder contains the same information as the SKILL *Quick Reference Guide*. You can save the online documentation to a file.

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

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

To use the Finder, follow these steps:

1. Specify a name pattern in the Search String pane.

2. Click the Search button.

3. Select a function in the Matches window pane.

    The Finder displays the abbreviated documentation for the selected function.

SKILL Finder Example

Enter partial string to match SKILL commands

Select from matches, click on columns to sort

View syntax and description of SKILL function
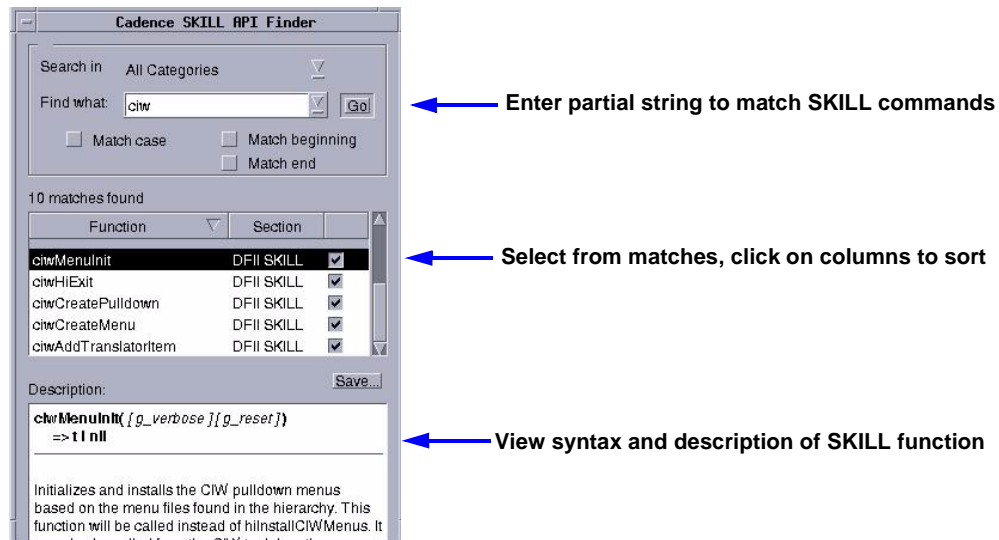
You can also click a check box next to the matched command and click the **Save** button to save the descriptions to a text file.

Search Assistant

The search assistant included with the Cadence application software has a feature to search for SKILL commands.

Select SKILL from pull-down menu

Enter partial string

Mouse-over the SKILL function

View syntax and description
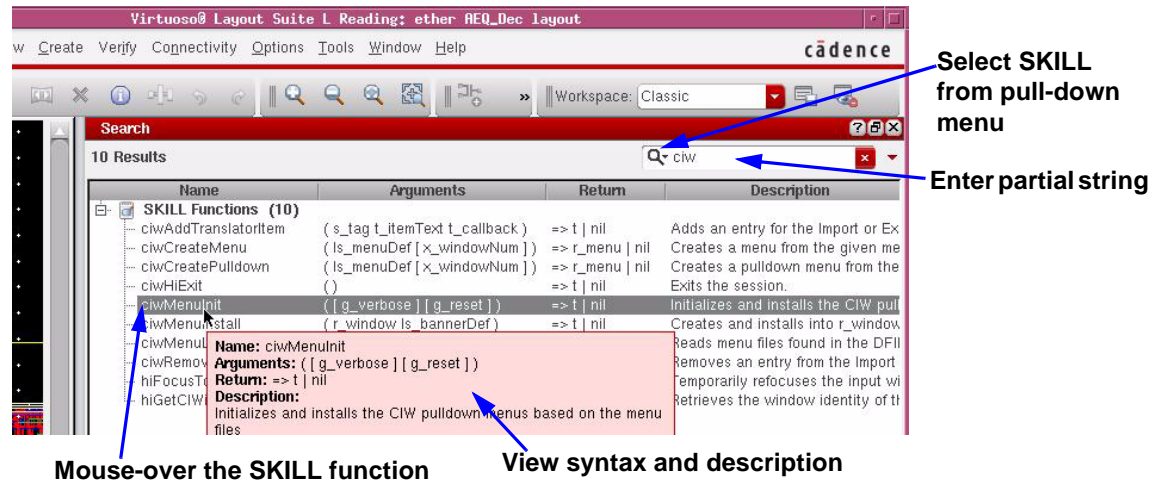
The Search assistant pane is initiated from the Windows—Assistants pull-down menu.

The Search assistant allows you to search through a variety of design information including SKILL commands.

Lab Exercises

There are two types of labs in this course—operational and programming.

■ Operational exercises

❏ Enter an example SKILL expression into the Command Interpreter Window (CIW) and observe the results. Modify the example.

❏ Examine and run source code.

Solutions to questions are usually on the next page.

■ Programming exercises

Section	Description
Requirements	Describe the functionality of the SKILL function you write.
Recommendations	Outline an approach to solving the problem.
Testing your solution	Run some tests that your SKILL function must pass.
Sample solutions	Study a solution that follows the recommendations.

Lab Overview

Lab 1-1 Locating SKILL Functions with the SKILL Finder

Lab 1-2 Locating SKILL Functions with the Search Assistant

Lab 1-3 Locating SKILL Solutions and Examples

The library definitions from the *cds.lib* file are:

```
DEFINE master  ./cell_design/master
DEFINE pCells  ./cell_design/pCells
DEFINE tutorial ./cell_design/tutorial
DEFINE basic   ${CDS_INST_DIR}/tools/dfII/etc/cdslib/basic
DEFINE cellTechLib ./cell_design/cellTechLib
DEFINE sample ${CDS_INST_DIR}/tools/dfII/samples/cdslib/sample
```

Module Summary

In this module, we covered:

- ■ Course objectives

- ■ Course agenda

- ■ Curriculum planning

- ■ Getting information about the SKILL language from the CDSDoc documentation system, SourceLink online customer support, and the Finder

- ■ The format of the lab exercises

- ■ The libraries used in the course

SKILL Programming Fundamentals

Module 2

April 30, 2007

Module Objectives

■ Start the Virtuoso® Design Environment.

■ Examine the Command Interpreter Window (CIW).

■ Describe the role of the SKILL Evaluator.

■ Examine the *CDS.log* file.

■ Summarize SKILL® syntax.

■ Display data in the CIW output pane.

■ Get the most out of SKILL error messages.

Terms and Definitions

CIW	Command Interpreter Window.
SKILL Evaluator	The SKILL Evaluator executes SKILL programs within the Virtuoso Design Environment. It compiles the program's source code before running the program.
Compiler	A compiler translates the source code into the machine language of a target machine. The compiler does not execute the program. The target machine can itself be a virtual machine.
Evaluation	Evaluation is the process whereby the SKILL Evaluator determines the value of a SKILL expression.
SKILL expression	The basic unit of source code. An invocation of a SKILL function, often by means of an operator supplying required parameters.
SKILL function	A SKILL function is a named, parameterizable body of one or more SKILL **expressions**. You can invoke any SKILL function from the CIW by using its name and providing appropriate parameters.
SKILL procedure	This term is used interchangeably with SKILL **function**.

What Is the SKILL Language?

The SKILL programming language is a high-level, interactive language.

It is the command language of the Virtuoso Design Environment.

Whenever you use forms, menus, and bindkeys, the Virtuoso Design Environment software calls SKILL functions to complete your task.

You can enter SKILL functions directly into the CIW input pane to bypass the normal user interface.

The SKILL language was developed from LISP (LISt Processing language). To learn more about the SKILL interpreter core language you can consult literature pertaining to LISP or SCHEME (a newer implementation of language very similar to LISP).

The IO used in the SKILL language resembles that of C.

What Can SKILL Functions Do?

The SKILL language acts as an extension to the Virtuoso Design Environment.

Some SKILL functions control the Virtuoso Design Environment or perform tasks in design tools. For example, SKILL functions can:

- Open a design window.

- Zoom in by 2.

- Place an instance or a rectangle in a design.

Other SKILL functions compute or retrieve data from the Virtuoso Design Environment or from designs. For example, SKILL functions can

- Retrieve the bounding box of the current window.

- Retrieve a list of all the shapes on a given layer purpose pair.

The Return Value of a SKILL 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 SKILL data can become a return value.

Starting the Virtuoso Design Environment

You can choose from two types of sessions:

- ■ Graphic

- ■ Nongraphic

You can replay a Virtuoso Design Environment session.

Session	UNIX command line	Notes
Graphic	*virtuoso* &	Use an ampersand (&)
Nongraphic	*virtuoso* -nograph	Do not use an ampersand (&)
Replay a session	*virtuoso* -replay *~/OldCDS.log* &	

Note: In the table above *virtuoso* is the name of the executable.

Graphic Sessions

In a graphic session, the Command Interpreter Window (CIW) is the first window you see.

Nongraphic Sessions

A nongraphic session is useful when you are using an ASCII terminal or modem or do not require graphics. For example, without graphics you still can open designs into virtual memory to query and update them.

After you launch a nongraphic session in an xterm window, the xterm window expects you to enter SKILL expressions.

During a nongraphic session, the Virtuoso® Design Environment suppresses any graphic output. It does not create any windows.

Replaying Sessions

When replaying a session, the Virtuoso Design Environment evaluates all the SKILL expressions contained in the *-replay* transcript file.

Initializing the Virtuoso Design Environment

During startup, the Virtuoso Design Environment initialization sequence searches the following directories for a *.cdsinit* file:

- ■ <install_dir>/tools/dfII/*local*

- ■ The current directory "."

- ■ The home directory

When the Virtuoso Design Environment finds a *.cdsinit* file, it stops searching and loads the *.cdsinit* file.

Typically, you use the *.cdsinit* file to define application bindkeys and load customer-specific SKILL utilities.

The site administrator has three ways of controlling the user customization.

Policy	Customization Strategy
The site administrator does all customization.	The <install_dir>/tools/dfII/*local/.cdsinit* contains all customization commands. There are no *./.cdsinit* or *~/.cdsinit* files involved.
The administrator does the site customization. The user can add further customization.	The <install_dir>/tools/dfII/*local/.cdsinit* file contains a command to load the *./.cdsinit* or *~/.cdsinit* files.
The user does all the customization.	The <install_dir>/tools/dfII/*local/.cdsinit* file does not exist. All customization is handled by either the *./.cdsinit* or *~/.cdsinit* files.

Consult the <install_dir>/tools/dfII/*cdsuser/.cdsinit* file for sample user customizations.

Command Interpreter Window

Output pane
Running history of commands

Window Manager Title

Menu banner

Virtuoso® 6.1.1 - Log: /hm/sdh/CDS.log

File Tools Options Help

cādence

```
t
printf("Muliple lines of input are allowed\n")
Muliple lines of input are allowed
t
```

```
printf("This area can be edited and resized\n" )
printf("Muliple lines of input are allowed\n")
```

Resizeable Input and Output panes

mouse L: mouseSingleSelectPt()   M: leHiMousePopUp()   R: printf("Muliple lines of input ar

1 >

Color-coded Syntax

Window number

Prompt pane

Mouse button cues

Input pane
SKILL functions or expressions

In a graphic session, the Command Interpreter Window (CIW) is the first window you see.

CIW Area	Description
Window manager title	The title indicates the name of the transcript log file
Window menu banner	The window banner contains several pull-down menus and a HELP button.
Output pane	This pane displays information from the session log file. You can control the kind of information the output pane displays. Colored text indicates warnings or errors.
Input pane	You can enter one or more SKILL expressions on this multi-line pane. The SKILL syntax is color-coded to assist you. When you type a carriage return in this pane, your input line is sent to the SKILL Evaluator.
Mouse button cues	When you enter data, this pane indicates the action available to you through the three mouse buttons. Other keys can also have a special meaning at this time.
Prompt pane	The pane displays the command prompt. When you enter data the prompt indicates what you need to do next.

Expandable, Multi-line Input Area

■ Set the number of input area lines in the User Preferences form -OR- graphically stretch the input window using the mouse

■ Set the number of input buffer lines in the User Preferences form. This specifies the number of previous commands that appear in the input area that may be edited.

User Preferences

Window Controls
- ☐ Place Manually   ☐ Create New Window When Descending
- ☑ Scroll Bars   ☑ Tear-Off Menus   ☑ Menu Shortcuts
- Mouse Prompts   Bottom ▾

Command Controls
- ☐ Infix (No Click is necessary for first point)
- ☐ Options Displayed When Commands Start
- Undo Limit   128 ▾   Nest Limit   5 ⬍
- Double Click Time   200
- Beep Volume   0
- Web Browser   netscape

CIW Controls
- Output History   1000   Input Buffer Lines   2
- Input Area Lines   4 ⬍   ☑ Enter Key Executes Command

OK Cancel Defaults Apply Help

icfb - Log: /home2/sdh/CDS.log.1

File Tools Options Help

```
*          90nm  Generic Standard Cell Library
*
* This library is provided as an education medium. It is
```

```
procedure( Tr61Example( a )
  let( (( b 4) c )
    c = a * b
  )
```

mouse L: M:

1 >

Use the left mouse button to select and stretch the input/output pane divider

SKILL Programming Fundamentals   2-15

Commands you can enter in the *.cdsenv* file to control the CIW are:

■ *ciwCmdHistorySize*: Specifies the number of previously executed commands that are displayed in the CIW.

■ *ciwCmdInputLines*: Specifies the initial number of lines in the input area of the CIW.

■ *ciwLogHistorySize*: Specifies the maximum number of commands in the output log.

■ *ciwCmdExecuteOnEnter*: Specifies whether the entire line is evaluated to execute a command when you press Enter, regardless of the position of the cursor.
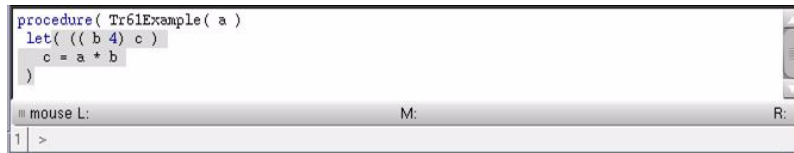
Color-coded SKILL Syntax

■ SKILL language keywords are highlighted in blue and strings in green.
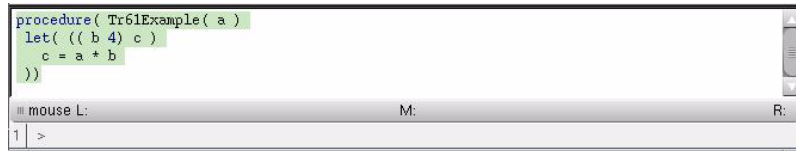
```
message = sprintf( nil "IC %2.1f has %s new features\n" 6.1 "many" )
procedure( Tr61Example( a )
  let( |
```
mouse L: M: R:
1 | >

■ Code groups between parens are highlighted in gray as you type or by clicking on a paren.

```
procedure( Tr61Example( a )
  let( (( b 4) c )
    c = a * b
  )
```
mouse L: M: R:
1 | >

■ Complete procedures are highlighted in light green as you type or by clicking on a paren.

```
procedure( Tr61Example( a )
  let( (( b 4) c )
    c = a * b
  ))
```
mouse L: M: R:
1 | >

To turn OFF auto-highlighting add the following line to your *.cdsenv* file:
```
ui ciwSyntaxHighlighting boolean nil
```

To control the highlight colors there are *.cdsenv* settings under the *ui* category as follows:
```
uiciwWarnColor string "#b87b00"
uiciwErrorColor string "dark red"
uiciwMatchParenColorstring "#dcdcdc"
uiciwMismatchParenColorstring "red"
uiciwMatchCmdColor string "#cce8c3"
```

Virtuoso Design Environment User Interface

```
Bindkey ─────────┐      ┌──► ( WAIT ) ◄──────────┐
                 │      │      │                 │
Menu Item ───────┤      │      ▼                 │
                 │      │  ┌──────────────────┐  │
Form ────────────┤      │  │ Read SKILL       │  │
                 ├──────┤  │ Expression       │  │
Loading a File ──┤      │  └──────────────────┘  │
                 │      │      │                 │
CIW Input Pane ──┤      │      ▼                 │
                 │      │  ┌──────────────────┐      Error?
Replay a Log File┤      │  │ SKILL Evaluator  │ ──────────►
                 │      │  └──────────────────┘
UNIX Process ────┘      │      │                 │
                        │      ▼                 │
                        │  ┌──────────────────┐  │
                        │  │Display Return    │──┘
                        │  │Value             │
                        │  └──────────────────┘
                        │      │
                        │      ▼
                     ┌────────┐   ┌──────────────┐
                     │CDS.log │──►│CIW Output Pane│
                     └────────┘   └──────────────┘
```
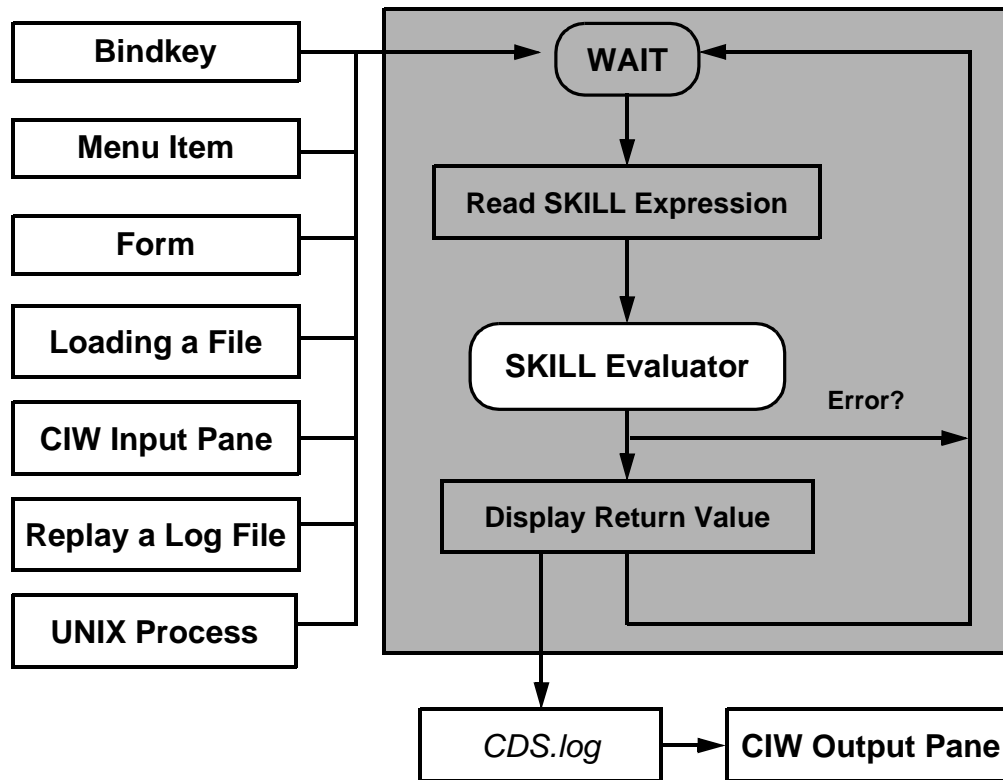
SKILL Evaluator

For each expression, the SKILL Evaluator parses it, compiles it, and then executes the compiled code. SKILL makes safety checks during each phase.

Bindkeys and Menu Items

Whenever you press a bindkey, choose a menu item, or click OK or Apply on a form, the Virtuoso Design Environment activates a SKILL function call to complete your task.

Loading SKILL Source Code

You can store SKILL code in a text file. The *load* or *loadi* function evaluates each SKILL expression in the file.

Replaying a Session File

You can replay a session file. The Virtuoso Design Environment successively sends each SKILL expression in the session file to the SKILL Evaluator. Only the input lines (prefixed by "*\i*") and the accelerated input lines (prefixed by "*\a*") are processed.

Sending a SKILL Expression from a UNIX Process

Using SKILL code, you can spawn a UNIX process that can send a SKILL expression to the SKILL Evaluator.

The *CDS.log* File

The Virtuoso Design Environment software transcribes the session in a file called *~/CDS.log*.

