

Cadence® SKILL Language Programming

Version 5.0

Lab Manual

November 22, 2002

© 1990-2002 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

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

Cadence Trademarks

Alanza SM	Gate Ensemble [®]	Quest [®]
Allegro [®]	HDL-ICE [®]	Quickturn [®]
Ambit [®]	how big can you dream? TM	Radium TM
Assura TM	Integration Ensemble TM	SignalStorm TM
BuildGates [®]	MegaSim TM	Silicon Ensemble [®]
Cadence [®] (brand and logo)	Mercury TM	SoC Encounter TM
CeltIC TM	NC-Verilog [®]	SourceLink [®] online customer support
ClockStorm TM	OpenBook [®] online documentation library	SPECCTRA [®]
CoBALT TM	Orcad [®]	SPECCTRAQuest TM
Concept [®]	Orcad Capture [®]	Spectre [®]
Connections [®]	Orcad Layout [®]	TtME [®]
Diva [®]	Palladium TM	Vampire [®]
Dracula [®]	Pearl [®]	Verifault-XL [®]
ElectronStorm TM	PowerSuite TM	Verilog [®]
Fire & Ice [®]	PSpice [®]	Verilog [®] -XL
First Encounter [®]	Q/Bridge [®]	Virtuoso [®]
FormalCheck [®]	QPlace [®]	VoltageStorm TM

Other Trademarks

All other trademarks are the exclusive property of their respective owners.

Confidentiality Notice

No part of this publication may be reproduced in whole or in part by any means (including photocopying or storage in an information storage/retrieval system) or transmitted in any form or by any means without prior written permission from Cadence Design Systems, Inc. (Cadence).

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

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

UNPUBLISHED This document contains unpublished confidential information and is not to be disclosed or used except as authorized by written contract with Cadence. Rights reserved under the copyright laws of the United States.

Table of Contents

SKILL Language Programming

Module 1 About This Course

Lab 1-1	No Labs for this Module	1-1
---------	-------------------------------	-----

Module 2 SKILL Statements and Variables

Lab 2-1	Starting the Software.....	2-1
	Starting the Environment	2-1
	Opening an xterm Window Using CDE	2-1
	Opening an xterm Window	2-1
Lab 2-2	Using the Command Interpreter Window.....	2-3
	Starting the Cadence Software.....	2-3
	Exploring the CIW Pull-Down Menus	2-4
	Setting the Log File Display Filter Options.....	2-4
	Scrolling the CIW Output Pane	2-5
	Editing Your <i>.cdsinit</i> File.....	2-5
	Entering SKILL Expressions in the CIW Input Pane	2-6
	Examining the Log File	2-6
	Pasting Previous Output into the CIW Input Pane.....	2-6
Lab 2-3	Exploring SKILL Numeric Data Types.....	2-8
	Entering Numeric Data	2-8
	Using Arithmetic Operators.....	2-8
	Observing Operator Precedence	2-9
	Observing SKILL Evaluation	2-9
	Controlling the Order of Evaluation	2-10
Lab 2-4	Exploring SKILL Variables.....	2-13
	Initializing a Variable	2-13
	Retrieving the Value of a Variable	2-13
	Incrementing a Variable.....	2-13
	Checking the Type of a Variable	2-14
Lab 2-5	Displaying Data in the CIW.....	2-15
	Using the <i>println</i> Function.....	2-15
	Using the <i>printf</i> Function.....	2-15

Lab 2-6	Solving Common Input Errors.....	2-18
	Confirming that the SKILL Evaluator Is Available.....	2-18
	Resolving Unbalanced String Quotes and Parentheses	2-18
	Resolving Problems with Inappropriate White Space	2-19
	Inappropriate Space Again.....	2-20
	Resolving Data Error Messages.....	2-20

Module 3 Lists

Lab 3-1	Creating New Lists	3-1
	Creating a New List Using the ' Operator	3-1
	Controlling the Number of List Items per Line	3-2
	Creating a New List Using the <i>list</i> Function.....	3-3
	Building a List Using the <i>cons</i> Function	3-4
	Building a List Using the <i>append</i> Function	3-5
	Comparing the <i>cons</i> and <i>append</i> Functions	3-5
	Exploring Restrictions for the <i>cons</i> and <i>append</i> Functions	3-5
	Adding an Element to the End of a List.....	3-6

Module 4 Windows

Lab 4-1	Opening Windows	4-1
	Opening a Design Window	4-1
	Opening a Text Window	4-2
Lab 4-2	Resizing Windows	4-3
	Retrieving the Bounding Box of a Window	4-3
	Resizing a Window	4-4
Lab 4-3	Storing and Retrieving Bindkeys.....	4-6
	Locating an Available Key	4-6
	Establishing the Bindkey Definitions	4-7
	Testing Schematic and CIW Bindkeys	4-8
	Testing Layout and CIW Bindkeys.....	4-8
Lab 4-4	Defining a Show File Bindkey.....	4-10
	Defining Your Bindkey.....	4-10
	Testing Your Bindkey Definition	4-10

Module 5 Database Queries

Lab 5-1	Querying Design Databases.....	5-1
	Opening a Design.....	5-1
	Retrieving the <i>cellView</i> Database Object.....	5-2
	Making Queries.....	5-2
	Querying the Other Designs.....	5-4

Module 6 Menus

Lab 6-1	Exploring Menus.....	6-1
	Examining the Code for a Pop-Up Menu	6-1
	Running the Code	6-2
	Examining the Code for a Pull-Down Menu	6-2
	Running the Code	6-3

Module 7 Customization

Lab 7-1	Defining Bindkeys in the <i>.cdsinit</i> File	7-1
	Editing Your <i>.cdsinit</i> File.....	7-1
	Testing the <i>.cdsinit</i> File.....	7-3

Module 8 Developing a SKILL Function

Lab 8-1	Developing a SKILL Function.....	8-1
	Requirements	8-1
	Suggestions	8-1
	Setting the <i>writeProtect</i> Switch	8-2
	Examining the SKILL Path.....	8-2
	Editing the Source Code File	8-3
	Writing the Source Code.....	8-3
	Loading Your Function.....	8-4
	Testing Your Solution.....	8-4
	Test Case Results	8-5

Module 9 Flow of Control

Lab 9-1	Writing a Database Report Program.....	9-1
	Requirements	9-1

Lab 9-2	Exploring Flow of Control.....	9-6
	Requirements	9-6
	Recommendations.....	9-6
	Testing Your Solution.....	9-7
Lab 9-3	More Flow of Control	9-9
	Requirements	9-9
	Testing Your Solution.....	9-10
Lab 9-4	Controlling Complex Flow	9-12
	Requirements	9-12
	Suggestions	9-13
	Testing Your Solution.....	9-14

Module 10 File I/O

Lab 10-1	Writing Data to a File	10-1
	Obtaining an Output Port on a File	10-1
	Writing Data to the File	10-1
	Closing the File	10-2
	Viewing the File.....	10-3
Lab 10-2	Reading Data from a Text File.....	10-4
	Obtaining an Input Port on a File.....	10-4
	Closing the File	10-6
	Reading the Data From a Text File.....	10-6
	Closing the File	10-7
	Reading Numeric Data from a Text File.....	10-7
Lab 10-3	Writing Output to a File.....	10-9
	Testing Your Solution.....	10-11
	Solutions	10-13