The log file adds a two-character tag that identifies the line.

The following text illustrates an example transcript file:

```
\p 1>                                 prompt
\i x = 0                              user type-in SKILL expression
\t 0                                  SKILL expression's return value
\p 1>                                 prompt
\i TrBump()                           user type-in SKILL expression
\o Old value: 0 New Value: 1          SKILL expression output
\t 1                                  SKILL expression's return value
\p 1>                                 Prompt
\a TrBump()                           Bindkey SKILL expression
\o Old value: 1 New Value: 2          SKILL expression output
\r 2                                  SKILL expression's return value
\i TrBump()                           user type-in SKILL expression
\o Old value: 2 New Value: 3          SKILL expression output
\t 3                                  SKILL expression's return value
```

The definition of the *TrBump* function is:

```
procedure( TrBump( )
  printf( "Old value: %d New Value: %d\n" x ++x ) x )
```

The following SKILL expression defines the *<Key>F7* bindkey for the CIW:

```
hiSetBindKey( "Command Interpreter" "<Key>F7" "TrBump()" )
```

By default mouse drag events are not logged. You can turn on logging by entering in the CIW:

```
hiLogDragEvents( t )
```

The *CDS.log* File Code

Tag	Description
\p	The prompt displayed in the CIW. This identifies the boundary between two user-level commands.
\i	A SKILL expression that the user typed into the CIW.
\o	The output, if any, that the SKILL expression generates.
\w	The warnings, if any, that the SKILL expression generates.
\e	The error, if any, that the SKILL expression caused.
\t	The return value for a SKILL expression that the user typed into the CIW.
\a	The SKILL expression activated by a bindkey or menu item.
\r	The return value for a SKILL expression activated by a bindkey or menu item.

When you replay a log file the replay function interprets each of these log file codes and passes those that represent input to the SKILL interpreter.

Setting the Log Filter

You can control the kinds of log file data that the CIW output pane displays. You can set the seven toggle options of the log filter in several ways.

■ Use the *hiSetFilterOptions* function in your *.cdsinit* file. For example, the following line sets up the most unrestrictive filter.

```
hiSetFilterOptions( t t t t t t t )
```

hiSetFilterOptions argument positions: (1) inputMenuCommands, (2) inputPrompts, (3) outputProgramResults, (4) outputMenuCommands, (5) outputUser, (6) messageErrors, (7) messageWarnings

■ From the CIW, use the **Options—Log Filter** command to display the Set Log File Display Filter form.
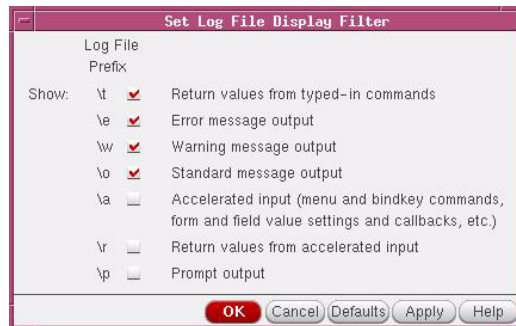
The arguments to the *hiSetFilterOptions* function correspond to the form as shown. Note the wording on the form.

Argument	Category	Toggle Option	Specific Meaning
a	Show input	Menu commands	SKILL expressions from menu commands, bindkeys, and your form interactions
p	Show input	Prompts	
o	Show output	Program results	*printf* and *println* output
r	Show output	Menu commands	Return results from bindkeys and menu commands
c	Show output	User	Return results from user type-in
e	Show messages	Errors	
w	Show messages	Warnings	

SKILL Syntax Summary

Syntax Category		Example
Comments		; remainder of the line /* several lines */
Data	integer floating point text string list boolean	5 5.3 "this is text" (1 "two" 3 4) t ;;; true nil ;;; false
Variables		line_Count1
	assignment retrieval	x = 5 x
Function call		**strcat(** "Good" " day" **)** **(strcat** "Good" " day" **)**
Operators		4 + 5 * 6 **plus(** 4 **times(** 5 6 **))**

Syntax Category	Notes
Comments	Do not use a semicolon (;) to separate expressions on a line. You will comment out the remainder of the line! Use /* .... */ to comment out one or more lines or to make a comment within a line. For example, 1+/*add*/2.
Data	Includes integer, floating-point, text string, list, and boolean data.
Variables	SKILL variables are case sensitive. The SKILL Evaluator creates a variable automatically when it first encounters it during a session. When the evaluator creates a variable, it gives the variable a special value to indicate you need to initialize the variable. The evaluator generates an error if you access an uninitialized variable.
Function	SKILL function names are case sensitive. The SKILL Evaluator allows you to specify a function call in two ways. You can put multiple function calls on a single line. Conversely, you can span a function call across multiple lines. Separate arguments with whitespace. => designates the return value of the SKILL function call.
Operators	SKILL parser rewrites operator expressions as function calls. Using operators does not affect execution speed.

Data

Each SKILL data type has an input syntax and a print representation.

- ■ You use the input syntax to specify data as an argument or to assign a value to a variable.

- ■ The SKILL Evaluator uses the print representation as the default format when displaying data, unless you specify another format.

An argument to a SKILL function usually must be a specific type. SKILL documentation designates the expected type with a single or double character prefix preceding the variable name. The letter *g* designates an unrestricted type.

The following table summarizes several common data types.

Data Type	Input Syntax	Print Representation	Type Character	Example Variable
integer	*5*	*5*	*x*	*x_count*
floating point	*5.3*	*5.3*	*f*	*f_width*
text string	*"this is text"*	*"this is text"*	*t*	*t_msg*
list	*'(1 "two" 3 4)*	*(1 "two" 3 4)*	*l*	*l_items*

When using the SKILL documentation to look up function details you see the type character used as the start of the arguments. This tells you the variable type that the argument expects.

To determine the *type* of a variable you use the type function.

The *type* function categorizes the data type of its single argument. The return value designates the data type of the argument.

Examples:
```
type( 4 ) => fixnum /* an integer */
type( 5.3 ) => flonum /* a floating point number */
type( "A SKILL program automates tasks" ) => string /* a string */
```

Variables

You do not need to declare variables in the SKILL language. The SKILL Evaluator creates a variable the first time you use it.

Variable names can contain

- Alphanumeric characters

- Underscores (_)

- Question marks

The first character of a variable cannot be a digit or a question mark. Variable names are case-sensitive.

Use the assignment operator to store a value in a variable. Enter the variable name to retrieve its value.

This example uses the *type* function to verify the data type of the current value of the variable.

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

SKILL Programming Fundamentals   2-31

Variables

The SKILL language allows both global and local variables. In Module 6, Developing a SKILL Function, see Grouping Expressions with Local Variables.

SKILL Symbols

The SKILL language uses a data type called symbol to represent both variables and functions. A SKILL symbol is a composite data structure that can simultaneously and independently hold the following:

- Data value. For example *x = 4* stores the value *4* in the symbol *x*.

- Function definition. For example, **procedure(*x(a b) a+b)*** associates a function definition with the symbol *x*. The function takes two arguments and returns their sum.

- Property list. For example, *x.raiseTime = .5* stores the name-value pair *raiseTime .5* on the property list for the symbol *x*.

You can use symbols as tags to represent one of several values. For example, *strength = 'weak* assigns the symbol as a value to the variable *strength*.

Function Calls

Function names are case sensitive.

SKILL syntax accepts function calls in three ways:

- State the function name first, followed by the arguments in a pair of matching parentheses. No spaces are allowed between the function name and the left parenthesis.

```
strcat( "A" " SKILL" " program" " automates" " tasks" )
    => "A SKILL program automates tasks"
```

- Alternatively, you can place the left parenthesis to the left of the function name.

```
( strcat "A" " SKILL" " program" " automates " " tasks" )
    => "A SKILL program automates tasks"
```

- For SKILL function calls that are not subexpressions, you can omit the outermost levels of parentheses.

```
strcat "A" " SKILL" " program" " automates " " tasks"
    => "A SKILL program automates tasks"
```

Use white space to separate function arguments.

You can use all three syntax forms together.

Multiple Lines

A literal text string cannot span multiple lines.

Function calls

■ You can span multiple lines in either the CIW or a source code file.

```
strcat(
"A" " SKILL" " program"
" automates" " tasks" ) => "A SKILL program automates tasks"
```

■ Several function calls can be on a single line. Use spaces to separate them.

```
gd = strcat("Good" " day" ) println( gd )
```

The SKILL Evaluator implicitly combines several SKILL expressions on the same line into a single SKILL expression.

■ The composite SKILL expression returns the return value of the last SKILL expression.

■ All preceding return values are ignored.

SKILL Programming Fundamentals 2-35

You can span multiple lines with a single command. You need to be careful with this ability. When you send a segment of your command to the SKILL compiler and it can be interpreted as a statement, the compiler treats it as one.

Example:

a = 2
a + 2 * (3 + a) => 12
however,
a + 2
* (3 + 2) =>
4
* error * - wrong number of arguments: mult expects 2 arguments

A text string can span multiple lines by including a \ before the return

Example:

myString = "this string spans \
two lines using a backslash at the end of the first line"

"this string spans two lines using a backslash at the end of the first line"

Understanding Function Arguments

Study the online documentation or the Cadence Finder to determine the specifics about the arguments for SKILL functions.

The documentation for each argument tells you

- The expected data type of the argument

- Whether the argument is required, optional, or a keyword argument

A single SKILL function can have all three kinds of arguments. But the majority of SKILL functions have the following type of arguments:

- Required arguments with no optional arguments

- Keyword arguments with no required and no optional arguments

The SKILL Evaluator displays an error message when you pass arguments incorrectly to a function.

To see the list of arguments for a given function use the *arglist* function.

```
arglist( 'printf )
(t_string \@optional g_general "tg")
```

SKILL Programming Fundamentals   2-37

Required Arguments

You must provide each required argument in the prescribed order when you call the function.

Optional Arguments

You do not have to provide the optional arguments. Each optional argument has a default value. If you provide an optional argument, you must provide all the preceding optional arguments in order.

```
view( t_file [g_boxSpec][g_title][g_autoUpdate ][l_iconPosition] )
```

Keyword Arguments

When you provide a keyword argument you must preface it with the name of the formal argument. You can provide keyword arguments in any order.

```
geOpen(
?window w_windowId
?lib t_lib
?cell t_cell
?view t_view
?viewType t_viewType
?mode t_mode )
=> t / nil
```

Operators

The SKILL language provides operators that simplify writing expressions. Compare the following two equivalent SKILL expressions.

```
( 3**2 + 4**2 ) **.5 => 5.0
expt( plus( expt( 3 2 ) expt( 4 2 )) .5 ) => 5.0
```

Use a single pair of parentheses to control the order of evaluation as this nongraphic session transcript shows.

```
> 3+4*5
23
> (3+4)*5
35
>  x=5*6
30
> x
30
> (x=5)*6
30
> x
5
```

However, when you use extra parentheses, they cause an error.

```
((3+4))*5
*Error* eval: not a function - (3 + 4)
```

Operator Precedence

In general, evaluation proceeds left to right. Operators are ranked according to their relative precedence. The precedence of the operators in a SKILL expression determine the order of evaluation of the subexpressions.

Each operator corresponds to a SKILL function.

Operator	Function	Use
++a a++	preincrement postincrement	Arithmetic
*a**b*	expt	Arithmetic
*a*b a/b*	times quotient	Arithmetic
a+b a-b	plus difference	Arithmetic
a==b a!=b	equal nequal	Tests for equality and inequality.
a=b	setq	Assignment

For more information, check the online documentation and search for preincrement.

Tracing Operator Evaluation

To observe evaluation, turn on SKILL tracing before executing an expression to observe evaluation. The arrow (-->) indicates return value.

Notice that the trace output refers to the function of the operator.

```
> tracef(t)
t
> (3+4)*5
|(3 + 4)
|plus --> 7
|(7 * 5)
|times --> 35
35
>
```

To turn off tracing you use:

```
untrace()
```

Note: *tracef(nil)* will not turn tracing off

Tracing Operator Evaluation

The trace function shows the order and intermediate results of operator evaluation.

SKILL Output	Explanation
>tracef(t)	The user executes the trace function to turn on tracing.
t	The SKILL Evaluator acknowledges successful completion of the function.
(3+4)*5	The user enters an expression for evaluation.
|(3 + 4)	The SKILL Evaluator begins evaluation starting from left to right.
|plus --> 7	The SKILL Evaluator executes the "plus" function resulting in 7.
|(7 * 5)	The 7 is returned to the original expression replacing (3 + 4)
|times --> 35	The "times" function is executed resulting in 35.
35	The result of the expression evaluation is returned.
>	The SKILL Evaluator is listening for the next command.

Lab Overview

Lab 2-1 Using the Command Interpreter Window

Lab 2-2 Exploring SKILL Numeric Data Types

Lab 2-3 Exploring SKILL Variables

SKILL Programming Fundamentals   2-43

Displaying Data in the CIW

Every SKILL data type has a default display format that is called the print representation.

Data Type	Print Representation
integer	5
floating point	1.3
text string	"SKILL Programming"
list	(1 2 3)

The SKILL Evaluator displays a return value with its print representation.

SKILL functions often display data before they return a value.

Both the *print* and *println* functions use the print representation to display data in the CIW output pane. The *println* function sends a newline character.

The *print* and *println* Functions

Both the *print* and *println* functions return *nil*.

This nongraphic session transcript illustrates *println*.

```
> x = 8
8
> println( x )
8
nil
>
```

This nongraphic session transcript shows an attempt to use the *println* function to print out an intermediate value *3+4* during the evaluation of *(3+4)*5*. The *println*'s return value of *nil* causes the error.

```
> println(3+4)*5
7
*Error* times: can't handle (nil * 5)
>
```

Displaying Data with Format Control

The *printf* functions writes formatted output to the CIW. This example displays a line in a report.

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

The first argument is a conversion control string containing directives.

```
%[-][width][.precision]conversion_code
    [-] = left justify
    [width] = minimum number of character positions
    [.precision] = number of characters after the decimal
    conversion_code
      d - digit(integer)
      f - floating point
      s - string or symbol
      c - character
      n - numeric
      L - default format
```

The *%L* directive specifies the default format. Use the print representation for each type to display the value.

The *printf* Function

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

```
> printf( "%d %d" 5 6.3 )
*Error* fprintf/sprintf: format spec. incompatible with data - 6.3
>
```

The %L Directive

The *%L* directive specifies the print representation. This directive is a very convenient way to intersperse application specific formats with default formats. Remember that *printf*
returns *t*.

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

Solving Common Problems

These are common problems you might encounter:

- ■ The CIW does not respond.

- ■ The CIW displays inexplicable error messages.

- ■ You pass arguments incorrectly to a function.

SKILL Programming Fundamentals

2-49

What If the CIW Doesn't Respond?

Situation:

- ■ You typed in a SKILL function.

- ■ You pressed Return.

- ■ Nothing happens.

You have one of these problems:

- ■ Unbalanced parentheses

- ■ Unbalanced string quotes

Solution:

The following steps trigger a system response in most cases.

- ■ You might have entered more left parentheses than right parentheses.

- ■ Enter a **]** character (a closing right square bracket). This character closes all outstanding right parentheses.

- ■ If still nothing happens, enter the **"** character followed by the **]** character.

- ■ **Control-c** cancels the current command and resets the SKILL Evaluator.

SKILL Programming Fundamentals   2-51

White Space Sometimes Causes Errors

White space can cause error messages.

Do **not** put any white space between the function name and the left parenthesis.

The error messages:

■ Do not identify the white space as the cause of the problem.

■ Vary depending on the surrounding context.

Examples

A SKILL function to concatenate several strings

```
strcat ( "A" "SKILL" " program" " automates" " tasks" )
*** Error in routine eval:
Message: *Error* eval: illegal function A
```

An assignment to a variable

```
greeting = strcat ( "work" " smarter" )
*** Error in routine eval:
Message: *Error* eval: unbound variable strcat
```

SKILL Programming Fundamentals   2-53

Passing Incorrect Arguments to a Function

All built-in SKILL functions validate the arguments you pass. You must pass the appropriate number of arguments in the correct sequence. Each argument must have the correct data type.

If there is a mismatch between what the caller of the built-in function provides and what the function expects, the SKILL Evaluator displays an error message.

Examples

The *strcat* function does not accept numeric data.

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

The *type template* mentioned in the error message encodes the expected argument types.

The *strlen* function expects at least one argument.

```
strlen()
*Error* strlen: too few arguments (1 expected, 0 given) - nil
```

SKILL Programming Fundamentals   2-55

Use the Cadence Finder to verify the following information for a SKILL function:

- The number of arguments that the function expects.

- The expected data type of each argument.

The following table summarizes some of the characters in the type template which indicate the expected data type of the arguments. The Cadence Finder and the Cadence online SKILL documentation follow the same convention.

Character in Type Template	Expected Data Type
x	integer
f	floating point
s	variable
t	text string
S	variable or text string
g	general

Lab Overview

Lab 2-4 Displaying Data in the CIW

Lab 2-5 Solving Common Input Errors

Module Summary

This module

- Introduced the SKILL language, the command language for the Virtuoso Design Environment

- Defined what user interface action sends SKILL expressions to the SKILL Evaluator

- Explored SKILL data, function calls, variables, and operators

- Showed you ways to solve common problems

Lists

Module 3

April 30, 2007

Module Objectives

■ Build lists.

■ Retrieve list elements.

■ Use lists to represent points and bounding boxes.

What Is a SKILL List?

A SKILL® list is an ordered collection of SKILL data objects.

The elements of a list can be of any data type, including variables and other lists.

The special data item *nil* represents the empty list.

SKILL functions commonly return lists you can display or process.

Design database list examples:

- Shapes in a design

- Points in a path or polygon

- Instances in a design

User interface list examples:

- Virtuoso® Design Environment windows currently open

- Pull-down menus in a window

- Menu items in a menu

You can use a list to represent many different types of objects. The arbitrary meaning of a list is inherent in the programs that manipulate it.

Object	Example list representation
Two-dimensional Point	List of two numbers. Each point is a sublist of two numbers.
Bounding Box	List of two points. Each point is a sublist of two numbers.
Path	List of all the points. Each point is a sublist of two numbers.
Circle	List of radius and center. The center is a sublist of two numbers.

How the SKILL Evaluator Displays a List

To display a list, the SKILL Evaluator surrounds the elements of the list with parentheses.

```
( "rect" "polygon" "rect" "line" )
( 1 2 3 )
( 1 ( 2 3 ) 4 5 ( 6 7 ))
( ( "one" 1 ) ( "two" 2 ) )
( "one" one )
```

To display a list as a return value, the SKILL Evaluator splits the list across multiple lines when the list:

■ Contains sublists

■ Has more than _itemsperline_ number of items

Use the *printf* or *println* functions to display a list. SKILL displays the output on a single line.

Consider the examples shown below based on these assignments. The output is taken from a nongraphical session.

```
aList = '( 1 2 3 )
aLongList = '( 1 2 3 4 5 6 7 8 )
aNestedList = '( 1 ( 2 3 ) 4 5 ( 6 7 ))
```

Lists   3-7

```
> aList = '( 1 2 3 )
(1 2 3)
> println( aList )
(1 2 3)
nil
> printf( "This is a list: %L\n" aList )
This is a list: (1 2 3)
t
> aLongList = '( 1 2 3 4 5 6 7 8 )
(1 2 3 4 5
    6 7 8
)
> println( aLongList )
(1 2 3 4 5 6 7 8)
nil
> printf( "This is a list: %L\n"  aLongList )
This is a list: (1 2 3 4 5 6 7 8)
t
>aNestedList = '( 1 ( 2 3 ) 4 5 ( 6 7 ))
(1
  (2 3) 4 5
  (6 7)
)
> println( aNestedList )
(1 (2 3) 4 5 (6 7))
nil
> printf( "This is a list: %L\n" aNestedList )
This is a list: (1 (2 3) 4 5 (6 7))
t
```

Creating New Lists

There are several ways to build a list of elements. Two straightforward ways are to do the following:

■ Specify all the elements literally. Apply the ' operator to the list.

Expressions	Return Result
'(1 2 3)	(1 2 3)
'("one" 1)	("one" 1)
'(("one" 1) ("two" 2))	(("one" 1) ("two" 2))

■ Make a list by computing each element from an expression. Pass the expressions to the *list* function.

Expressions	Return Result
a = 1	1
b = 2	2
list(a b 3)	(1 2 3)
list(a**2+b**2 a**2-b**2)	(5 -3)

Lists

Use Variables

Store the new list in a variable. Otherwise, you cannot refer to the list again.

The ' Operator

Follow these guidelines when using the ' operator to build a list:

■ Include sublists as elements with a single set of parentheses.

■ Do not use the ' operator in front of the sublists.

■ Separate successive sublists with white space.

The *list* Function

The SKILL Evaluator normally evaluates all the arguments to a function before invoking the function. The function receives the evaluated arguments. The *list* function allocates a list in virtual memory from its evaluated arguments.

Adding Elements to an Existing List

Here are two ways to add one or more elements to an existing list:

■ Use the *cons* function to add an element to an existing list.

Expressions	Return Result
result = '(2 3)	*(2 3)*
*result = **cons(** 1 result **)***	*(1 2 3)*

■ Use the *append* function to merge two lists together.

Expressions	Return Result
oneList = '(4 5 6)	*(4 5 6)*
aList = '(1 2 3)	*(1 2 3)*
*bList = **append(** oneList aList **)***	*(4 5 6 1 2 3)*

The *cons* Function

The construct (*cons*) function adds an element to the beginning of an existing list. This function takes two arguments. The first is the new element to be added. The second is the list to add the element to. The result of this function's execution is a list containing one more element than the input list.

Store the return result from *cons* in a variable. Otherwise, you cannot refer to the list subsequently. It is common to store the result back into the variable containing the target list.

The *append* Function

The *append* function builds a new list from two existing lists. The function takes two arguments. The first argument is a list of the elements to begin the new list. The second argument is a list of the elements to complete the new list.

Store the return result from *append* in a variable. Otherwise, you cannot refer to the list subsequently.

Points of Confusion

People often think that *nil, cons*, and the *append* functions violate common sense. Here are some frequently asked questions.

Question	Answer
What is the difference between *nil* and *'(nil)*?	*nil* is a list containing nothing. Its length is *0*. *'(nil)* builds a list containing the single element *nil*. The length is *1*.
How can I add an element to the end of a list?	Use the *append* and *list* functions. *aList = '(1 2)* *aList = **append(aList list(3)) => (1 2 3)***
Can I reverse the order of the arguments to the *cons* function? Will the results be the same?	You either get different results or an error. **cons('(1 2) '(3 4)) => ((1 2) 3 4)** **cons('(3 4) '(1 2)) => ((3 4) 1 2)** **cons(3 '(1 2)) => (3 1 2)** **cons('(1 2) 3) =>** *** Error in routine cons *Error* cons: argument #2 should be a list
What is the difference between *cons* and *append*?	**cons('(1 2) '(3 4)) => ((1 2) 3 4)** **append('(1 2) '(3 4)) => (1 2 3 4)**

Lists

Question	Answer
What is the difference between *nil* and *'(nil)*?	*nil* is a list containing nothing. Its length is *0*. *'(nil)* builds a list containing a single element. The length is 1.
How can I add an element to the end of a list?	Use the *list* function to build a list containing the individual elements. Use the append function to merge it to the first list. There are more efficient ways to add an element to the end of a list. They are beyond the scope of this course.
Can I reverse the order of the arguments to the *cons* function? Will the results be the same?	Common sense suggests that simply reversing the elements to the *cons* function will put the element on the end of the list. This is not the case.
What is 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 resulting list is the sum of the lengths of the two argument lists.

Working with Existing Lists

Task	Function	Example	Return Result
Retrieve the first element of a list	car	numbers = '(1 2 3) **car(** numbers **)**	(1 2 3) 1
Retrieve the tail of the list	cdr	**cdr(** numbers **)**	(2 3)
Retrieve an element given an index	nth	**nth(** 1 numbers **)**	2
Tell if a given data object is in a list	member	**member(** 4 numbers **)** **member(** 2 numbers **)**	nil (2 3)
Count the elements in a list	length	**length(** numbers **)**	3
Apply a filter to a list	setof	**setof(** x '(1 2 3 4) **oddp(** x **))**	(1 3)

The *nth* Function

Lists in the SKILL language are numbered from 0. The 0 element of a list is the first element, the 1 element of a list is the second element and so on.

The *member* Function

The member function returns the tail of the list starting at the element sought or nil, if the element is not found. Remember, if it is not nil - it is true.

The *setof* Function

The *setof* function makes a new list by copying only those top-level elements in a list that pass a test. You must write the test in terms of a single variable. The first parameter to the *setof* function identifies the variable you are using in the test.

■ The first argument is the variable that stands for an element of the list.

■ The second argument is the list.

■ The third argument is one or more expressions that you write in terms of the variable. The final expression is the test. It determines if the element is included in the new list.

Traversing a list with *car* and *cdr*

A list can be conceptually represented as an inverted tree. To traverse a left branch use **car** and to traverse a right branch use **cdr**.

1: **alist = '(1 (2 3))**

2: **car(alist) => 1**

3: **cdr(alist) => ((2 3))**

4: **car(cdr(alist)) = (2 3)**

4a: **cadr(alist) = (2 3)**

5: **car(car(cdr(alist))) => 2**

5a: **caadr(alist) => 2**

6: **car(cdr(car(cdr(alist)))) => 3**

6a: **cadadr(alist) => 3**

7: **cdr(cdr(alist)) => nil**

'(1 (2 3))
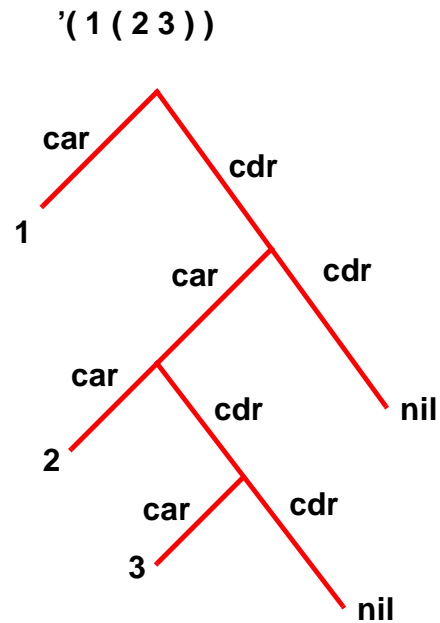
You can use *car* and *cdr* to traverse a list. You can draw a tree as shown to layout your list structure. Make each sub-list a sub-tree. Label each left branch with a *car* and each right branch with a *cdr*. From the item you wish to locate in the list follow the tree back to the root and construct your nested *car* and *cdr* function calls.

You can abbreviate your nested car and cdr function calls by using just the "a" for *car* and "d" *for* cdr to create a function call. For example, you can abbreviate *car(cdr (alist))* as *cadr(alist)*.

The SKILL language provides a family of functions that combine *car* and *cdr* operations. You can use these functions on any list. Bounding boxes provide a good example of working with the *car* and *cdr* functions.

Functions	Combination	Bounding box examples	Expression
car	**car(...)**	lower left corner	ll = **car(bBox)**
cadr	**car(cdr(...))**	upper right corner	ur = **cadr(bBox)**
caar	**car(car(...))**	x-coord of lower left corner	llx = **caar(bBox)**
cadar	**car(cdr(car(...))**	y-coord of lower left corner	lly = **cadar(bBox)**
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)**

Frequently Asked Questions

Students often ask these questions:

- Why are such critical functions as *car* and *cdr* called such weird names?

- What is the purpose of the *car* and *cdr* functions?

- Can the *member* function search all levels in a hierarchical list?

- How does the *setof* function work? What is the variable *x* for?

| Questions	Answers
Why are such critical functions as *car* and *cdr* called such weird names?	*car* and *cdr* were machine language instructions on the first machine to run Lisp. *car* stands for *contents of the address register* and *cdr* stands for *contents of the decrement register*.
What is the purpose of the *car* and *cdr* functions?	Lists are stored internally as a series of doublets. The first element is the list entry, the second element of the doublet is a pointer to the rest of the list. The *car* function returns the first element of the doublet, the *cdr* function returns the second. For any list *L* it is true that ***cons(car(L) cdr(L))*** builds a list equal to *L*. This relates the three functions *cons*, *car*, and *cdr*.
Can the *member* function search all levels in a hierarchical list?	No. 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.
How does the *setof* function work? What is the variable *x* for?	The *setof* function makes a new list by copying only those top-level elements in a list that pass a test. The test must be written in terms of a single variable. The first parameter to the *setof* function identifies the variable you are using in the test.

Two-Dimensional Points

The SKILL language represents a two-dimensional point as a two-element list.

The binary operator (:) builds a point from an x-value and a y-value.

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

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

```
xCoord( P ) => 300
yCoord( P ) => 400
```
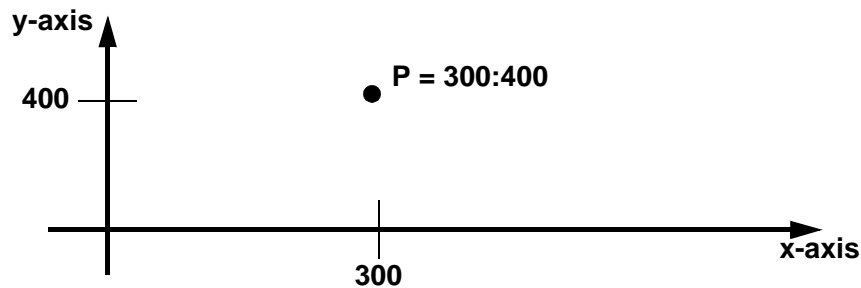
The : Operator

You can use the ' operator or list function to build a coordinate.

```
P = '( 3.0 5.0 )
P = list( xValue yValue )
```

The : operator expects both of its arguments to be numeric. The *range* function implements the : operator.

```
> "hello":3
*Error* range: argument #1 should be a number
(type template = "n") - "hello"
```

The *xCoord* and *yCoord* Functions

Alternatively, you can use the *car* function to access the x-coordinate and *car(cdr (...))* to access the y-coordinate.

```
xValue = car( P )
yValue = car( cdr( P ))
```

Computing Points

The : operator combines naturally with arithmetic operators. It has a lower precedence than the + or the * operator.

```
3+4*5:4+7*8 => (23 60)
```

Computing a point from another point is easy.

For example, given a point *P*, apply an offset *dx* and *dy* in both directions.
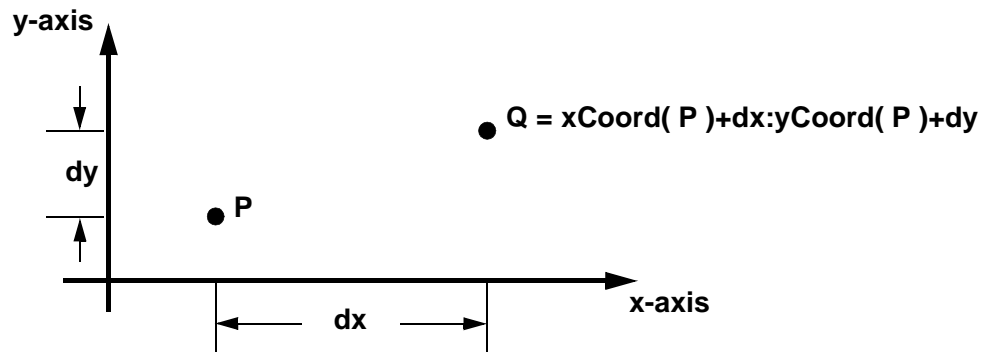
Bounding Boxes

The SKILL language resents a bounding box as a two-element list. The first element is the lower-left corner and the second element is the upper-right corner.

y-axis

ur = 500:450

ll = 300:400

x-axis
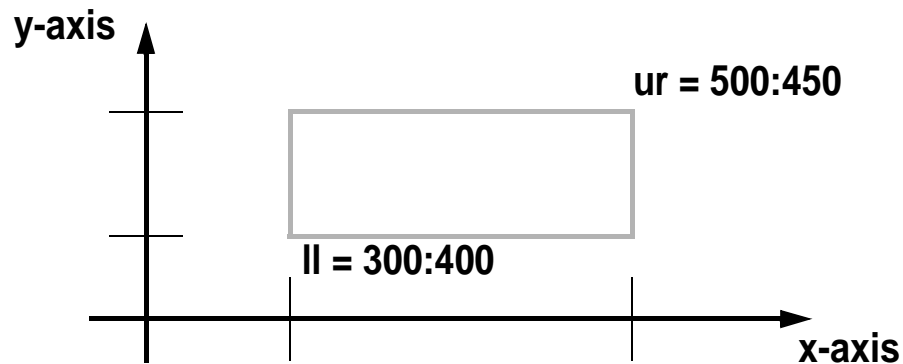
This is returned to you by the system as: ((300 400) (500 450))

Lists 3-25

Remember that : is a point operator.

'(300:400 500:450) does not create a list of two lists since the : operator is not evaluated.

Use list(300:400 500:450) which will evaluate the point and thus create a list containing two lists and so, in this case, also a bounding box.

Creating a Bounding Box

Use the *list* function to build a bounding box. Specify the points by variables or by using the : operator.

```
ll = 300:400 ur = 500:450
bBox = list( ll ur ) =>
        (( 300 400 ) ( 500 450 ))

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

When you create a bounding box, put the points in the correct order. When SKILL prompts the user to digitize a bounding box, it returns the bounding box with the lower-left and upper-right corner points correctly ordered, *even though the user may have digitized the upper-left and lower-right corners!*

You may use the ' operator to build the bounding box ONLY if you specify the coordinates as literal lists.

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

Retrieving Elements from Bounding Boxes

Use the *lowerLeft* and *upperRight* functions to retrieve the lower-left corner and the upper-right corner points of a bounding box.

```
lowerLeft( bBox ) => ( 300 400 )
upperRight( bBox ) => ( 500 450 )
```

These functions assume that the order of the elements is correct.

Use the *xCoord* and *yCoord* functions to retrieve the coordinates of these corners.

```
xCoord( lowerLeft( bBox )) => 300
yCoord( upperRight( bBox )) => 450
```

Lists

Lecture Exercises

The exercises on the next few pages illustrate techniques for manipulating bounding boxes:

- Offsetting a box

- Finding the smallest bounding box

- Finding the intersection of two bounding boxes

Offsetting a Box

Assume the variable *Box* contains the lower-left bounding box. The upper-right box is the same size as the lower-left box.

Using variables, write an expression for the upper-right bounding box.
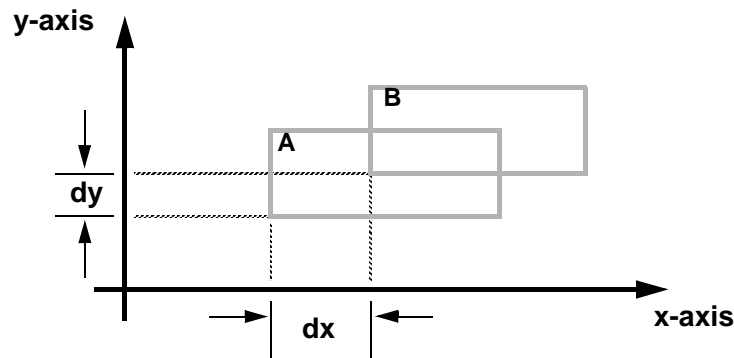
Use this template as a starting point for writing the expression.

```
boxLL = ... : ...
boxUR = ... : ...
list( boxLL boxUR )
```

To view the solution, turn the page upside down.

```
boxLLx = xCoord( lowerLeft( box ) )+dx
boxLLy = yCoord( lowerLeft( box ) )+dy
boxURx = xCoord( upperRight( box ) )+dx
boxURy = yCoord( upperRight( box ) )+dy
list(
    boxLLx:boxLLy
    boxURx:boxURy)
```

Finding the Smallest Bounding Box

Write expressions that compute the smallest bounding box containing two boxes *A* and *B*.
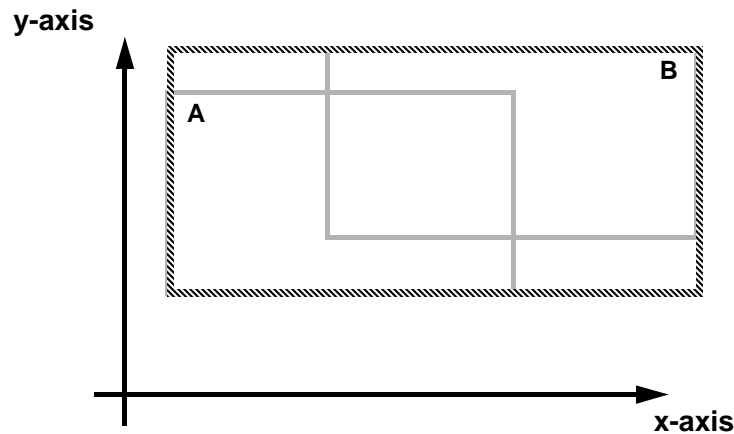
y-axis

A

B

x-axis

Use the *xCoord*, *yCoord*, *lowerLeft*, and *upperRight* functions in the following template.

```
boxLL = min( ... ... ):min( ... ... )
boxUR = max( ... ... ):max( ... ... )
list( boxLL boxUR )
```

Lists   3-35

To view the solution, turn the page upside down.

```
llAx = xCoord( lowerLeft( A ))
llBx = xCoord( lowerLeft( B ))
llAy = yCoord( lowerLeft( A ))
llBy = yCoord( lowerLeft( B ))
urAx = xCoord( upperRight( A ))
urBx = xCoord( upperRight( B ))
urAy = yCoord( upperRight( A ))
urBy = yCoord( upperRight( B ))
boxLL = min( llAx llBx ):min( llAy llBy )
boxUR = max( urAx urBx ):max( urAy urBy )
list( boxLL boxUR )
```

Finding the Intersection of Two Bounding Boxes

Write an expression that describes the overlap between two boxes A and B.
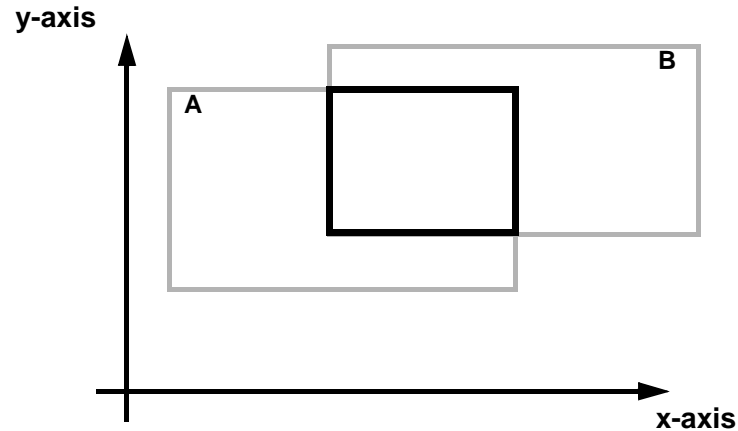
y-axis

B

A

x-axis

Use the *xCoord*, *yCoord*, *lowerLeft*, and *upperRight* functions in the following template.

```
boxLL = max( ... ... ):max( ... ... )
boxUR = min( ... ... ):min( ... ... )
list( boxLL boxUR )
```

To view the solution, turn the page upside down.