Module 11 SKILL Development Environment

Lab 11-1	Analyzing an Error.....	11-1
	Installing the SKILL Debugger	11-1
	Adjusting the <i>tracelevel</i> and <i>tracelength</i> Variables.....	11-1
	Loading the Program.....	11-2
	Opening an Example Design	11-2
	Generating the Error	11-2
	Displaying the Stack	11-3
	Determining the Cause of the Error	11-3
	Quitting the Debugger.....	11-3
	Editing and Loading the Fixed Program.....	11-4
	Testing the Fixed Program.....	11-4
Lab 11-2	Debugger Sessions.....	11-6
	Verifying the Window ID of the Design Window.....	11-6
	Verifying that the SKILL Debugger Is Installed	11-6
	Loading the Application	11-7
	Setting a Breakpoint.....	11-7
	Running the Shape Report Program	11-8
	Examining Function Call Arguments	11-8
	Continuing Execution	11-9
	Running the Shape Report Program Again.....	11-9
	Using the Step Command	11-9
	Using the Next Command.....	11-10
	Stepping into a <i>foreach</i> Loop.....	11-10
	Using the Step Out Command	11-10
Lab 11-3	Using SKILL Lint.....	11-12
	Running SKILL Lint.....	11-12
	Examining the SKILL Lint Output.....	11-12
	Resolving SKILL Lint Problems	11-13

Module 12 List Construction

Lab 12-1	Revising the Layer Shape Report	12-1
	Requirements	12-1
	Testing Your Solution.....	12-1
Lab 12-2	Describing the Shapes in a Design.....	12-3
	Requirements	12-3
	Recommendations.....	12-3
	Testing Your Solution.....	12-4

Module 13 Data Models

Lab 13-1	Enhancing the Layer Shape Report SKILL Program	13-1
	Requirements	13-1
	Suggestions	13-2
	Testing Your Solution.....	13-2
	Sample Solution.....	13-3
Lab 13-2	Creating a Cellview.....	13-4
	Requirements	13-4
	Getting Started	13-4
	Writing Your Function.....	13-5
	Testing Your Solution.....	13-5
Lab 13-3	Aligning Rectangles.....	13-8
	Requirements	13-8
	Testing Your Solution.....	13-8
Lab 13-4	Exploring Hierarchy.....	13-11
	Opening a Design.....	13-11
	Retrieving a Cellview Database Object from a Window	13-11
	Examining the Instance Masters	13-12
	Examining an Instance.....	13-12
	Querying the Master	13-13
	Viewing the Master.....	13-13
Lab 13-5	Traversing a Hierarchical Design	13-15
	Opening a Design.....	13-15
	Closing All the Master Cellviews.....	13-17
	Checking for Purged Cellview Database Objects.....	13-17
Lab 13-6	Developing a Netlister	13-19
	Requirements for the <i>TriNetList</i> Function	13-19
	Requirements for the <i>TrNetList</i> Function	13-19
	Testing Your Solution.....	13-20
	Sample Solution.....	13-22
Lab 13-7	Developing an Instance-based Netlister.....	13-23
	Requirements for the <i>TriInstNetList</i> Function	13-23
	Requirements for the <i>TrInstNetList</i> Function	13-23
	Testing Your Solution.....	13-24
	Sample Solution.....	13-27

Lab 13-8	Exploring User-Defined Properties.....	13-28
	Opening a Design.....	13-28
	Displaying Property Names and Values	13-28
	Building a List of Property Name and Value Pairs.....	13-29
Lab 13-9	Dumping Database Objects.....	13-32
	Opening a Design.....	13-32
	Using the ~> Operator to Dump Database Objects	13-32
	Using the Show File Browser	13-33
	Browsing the First Instance	13-34
	Navigating Among Show File Browser Windows.....	13-35
	Raising the CIW.....	13-35
	Clearing the Selection.....	13-35
Lab 13-10	Building the Cellview Data Model Road Map	13-37

Module 14 User Interface

Lab 14-1	Exploring Fixed Menus.....	14-1
	Building a Sample Fixed Menu	14-1
	Displaying the Fixed Menu.....	14-2
	Opening a Design Window.....	14-3
	Retrieving the Attached Fixed Menu.....	14-3
	Removing the Fixed Menu.....	14-3
	Attaching the Sample Fixed Menu	14-4
	Restoring the Original Fixed Menu	14-4
Lab 14-2	Exploring Dialog Boxes.....	14-5
	Using a Modeless Dialog Box	14-5
	Using a Modal Dialog Box	14-6
	Using a System Modal Dialog Box	14-6
Lab 14-3	Exploring List Boxes	14-8
	Examining a Simple List Box Application	14-8
	Exploring an Advanced List Box Application.....	14-9
Lab 14-4	Exploring Forms	14-10
	Examining the File Form Application	14-10
	Sampling Various Form Fields	14-10

Lab 14-5	Writing a Form Application.....	14-11
	Requirements	14-11
	Assumptions.....	14-11
	Suggestions	14-12
	Developing the <i>TrCreateWindowForm</i> Function	14-13
	Developing the <i>TrWindowFormCB</i> Function.....	14-14
	Testing Your Solution.....	14-14
	Optional Enhancements	14-15
	Generating Form Symbols	14-15
	Testing Your Solution.....	14-16
	Sample Solution.....	14-17
 Module 15 Advanced Customization		
Lab 15-1	Adding a Menu Item to the Composer Edit Menu.....	15-1
	Studying the SKILL Source Code	15-1
	Loading the SKILL Source Code	15-1
	Checking the Trigger Function	15-1
	Opening a Design.....	15-2
	Descending the Hierarchy.....	15-2
	Ascending the Hierarchy.....	15-3
Lab 15-2	Adding a Pull-Down Menu to a Composer Window.....	15-5
	Studying the SKILL Source Code	15-5
	Loading the SKILL Source Code	15-5
	Checking the User Postinstall Trigger	15-5
	Opening a Design.....	15-6
	Descending the Hierarchy.....	15-6
	Ascending the Hierarchy.....	15-7
Lab 15-3	Reversing the Layout Editor Pull-Down Menus.....	15-9
	Studying the SKILL Source Code	15-9
	Loading the SKILL Source Code	15-9
	Opening a Design.....	15-9
	Descending the Hierarchy.....	15-10
	Ascending the Hierarchy.....	15-10
Lab 15-4	Customizing the Initial Window Placement	15-12
	Requirements	15-12
	Planning Your Solution	15-12
	Developing Your Trigger Function	15-13

Module 16 Interprocess Communication

Lab 16-1	Compiling a C Program	16-1
	Studying the <i>IPC/toUpperCase.c</i> Program.....	16-1
	Compiling the <i>IPC/toUpperCase.c</i> Program	16-1
	Running the <i>IPC/toUpperCase.out</i> Program	16-1
Lab 16-2	Exploring Synchronous Communication.....	16-3
	Studying the SKILL Sample Source Code	16-3
	Loading the SKILL Sample Source Code	16-4
	Running the Example.....	16-4
	Tracing the Example	16-4
Lab 16-3	Exploring Asynchronous Communication.....	16-6
	Studying the SKILL Sample Source Code	16-6
	Loading the SKILL Sample Source Code	16-6
	Running the Example.....	16-7
	Tracing the Example	16-7

Module 17 Data Structures



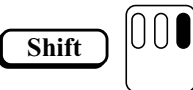


	Introduction.....	17-1
Lab 17-1	Exploring Associative Tables	17-2
	Requirements	17-2
	Suggestions	17-2
	Testing Your Solution.....	17-3
Lab 17-2	Exploring Association Lists.....	17-6
	Requirements	17-6
	Suggestions	17-6
	Testing Your Solution.....	17-8
Lab 17-3	Exploring Disembodied Property Lists.....	17-12
	Requirements	17-12
	Suggestions	17-12

Module 18 OpenAccess

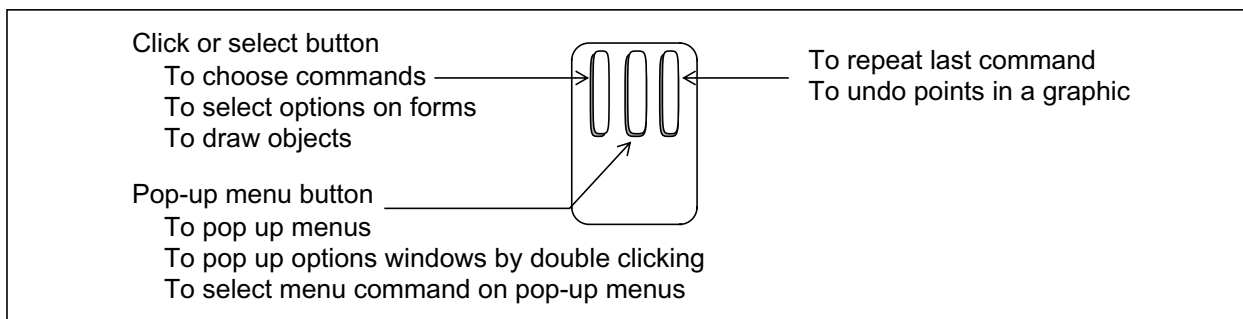
Lab 18-2	No Labs for this Module	18-1
----------	-------------------------------	------

Mouse Use and Terminology

Use the mouse to move the cursor on the screen, to make selections from a menu, and to draw.

Term	Action	Icon Example
click	Quickly press and release the specified mouse button. On menus and forms, you use the <u>left</u> mouse button most of the time.	 Click <u>left</u>
double click	Rapidly press the specified mouse button twice.	 Double click
Shift-click Control-click Shift-Control-click	Hold down the appropriate key or keys and click a specified mouse button.	 Shift- click <u>right</u>
draw through	Define a box by pressing the mouse button at one corner of the box, moving to the diagonally opposite corner of the box with the mouse button held down, and releasing the button.	 Draw through
pop up	Press the <u>middle</u> mouse button.	 Click <u>middle</u>
pull down	Move the mouse cursor to the menu name on the menu banner, press and hold the <u>left</u> mouse button, move the cursor down to highlight the menu selection, release the mouse button to execute the selection.	
Enter	Type a command in an xterm window and press Return to execute the command.	
Select	Position the cursor over a command and press the <u>left</u> mouse button. Choose or pick are synonyms for select.	

The basic uses of mouse buttons are shown in this graphic.

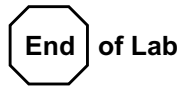


Labs for Module 1

About This Course

No Labs for This Module

Lab 2-1 No Labs for this Module



Labs for Module 2

SKILL Statements and Variables

Lab 2-1 Starting the Software

Objective: Login and start a Design Framework II executable.

Starting the Environment

1. Log on to the system. At the **Login** prompt, enter your login ID.

`user1`

2. At the **Password** prompt, enter your password

`training`

The prompt now identifies the host machine: `cdsXXX>`

Opening an xterm Window Using CDE

If you are not using CDE go to the section named **Opening an xterm Window**.

1. Move the cursor to an empty area of the screen.
2. Display the Root Menu by holding down the right mouse button.
3. Select **Programs** and then **Terminal**
4. Go to the section titled **Verifying the Results**.

Opening an xterm Window

1. Start the X Window System. Enter:

`xwin`

2. Move the cursor to an empty area of the screen.
3. Display the Root Menu by holding down the middle mouse button.

4. Release the middle mouse button on **New Window**.

An *xterm* window opens in the upper left corner of the screen.

5. Continue with the next section.

Verifying the Results

1. In the *xterm* window, enter

```
ls ~/SKILL/cell_design
```

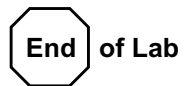
The window displays the following:

<code>cds.lib</code>	<code>cellTechLib</code>	<code>display.drf</code>
<code>master</code>	<code>pCells</code>	<code>tutorial</code>

2. In the *xterm* window, enter

```
ls ~/SKILL/display.drf
```

The system displays the path to the file.



Lab 2-2 Using the Command Interpreter Window

Objective: To start the Design Framework II environment and learn about the CIW.

Starting the Cadence Software

1. Change to the working directory in an *xterm* window. Enter
`cd SKILL`
2. Start the Cadence software in background mode in the same *xterm* window. Enter
`icfb &`

When the Cadence® Design Framework II environment starts, the Command Interpreter Window (CIW) opens at the bottom of the screen. The CIW is numbered 1 in the upper right corner. The CIW banner displays the names of a number of pull-down menus. You use these menus to open one of the following:

- A cellview for editing or viewing
- A cell library for browsing
- A text file for editing or viewing

Exploring the CIW Pull-Down Menus

Here is a summary of the CIW pull-down menus for the *icfb* executable.

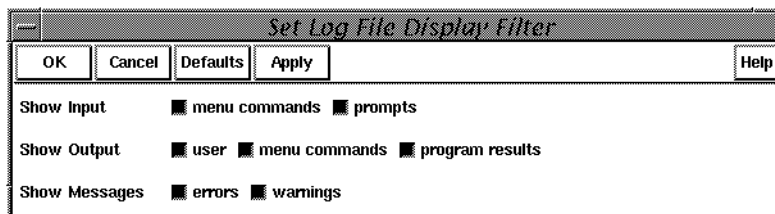
Menu Name	Most Common Use
File	Open or create a design.
Tool	Launch the Library Manager. Launch the SKILL Development Environment.
Options	
Technology File	Manage technology files.

Which menu has the User Preferences menu item?

Which menu has the Licenses item?

Setting the Log File Display Filter Options

1. Resize the CIW so that you can see at least 10 lines in the output field of the CIW.
2. In the CIW window banner, select **Options** → **Log Filter**.
The Set Log File Display Filter form appears.
3. Set the toggles buttons as shown.