```
llAx = xCoord( lowerLeft( A ))
llBx = xCoord( lowerLeft( B ))
llAy = yCoord( lowerLeft( A ))
llBy = yCoord( lowerLeft( B ))
urAx = xCoord( upperRight( A ))
urBx = xCoord( upperRight( B ))
urAy = yCoord( upperRight( A ))
urBy = yCoord( upperRight( B ))
boxLL = max( llAx llBx ):max( llAy llBy )
boxUR = min( urAx urBx ):min( urAy urBy )
list( boxLL boxUR )
```

Combinations of *car* and *cdr* Functions

The SKILL language provides a family of functions that combine *car* and *cdr* operations. You can use these functions on any list.

Bounding boxes provide a good example of working with the *car* and *cdr* functions.

Functions	Combination	Bounding box examples	Expression
car	**car(...)**	lower left corner	ll = **car(** bBox **)**
cadr	**car(cdr(...))**	upper right corner	ur = **cadr(** bBox **)**
caar	**car(car(...))**	x-coord of lower left corner	llx = **caar(** bBox **)**
cadar	**car(cdr(car(...))**	y-coord of lower left corner	lly = **cadar(** bBox **)**
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 **)**

Using the *xCoord*, *yCoord*, *lowerLeft* and *upperRight* functions is preferable in practice to access coordinates, bounding boxes, and paths.

The functions *cadr*, *caar*, and so forth are built in for your convenience.
Any combination of four a's or d's.

Lab Overview

Lab 3-1 Creating New Lists

Lab 3-2 Extracting Items from Lists

Module Summary

In this module we covered:

- ■ SKILL lists can contain any type of SKILL data. *nil* is the empty list.

- ■ Using the *'* operator and the *list* function to build lists.

- ■ Using the *cons* and *append* functions to build lists from existing lists.

- ■ Using the *length* function to count the number of elements in a list.

- ■ Using the *member* function to find an element in an existing list.

- ■ Using the *setof* function to filter a list according to a condition.

- ■ How two-dimensional points are represented by two element lists.

- ■ How bounding boxes are also represented by two element lists.

Windows

Module 4

April 30, 2007

Module Objectives

■ Understand the window ID data type.

■ Open design windows.

■ Examine session windows

■ Define application bindkeys.

■ Open read-only text windows.

■ Manage windows.

Terms and Definitions

Callback	A callback is a SKILL® expression that the software sends to the SKILL Evaluator in response to a keyboard or mouse event. To retrieve the callback, the software notes where the cursor is when the keyboard event happens.
Application type	Each Virtuoso® Design Environment window has an application type. The system uses this type to determine the table of **bindkey** definitions.
Bindkey	A bindkey associates a SKILL expression with a key or mouse button.

Virtuoso Design Environment Windows

Most Virtuoso® Design Environment windows have a window number.

■ CIW

■ Application windows

■ The Layer Selection Window (LSW) is not one of these windows.

The *window* function converts a window number to a window ID.

```
window( 3 ) => window:3
```

The data type *wtype* represents the underlying data structure for a window.

```
type( window( 3 ) ) => wtype
```

To make a window current, the user puts the mouse cursor in the window and presses either a mouse button or a key.

The CIW is never the current window.

SKILL® functions act on the current window by default.

The *hiGetWindowList* Function

The *hiGetWindowList* function returns a list of the window IDs of the existing Virtuoso Design Environment windows.

```
hiGetWindowList() => ( window:1 window:2 )
```

The *hiGetCurrentWindow* Function

The *hiGetCurrentWindow* function identifies the current window. If there is no current window, the function returns *nil*.

```
hiGetCurrentWindow() => window:4
```

Session Windows

A session window, also know as a *swindow*:

■ Is a container for design windows which are organized using tabs

■ Provides for dockable assistant windows and toolbars along the edges

■ Has a menu bar that can be docked or floated
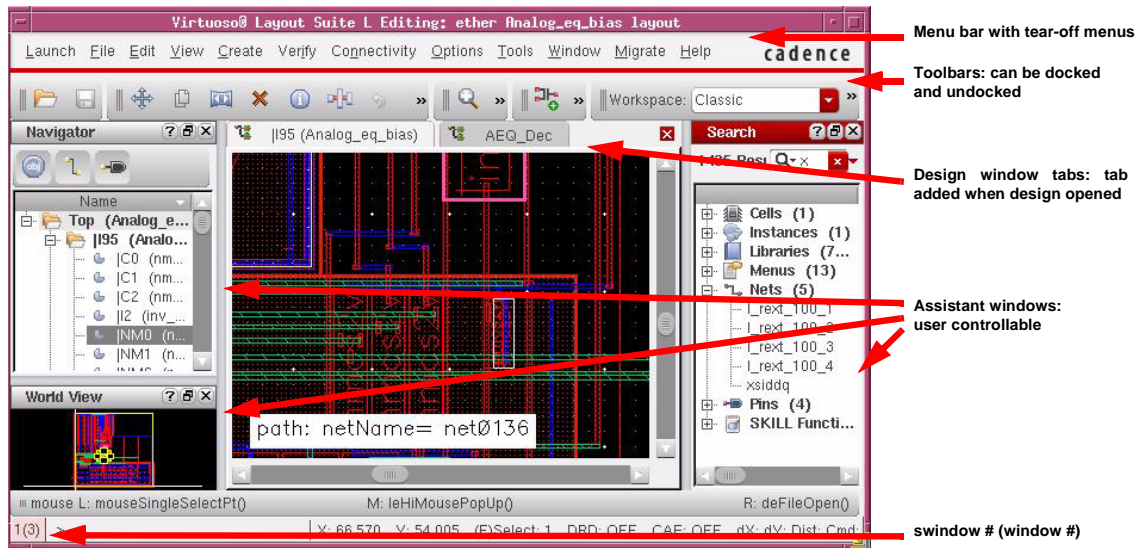
Windows 4-7

Opening a new design window:
```
geOpen(...)
hiOpenWindow(...)
```

To locate a session window you provide the design window ID. The *swindow* command looks up the session window that contains the window ID.
```
swindow( window(#) )
```

Opening a Design Window

Use the *geOpen* function to open a design in a window.

The *geOpen* function

■ Loads the design into virtual memory.

■ Creates a window displaying the design.

■ Returns the window ID.

The following expression opens the *master mux2 schematic* view for editing.

```
geOpen(
    ?lib        "master"
    ?cell       "mux2"
    ?view       "schematic"
    ?mode       "w"
    ) => window:12
```

The following expression displays the Open File form with the Library Name field set to *master*.

```
geOpen( ?lib "master" )
```

The *geOpen* Function

Keyword	Example Parameter	Meaning
?lib	*"master"*	the library name
?cell	*"mux2"*	the cell name
?view	*"schematic"*	the view name
?mode	*"r", "a" or "w"*	read, append, overwrite

The *geOpen* function requires keyword parameter passing. Precede each argument with the corresponding keyword. Keywords correspond to the formal arguments.

Prompt Line

By default every graphic window has a prompt line. This line tells the user what the command requires next. You can remove the prompt line for the current window with the command:

```
hiRemovePromptLine()
```

or optionally specify a window ID as an input parameter to the command.

Using Bindkeys

Bindkeys make frequently used commands easier for the user to execute.

There are several different uses for bindkeys:

- ■ To initiate a command, such as the Zoom In command.

- ■ To use the mouse during a command to enter points.

- ■ To perform subsidiary actions, such as panning the window, during a command.

A bindkey associates a SKILL expression with a key or mouse button.

When the mouse cursor is in an application window, and the user presses a key or mouse button, the SKILL Evaluator evaluates the bindkey expression.

Each application can associate different SKILL expressions with the same key or mouse button.

While the user digitizes points during a command, a key or mouse button can trigger a different SKILL expression than it normally does.

This example illustrates the different uses of bindkeys.

1. With the mouse cursor in a Layout Editor window, the user presses the **z** key.

   The Zoom In command starts.

2. The user clicks the left mouse button to indicate the first corner of the region to zoom.

3. The user presses the **Tab** key.

   The Pan command starts.

4. The user clicks the left mouse button to indicate the center point of the pan command.

   The Pan command finishes.

5. Finally, to indicate the second corner of the region, the user clicks the left mouse button again.

   The Zoom In command finishes.

Defining Bindkeys

When you define a bindkey, you specify the following information:

- The application type, which identifies the application by means of a text string. Typical application types include the following:
    - *"Command Interpreter"*
    - *"Layout"*
    - *"Schematics"*
    - *"Graphics Browser"*

- The keyboard or mouse event that triggers the SKILL expression. Typical events include the following:
    - Press the **a** key.
    - Press the left mouse button.
    - Draw through with the left mouse button.

- The mode that governs the bindkey. The bindkey is either modeless or is in effect only when the user enters points.

- The SKILL expression that the bindkey triggers.

The *hiGetAppType* Function

Use the *hiGetAppType* function to determine the appropriate application type.

```
hiGetAppType( window( 1 ) ) =>
    "Command Interpreter"
```

The *hiSetBindKey* Function

Use the *hiSetBindKey* function to define a single bindkey.

```
hiSetBindKey( "Schematics"
  "Shift Ctrl<Btn2Down>(2)"
  "hiRaiseWindow( window(1))"
)
```

Use the curly braces, { }, to group several SKILL expressions together when defining the callback (the third argument in *hiSetBindKey*).

Describing Events

To determine the syntax to describe an event, do one of the following:

■ Study the Cadence® online documentation.

■ Display the bindkeys for an application that uses the event.

Examples

Event Description	Event Syntax
The user pressed the **a** key.	*"<Key>a"*
The user clicked the left mouse button.	*"<Btn1Down>"*
The user draws through an area with the left mouse button.	*"<DrawThru1>"*
While holding down Shift and Control keys, the user double clicked the middle mouse button.	*"Shift Ctrl<Btn2Down>(2)"*

To limit the event to entering points, append EF to the event syntax:

```
"<Btn1Down>EF"
```

Windows   4-15

Modes

If *t_key* ends with "EF", you use the SKILL command in enterfunction mode. Otherwise, it is a non-enterfunction mode command. If there is no enterfunction mode command defined when a key or mouse event happens in enterfunction mode, the system uses the non-enterfunction mode command for this key.

Note that even an empty string is a valid bindkey command. Therefore, if you want a non-enterfunction mode command to be executed during an enterfunction, do not define an enterfunction mode command for this key.

Enterfunctions

An enterfunction in the SKILL language is a built-in function which allows you to enter points graphically. The enterfunctions then collect these points and pass them to your procedure which uses the points to perform some action. These are very useful in the graphical environment.

The list of enterfunctions that collect points are:

enterArc, enterBox, enterCircle, enterDonut, enterEllipse, enterLine, enterPath, enterPoint, enterPoints, enterPolygon, enterScreenBox, enterSegment, enterMultiRep

Additional enterfunctions are *enterNumber* and *enterString*.

Displaying Bindkeys

To display the current bindings for the application, perform these steps:

1. In the CIW, use the **Options—Bind Key** command to display the Key or Mouse Bindings form.

2. In the Application Type Prefix cyclic field, choose the name of the application.

3. Click the Show Bind Keys button.

You can save the displayed file and load it from your *.cdsinit* file.

The file uses the *hiSetBindKeys* function, instead of the *hiSetBindKey* function, to define the bindkeys.

Can you describe the difference between the arguments for these two functions?

The *hiGetBindKey* Function

Use the *hiGetBindKey* function to determine the SKILL command associated with a mouse or keyboard event.

```
hiGetBindKey( "Schematics" "None<Btn1Down>" ) =>
 "schSingleSelectPt()"
hiGetBindKey( "Schematics" "<Key>z" ) => "hiZoomIn()"
```

The *hiSetBindKeys* Function

Use the *hiSetBindKeys* function to set multiple bindkeys for an application at one time. The first parameter is the application type. The second parameter is a list of bindkey lists, where a bindkey list is a two element list. The first element is the bindkey description, the second is the bindkey action.

Standard Bindkey Definitions

The following files contain the standard bindkey definitions:

- ◼ <install_dir>/tools/dfII*/samples/local/schBindKeys.il*

- ◼ <install_dir>/tools/dfII*/samples/local/leBindKeys.il*

Notice that these files use the *alias* function to give a shorter name to the *hiSetBindKey* function.

Examine the <install_dir>/tools/dfII*/cdsuser/.cdsinit file*, particularly the section entitled *LOAD APPLICATION BIND KEY DEFINITIONS,* to study the SKILL code that loads these two files.

The *alias* Function

Use this function to give a more convenient name to a SKILL function. This example gives the shorter name *bk* to the *hiSetBindKey* function.

```
alias( bk hiSetBindKey )
```

Opening a Text Window

Use the *view* function to display a text file in a read-only window.

Example

This example displays the bindkey file from the Virtuoso® Schematic Editor.

```
view(
   prependInstallPath( "samples/local/schBindKeys.il" )
)
```

Use the *prependInstallPath* function to make a pathname relative to the Virtuoso Design Environment installation directory. This function prepends *<install_dir>/tools/dfII* to the path name.

Example

This example displays the same file in a window entitled *Schematics Bindkeys*.

```
view(
   prependInstallPath("samples/local/schBindKeys.il" )

      ;; path to file
   '((406 506) (1032 806)) ;;; window bounding box
   "Schematics Bindkeys" ;;; window title
) => window:6
```

The *view* Function

The *view* function takes several optional arguments.

Argument	Status	Type	Meaning
file	required	text	Pathname
winSpec	optional	bounding box/ nil	Bounding box of the window If you pass *nil*, the default position is used.
title	optional	text	The title of the window. The default is the value of file parameter.
autoUpdate	optional	t/nil	If *t*, then the window will update for each write to the file. The default is no autoUpdate.
appName	optional	text	The Application Type for this window. The default is *"Show File"*.
help	optional	text	Text string for online help. The default means no help is available.

Manipulating Windows

hiGetWindowName(window(5)) => "Virtuoso..."

hiRaiseWindow(window(5))

Y

870

1

3

Virtuoso...

5

1140

X
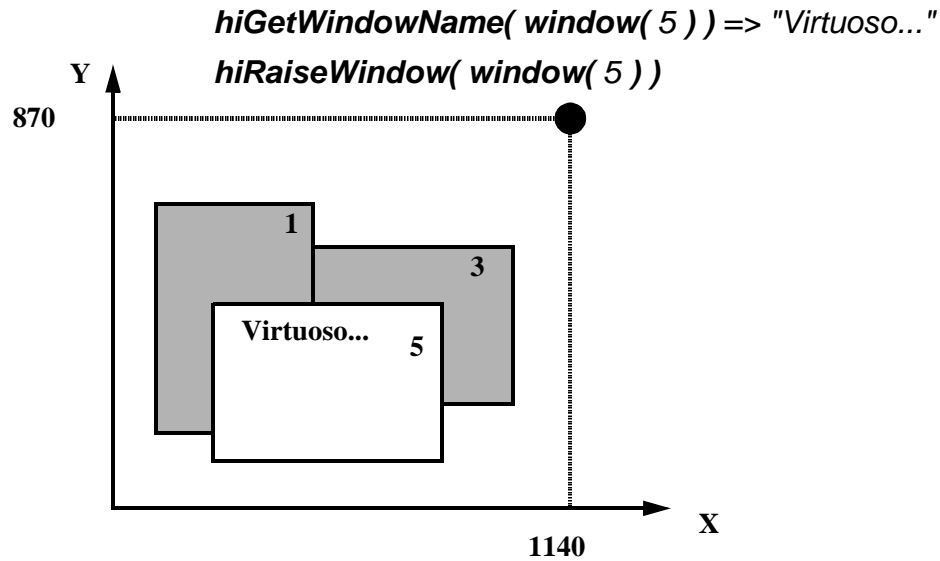
Naming Windows

- The **hiGetWindowName** Function

  Use the *hiGetWindowName* function to retrieve a window title.

  ```
  hiGetWindowName( window( 5 )) => "Virtuoso ... "
  ```

- The **hiSetWindowName** Function

  Use the *hiSetWindowName* function to set a window title.

  ```
  hiSetWindowName( window( 5 ) "My Title" ) => t
  ```

Raising and Lowering Windows

- The **hiRaiseWindow** Function

  Use the *hiRaiseWindow* function to bring a window to the top of the desktop.

  ```
  hiRaiseWindow( window( 5 )) => t
  ```

- The **hiLowerWindow** Function

Resizing Windows

The origin of the screen coordinate system is the lower-left corner.

The unit of measurement for screen coordinates is a pixel.

hiResizeWindow(window(3) list(300:50 1000:600))
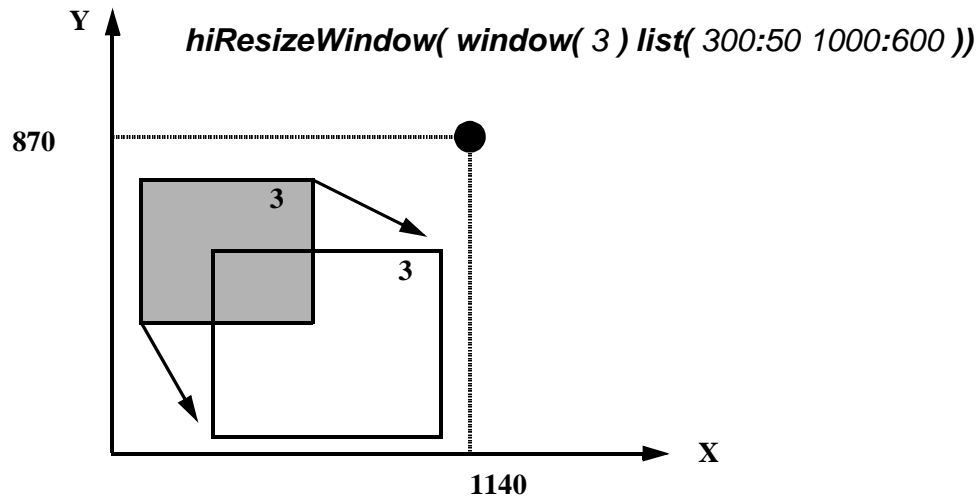
The *hiGetMaxScreenCoords* Function

Use the *hiGetMaxScreenCoords* function to determine the maximum x-coordinate and maximum y-coordinate value.

```
hiGetMaxScreenCoords() => ( 1140 870)
```

The *hiGetAbsWindowScreenBBox* Function

Use the *hiGetAbsWindowScreenBBox* function, passing *t* as the second argument, to retrieve the bounding box of a window.

```
hiGetAbsWindowScreenBBox( window(1) t ) =>
  ((200 300) (650 700))
```

The *hiResizeWindow* Function

Use the *hiResizeWindow* function to resize a window. The bounding box you pass to the *hiResizeFunction* will be the return value of the next call to the *hiGetAbsWindowScreenBBox* function.

```
hiResizeWindow( window(1)
  '((200 300) (650 700)) ) => t
```

Iconifying Windows

Y

870

3

hiIconifyWindow(window(3))

1140

X
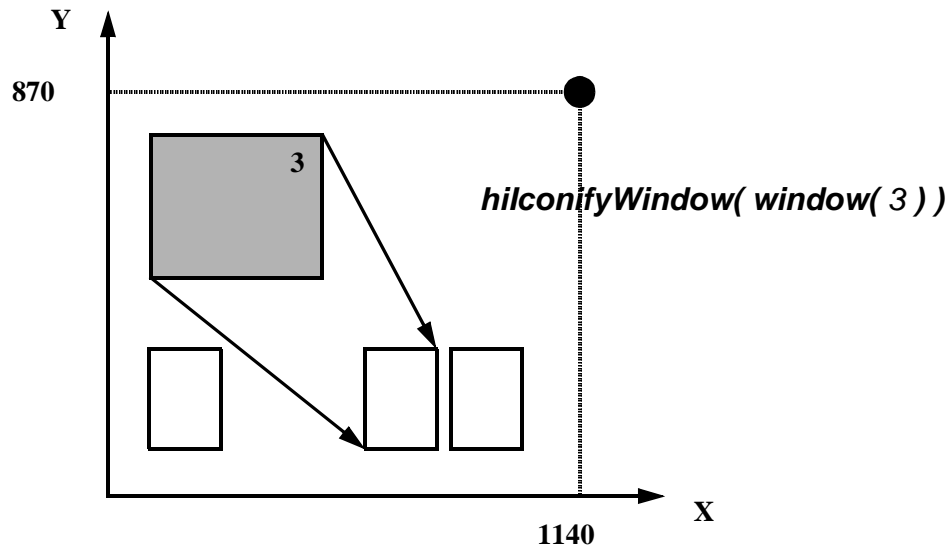
The *hiIconifyWindow* Function

The *hiIconifyWindow* function iconifies an open window.

```
hiIconifyWindow( window( 3 ) ) => t
```

The *hiGetWindowIconifyState* Function

The *hiGetWindowIconifyState* function returns *nil* if the window is uniconified. If the window is iconified, it returns the upper-left corner of the iconified window.

```
hiGetWindowIconifyState( window( 3 ) ) => (1108 490)
```

The *hiDeiconifyWindow* Function

The *hiDeinconifyWindow* function opens an iconified window.

```
hiDeiconifyWindow( window( 3 ) ) => t
```

Lab Overview

Lab 4-1 Opening Windows

Lab 4-2 Resizing Windows

Lab 4-3 Storing and Retrieving Bindkeys

Lab 4-4 Defining a Show File Bindkey

Module Summary

In this module, we discussed

- ■ The window ID data type

- ■ Opening design windows

- ■ Defining application bindkeys

- ■ Opening read-only text windows

- ■ Managing windows

Category	Functions
Basic	*window* *hiGetWindowList* *hiGetCurrentWindow* *hiGetAbsWindowScreenBBox*
Window manipulation	*hiGetWindowName, hiSetWindowName* *hiRaiseWindow* *hiResizeWindow* *hiGetWindowIconifyState,* *hiIconifyWindow,* *hiDeiconifyWindow*
Opening design windows	*geOpen*
Opening text windows	*view*
Bindkeys	*hiGetAppType* *hiGetBindKey* *hiSetBindKey*

Database Queries

Module 5

April 30, 2007

Module Objectives

- Use the SKILL® language to query design databases.
  - ❏ What is the name of the design in the current window?
  - ❏ How many nets are in this design?
  - ❏ What are the net names?
  - ❏ Are there any instances in this design? If so, how many?
  - ❏ Are there any shapes in this design? If so, how many?
- Understand database object concepts.
- Use the ~> operator to retrieve design data.

Terms and Definitions

Library	A collection of design objects referenced by logical name, such as cells, views, and cellviews.
Cell	A component of a design: a collection of different representations of the components implementation, such as its schematic, layout, or symbol.
Cellview	A particular representation of a particular component, such as the layout of a flip-flop or the schematic of a NAND gate.
View	An occurrence of a particular viewtype identified by its user-defined name, "XYZ". Each "XYZ" view has an associated *viewType* such as maskLayout, schematic, or symbolic.
Pin	A physical implementation of a terminal.
Terminal	A logical connection point of a component.

Database Objects

You can use the SKILL language to access schematic, symbol, and mask layout design data.

- Physical information (rectangles, lines, and paths)

- Logical information (nets and terminals)

The SKILL Evaluator organizes design data in virtual memory in terms of database objects. Each database object belongs to an object type that defines a set of common attributes that describe the object. The set of object types and their attributes is fixed by Cadence.

This module presents several object types and some of their attributes.

- The *cellView* object type

- The *inst* object type

- The *net* object type

- The Figure object types with their common attributes

- The Shape object types with their common attributes

Each database object can have one or more user-defined properties.

Database Queries   5-5

User actions can create, modify, and save database objects to disk.

SKILL variables can contain data of any type. However, for each attribute, the Database Access software constrains the value of the attribute to one of these SKILL data types:

- A string

- An integer

- A floating-point number

- A database object

- A list, possibly of database objects

Complete List of Object Types

The complete list of database object types and attributes is described in the *Virtuoso Design Environment SKILL Functions Reference*.

The objects outlined in red are covered in this course.

See the *Attribute Retrieval and Modification* section of the manual for more details.

Database Queries   5-7
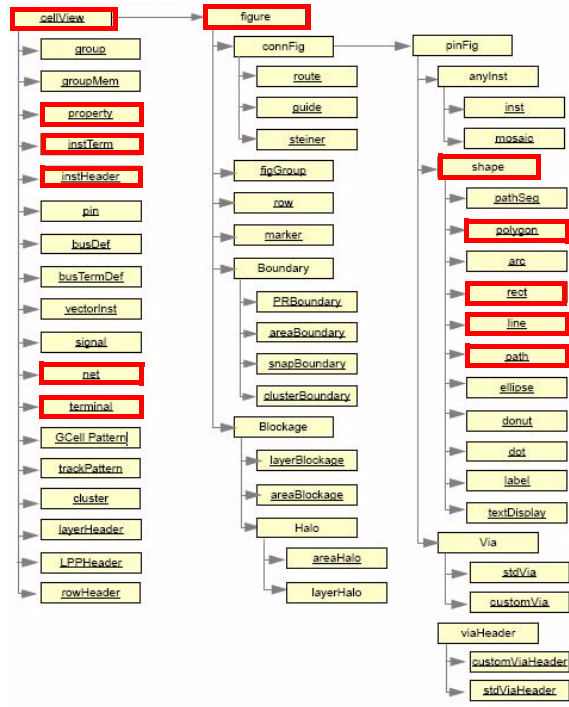
Querying a Design

When the design is in a graphics window, use the *geGetWindowCellView* function to retrieve the database object for the design as in this example.

```
geOpen(
   ?lib "master"
   ?cell "mux2"
   ?view "schematic"
   ?mode "r" ) => window:7

mux2 = geGetWindowCellView( window(7))⟹ db:38126636
```

Assign the database object to a SKILL variable to refer to it subsequently.

- *db:38126636* represents the database object.

- The SKILL Evaluator does not accept *db:38126636* as valid user input.

You can also use the *geGetEditCellView* function to retrieve the database object for the design that is open in the window specified. (default = current window)

```
mux2 = geGetWindowCellView( )⟹ db:38126636
```

These two functions are equivalent except when you are doing an edit-in-place. In that case *geGetEditCellview* returns the cellview being edited rather than the cellview in the window returned by *geGetWindowCellView*.

Database Queries   5-9

The *geGetWindowCellView* Function

See the Cadence® online documentation to read about this function.

The *geGetEditCellView* Function

See the Cadence online documentation to read about this function.

The ~> Operator

Use the ~> operator to access the attributes of a database object.

```
mux2~>objType => "cellView"
mux2~>cellViewType => "schematic"
```

Summary of ~> Syntax

Left Side	Right Side	Action
Database object	Attribute or user-defined property	Retrieve value or *nil* if no such attribute or property.
Database object	?	Returns a list of attribute names.
Database object	??	Returns list of name and values.
List of database objects	Attribute or user-defined property	Retrieves values individually and returns them in a list.

The underlying function for the ~> operator is the *getSGq* function. You sometimes see *getSGq*, *get*, or *getq* in error messages if you apply it to the wrong data.

The error message summarizes the data types to which the ~> operator is applicable. A database object is a kind of user type.

```
mux2 = 5
mux2~>objType
*Error* get/getq: first arg must be
either symbol, list, defstruct or user type - 5
```

Querying Designs with the ~> Operator

The queries stated in the objectives for this module are previewed below.

Query	The ~> Expression
What is the name of the design in the current window?	*cv = geGetWindowCellView( )* *cv~>libName* *cv~>cellName* *cv~>viewName*
How many nets are in the design? What are their names?	*length(cv~>nets)* *cv~>nets~>name*
What are the terminal names in the design?	*cv~>terminals~>name*
How many shapes are in the design? What kinds of shapes are they?	*length(cv~>shapes)* *cv~>shapes~>objType*

Database Queries

5-13

The *cellView* Object Type

Example: *master mux2 schematic*
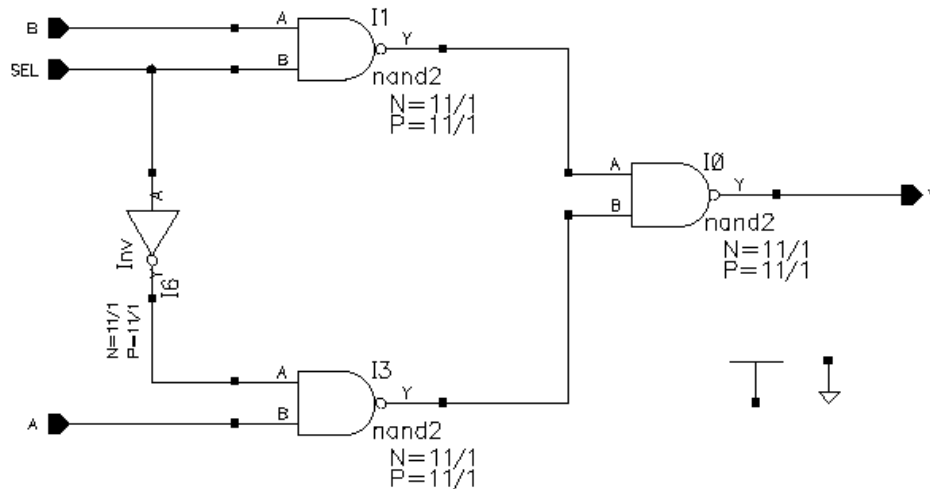
The *cellView* object type includes the following attributes among others:

Attribute	Data Type	Description
objType	String	*"cellView"*
libName	String	The library name of the design.
cellName	String	The cell name of the design.
viewName	String	The view name of the design.
cellViewType	String	The type of design data. Examples include *"schematic"*, *"maskLayout"*, and *"schematicSymbol"*.
instances	List of database objects	The list of instances in the design. Can be *nil*.
shapes	List of database objects	The list of shapes in the design. Can be *nil*.
nets	List of database objects	The list of nets in the design. Can be *nil*.
terminals	List of database objects	The list of terminals in the design. Can be *nil*.

Instances

Example: Instance *I1* of *master mux2 schematic*
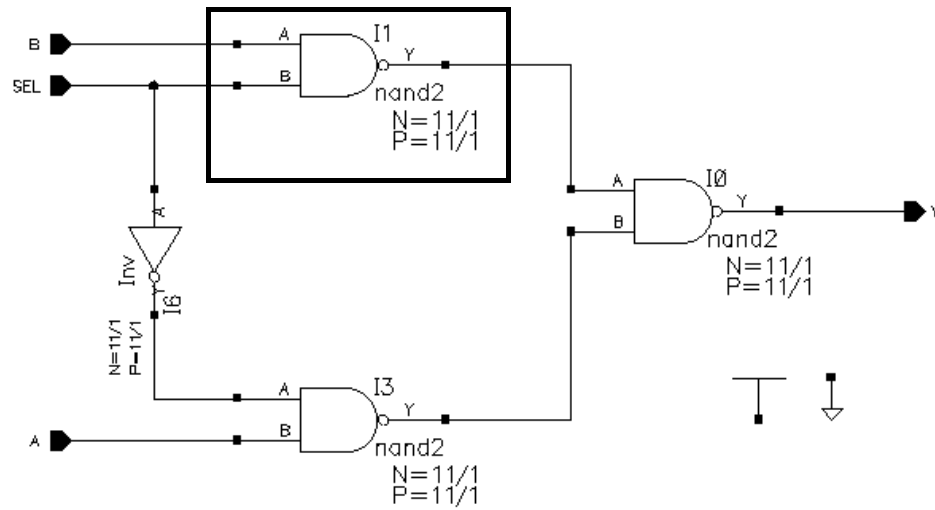
In this design, each of the instances has user-defined properties that describe the physical sizes of the transistors.

The *inst* Object Type

You can apply the ~> operator to the result of another ~> expression as in these examples:

- The list of the instances in the design

```
mux2~>instances =>
   ( db:39520396 db:39523572 db:39522480 .... )
I1 = dbFindAnyInstByName( mux2 "I1" ) => db:38127508
```

- The list of instance names

```
mux2~>instances~>name =>
   ( "I6" "I9" "I8" "I3" "I1"
     "I0" "I5" "I7" "I4" "I2" )
```

- The list of user-defined properties on the I1 instance

```
I1~>prop~>name =>("pw" "pl" "nl" "nw")
I1~>prop~>value => ("11" "1" "1" "11")
I1~>pw => "11"
```

- The list of master cell names

```
mux2~>instances~>cellName
   ( "Inv" "gnd" "vdd" "nand2" "nand2"
     "nand2" "opin" "ipin" "ipin" "ipin" )
```

The *inst* Object Type

The *inst* object type has the following attributes among others:

Attribute	Data Type	Description
objType	string	*"inst"*
libName	string	The library name of the master.
cellName	string	The cell name of the master.
viewName	string	The view name of the master.
name	string	The instance name.
master	a database object	The master cell view of this instance. Can be *nil* if master hasn't been read into virtual memory.
instTerms	list of database objects	The list of instance terminals. Can be *nil*.

The *dbFindAnyInstByName* Function

The *dbFindAnyInstByName* function returns the database object of the instance, given the database object of the design and the instance name.

See the Cadence online documentation to read about this function.

Nets

Example: net *SEL* in *master mux2 schematic*
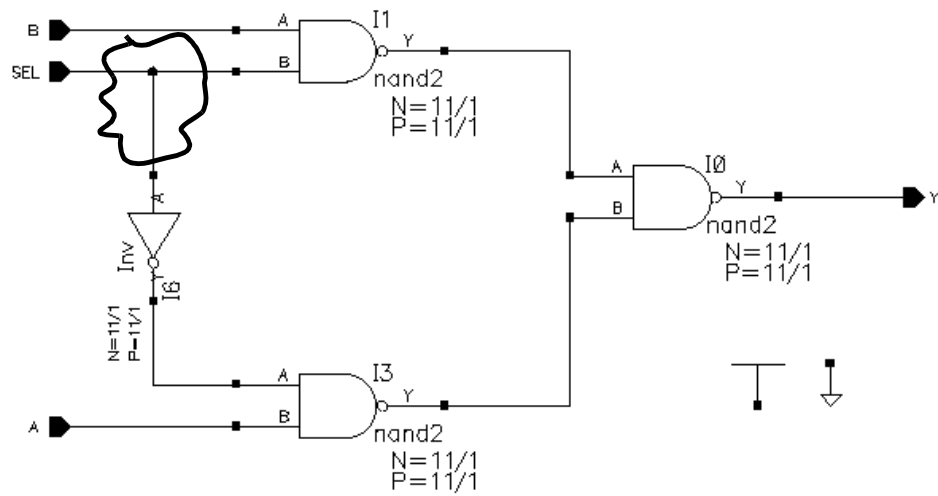
The *net* Object Type

Use the ~> operator and *cellView* and *net* attributes to retrieve the following:

- ■ The nets in the design

```
mux2~>nets => (
    db:41088056 db:41087980 db:41087752 db:41087676
    db:41087640 db:41087380 db:41087284 db:39523392
    db:39522784 )
dbFindNetByName( mux2 "SEL" ) => db:41087284
```

- ■ The number of nets

```
length( mux2~>nets ) => 9
```

- ■ The names of the nets

```
mux2~>nets~>name =>
    ("B" "A" "net4" "Y" "net6"
    "net7" "SEL" "gnd!" "vdd!" )
```

The *net* Object Type

The *net* object type has the following attributes among others:

Attribute	Data Type	Description
objType	string	*"net"*
name	string	The name of the net.
term	database object	The unique terminal. Can be *nil* if net is internal.
instTerms	list of database objects	The list of instance terminals.

The *dbFindNetByName* Function

See the Cadence online documentation to read about this function.

Instance Terminals

Instance terminals provide an interface between instance and the nets in a design.

- Each instance contains zero or more instance terminals.

- Each net connects zero or more instance terminals.

Example: The *B* instance terminal on the *I1* instance.
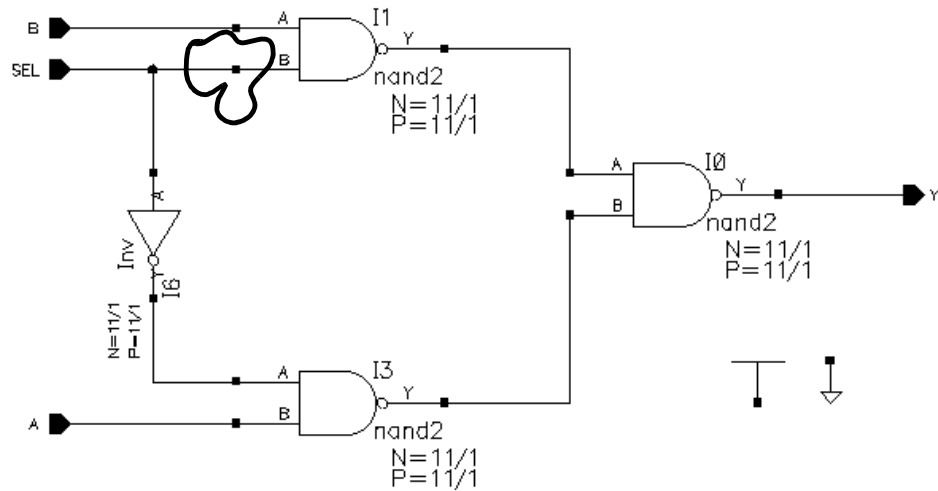
The *instTerm* Object Type

You can retrieve the following data:

■ The names of *instTerm* objects associated with the *I1* instance

```
dbFindAnyInstByName( mux2 "I1" )~>instTerms~>name =>
   ("B" "Y" "A")
```

■ The name of the net that attaches to the *B instTerm*

```
dbFindAnyInstByName( mux2 "I1" )~>instTerms~>net~>name =>
   ( "SEL" "net4" "B" )
```

Terminals

A terminal provides a way to connect to a net within the design.

Internal nets do not have terminals.
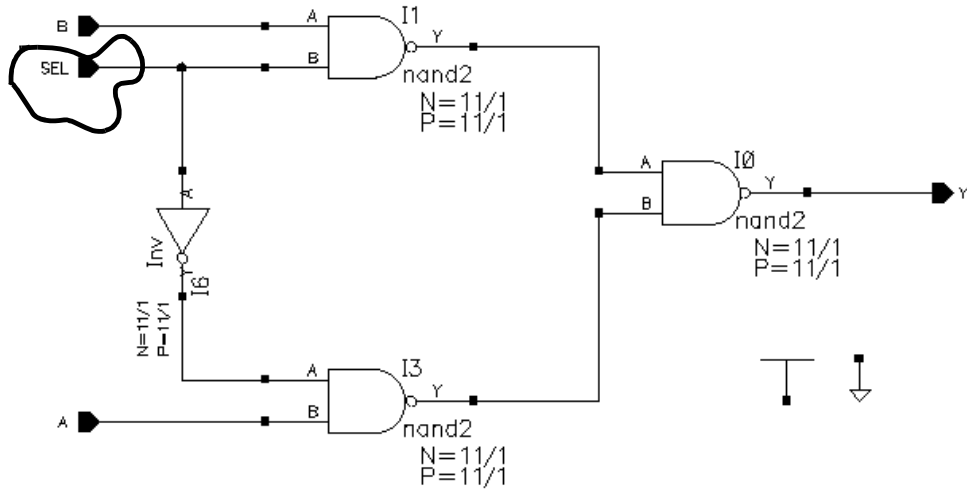
The *term* Object Type

Every design contains a list of its terminals. For example, the SEL terminal.

```
mux2~>terminals =>
   (db:39521220 db:39520296 db:39895624 db:39895292)
mux2~>terminals~>name =>
   ("SEL" "Y" "B" "A")
dbFindTermByName( mux2 "SEL" ) => db:39521220
```

Every net connects to one terminal at most. *nil* means that the corresponding net doesn't connect to a terminal object.

```
mux2~>nets~>term =>
   ( db:27850008 nil nil db:27850348 db:27849804
    nil nil db:27847992 nil )
```

Internal nets do not connect to any terminal. For example, the *net4* net does not connect to a terminal.