4. Click **OK** to execute the command and remove the form from the screen.

The CIW output pane shows

- SKILL function calls
- SKILL output
- SKILL function results
- Warnings
- Error messages

Scrolling the CIW Output Pane

The CIW has a scroll bar on the right side.

1. Position the mouse pointer over the triangle at the top of the scroll bar and click the left mouse button.
2. With the pointer still over the triangle at the top, press and hold the left mouse button until you scroll all the way back.
3. Move the pointer over the rectangle in the scroll bar. Press the middle mouse button and drag the rectangle to the bottom of the scroll bar.

Editing Your *.cdsinit* File

You'll use *vi* to edit your *.cdsinit* file so that the next time you enter the environment, the CDS log file display filter will be set as it is now (which is not the default).

1. In the CIW input pane, enter

```
edit( "~/SKILL/.cdsinit" )
```

A *vi* window appears.
2. Find this line

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

3. By deleting the three semi-colons, change the line to

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

4. Save your edits and exit the editor.

You have uncommented a call to the *hiSetFilterOptions* function. The next time you enter the Cadence environment, the *hiSetFilterOptions* function will be invoked.

Entering SKILL Expressions in the CIW Input Pane

The Cadence® SKILL programming language supports the full-range of arithmetic operations.

1. In the CIW input pane, enter

```
6*5
```

The SKILL Evaluator executes *6*5* and displays your input, *6*5*, and the return result, *30*, in the CIW output pane.

Examining the Log File

Use the *view* function to open a text file.

1. In the CIW input pane, enter

```
view( "~/CDS.log" )
```

A Show File window appears displaying the file.

Can you find your input lines in the log file?

Pasting Previous Output into the CIW Input Pane

1. Click the left mouse button on the line *6*5* in the CIW output pane.

The line is pasted into the CIW input pane.

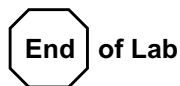
2. Press **Return** to execute the command.

The SKILL Evaluator executes *6*5* a second time and displays the return result, *30*, in the CIW output pane.

Lab Summary

In this lab, you

- Started the Design Framework II environment.
- Examined pull-down menus on the CIW.
- Set the Log File Display Filter options interactively.
- Scrolled the CIW output pane.
- Entered SKILL expressions in the CIW input pane.
- Pasted a line from the CIW output pane into the input pane to execute a SKILL expression again.



Lab 2-3 Exploring SKILL Numeric Data Types

Objective: Become familiar with fixed-point and floating-point numbers and arithmetic operators.

Entering Numeric Data

SKILL accepts a variety of means of denoting numeric data, including scaling factors and scientific notation.

1. In the CIW, enter

5
5.3
1e10
5M
3m
2.0n

What is displayed in the CIW output pane?

Using Arithmetic Operators

SKILL provides familiar operators for arithmetic computation. When all the operands are integers, SKILL returns an integer result.

1. In the CIW, enter

7 / 5

What is displayed in the CIW output pane?

2. In the CIW, enter

9 / 5

What is displayed in the CIW output pane?

If the operands include both integer and floating-point numbers, SKILL returns a floating-point result.

3. In the CIW, enter

$1+0$

What is displayed in the CIW output pane?

4. In the CIW, enter

$1+0.0$

What is displayed in the CIW output pane?

5. In the CIW, enter

$5 + 4.1$

$5 + 4$

$5 + 4.0$

What is displayed in the CIW output pane?

Observing Operator Precedence

SKILL operators have precedence. Can you predict the values of the following expressions?

1. In the CIW, enter

$5+3/2$

Which operator has higher precedence?

2. In the CIW, enter

$5*3/2$

3. In the CIW, enter

$5*3**2$

4. In the CIW, enter

$20-9/3*2**3$

Observing SKILL Evaluation

You can use the SKILL trace facility to verify the order of evaluation.

When the SKILL trace facility is active, SKILL notifies you in the CIW output pane before it evaluates an expression.

Note: *The notification includes one or more vertical bars that include the nesting depth of the expression within in the current top-level expression under evaluation.*

After evaluating the expression, SKILL notifies you of the return result.

Note: *The SKILL Parser translates each operator expression into a corresponding SKILL function call. The trace output displays the SKILL function that implements the operator.*

1. In the CIW, enter

```
tracef( t)
```

The trace facility is now active.

2. In the CIW, enter

```
20-9/3*2**3
```

The system displays the following in the CIW output pane.

```
20-9/3*2**3
| (9 / 3)
| quotient --> 3
| (2**3)
| expt --> 8
| (3 * 8)
| times --> 24
| (20 - 24)
| difference --> -4
-4
```

3. In the CIW, enter

```
untrace()
```

The SKILL trace facility is now inactive.

Controlling the Order of Evaluation

You can control the order of evaluation by enclosing expressions in parentheses.

1. In the CIW, enter

$(5+3)/2$

2. In the CIW, enter

$(5*3)**2$

Nesting parentheses redundantly can cause unpredictable results.

3. In the CIW, enter

$((5+3))/2$

SKILL responds with the following error

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

Can you correct this error?

Place parentheses appropriately in the expression

$20-9/3*2**3$

- to make it return

-196

- to make it return

2744

- to make it return

24

— Check your work against the solution on the next page.

Answers

1. In the CIW, enter

$(5+3)/2$

2. In the CIW, enter

$20-(9/3*2)**3$

3. In the CIW, enter

$(20-9/3*2)**3$

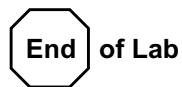
4. In the CIW, enter

$(20-9)/3*2**3$

Lab Summary

In this lab, you

- Explored arithmetic computations with fixed-point and floating-point numbers.
- Used the SKILL trace facility to observe the order of evaluation.
- Used parentheses to control the order of evaluation.



Lab 2-4 Exploring SKILL Variables

Objective: Store and retrieve values in SKILL variables.

Initializing a Variable

Use the = operator to assign a value to a SKILL variable.

1. In the CIW, enter
`lineCount = 0`

Retrieving the Value of a Variable

Use the variable name to retrieve the current value of the variable.

1. In the CIW, enter
`lineCount`

Incrementing a Variable

Use the SKILL pre-increment operator ++ to increment the value of a variable and return the new value.

1. In the CIW, enter
`++lineCount`
`lineCount`

Use the SKILL postincrement operator ++ to increment the value of a variable and to return the old value.

2. In the CIW, enter
`lineCount = 0`
`lineCount++`
`lineCount`
`lineCount++`
`lineCount`

Checking the Type of a Variable

A SKILL variable can store any type of data. Use the *type* function to tell what kind of data is currently in a variable.

1. In the CIW, enter

```
lineCount = 5
```

2. In the CIW, enter

```
type( lineCount )
```

What is displayed in the CIW output pane?

3. In the CIW, enter

```
lineCount = "many lines"
```

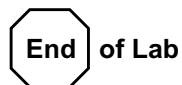
4. In the CIW, enter

```
type( lineCount )
```

What is displayed in the CIW output pane?