```
mux2~>nets~>name =>
   ("B" "net7" "net6" "Y" "A"
    "vdd!" "gnd!" "SEL" "net4" )
```

The *dbFindTermByName* Function

The *dbFindTermByName* function returns the terminal database object, given the database object of the design and the name of the terminal.

See the Cadence online documentation to read about this function.

Figures and Shapes
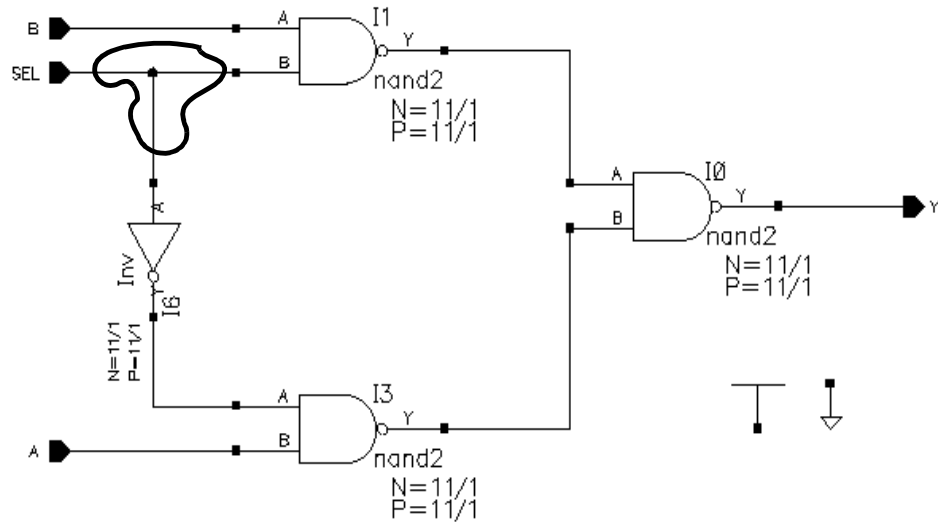
- *geGetSelSet* — function used to retrieve the currently selected set

- *geSelectFig* — adds an object to the selected set if it passes the selection filter.

Figures

Anything the user can select with the mouse is a Figure.

Every Figure is either an instance or a shape. Every Figure has a *bBox* attribute that describes its bounding box.

Each net can have one or more associated Figures. In a schematic, these are wire segments with objType *"line"*.

The *geGetSelSet* Function

Use the *geGetSelSet* function to retrieve the selected set. See the Cadence online documentation to read about this function.

The *geSelectFig* Function

Use the *geSelectFig* function to select a specific figure database object. See the Cadence online documentation to read about this function.

Shape Attributes

All shapes have several common attributes.

Attribute	Data Type	Description
objType	string	*"line", "rect"*, or *"path"*, or *"polygon"* etc.
bBox	list of coordinates	The bounding box of the shape.
layerName	string	The name of the layer on which the shape is found.
layerNum	integer	The number of the layer on which the shape is found.
lpp	list of two strings	The list of layer name and layer purpose.

Lab Overview

Lab 5-1 Querying Design Databases

In this lab, you open these designs:

- master mux2 schematic

- master mux2 layout

- master mux2 extracted

- master mux2 symbol

You enter SKILL expressions to answer the following questions:

- How many nets are in the design?

- What are the net names?

- Are there any instances in the design? If so, how many?

- Are there any shapes in the design? If so, how many?

Module Summary

In this module, we covered

- ■ Database object concepts

- ■ Several specific object types
    - ❑ The *cellView* object type
    - ❑ The *inst* object type
    - ❑ The various shape object types
    - ❑ The net object type

- ■ The *geGetWindowCellView* function

- ■ The *geGetSelSet* function

- ■ Using the ~> operator to retrieve attributes and user-defined properties

Database Queries   5-39

cadence®

Developing a SKILL Function

Module 6

April 30, 2007

Module Objectives

■ Group several SKILL® expressions into a single SKILL expression

■ Declare local variables

■ Declare a SKILL function

■ Understand the SKILL software development cycle

❏ Loading your SKILL source code

❏ Redefining a SKILL function

Terms and Definitions

load	A SKILL procedure that opens a file and reads it one SKILL expression at a time. The *load* function evaluates immediately. Usually, you set up your *.cdsinit* file to load your SKILL source code during the initialization of the Virtuoso® Design Environment.

Grouping SKILL Expressions Together

Sometimes it is convenient to group several SKILL statements into a single SKILL statement.

Use the curly braces, { }, 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 height of the bounding box stored in *bBox*.

```
bBox = list( 100:200 350:450 )
bBoxHeight = {
   ll  = lowerLeft( bBox )
   ur  = upperRight( bBox )
   lly = yCoord( ll )
   ury = yCoord( ur )
   ury - lly
}
=> 250
```

The variables *ll*, *ur*, *lly*, and *ury* are global variables. The curly braces, { }, do not make them local variables.

Developing a SKILL Function   6-5

Curly Braces

The following statements refer to the example above:

- The *ll* and *ur* variables hold the lower-left and upper-right points of the bounding box respectively.

- The *xCoord* and *yCoord* functions return the *x* and *y* coordinate of a point.

- The *ury-lly* expression computes the height. It is the last statement in the group and consequently determines the return value of the group.

- The return value is assigned to the *bBoxHeight* variable.

All of the variables, *ll*, *ur*, *ury*, *lly, bBoxHeight* and *bBox* are global variables.

Grouping Expressions with Local Variables

Use the *let* function to group SKILL expressions and declare one or more local variables.

■ Include a list of the local variables followed by one or more SKILL expressions.

■ These variables will be initialized to *nil*.

The SKILL expressions compose the body of the *let* function. The *let* function returns the value of the last expression computed within its body.

This example computes the height of the bounding box stored in *bBox*.

```
bBox = list( 100:200 350:450 )
bBoxHeight = let( ( ll ur lly ury )
  ll  = lowerLeft( bBox )
  ur  = upperRight( bBox )
  lly = yCoord( ll )
  ury = yCoord( ur )
  ury - lly ) ; let
=> 250
```

■ The local variables *ll*, *ur*, *lly*, and *ury* are initialized to *nil*.

■ The return value is the *ury-lly*.

The *let* Function

You can freely nest *let* statements.

You can access the value of a variable any time from anywhere in your program. SKILL transparently manages the value of a variable like a stack. Each variable has a stack of values.

■ The current value of a variable is simply the top of the stack.

■ Assigning a value to a variable changes only the top of the stack.

Whenever the flow of control enters a *let* function, the SKILL Evaluator pushes a temporary value onto the value stack of each variable in the local variable list. The local variables are normally initialized to *nil*.

When the flow of control exits a *let* function, SKILL pops the top value of each variable in the local variable list. If a variable with the same name existed outside of the *let* it will have the same value that it had before the *let* was executed.

Two Common *let* Errors

The two most common *let* errors are:

■ Including whitespace after *let*.
The error message depends on whether the return value of the *let* is assigned to a variable.

```
let ( ( ll ur lly ury )
   ll  = lowerLeft( bBox )
   ur  = upperRight( bBox )
   lly = yCoord( ll )
   ury = yCoord( ur )
   ury - lly
) ; let
*Error* let: too few arguments (at least 2 expected, 1 given)
```

■ Omitting the list of local variables.
The error messages vary and can be obscure and hard to decipher.

```
let(
   ll  = lowerLeft( bBox )
   ur  = upperRight( bBox )
   lly = yCoord( ll )
   ury = yCoord( ur )
   ury - lly
) ; let
```

Developing a SKILL Function   6-9

Defining SKILL Functions

Use the *procedure* function to associate a name with a group of SKILL expressions. The name, a list of arguments, and a group of expressions compose a SKILL function declaration.

■ The name is known as the function name.

■ The group of statements is the function body.

Example

```
bBox = list( 100:200 350:450 )                    Write global variable

procedure( TrBBoxHeight( )                         Define function
   let( ( ll ur lly ury )                          Local variables
      ll      = lowerLeft( bBox )                   Read global variable
      ur     = upperRight( bBox )                   Read global variable
      lly    = yCoord( ll )
      ury    = yCoord( ur )
      ury - lly                                    Return value
   ) ; let
) ; procedure

bBoxHeight = TrBBoxHeight()                        Invoke function
```

Developing a SKILL Function   6-11

The *procedure* Function

Local Variables

You can use the *let* syntax function to declare the variables *ll*, *ur*, *ury*, and *lly* to be local variables. The arguments presented in an argument list of a procedure definition are also local variables. All other variables are global variables.

Global Variable

The *bBox* variable is a global variable because it is neither an argument nor a local variable.

Return Value

The function returns the value of the last expression evaluated. In this example, the function returns the value of the *let* expression, which is the value of the *ury-lly* expression.

Three Common *procedure* Errors

The three most common *procedure* errors are:

■ Including whitespace after the *procedure* function.

```
procedure ( TrBBoxHeight( ) ...
) ; procedure
*Error* procedure: too few arguments (at least 2 expected, 1 given)
```

■ Including whitespace after your function name.

```
procedure( TrBBoxHeight ( ) ...
) ; procedure
*Error* procedure: illegal formal list - TrBBoxHeight
```

■ Omitting the argument list.

```
procedure( TrBBoxHeight ...
) ; procedure
*Error* procedure: illegal formal list - TrBBoxHeight
```

Notice that the error message refers to the *procedure* function. The arguments to *procedure* are the function name, the argument list of the function, and the expression composing the body of the function.

Developing a SKILL Function   6-13

Defining Required Function Parameters

The fewer global variables you use, the more reusable your code is.

Turn global variables into required parameters. When you invoke your function, you must supply a parameter value for each required parameter.

In our example, make *bBox* be a required parameter.

```
procedure( TrBBoxHeight( bBox )
  let( ( ll ur lly ury )
    ll  = lowerLeft( bBox )
    ur  = upperRight( bBox )
    lly = yCoord( ll )
    ury = yCoord( ur )
    ury - lly
  ) ; let
) ; procedure
```

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

```
bBoxHeight = TrBBoxHeight( list( 50:150 200:300 ) )
  => 150
```

You can use the *let* function to declare the variables *ll*, *ur*, *ury*, and *lly* to be local variables.

In the example, the variable *bBox* occurs both as a global variable and as a formal parameter.

Here's how the SKILL Evaluator keeps track. When you call the *TrBBoxHeight* function, the SKILL Evaluator:

- Saves the current value of *bBox*.

- Evaluates **list(*50:150 200:300*)** and temporarily assigns it to *bBox*.

- Restores the saved value of *bBox* when the *TrBBoxHeight* function returns.

Defining Optional Function Parameters

Include **@optional** before any optional function parameters.

Use parentheses to designate a default value for an optional parameter.

```
procedure( TrOffsetBBox( bBox @optional (dx 0)(dy 0))
  let( ( llx lly urx ury )
    ...
    list( ;;; return the new bounding box
      llx+dx:lly+dy
      urx+dx:ury+dy
      )
  ) ; let
) ; procedure
```

To call the *TrOffsetBBox* function, specify the required and the optional arguments as follows:

■ **TrOffsetBBox(*someBBox*)**
  dx and *dy* default to *0.*

■ **TrOffsetBBox(*someBBox* 0.5)**
  dy defaults to *0.*

■ **TrOffsetBBox(*someBBox* 0.5 0.3)**

Developing a SKILL Function   6-17

The procedure above takes as input the original bounding box and the desired change in the *x* and *y* directions. The procedure returns a new bounding box that has been offset.

Unless you provide a default value for an optional parameter in the procedure declaration, the default value will be *nil*. This can lead to an error if *nil* is not appropriate for the operations executed using the parameter. This is the case for the procedure shown here.

You provide a default value for a parameter using a list (*<parameter> <default value>*).

Defining Keyword Function Parameters

Include **@key** before keyword function parameters.

Use parentheses to designate a default value for a keyword parameter.

```
procedure( TrOffsetBBox( bBox @key (dx 0)(dy 0))
   let( ( llx lly urx ury )
      ...
      list(
         llx+dx:lly+dy
         urx+dx:ury+dy
         )
   ) ; let
) ; procedure
```

To call the *TrOffsetBBox* function, specify the keyword arguments as follows:

- **TrOffsetBBox(*someBBox ?dx 0.5 ?dy 0.4*)**

- **TrOffsetBBox(*someBBox ?dx 0.5*)**
  dy defaults to *0.*

- **TrOffsetBBox(*someBBox ?dy 0.4*)**
  dx defaults to *0.*

Keyword parameters free you from the need for a specific order for your parameters. This can be very valuable when you have a significant number of optional parameters. When you must preserve the order for optional parameters you need to supply values for all parameters preceding the one you actually want to set.

Keyword parameters are always syntactically optional. Care should be taken however that the procedure will give correct results without a parameter specified. If you cannot do this check each parameter for an acceptable value and emit an error message when execution of the function will not yield a sensible answer.

As with all parameters, unless you provide a default value for a keyword parameter the default value will be *nil*.

Collecting Function Parameters into a List

An **@rest** argument allows your procedure to receive all remaining arguments in a list. Use a single **@rest** argument to receive an indeterminate number of arguments.

Example

The *TrCreatePath* function receives all of the arguments you pass in the formal argument *points*.

```
procedure( TrCreatePath( @rest points )
  printf(
     "You passed these %d arguments: %L\n"
     length( points ) points
  )
) ; procedure
```

The *TrCreatePath* function simply prints a message about the list contained in *points*.

To call the *TrCreatePath* function, specify any number of points.

```
TrCreatePath( 3:0 0:4 )
TrCreatePath( 3:0 0:4 0:0 )
```

The *@rest* parameter structure allows you to specify input parameters even when you do not know how many input elements the user will want to operate on. This type of argument specification is perfect for functions like *dbCreatePath* or *dbCreatePolygon*. It is also very convenient for inputting the elements of an arbitrary length list.

As written, the *TrCreatePath* function doesn't really create a path. You can make it create a path in a cellview by passing the list *points* to the *dbCreatePath* function.

SKILL Development Cycle

```
        ┌──────────────────┐ ◄─────────┐
        │ Define Functions │           │
        └──────────────────┘           │
                 │                      │
                 ▼                      │
        ┌──────────────────┐           │
        │  Test Functions  │           │
        └──────────────────┘           │
                 │                      │
                 ▼                      │
               ╱   ╲        Yes         │
              ╱ Bug? ╲ ──────────────►──┤
              ╲      ╱                   │
               ╲    ╱                    │
                │ No                     │
                ▼                        │
               ╱       ╲     Yes         │
              ╱ Enhance? ╲ ───────────►──┘
              ╲          ╱
               ╲        ╱
                │ No
                ▼
        ┌──────────────────┐
        │       Done       │
        └──────────────────┘
```
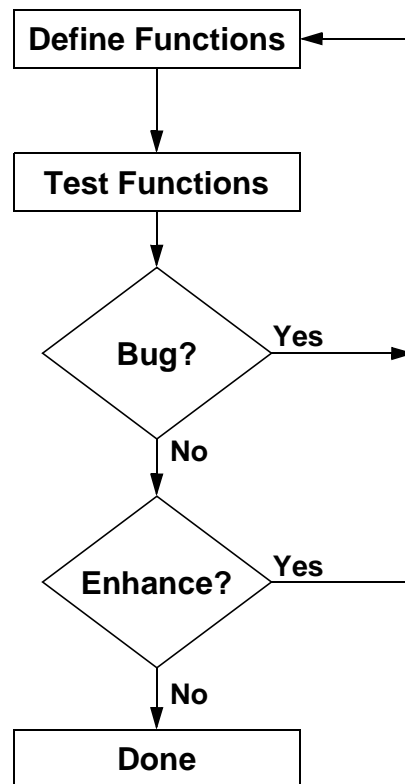
Developing a SKILL Function   6-23

Defining SKILL Functions

The SKILL Evaluator allows you to redefine functions without exiting the Virtuoso
Design Environment. A rapid edit-and-run cycle facilitates bottom-up software
development.

Launching an Editor from the Virtuoso Design Environment

The *edit* function launches the editor from the Virtuoso Design Environment. When you
use a path name, the *edit* function uses the UNIX path variable.

```
edit( "~/SKILL/YourCode.il" )
```

Before you use the *edit* function, set the editor global to a string that contains the UNIX
shell command to launch your editor.

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

If you install the SKILL Debugger before you load your code, you can edit the source
code for one of your functions by passing the function name to the *edit* function.

```
edit( TrExampleFunction )
```

Testing SKILL Functions

You can initially test your application SKILL functions directly in the CIW. Then, you
can create a user interface for your application.

Loading Source Code

The *load* function evaluates each expression in a given SKILL source code file.

You use *load* function to define SKILL functions and execute initialization expressions.

The *load* function returns *t* if all expressions evaluate without errors. Typical errors include:

■ Syntax problems with a *procedure* definition.

■ Attempts to *load* a file that does not exist.

Any error aborts the *load* function. The *load* function does not evaluate any expression that follows the offending SKILL expression.

When you pass a relative pathname to the *load* function, the *load* function resolves it in terms of a list of directories called the SKILL path.

You usually establish the SKILL path in your *.cdsinit* file by using the *setSkillPath* or *getSkillPath* functions.

The *loadi* Function

The *loadi* function also loads the SKILL code contained in a file. The difference in this function is that it continues despite errors completing all the statements within the file. In fact, no error messages are passed to the user and *loadi* always completes with a return value of *t*. This function is very valuable when the statements within a file are independent, for examples statements that set the bindkeys for a session.

The SKILL Path Functions

■ The *getSkillPath* function returns the current list of directories in search order.

■ The *setSkillPath* function sets the path to a list of directories.

Pasting Source Code into the CIW

The *load* function accesses the current version of a file. Any unsaved edits are invisible to the *load* function.

Sometimes you want to define a function without saving the source code file.

You can paste source code into the CIW with the following procedure:

1. Use the mouse in your editor to select the source code.

2. Move the cursor over the CIW input pane.

3. Click the middle mouse button.
   Your selection is displayed in the CIW input pane.

4. Press Return.
   Your entire selection is displayed in the CIW output pane.

Developing a SKILL Function   6-27

If you are using the *vi* editor, make sure line numbering is turned off. Otherwise, when you paste your code into the CIW, the line numbers become SKILL expressions in the virtual memory definition of your functions. To turn off line numbers, enter the following into your *vi* window:

```
:set nonumber
```

Make sure you select all the characters of the SKILL function definition. Be particularly careful to include the final closing parentheses.

If you paste the definition of a single SKILL function into the CIW, then the function name is the last word displayed in the CIW output pane.

> *Why is that?*

Because the SKILL *procedure* function returns the function symbol.

Redefining a SKILL Function

While developing SKILL code, you often need to redefine functions.

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

By default the *writeProtect* switch is set to *nil*. If you first define a SKILL function with *writeProtect t* you cannot redefine the function during the session.

You set the *writeProtect* switch in your *.cdsinit* file.

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

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

```
sstatus( writeProtect t ) => t
procedure( TrReciprocal( x ) 1.0 / x ) => TrReciprocal
procedure( TrReciprocal( x ) when( x != 0.0 1.0 / x ))
```

Error def: function name write protected
and cannot be redefined - TrReciprocal

The *writeProtect* switch has nothing to do with write protecting the source code file itself.

Lab Overview

Lab 6-1 Developing a SKILL Function

○ Develop a SKILL function, *TrBBoxArea*, to compute the area of a bounding box. The bounding box is a parameter.

○ Model your SKILL function after the example SKILL function *TrBBoxHeight*.

Developing a SKILL Function   6-31

Module Summary

In this module, we covered

- ■ Using curly braces, { }, to group SKILL statements together

- ■ Using the *let* function to declare local variables

- ■ Declaring SKILL functions with the procedure function
    - ❏ Name
    - ❏ Arguments
    - ❏ Body

- ■ The SKILL development cycle

- ■ Maintaining SKILL code
    - ❏ The *edit* function and *editor* variable
    - ❏ The *load* function
    - ❏ The *writeProtect* switch

Developing a SKILL Function   6-33

cadence®

Flow of Control

Module 7

April 30, 2007

Module Objectives

■ Review relational operators.

■ Describe logical operators.

■ Examine branching statements.

■ Discuss several methods of iteration.

Terms and Definitions

Iteration	To repeatedly execute a collection of SKILL® expressions.

Relational Operators

Use the following operators to compare data values.

These operators all return *t* or *nil.*

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

Use parentheses to control the order of evaluation. This example assigns *3* to *x*, returning *3*, and next compares *3* with *5*, returning *t*.

```
(x=3)<5 => t
```

The SKILL Evaluator generates an error if the data types are inappropriate. Error messages mention the function in addition to the operator.

```
1 > "abc"
*** Error in routine greaterp:
Message: *Error* greaterp: can't handle (1 > "abc")
```

Logical Operators

The SKILL Evaluator considers *nil* as FALSE and any other value as TRUE.

The SKILL® language provides generalized boolean operators.

Operator	Arguments	Function	Example	Return Value
!	general	null	!3 !nil !t	nil t nil
&&	general	and	x = 1 y = 5 x < 3 && y < 4 x < 3 && 1/0 y < 4 && 1/0	5 nil *** Error nil
||	general	or	x < 3 || y < 4 x < 3 || 1/0 y < 4 || 1/0	t t *** Error

The *&&* and *||* operators only evaluate their second argument if they must to determine the return result.

The *&&* and *||* operators return the value last computed.

Flow of Control   7-7

The *&&* Operator

Evaluates its first argument. If it is *nil*, then *&&* returns *nil*.
The second argument is not evaluated.

If the first argument evaluates to non-*nil*, then *&&* evaluates the second argument. The *&&* operator returns the value of the second argument.

The *||* Operator

Evaluates its first argument. If it is non-*nil*, then *||* returns the value of the first argument. The second argument is not evaluated.

If the first argument evaluates to *nil*, then the second argument is evaluated. The *||* operator returns the value of the second argument.

Using the && and || Operators to Control Flow

You can use both the *&&* and *||* operators to avoid cumbersome *if* or *when* expressions.

Example of Using the || Operator

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

```
theUser = userName || "noName"
```

Flow of Control   7-9

Branching

Branching Task	Function
Binary branching	if when unless
Multiway branching	case cond
Arbitrary exit	prog return

Flow of Control

The *if* Function

Use the *if* function to selectively evaluate two groups of one or more expressions.

The selection is based on whether the condition evaluates to *nil* or non-*nil*.

The return value of the *if* expression is the value last computed.

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

In the above expression:

- If the shapeType is *rect* the return value is the value of *rectCount*.

- If the shapeType is not *rect* the return value is the value of *miscCount*.

Alternatives expressions:

- Use *if(exp ...)* instead of *if(exp != nil ...)*
- Use *if(!exp ...)* instead of *if(exp == nil ...)*

Two Common *if* Errors

Two common *if* errors are:

■ Including whitespace after *if*

```
if ( shapeType == "rect"
   ...
)
*** Error in routine if
Message: *Error* if: too few arguments ...
```

■ Including a right parenthesis after the conditional expression

```
if( shapeType == "rect" )
***Error* if: too few arguments (at least 2 expected, 1 given)
```

The SKILL Evaluator does most of its error checking during execution. Error messages involving *if* expressions can be obscure.

Remember

■ To avoid whitespace immediately after the *if* syntax function.

■ To place parentheses as follows:

   if(... then ... else ...)

Nested *if-then-else* Expressions

Be careful with parentheses. Comment the closing parentheses and indent consistently as in this example.

```
if( shapeType == "rect" then
  ++rectCount
else
  if( shapeType == "line" then
   ++lineCount
  else
   ++miscCount
  ) ; if line
) ; if rect
```

The *when* and *unless* Functions

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

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

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

Use the *unless* function to avoid negating a condition.

```
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 sequentially compares a candidate value against a series of target values. The target must be a value and cannot be a symbol or expression that requires evaluation.

When it finds a match, it evaluates the associated expressions and returns the value of the last expression evaluated.

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

If you have expressions to evaluate when no target value matches the candidate value, include those expressions in an arm at the end. Use *t* for the target value for this last arm. If the flow of control reaches an arm whose target value is the *t* value, then the SKILL Evaluator unconditionally evaluates the expressions in the arm.

When target value of an arm is a list, the SKILL Evaluator searches the list for the candidate value. If the candidate value is found, all the expressions in the arm are evaluated.

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

The *cond* Function

Use the *cond* function when your logic involves 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 arm, until it finds one that is non-*nil.* It then executes all the expressions in the arm and exits.

■ Returns the last value computed in the arm it executes.

The *cond* function is equivalent to

```
if        condition1  then exp11exp12 ...
else if   condition2  then exp21exp22 ...
else if   condition3  then exp31exp32 ...
...
else                       expN1expN2 ....
```

Flow of Control   7-23

This example *TrClassify* function includes the *cond* function and the *numberp* function.

```
procedure( TrClassify( signal )
  cond(
    ( !signal nil )
    ( !numberp( signal ) nil )
    ( signal >= 0 && signal < 3 "weak" )
    ( signal >= 3 && signal < 10 "moderate" )
    ( signal >= 10 "extreme" )
    ( t  "unexpected" )
  ) ; cond
) ; procedure
```

The *numberp, listp, stringp,* and *symbolp* Functions

The SKILL language provides many functions that recognize the type of their arguments, returning *t* or *nil*. Traditionally, such functions are called *predicates*. Their names end in *"p".*

Each function takes one argument, returning *t* if the argument is of the type specified, otherwise returning *nil*. See the Cadence® online documentation to read more about this function.

Iteration

Iteration task	Function
Numeric range	for
List of values	foreach
While a condition is non-*nil*	while

These functions repeat a statement block for a specific number of times, or through each element of a list, or until a certain condition is satisfied.

The *for* Function

This example adds the integers from *1* to *5* using a *for* function.

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

The *for* function increments the index variable by 1.

The *for* function treats the index variable as a local variable.

- Saves the current value of the index variable before evaluating the loop expressions.

- Restores the index variable to its saved value after exiting the loop.

- Returns the value *t*.

The SKILL Evaluator does most of its error checking during execution. Error messages about *for* expressions can be obscure. Remember

- The placement of the parentheses: *for(...)*.

- To avoid putting whitespace immediately after the *for* syntax function.

Flow of Control 7-27

The only way to exit a *for* loop early is to call the *return* function. To use the *return* function, you must enclose the *for* loop within a *prog* expression. This example finds the first odd integer less than or equal to *10*.

```
prog( ( )
  for( i 0 10
    when( oddp( i )
      return( i )
    ) ; when
  ) ; for
) ; prog
```

The *foreach* Function

Use the *foreach* function to evaluate one or more expressions for each element in a list of values.

```
rectCount = lineCount = miscCount = 0
shapeTypeList = '( "rect" "polygon" "rect" "line" )

foreach( shapeType shapeTypeList
  case( shapeType
     ( "rect"    ++rectCount )
     ( "line"    ++lineCount )
     ( t         ++miscCount )
  ) ; case
) ; foreach

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

When evaluating a *foreach* expression, the SKILL Evaluator 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, the SKILL Evaluator saves the current value of *shapeType*. The SKILL Evaluator restores the saved value after completing the *foreach* loop.

■ The variable *shapeTypeList* contains the list of values. The SKILL Evaluator 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 *case* statement.

■ The return value of the *foreach* loop is the list contained in the *shapeTypeList* variable.

The *while* Function

Use the *while* function to execute one or more expressions repeatedly as long as the condition is non-*nil*.

Example of a *while* expression.

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

The *prog* and *return* Functions

If you need to exit a collection of SKILL statements conditionally, use the *prog* function.

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

Use the *return* function to force the *prog* function to immediately return a value. The *prog* function does not execute any more SKILL statements.

If you do not call the *return* function within the *prog* body, *prog* returns *nil*.

For example, the *TrClassify* function returns either *nil*, "*weak*", "*moderate*", "*extreme*", or "*unexpected*", depending on the *signal* argument. This *prog* example does not use any local variables.

```
procedure( TrClassify( signal )
  prog( ()
    unless( signal return( nil ))
    unless( numberp( signal ) return( nil ))
    when( signal >= 0 && signal < 3 return( "weak" ))
    when( signal >= 3 && signal < 10 return( "moderate" ))
    when( signal >= 10 return( "extreme" ))
    return( "unexpected" )
  ) ; prog
) ; procedure
```

Use the *prog* function and the *return* function to exit early from a *for* loop. This example finds the first odd integer less than or equal to *10*.

```
prog( ( )
  for( i 0 10
    when( oddp( i )
      return( i )
    ) ; when
  ) ; for
) ; prog
```

A *prog* function can also establish temporary values for local variables. All local variables receive temporary values initialized to *nil*.

The current value of a variable is accessible at any time from anywhere.

The SKILL Evaluator transparently manages a value slot of a variable as if it were a stack.

■ The current value of a variable is simply the top of the stack.

■ Assigning a value to a variable changes only the top of the stack.

Whenever your program invokes the *prog* function, the SKILL Evaluator pushes a temporary value onto the value stack of each variable 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.

Lab Overview

Lab 7-1 Writing a Database Report Program

You write a SKILL function to count the shapes in a design.
You enhance this program in subsequent labs.

Lab 7-2 Exploring Flow of Control

You write a SKILL function to validate 2-dimensional points.

Lab 7-3 More Flow of Control

You write a SKILL function to compare 2-dimensional points.

Lab 7-4 Controlling Complex Flow

You write a SKILL function to validate a bounding box.

Module Summary

In this module, we covered

Category	Function
Relational Operators	<
	<=
	>
	>=
	==
	!=
Logical Operators	!
	&&
	||
Branching	if when unless
	case
	cond
Iteration	for
	foreach
	while
Miscellaneous	prog return

List Construction

Module 8

April 30, 2007

Module Objectives

- Review the techniques for building a list.
    - The ' operator
    - The *list* function
    - The *cons* function
    - The *append* function

- Use the *foreach mapcar* function to make a list that corresponds one to one with a given list.

- Use the *setof* function to make a filtered copy of a list.

- Use the *foreach mapcar* function and *setof* function together to make a list that corresponds one to one with a filtered list.

List Construction   8-3

List Review

You can make a new list by:

- Specifying all the elements literally

```
'( ( "one" 1 ) ( "two" 2 ) )
  => ( ( "one" 1 ) ( "two" 2 ) )
```

- Computing each element from an expression

```
a = 1 => 1
b = 2 => 2
list( a**2+b**2 a**2-b**2) => ( 5 -3 )
```

You can add one or more elements to an existing list by:

- Adding an element to the front of a list

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

- Merging two lists together

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

The *foreach mapcar* Function

Use the *foreach mapcar* function to build a list in one-to-one correspondence with a base list.

The body of the *foreach mapcar* loop computes each element in the new list from the corresponding element in the base list.

Base list   **New list**
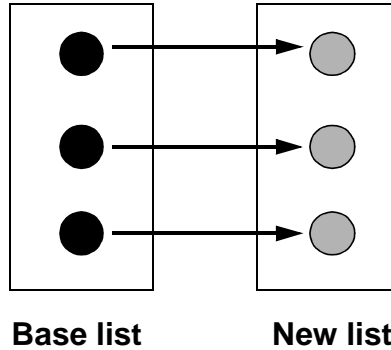
Example

```
L = '( 1 2 3 )
foreach( x L x**2 ) => ( 1 2 3 )
Squares = foreach( mapcar x L x**2 ) => ( 1 4 9 )
```

List Construction   8-7

The *foreach mapcar* function and the *foreach* function behave similarly with one exception. Although they compute the same values during each loop iteration, they return different lists.

■ The *foreach mapcar* function returns the list of values that each loop iteration computes.

■ The *foreach* function returns the base list.

Questions	Answers	
	The *foreach* Function	**The *foreach mapcar* Function**
What happens to the return result of the last expression in the loop body?	Each loop result is ignored.	Each loop result is collected into a new list.
What is the return result?	The original base list is the return result.	The new list is the return result.

Extended *foreach mapcar* Example

Suppose we want to build a list of the window titles for all the open windows. Use the *foreach mapcar* function together with the *hiGetWindowList* function and the *hiGetWindowName* function.

```
winNames = foreach( mapcar wid hiGetWindowList()
    hiGetWindowName( wid )
) ; foreach
```

As an alternative to using the *foreach mapcar* function, you can use the normal *foreach* function.

■ However, you are responsible for collecting the return results. Use the **cons** function.

■ In addition, the list will be in reverse order. Use the *reverse* function to make a copy of the list in the correct order.

```
winNames = nil
foreach( wid hiGetWindowList()
  winNames = cons( hiGetWindowName( wid ) winNames )
) ; foreach

winNames = reverse( winNames )
```

List Construction   8-9

The *mapcar* Function

The *SKILL Language Reference Manual* documents the *mapcar* function.

You can use the *foreach mapcar* expressions without understanding *mapcar*.

The *mapcar* function accepts two arguments:

- A function of one argument
- A list

The *mapcar* function

- Invokes the function for each element of a list.
- Returns the list of results.

Examples

- A *foreach mapcar* expression

    ```
    foreach( mapcar x '( 1 2 3 4 ) x**2 )
    ```

- An equivalent *mapcar* expression

    ```
    procedure( TrSquare( x ) x**2 )
    mapcar( 'TrSquare '( 1 2 3 4 ) ) => ( 1 4 9 16 )
    ```

List Construction   8-11

During compilation, each *foreach mapcar* expression is translated into a *mapcar* function call. You can use the trace facility to reveal the details in these examples:

- A trace of a *foreach* expression

    ```
    foreach( x '(1 2 3 ) x**2 )
    |(1**2)
    |expt --> 1
    |(2**2)
    |expt --> 4
    |(3**2)
    |expt --> 9
    (1 2 3)
    ```

- A trace of a *foreach mapcar* expression
  funobj:0x21980a8 represents the function that the SKILL Evaluator dynamically generates to represent *x**2*.

    ```
    foreach( mapcar x '(1 2 3 ) x**2 )
    |mapcar(funobj:0x21980a8 (1 2 3))
    ||(1**2)
    ||expt --> 1
    ||(2**2)
    ||expt --> 4
    ||(3**2)
    ||expt --> 9
    |mapcar --> (1 4 9)
    (1 4 9)
    ```

Filtering Lists

The *setof* function makes a filtered copy of a list, including all elements that satisfy a given filter.

For example, you can build a list of the odd elements of *(1 2 3 4 5)*.

Filtered list
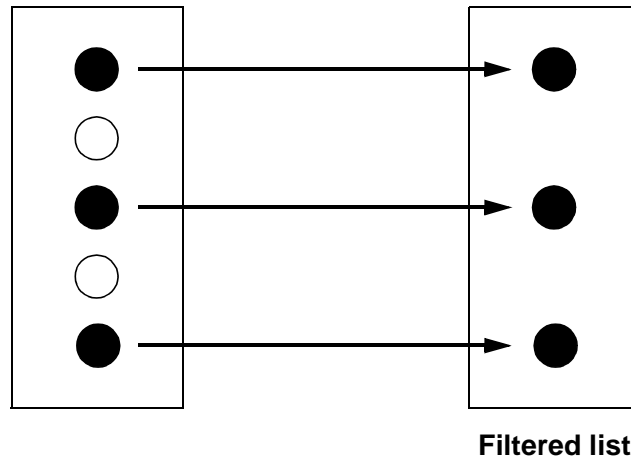
The *oddp* function returns *t/nil* if its argument is odd/even.

```
setof( x '( 1 2 3 4 5 ) oddp( x ) ) => ( 1 3 5 )
```

List Construction   8-13

The *setof* Function

The *setof* function accepts several arguments:

- A local variable that holds a list element in the filter expressions.

- The list to filter.

- One or more expressions that compose the filter.

For each element of the list, the *setof* function:

- Binds each element to the local variable.

- Evaluates the expressions in its body.

- Uses the result of the last expression to determine whether to include the element.

The *setof* function returns the list of elements for which the last body expression returned a non-*nil* value.

The following nongraphic session uses the trace facility to illustrate how the *setof* function works.

```
> tracef( t )
t
> setof( x '( 1 2 3 4 5 ) oddp( x ) )
|oddp(1)
|oddp --> t
|oddp(2)
|oddp --> nil
|oddp(3)
|oddp --> t
|oddp(4)
|oddp --> nil
|oddp(5)
|oddp --> t
(1 3 5)
> untrace()
t
```

A *setof* Example

Suppose you want to retrieve all the rectangles in a design.

The following code uses the *setof* function to retrieve all rectangles in a design.

```
cv = geGetWindowCellView()
setof( shape cv~>shapes shape~>objType == "rect" )
```

- The *cv~>shapes* expression retrieves the list of shape database objects.

- The *shape~>objType == "rect"* expression returns *t/nil* if the database object is/isn't a rectangle.

List Construction   8-17

Another *setof* Example

Suppose you want to compute the intersection of two lists.

One way to proceed is to use the *cons* function as follows:

```
procedure( TrIntersect( list1 list2 )
  let( ( result )
    foreach( element1 list1
      when( member( element1 list2 )
        result = cons( element1 result )
      ) ; when
    ) ; foreach
    result
  ) ; let
) ; procedure
```

The more efficient way is to use the *setof* function.

```
procedure( TrIntersect( list1 list2 )
  setof(
    element list1
    member( element list2 ))
) ; procedure
```

List Construction   8-19

Transforming Elements of a Filtered List

You can build a list by transforming elements of a filtered list.

setof   *foreach mapcar*
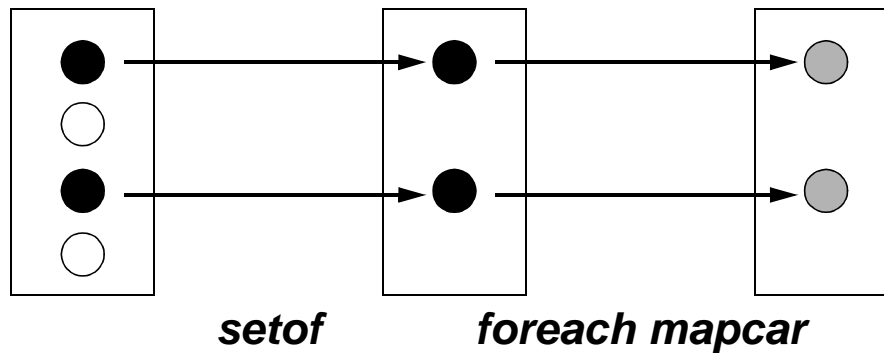
■ Use the *setof* function to filter the list.

■ Use the *foreach mapcar* function to build the resulting list.

This example computes the list of squares of odd integers.

```
procedure( TrListOfSquares( aList )
  let( ( filteredList )
    filteredList =
      setof( element aList oddp( element ))
    foreach( mapcar element filteredList
      element * element
    ) ; foreach
  ) ; let
) ; procedure

TrListOfSquares( '( 1 2 3 4 5 6 )) => ( 1 9 25 )
```

Lab Overview

Lab 8-1 Revising the Layer Shape Report

Rewrite your Shape Report program to count shapes by using the *length* and *setof* functions. For example, to count the rectangles, use this code:

```
rectCount = length(
    setof( shape cv~>shapes shape~>objType == "rect" )
)
```

Lab 8-2 Describing the Shapes in a Design

Develop a function *TrShapeList* to return a list of shape descriptions for all the shapes in a design. A shape description is a list that identifies the object type, the layer name, and the layer purpose as in this example:

```
( "rect" "metal1" "drawing" )
```

List Construction   8-23

Module Summary

In this module, we covered

■ The *foreach mapcar* function to build lists

■ The *setof* function to make a filtered copy of a list

■ The *foreach mapcar* function and the *setof* function used together to build a list by transforming each element

List Construction   8-25

Menus

Appendix A

April 30, 2007

Module Objectives

■ Create and display pop-up menus.

■ Create pull-down menus.

■ Install and remove pull-down menus from window banners.

Menus A-3

Terms and Definitions

Callback	SKILL® code that the system calls when the user does something in the user interface.

Creating a Pop-Up Menu

Use the *hiCreateSimpleMenu* function to build a pop-up menu with choices.

This example creates a pop-up menu with *Example Menu* as the title.

```
hiCreateSimpleMenu(
   'TrExampleMenu
   "Example Menu"
   '( "One" "Two" )
   '( "println( \"One\" )" "println( \"Two\" )" )
) => hiMenu@0x3b2aae8
```

Use the *hiDisplayMenu* function to display the menu. Once it's displayed,

- Choosing *One* causes *println("One")* to execute.

- Choosing *Two* causes *println("Two")* to execute.