Lab Summary

In this lab, you manipulated SKILL variables.



Lab 2-5 Displaying Data in the CIW

Objective: To use *printf* and *println* functions.

In this lab, you use the *printf* and *println* to display the values of variables in the CIW output pane.

Store values in *x*, *y*, and *z*.

1. In the CIW, enter

```
x = 4
y = 5.3
z = "layout"
```

Using the *println* Function

The *println* function uses the argument's data type to determine how to display it's value.

1. In the CIW, enter

```
println( z )
```

SKILL displays the following in the CIW output pane:

```
"layout"
nil
```

nil is the return result.

Using the *printf* Function

You supply the desired format to the *printf* function.

1. In the CIW, enter

```
printf( "The value of x is %n" x )
```

SKILL displays the following in the CIW output pane:

```
The value of x is 4
t
```

t is the return result.

Use the *printf* Function to Format Your Output

1. Assign values to the *libName*, *cellName*, *viewName*, and *shapeCount* variables and use the *printf* function to display the following output in the CIW.

```
The design Cells inverters schematic has 200
shapes
```

2. Modify your solution to display the following line in the CIW output pane.

```
The design Cells inverters schematic has many
shapes
```

— Check your work against the solution on the next page.

Answers

1. In the CIW, enter

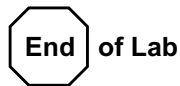
```
libName = "Cells"
cellName = "inverters"
viewName = "schematic"
shapeCount = 200
printf( "\nThe design %s %s %s has %d shapes"
        libName cellName viewName shapeCount )
```

2. Change the value of *shapeCount* and change the format specification from *%d* to *%s*.

```
libName = "Cells"
cellName = "inverters"
viewName = "schematic"
shapeCount = "many"
printf( "\nThe design %s %s %s has %s shapes"
        libName cellName viewName shapeCount )
```

Lab Summary

In this lab, you used the *println* and *printf* functions to display data in the CIW.



Lab 2-6 Solving Common Input Errors

Objective: To trigger common errors and to solve them.

Confirming that the SKILL Evaluator Is Available

1. In the CIW input pane, enter

1+2

You see the following in the CIW output pane

3

If you see a 3, then the SKILL Evaluator is available. You are ready to perform the next section of this laboratory exercise.

Resolving Unbalanced String Quotes and Parentheses

1. Enter the following into the CIW exactly as shown. Notice there is a missing " after the word *lamb*.

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

2. Press **Return**.

Nothing happens.

3. In the CIW, enter

1+2

4. Press **Return**.

Nothing happens.

Is the SKILL Evaluator occupied and therefore unable to respond?

Enter several characters to complete the SKILL expression.

5. In the CIW, type

]

Nothing happens.

6. In the CIW, type

```
"
```

Nothing happens.

7. In the CIW, type

```
]
```

You see the following return value in the CIW:

```
"mary had a little lamb ) 1+2 ]"
```

Confirm that the SKILL Evaluator is available for your next command.

8. In the CIW input pane, enter

```
1+2
```

You see the following in the CIW output pane:

```
3
```

The CIW is available for your next command.

Resolving Problems with Inappropriate White Space

1. Enter the following into the CIW. Notice there is space after the word *strcat* and the (

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

2. You see the following error message in the CIW:

```
*Error* eval: illegal function - "mary had a little"
```

3. Use the mouse to select your input line in the CIW output pane and to paste it into the CIW input pane.

4. Remove the offending space character.

5. Press **Return**.

You see the following in the CIW:

```
"mary had a little lamb"
```

Inappropriate Space Again

1. In the CIW, enter

```
storyLine = strcat ( "mary had a little" " lamb" )
```

You see the following error message in the CIW:

```
*Error* eval: unbound variable - strcat
```

2. Paste the line back into the input pane.
3. Remove the offending space.
4. Press **Return**.

You see the following in the CIW:

```
"mary had a little lamb"
```

5. Check the value of the *storyLine* variable. In the CIW, enter

```
storyLine
```

You see the following in the CIW:

```
"mary had a little lamb"
```

Resolving Data Error Messages

1. In the CIW, enter

```
strcat( "mary had a little" 5)
```

You see the following error message on a single line:

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

2. Enter the following into the CIW to correct the error:

```
strcat( "mary had a little" " 5" )
```

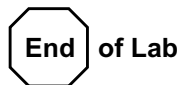
You see the following in the CIW:

```
"mary had a little 5"
```


Summary

In this lab, you triggered input common errors and resolved them. The errors were due to

- Unbalanced string quotes and unbalanced parentheses.
- Inappropriate space.
- Mismatched data.



Labs for Module 3

Lists

Lab 3-1 Creating New Lists

Objective: To build lists with the `'` operator and the *list*, *cons*, and *append* functions.

Creating a New List Using the `'` Operator

1. In the CIW, enter

```
numbers = '( 1 2 3 )  
numbers  
type( numbers )
```

By entering these expressions, you have

- Built a list containing the integers *1*, *2*, and *3*.
- Stored the list in the variable *numbers*.
- Fetched the value of *numbers*.
- Used the *type* function to confirm that *numbers* contains a list.

2. Create a list containing the numbers *4*, *5*, and *6*. Store the list in the variable *myNumbers*.

— Check your work against the solution on the next page.

Answers

1. In the CIW, enter

```
myNumbers = '( 4 5 6 )  
myNumbers
```

Did you remember to include the ' operator in front of (4 5 6)?

Continue with the lab.

Controlling the Number of List Items per Line

The global variable `_itemsperline` governs how many list items per line SKILL displays for a return result. Examine the current value of the variable.

1. In the CIW, enter

```
_itemsperline
```

2. In the CIW, enter

```
'(1 2 3 4 5 6 7 )
```

SKILL displays the list on two lines.

A variable whose name starts with an underscore character is generally a global variable you should not set. However, it is OK to set this global variable. Set the global variable to 15.

3. In the CIW, enter

```
_itemsperline = 15
```

4. In the CIW, enter

```
'(1 2 3 4 5 6 7 )
```

SKILL displays the list on a single line.

Creating a New List Using the *list* Function

1. In the CIW, enter

```
one = 1
two = 2
three = 3
moreNumbers = list( one two three )
moreNumbers
```

What is the value of moreNumbers?

2. Use the variables *one*, *two*, and *three* to create a list containing *1,2,3,2,1* in that order. Store the list in the variable *evenMoreNumbers*.

— Check your work against the solution on the next page.

Answers

1. In the CIW, enter

```
evenMoreNumbers = list( one two three two one
)
evenMoreNumbers
```

Did you remember to use the list function instead of the ' operator?

Continue with the lab.

Building a List Using the *cons* Function

1. In the CIW, enter

```
result = nil
result = cons( 1 result )
result = cons( 2 result )
result = cons( 3 result )
```

You have incrementally built up a list in *result*.

2. In the CIW, enter

```
reverse( result )
result
```

You have built a reversed copy of the list in *result*. Yet, as you next verified, the list in *result* is untouched.

What do you need to do so that the variable result contains the reversed list?

— Check your work against the solution on the next page.

Answer

1. In the CIW, enter

```
result = reverse( result )
```

Continue with the lab.

Building a List Using the *append* Function

1. In the CIW, enter

```
left = '( 1 2 3 )  
right = '( 4 5 6 )  
leftRight = append( left right )  
left  
right
```

Can you describe the result?

Comparing the *cons* and *append* Functions

1. In the CIW, enter

```
append( left right )  
append( right left )
```

Can you describe the difference between the two results?

2. In the CIW, enter

```
cons( left right )  
cons( right left )
```

Can you describe the difference between the two results?

Exploring Restrictions for the *cons* and *append* Functions

The second argument to the *cons* function must be a list.

1. In the CIW, enter

```
cons( 7 right )  
cons( right 7 )
```

You see the following error message:

```
*Error* cons: argument #2 should be a list (type template = "gl") - 7
```

Both arguments to the *append* function must be lists.

2. In the CIW, enter

```
append( right 7 )
```

The following error message appears:

```
*Error* append: argument #2 should be a list - 7
```

Adding an Element to the End of a List

If we put 7 into a list, we could use the *append* function to add 7 to the end of the list *right*.

How can we turn 7 into a list?

1. In the CIW, enter

```
list( 7 )  
append( right list( 7 ))  
right  
right = append( right list( 7 ))
```

2. Add 4 to the end of the list *left*.

— Check your work against the solution on the next page.

Answer

1. In the CIW, enter

```
left = append( left list( 4 ) )
```

Continue with the lab.

Building Hierarchical Lists (Optional)

By using the *list*, *cons* and *append* functions, you use construct lists of arbitrary depth and complexity.

1. In the CIW, enter

```
left = '( 1 2 3 )
right = '( 4 5 6 )
```

2. In the CIW, enter this example:

```
cons( left right )
cons( cons( left right ) right )
```

You see the following results:

```
((1 2 3) 4 5 6)
(((1 2 3) 4 5 6) 4 5 6)
```

3. Build list A with various combinations of the *list* function, the *cons* function, the *append* function and the variables *left* and *right*.

Use the *println* function to display the list on a single line.

List A

```
( 1 2 3 ( 4 5 6 ) 1 2 3 )
```

4. Build list B with various combinations of the *list* function, the *cons* function, the *append* function and the variables *left* and *right*.

Use the *println* function to display the list on a single line.

List B

```
( ( 1 2 3 ( 4 5 6 ) ) 1 2 3 )
```

5. Build list C with various combinations of the *list* function, the *cons* function, the *append* function and the variables *left* and *right*.

Use the *println* function to display the list on a single line.

List C

```
( ( 1 2 3 ) ( 4 5 6 ) ( 1 2 3 ) )
```

— Check your work against the solution on the next page.

Answers

- To build list A, in the CIW enter

```
println( append( left cons( right left )))
```

- To build list B, in the CIW enter

```
println( cons( append( left list( right))  
left))
```

- To build list C, in the CIW enter

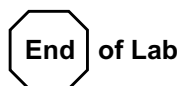
```
println( list( left right left ) )
```

How did you do?

Lab Summary

In this lab, you

- Built lists with the ' operator.
- Built a list with the *list* function.
- Used *cons* to add an element to the front of a list.
- Merged two lists with the *append* function.
- Compared the *cons* and *append* functions.
- Used the *append* function to add an element to the end of a list.
- Built various hierarchical lists with the *list*, *cons* and *append* functions.



Labs for Module 4

Windows

Lab 4-1 Opening Windows

Objective: To use the *geOpen* and *view* functions to open windows.

Opening a Design Window

1. In the Command Interpreter Window (CIW), enter

```
geOpen( ?lib "master" ?cell "mux2" ?view "layout" )
```

The design appears in a window in edit mode.

The *geOpen* function returns the window ID.

Note the window number for the next step.

2. Look at the window and use its window number in place of ***INSERT THE WINDOW NUMBER*** below. In the CIW, enter

```
designWindow = window( INSERT THE WINDOW NUMBER )
```



Type the window number here.

The *window* function returns the window ID.

SKILL assigns the window ID to the *designWindow* variable. You will refer to *designWindow* in a subsequent lab.

3. Open a window to edit the *master inv layout* design by entering

```
geOpen( )
```

Because you haven't specified certain required parameters, the Open File form appears.

4. Supply the library name, cell name, and view name and click on **OK**.

The design appears.

Opening a Text Window

1. In the CIW, enter

```
view( "~/SKILL/.cdsinit" )
```

A window displays the *~/SKILL/.cdsinit* file.

The *view* function returns the window ID of the new window. You cannot use this window to edit the file.

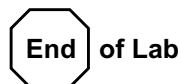
2. Use the *view* function to display your *.cshrc* file.
3. Assign the window ID of the text window to the variable *textWindow*.

You will refer to *textWindow* in a subsequent lab.

Lab Summary

In this lab, you used the

- *geOpen* function to open two designs.
- *view* function to view a text file.



Lab 4-2 Resizing Windows

Objective: To use the *hiGetAbsWindowScreenBBox* and *hiResizeWindow* functions to resize windows.

Retrieving the Bounding Box of a Window

1. In the CIW, enter

```
ciwBBox = hiGetAbsWindowScreenBBox( window(1) t )
```

The *hiGetAbsWindowScreenBBox* function returns the bounding box for window 1. This is the CIW.

2. Retrieve the bounding box of the text window you previously opened. Assign it to the variable *textBBox*.
3. Retrieve the bounding box of the first design window you previously opened. Assign it to the variable *designBBox*.

— Check your work against the solution on the next page.

Answers

1. In the CIW, enter

```
textBBox = hiGetAbsWindowScreenBBox( textWindow t )
```

2. In the CIW, enter

```
designBBox = hiGetAbsWindowScreenBBox( designWindow t )
```

Resizing a Window

1. In the CIW, enter

```
hiResizeWindow( window(1) designBBox )
```

The CIW changes to match the size and position of the design window exactly.

2. Resize the CIW to match the size and position of the text window exactly.

— Check your work against the solution on the next page.

Answers

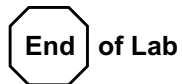
1. In the CIW, enter

```
hiResizeWindow( window(1) textBBox )
```

Lab Summary

In this lab, you

- Retrieved the bounding box of a window with the *hiGetAbsWindowScreenBBox* function.
- Resized a window with the *hiResizeWindow* function.



Lab 4-3 Storing and Retrieving Bindkeys

Objective: To use the *hiGetBindKey* and *hiSetBindkey* functions to store and retrieve bindkey definitions.

You establish a bindkey definition for both the *Schematics* and *Layout* applications to raise the Command Interpreter Window.

You establish another bindkey definition for the *Command Interpreter* application to raise the current window.

After defining these two bindkeys, you can press a single key to view the CIW or to view the current window.

You can thereby use a large CIW. Conversely, you can hide the CIW until you need it.