```
┌─────────────────┐
│  Example Menu   │
├─────────────────┤
│      One        │
│      Two        │
│                 │
│                 │
└─────────────────┘
```
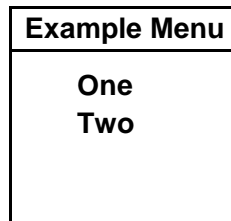
The *hiCreateSimpleMenu* Function

Example Arguments	Meaning
'TrExampleMenu	The menu variable. *hiCreateSimpleMenu* stores the data structure of the menu in this variable. *TrExampleMenu => hiMenu@0x3b2aae8*
"Example Menu"	The title of the menu.
'("One" "Two")	The list of the choices. No repetitions.
'("println(\"One\")" "println(\"Two\")")	The list of callbacks. Each callback is a string representing a single expression. Use \" to embed a single quote. Use curly braces, **{ }**, to group multiple expressions into a single expression.

Displaying a Pop-Up Menu

Use the *hiDisplayMenu* function to display a pop-up menu.

Pass the menu data structure to the *hiDisplayMenu* function.

This example defines a bindkey for the Schematics application. The bindkey displays your pop-up menu.

```
hiSetBindKey( "Schematics"
   "Shift Ctrl<Btn2Down>(2)"
   "hiDisplayMenu( TrExampleMenu )" )
```

hiDisplayMenu

The *hiDisplayMenu* Function

Example Argument	Meaning
TrExampleMenu	Use the variable you passed to *hiCreateSimpleMenu*. Do not apply the ' operator to the *TrExampleMenu* variable because you want to refer to the value of the variable.

Creating a Pull-Down Menu

General characteristics of a pull-down menu include the following:

■ You can insert a pull-down menu in one or more menu banners or in slider menus.

■ These menus can contain textual, iconic, or slider items.

Use the *hiCreatePulldownMenu* function to create a pull-down menu.

Use the *hiCreateMenuItem* function to create the menu items.

```
TrMenuItemOne = hiCreateMenuItem(
    ?name        'TrMenuItemOne
    ?itemText    "One"
    ?callback    "println( \"One\" )"
)
```

Pass the list of menu items to the *hiCreatePulldownMenu* function.

```
hiCreatePulldownMenu(
    'TrPulldownMenu       ;;; the menu variable
    "Example Menu"        ;;; menu title
    list( TrMenuItemOne TrMenuItemTwo )
)
```

The *hiCreateMenuItem* Function

Formal Argument	Actual Argument	Meaning
?name	*'TrMenuItemOne*	The symbolic name of the item. Quote this variable. Make sure that you also store the return result in this variable.
?itemText	*"One"*	The text that appears on the menu.
?callback	*"println(\"One\")"*	The action triggered by this menu item.

The *hiCreatePulldownMenu* Function

Example Argument	Meaning
'TrPulldownMenu	The menu variable. Quote this variable. The function stores the pull-down data structure menu in this variable.
"Example Menu"	The menu title.
list(TrMenuItemOne TrMenuItemTwo)	The list of menu items.

Inserting a Pull-Down Menu

Assume that a single window is already open.

Use the *hiInsertBannerMenu* function to insert a pull-down menu in the window menu banner.

The following line inserts a pull-down menu in the leftmost position of the CIW:

```
hiInsertBannerMenu( window( 1 ) TrPulldownMenu 0 )
```

If you attempt to insert a pull-down menu that is already present in a window banner, the following actions occur:

■ You get a warning message.

■ Your request is ignored.

To replace a pull-down menu, first delete the menu you want to replace.

The *hiInsertBannerMenu* Function

Example Argument	Meaning
window(1)	The window ID
TrPulldownMenu	The data structure for the pull-down menu. Use the variable you passed to *hiCreatePulldownMenu.* Do not apply the ' operator to the *hiCreatePulldownMenu* variable because you want to refer to its contents.
0	The index of the menu after it has been inserted. 0 is the leftmost.

The *hiInsertBannerMenu* function only works on a single window.

Deleting a Pull-Down Menu

Use the *hiDeleteBannerMenu* function to remove a pull-down menu from the window banner.

The pull-down menus on the right move one position to the left.

This example removes the leftmost menu in the CIW.

```
hiDeleteBannerMenu( window( 1 ) 0 )
```

Notice that the *hiDeleteBannerMenu* function requires the index of the pull-down menu. You can use the result of *hiGetBannerMenus()* to figure out the index by counting down the list to the position of the menu. The first menu in the list is index 0.

The *hiDeleteBannerMenu* Function

Example Argument	Meaning
window(1)	The window ID.
0	The index of the menu. 0 is the leftmost.

Lab Overview

Lab A-1 Exploring Menus
○ Exploring Pop-Up Menus
○ Exploring Pull-Down Menus

Menus   A-15

Module Summary

In this module, we covered

- ■ Creating and displaying a pop-up menu

- ■ Creating a pull-down menu

- ■ Installing a pull-down menu in a window banner

- ■ Deleting a pull-down menu from a window banner

Function	Description
hiCreateSimpleMenu	Builds a pop-up menu with text choices only.
hiCreateMenu	Builds a pop-up menu text or icon choices.
hiDisplayMenu	Displays a pop-up menu built by *hiCreateSimpleMenu* or *hiCreateMenu*.
hiCreateMenuItem	Builds a menu item. Several menus can share this item simultaneously.
hiCreatePulldownMenu	Builds a pull-down menu with text or icon choices.
hiInsertBannerMenu	Installs a pull-down menu in a window banner.
hiDeleteBannerMenu	Removes a pull-down menu from the banner.

Customization

Appendix B

April 30, 2007

Module Objectives

- Provide an overview of the Virtuoso® Design Environment initialization sequence.

- Basic tasks to accomplish in your *.cdsinit* file.

- Survey several useful SKIL® functions.

- Optional tasks to accomplish in your *.cdsinit* file.

- Preview the laboratory exercise.

Terms and Definitions

Context	A SKILL context is a binary file containing compiled SKILL code. Each SKILL application is associated with one or more contexts.

Virtuoso Design Environment Initialization Sequence

Cadence partitions applications into one or more SKILL contexts. A SKILL context is a binary file containing compiled SKILL code and SKILL data structures.

When you start the Virtuoso Design Environment, it loads these items:

1. Loads one or more SKILL contexts.

2. Loads all *.cdsenv* files.

3. Loads the *.cdsinit* customization file.
   The search order for the *.cdsinit* file is <install_dir>/tools/dfII/*local*, the current directory, and the home directory.
   Once the environment finds a *.cdsinit* file, the search stops. However, a *.cdsinit* can load other *.cdsinit* files.

4. Responds to user activity by loading other SKILL contexts.

```
  ●1        ●1        ●1        ●1        ●2        ●3        ●4
PRshare  amSynUI  schView  selectSv  .cdsenv  .cdsinit  schematic
```
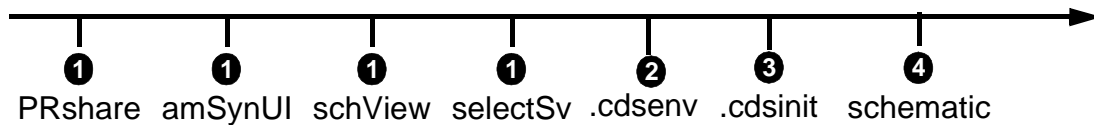
Customization   B-5

The context files are in the *<install_dir>/tools.dfII/etc/context* directory.

Each executable loads different contexts.

- Study the log file to determine the contexts that your executable loads prior to the *.cdsinit* file.

- The *icfb* executable loads these contexts, as the following excerpt for the *CDS.logs* indicates.

```
Loading PRshare.cxt
Loading amSynUI.cxt
Loading schView.cxt
Loading selectSv.cxt
```

The *.cdsinit* File

The site administrator has three strategies to control the user customization.

	Policy	Customization Strategy
1.	The site administrator does all the customization.	The <install_dir>/tools/dfII*/local/.cdsinit* contains all customization commands. There are no *./.cdsinit* or *~/.cdsinit* files involved.
2.	The site administrator does some customization. Each user can customize further.	The <install_dir>/tools/dfII*/local/.cdsinit* file contains a command to load the *./.cdsinit* or *~/.cdsinit* files
3.	Each user does all his/her own customization.	The <install_dir>/tools/dfII*/local/.cdsinit* file does not exist. All customization handled by either the *./.cdsinit* or *~/.cdsinit* files.

Study the <install_dir>/tools/dfII*/cdsuser/.cdsinit* file for sample user customizations.

Customization   B-7

The Basic *.cdsinit* Tasks

Make the *.cdsinit* file accomplish these three tasks:

- Set the SKILL path.
  The SKILL path is a list of directories containing SKILL source code and data files. The *load, loadi, view,* and other file system interface functions refer to the SKILL path to resolve relative pathnames.

- Load your SKILL utilities.
  Use the *load* function or the *loadi* functions.

- Define application bindkeys.
  Use the either the *hiSetBindKey* function or the *hiSetBindKeys* functions.

For maximum flexibility, implement your *.cdsinit* file so that another *.cdsinit* file can load it as in these examples:

- Another *.cdsinit* file can set the SKILL path before loading your *.cdsinit* file.

- Set the SKILL path to the current SKILL path plus the directories that contain your SKILL source code files.

Customization   B-9

The SKILL Path

The *getSkillPath* Function

The *getSkillPath* function returns a list of directory pathnames in search order.

The *setSkillPath* Function

The *setSkillPath* function sets the path to a list of directory pathnames.

Loading SKILL Source Code

Use the *load* or *loadi* functions to load SKILL source code.

- The *load* functions returns *t* if all SKILL expressions run without errors. Any error causes the *load* function to abort.

- The *loadi* function executes every SKILL expression in the file, regardless of errors.

Example

Load standard Virtuoso® Schematic Editor bindkey definitions.

```
load( prependInstallPath( "samples/local/schBindKeys.il" ) )
```

Customization   B-11

Standard Bindkey Definitions

These files contain standard bindkey definitions for the Virtuoso software.

Application	Pathname
Virtuoso Schematic Editor	<install_dir>/tools/dfII/*samples/local/schBindKeys.il*
Virtuoso Layout Editor	<install_dir>/tools/dfII/*samples/local/leBindKeys.il*

Other Useful *.cdsinit* Functions

SKILL functions can answer the following questions:

- What is the installation path?

- What is the value of a particular shell environment variable?

- Does a particular file exist?

- What are all the files in a directory?

Use the Cadence® online documentation to study the functions on the next few pages.

Determining the Installation Path

Use the *prependInstallPath* function to make a pathname relative to the Virtuoso Design Environment installation directory. The function prepends *<install_dir>/tools/dfII* to the path name.

Example

This example code adds three directories to the front of the current SKILL path. The three directories are in the installation hierarchy.

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

setSkillPath(       ;;; requires a list of strings
   append(
      TrSamplesPath ;;; a list of strings
      getSkillPath();;; the current SKILL path
   ) ; append
) ; setSkillPath
```

Customization

Interfacing with UNIX Files and Directories

Use the *simplifyFilename* function to determine the full pathname to a file. It correctly deals with the following items:

- The user's home directory

- The current working directory

- The SKILL path

Example

```
simplifyFilename( "~/SKILL" ) => "/usr1/mnt/skilldev/SKILL"
simplifyFilename( "./.cdsinit" )=>
   "/usr1/mnt/skilldev/SKILL/.cdsinit"
simplifyFilename( "Menus" ) =>
   "/usr1/mnt/skilldev/SKILL/Menus"
```

Use the *getDirFiles* function to retrieve the files and subdirectories in a directory.

Example

```
getDirFiles( "Menus" ) =>
   ("." ".." "Solutions" "Pulldown.il" "PopUp.il"
       "MenuItemUtilities.il" "SimpleMenuExample.il" )
```

The *simplifyFilename* Function

The *simplifyFilename* function expands the specified path to a full path. The tilde, /, and ./ are resolved, and redundant backslashes are removed. This shows exactly where the files are accessed for reporting information or error reporting by functions that require paths.

The *isFile* Function

The *isFile* function returns *t* if the files exists and returns *nil* otherwise. Use the *isFile* function to test for the existence of the file in the SKILL search path before attempting to load the file.

```
when( isFile( "SomeUtilities.il" )
   loadi( "SomeUtilities.il" ) ) ; when
```

The *when* Function

The *when* function evaluates the first expression to determine whether or not to evaluate the subsequent expressions in its body.

The *isDir* Function

The *isDir* function returns t if the directory exists and returns nil otherwise.

The *getDirFiles* Function

The *getDirFiles* function returns the list of files in the directory or subdirectory.

Reading Shell Environment Variables

Use the *getShellEnvVar* function to determine the value of a shell environment variable.

For example, the following line returns the value of the USER variable.

```
getShellEnvVar("USER")=> "user1"
```

Manipulating a List of Strings

Use the *buildString* function to make a single string from a list of strings. The *buildString* function provides separating space by default.

Example

```
buildString( '( "a" "b" "c" "d" )) => "a b c d"
buildString( '( "a" "b" "c" "d" ) "/" ) => "a/b/c/d"
```

Use the *parseString* function to parse a single string into a substring based on delimiter characters.

Example

```
parseString( "a b c d" ) => ( "a" "b" "c" "d" )
parseString( "a/b/c/d" "/" ) => ( "a" "b" "c" "d" )
```

Customization   B-21

Manipulating the CIW in the *.cdsinit* File

You can accomplish the following optional tasks in your *.cdsinit* file:

■ Positioning the CIW.

```
hiResizeWindow(window(1) list( 10:10 750:200))
```

■ Adding pull-down menus to the CIW. Consider the following example from the previous module.

```
hiInsertBannerMenu( window( 1 ) TrPulldownMenu 0 )
```

■ Removing pull-down menus from the CIW menu banner. Consider the following example from the previous module.

```
hiDeleteBannerMenu( window( 1 ) 0 )
```

Customization   B-23

SKILL Development *.cdsinit* Tasks

SKILL programmers can accomplish the following optional tasks in the *.cdsinit* file:

■ Set the log filter options.
The CIW will display the maximum amount of SKILL information.

```
hiSetFilterOptions( t t t t t t t )
```

■ Install the SKILL debugger.
The SKILL debugger will be available to trap errors.

```
installDebugger()
alias( q debugQuit )
alias( c continue )
```

Customization   B-25

Lab Preview

In the lab, you make several bindkey definitions immediately available when the Virtuoso Design Environment starts.

You define a bindkey for both the *Schematics* and *Layout* applications to raise the Command Interpreter Window.

You define a bindkey for the *Command Interpreter* application to raise the current window.

With a single key stroke, you can toggle back and forth between the current window and the CIW. This feature has the following benefits:

■ You can keep the CIW large without loosing track of your current window.

■ You can bury the CIW until you need it.

Customization

Lab Overview

Lab B-1 Defining Bindkeys in the .cdsinit File

Customization   B-29

Module Summary

In this module, we covered

■ The Virtuoso Design Environment initialization sequence

■ Basic tasks to accomplish in your *.cdsinit* file

■ Several useful SKILL functions

■ Optional tasks to accomplish in your *.cdsinit* file

■ A preview of the laboratory exercise

File I/O

Appendix C

April 30, 2007

Module Objectives

- Write UNIX text files.

- Read UNIX text files.

- Open a text window.

Writing Data to a File

Instead of displaying data in the CIW, you can write the data to a file.

Both *print* and *println* accept a second, optional argument that must 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.

This example uses the *for* function to execute the *println* function iteratively. The *i* variable is successively set to the values 1,2,3,4, and 5.

```
myPort = outfile( "/tmp/myFile" ) => port:"/tmp/myFile"
for( i 1 5
  println( list( "Number:" i) myPort )
  ) => t
close( myPort ) => t
myPort = nil
```

After you execute the *close* function, */tmp/myFile* contains the following lines:

```
("Number:" 1)
("Number:" 2)
("Number:" 3)
("Number:" 4)
("Number:" 5)
```

File I/O   C-5

The *print* and *println* Functions

Notice how the SKILL Evaluator displays a port in the CIW.

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

Use a full pathname with the *outfile* function. Keep in mind that *outfile* returns *nil* if you do not have write access to the file, or if you cannot create the file in the directory you specified in the pathname.

The *print* and *println* functions raise an error if the port argument is *nil*. Observe that the type template uses a *p* character to indicate a port is expected.

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

The *close* Function

The *close* function does not update your port variable after it closes the file. To facilitate robust program logic, set your port variable to *nil* after calling the *close* function.

Writing Data to a File (continued)

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. The first argument must be an output port associated with the file.

Example

```
myPort = outfile( "/tmp/myFile" )
for( i 1 5
   fprintf( myPort "\nNumber: %d" i )
) ; for
close( myPort )
```

The example above writes the following data to */tmp/myFile*:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Reading Data from a File

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 *~/.cshrc* to the CIW.

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

The *infile* Function

The *gets* Function

The *gets* function reads the next line from the file. The arguments of the *gets* function are the:

- Variable that receives the next line.

- Input port.

The *gets* function returns the text string or returns *nil* when the end of file is reached.

The *when* Function

The first expression within a *when* expression is a condition. The SKILL Evaluator evaluates the condition. If it evaluates to a non-*nil* value, then the SKILL Evaluator evaluates all the other expressions in the *when* body. The *when* function returns either *nil* or the value of the last expression in the body.

The *while* Function

The SKILL Evaluator repeatedly evaluates all the expressions in the *while* body as long as the condition evaluates to a non-*nil* value. The *while* function returns *nil*.

The *fscanf* Function

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

The arguments of *fscanf* are

■ The input port

■ The conversion control string

■ The variable(s) that receive(s) the matching data values.

The *fscanf* function returns the number of data items matched.

This example prints every word in *~/.cshrc* to the CIW.

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

The format directives commonly found include the ones in this table.

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

The following is an example of output from the *~/CDS.log*.

```
\o "#.cshrc"
\o "for"
\o "Solaris"
\o "#Cadence"
\o "Training"
\o "Database"
\o "setup"
\o "for"
\o "the"
\o "97A"
\o "release"
\o "#"
```

Opening a Text Window

Use the *view* function to display a text file in a read-only window.

```
view( "~/SKILL/.cdsinit" ) => window:5
```

The *view* function is very useful for displaying a report file to the user.

The *view* function resolves a relative pathname in terms of the SKILL® path. This is a list of directories you can establish in your *.cdsinit*.

See Module 7, *Customization*, for specifics on the SKILL path.

To select text from your *CDS.log* file, try the following:

```
view(
   "~/CDS.log" ;;; pathname to CDS.log
   nil           ;;; default location
   "Log File"    ;;; window title
   t             ;;; auto update
) => window:6
```

The *view* Function

The *view* function takes several optional arguments.

Argument	Status	Type	Meaning
file	required	text	Pathname
winSpec	optional	bounding box/*nil*	Bounding box of the window. If you pass *nil*, the default position is used.
title	optional	text	The title of the window. The default is the value of the file parameter.
autoUpdate	optional	*t/nil*	If *t*, then the window updates for each write to the file. The default is *nil*.
appName	optional	text	The Application Type for this window. The default is *"Show File"*.
help	optional	text	Text string for online help. The default means no help is available.

Lab Overview

Lab C-1 Writing Data to a File

Lab C-2 Reading Data from a Text File

Lab C-3 Writing Output to a File
> Enhance your *TrShapeReport* function to write the output to a file.

File I/O

C-15

1

Lab Overview

Lab C-1 Writing Data to a File

Lab C-2 Reading Data from a Text File

Lab C-3 Writing Output to a File
> Enhance your *TrShapeReport* function to write the output to a file.

File I/O

C-15

Module Summary

In this module, we covered

- ■ Writing text data to a file by using
    - ❏ The *outfile* function to obtain an output port on a file
    - ❏ An optional output port parameter to the *print* and *println* functions
    - ❏ A required port parameter to the *fprintf* function
    - ❏ The *close* function to close the output port

- ■ Reading a text file by using
    - ❏ The *infile* function to obtain an input port
    - ❏ The *gets* function to read the file a line at a time
    - ❏ The *fscanf* function to convert text fields upon input
    - ❏ The *close* function to close the input port