Locating an Available Key

1. In the CIW, select the **Options** → **Bindkey** command to verify that the **F8** key has no bindkey definition for the following applications:

Command Interpreter
Schematics
Layout

2. Use the *hiGetBindKey* function to verify that the **F8** key has no bindkey definition for the following applications:

Command Interpreter
Schematics
Layout

— Check your work against the solution on the next page.

Answers

To use the *hiGetBindKey* function to verify that the **F8** key has no bindkey definition, in the CIW enter

```
hiGetBindKey( "Command Interpreter" "<Key>F8" )
hiGetBindKey( "Schematics" "<Key>F8" )
hiGetBindKey( "Layout" "<Key>F8" )
```

The *Command Interpreter* and *Schematics* environments function call returns *nil*.

The Layout environment has a bindkey defined called *leToggleGuidedPathCreae()*. As you continue with this lab you will overwrite this function. Do not worry about it.

Continue with the lab.

Establishing the Bindkey Definitions

1. Use the *hiSetBindKey* function to establish the following bindkey definition for the **F8** key in the Command Interpreter application.

```
"hiRaiseWindow( hiGetCurrentWindow( ) )"
```

2. Use the *hiSetBindKey* function to establish the following bindkey definition for the **F8** key in the Schematics and Layout applications.

```
"hiRaiseWindow( window( 1 ) )"
```

— Check your work against the solution on the next page.

Answers

To establish the Command Interpreter bindkey definition.

```
hiSetBindKey( "Command Interpreter"
  "<Key>F8" "hiRaiseWindow( hiGetCurrentWindow() )" )
```

To establish the Schematics bindkey definition for the **F8** key.

```
hiSetBindKey( "Schematics"
  "<Key>F8" "hiRaiseWindow( window( 1 )" )
```

To establish the Layout bindkey definition for the **F8** key.

```
hiSetBindKey( "Layout"
  "<Key>F8" "hiRaiseWindow( window( 1 )" )
```

Continue with the lab.

Testing Schematic and CIW Bindkeys

1. Enlarge the CIW to occupy the lower half of the screen.
2. Open the *master mux2 schematic* cellview. Move the design window so that it partially overlaps the CIW.
3. Move the cursor over the Schematics application window.
4. Press the **F8** key.

The CIW comes to the front of the screen. Click on the header to make it active.

5. Press the **F8** key with the mouse over the CIW.

The schematic application window comes to the front of the screen.

Testing Layout and CIW Bindkeys

Open the *master mux2 layout* cellview and perform similar tests.

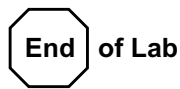
When you are satisfied that your bindkey definitions work, go on to the Lab Summary.

Lab Summary

In this lab, you used the *hiSetBindkey* function to define the following bindkeys:

- A bindkey that raises the CIW for the Schematics and Layout applications.
- A bindkey that raises the current window for the Command Interpreter.

What productivity enhancements can you implement with bindkeys?



Lab 4-4 Defining a Show File Bindkey

Objective: Clear the text selection with a bindkey.

In this lab, you establish a bindkey on `<Key>F8` for *Show File* windows to clear the text selection.

Defining Your Bindkey

Use this expression in your bindkey definition to clear the text selection in the current window.

```
hiUnselectTextAll(  
  hiGetCurrentWindow()  
)
```

Testing Your Bindkey Definition

1. Open a *Show File* application window. In the CIW, enter

```
view( "~/SKILL/.cdsinit" )
```

Use this window to test your bindkey.

— Check your work against the solution on the next page.

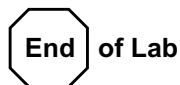
Answers

1. In the CIW, enter

```
hiSetBindKey( "Show File" "<Key>F8"  
  "hiUnselectTextAll( hiGetCurrentWindow())"  
  )
```

Lab Summary

In this lab, you used the *hiSetBindKey* function to define a bindkey for the Show File application.



Labs for Module 5

Database Queries

Lab 5-1 Querying Design Databases

Objective: Use the ~> operator to query several design databases.

In this lab, you open several designs and make the following queries using the ~> operator.

- How many nets are in the design?
- What are the net names?
- How many instances are in the design?
- What are the instance names?
- What are the master cell names?
- How many shapes in the design?
- What kinds of shapes are in the design?

Opening a Design

1. Open a design file by selecting, **File → Open**.

The Open File form appears.

2. In the form, enter the values in this table.

Library	master
Cell Name	mux2
View Name	layout
Mode	read

3. Click **OK**.

A Layout application window appears as the current window.

Retrieving the *cellView* Database Object

To query the design, you need to retrieve the *cellView* database object and store it in a variable. The *geGetWindowCellView* function works on the current window by default.

1. In the CIW, enter

```
cv = geGetWindowCellView()
```

The system displays a value resembling *db:23480364* in the CIW.

You are now ready to make several queries regarding the design.

Making Queries

How many nets are in the design?

1. In the CIW, enter

```
length( cv~>nets )
```

The system displays 6 in the CIW. There are 6 nets in this design.

What are the net names?

2. In the CIW, enter

```
cv~>nets~>name
```

The system displays the following list in the CIW:

```
( "SEL" "B" "A" "gnd!" "Y"
  "vdd!"
)
```

How many instances are there in the design?

3. In the CIW, enter

```
length( cv~>instances )
```

The system displays 17 in the CIW.

There are 17 instances in this design.

What are the instance names?

4. In the CIW, enter

```
cv~>instances~>name
```

The system displays the following list in the CIW:

```
("I35" "I31" "I30" "I32" "I36"
 "I4" "I5" "I34" "I41" "I40"
 "I37" "I38" "I6" "I3" "I18"
 "I1" "I2")
```

These are the instances names.

What are the master cell names?

5. In the CIW, enter

```
cv~>instances~>cellName
```

The system displays the following list in the CIW:

```
("M2_M1" "M2_M1" "M2_M1" "M2_M1" "M2_M1"
 "M2_M1" "M2_M1" "M1_POLY1" "M1_POLY1" "M1_POLY1"
 "M1_POLY1" "M1_POLY1" "M1_POLY1" "Inv" "nand2"
 "nand2" "nand2"
 )
```

These are the master cell names.

How many shapes are in the design?

6. In the CIW, enter

```
length( cv~>shapes )
```

The system displays *21* in the CIW.

There are *21* shapes in this design.

What kinds of shapes are in the design?

7. In the CIW, enter

```
cv~>shapes~>objType
```

The system displays the following list in the CIW:

```
( "textDisplay" "textDisplay" "textDisplay" "textDisplay" "textDisplay"
  "textDisplay" "rect" "rect" "rect" "path"
  "path" "rect" "path" "path" "rect"
  "path" "path" "rect" "path" "rect"
  "rect"
)
```

These shape types describe the kind of shapes in the design.

Continue with the lab on the next page.

Querying the Other Designs

Query the following designs:

```
master mux2 schematic
master mux2 extracted
master mux2 symbol
```

In these queries, find out the following information:

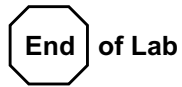
- How many nets are in the design?
- What are the net names?
- How many instances are in the design?
- What are the instance names?
- What are the master cell names?
- How many shapes in the design?
- What kinds of shapes are in the design?

What queries are the most relevant to your projects at work?

Can you express your queries using the ~> operator and SKILL functions, such as the length function?

Lab Summary

In this lab, you queried several designs using $\sim>$ expressions. Each expression retrieved one or more database object attributes.



Labs for Module 6

Menus

Lab 6-1 Exploring Menus

Objective: Correlate SKILL source code with pop-up menus and pull-down menus.

In this lab, you examine SKILL source code to determine what the code does. You validate your understanding by loading the source code and interacting with menus.

Examining the Code for a Pop-Up Menu

Use the *view* command to examine the file *~/SKILL/Menus/opiPopUp.il*.

1. In the CIW, enter

```
view( "~/SKILL/Menus/PopUp.il" )
```

Can you describe the pop-up menu that this code builds?

When is the pop-up menu displayed?

— Check your work against the solution on the next page.

Answer

The source code does the following:

- Creates a pop-up menu named *"Example"* and stores a reference to its data structure in the *TrPopUpMenu* variable.
- Defines a bindkey for the Layout application to display a pop-up menu.

Key Description	Callback
Ctrl Shift<Btn2Down>(2)	<i>hiDisplayMenu(TrPopUpMenu)</i>

Running the Code

Load the code in *~/SKILL/Menus/PopUp.il*.

1. In the CIW, enter


```
load( "~/SKILL/Menus/PopUp.il" )
```
2. Open a *Layout* application window on a design of your choice.
3. Display the pop-up menu with the **Ctrl Shift<Btn2Down>(2)** bindkey.
Use the middle mouse button to choose a menu item.

Is everything as you expected it to be?

Continue with the lab.

Examining the Code for a Pull-Down Menu

Use the *view* command to examine the *~/SKILL/Menus/Pulldown.il* file.

1. In the CIW, enter


```
view( "~/SKILL/Menus/Pulldown.il" )
```

What does the code do?

What is the name of the pull-down menu?

— Check your work against the solution on the next page.

Answer

The source code creates a pull-down menu named "*Navigation*" and installs it in the leftmost position of the CIW menu banner.

Running the Code

Load the code in *~/SKILL/Menus/Pulldown.il*.

1. In the CIW, enter

```
load( "~/SKILL/Menus/Pulldown.il" )
```

The Navigation menu appears in the leftmost position in the CIW.

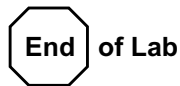
2. Display the *Navigation* pull-down menu.

Is everything as you expected it to be?

Lab Summary

In this lab, you

- Examined two SKILL source code files to determine what the code does.
- Loaded each file to verify the actual behavior of the SKILL source code.



Labs for Module 7

Customization

Lab 7-1 Defining Bindkeys in the .cdsinit File

Objective: Call the *hiSetBindKey* function in your .cdsinit file.

In this lab, you define three bindkeys in your .cdsinit file. These three bindkeys are immediately available the next time you access the Design Framework II environment.

In Lab 4-3, you defined three related bindkeys.

- Two bindkeys raise the CIW to the top of the screen. One bindkey applies to the *Schematics* application windows and the other applies to *Layout* application windows.
- A third bindkey raises the current window to the top of the screen. This bindkey applies to the *Command Interpreter* window.

1. Review your solution to Lab 4-3.

In Lab 7-1, you include the solution from Lab 4-3 in your .cdsinit file.

2. Review the tests you performed to verify that the bindkeys behaved appropriately.

In this lab, you perform these tests again.

Continue with the lab.

Editing Your .cdsinit File

1. Use the *edit* function to study the .cdsinit file for this course. In the CIW, enter:

```
edit( "~/SKILL/.cdsinit" )
```

An editor window displays the ~/SKILL/.cdsinit file.

2. Locate the section of the file that defines bindkeys.

```
;;;;;;;;;;;;; Load Bindkey definitions ;;;;;;;;;;;;;;
```

```
when( isFile( "leBindKeys.il" )  
  TrLoad( "leBindKeys.il" )  
  ) ; when
```

```
when( isFile( "schBindKeys.il" )  
  TrLoad( "schBindKeys.il" )  
  ) ; when
```

3. Use the editor to add your *hiSetBindKey* function calls after two calls to the *TrLoad* function.

— Check your work against the solution on the next page.

Answers

Verify that the portion of your *.cdsinit* resembles the following:

```

;;;;;;;;;;;;; Load Bindkey defintions ;;;;;;;;;;;;;;

when( isFile( "leBindKeys.il" )
      TrLoad( "leBindKeys.il" )
    ) ; when

when( isFile( "schBindKeys.il" )
      TrLoad( "schBindKeys.il" )
    ) ; when

hiSetBindKey( "Command Interpreter"
              "<Key>F8" "hiRaiseWindow( hiGetCurrentWindow() )" )

hiSetBindKey( "Schematics"
              "<Key>F8" "hiRaiseWindow( window(1))" )

hiSetBindKey( "Layout"
              "<Key>F8" "hiRaiseWindow( window(1))" )

```

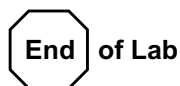
Continue with the lab.

Testing the *.cdsinit* File

1. Exit the Design Framework II session.
2. Start the Design Framework II environment.
3. Perform the tests for Lab 4-3.

Lab Summary

In this exercise, you edited the course *.cdsinit* file to make several bindkeys immediately available to the user.



Labs for Module 8

Developing a SKILL Function

Lab 8-1 Developing a SKILL Function

Objective: To develop a SKILL function that computes the area of a bounding box.

Requirements

Develop a SKILL function named *TrBBoxArea* that satisfies the following requirements:

- Make the *TrBBoxArea* function take a single argument named *bBox*.
- Make the *TrBBoxArea* function print a message in the CIW to identify the bounding box and the area.
- Make the *TrBBoxArea* function return the area of *bBox*.

For example, this function call shown here:

```
TrBBoxArea( list( 100:100 250:390 ) )
```

displays the following message in the CIW and returns *43500*.

```
Box ((100 100) (250 390)) area is: 43500
```

Suggestions

Base your work on the *TrBBoxHeight* function discussed in the lecture. The source code appears here:

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

Make sure that you do the following:

- Write your version of the *TrBBoxArea* function in a new file called *~/SKILL/Functions/TrBBoxArea.il*.

- Use the variables *ll*, *ur*, *llx*, *lly*, *urx*, *ury*, and *area*.

- Compute the area of the bounding box with this code.

```
area = ( urx - llx )*( ury - lly )
```

- Use the *printf* function to display this message.

```
printf( "Box %L area is: %n" bBox area )
```

Note that you must use the *%n* specification because *area* might be a floating point number.

Setting the *writeProtect* Switch

It is important for you to be able to redefine the *TrBBoxArea* function during development.

1. Check the value of the *writeProtect* switch. In the CIW, enter

```
status( writeProtect )
```

Is it *nil*?

2. If the value of *writeProtect* is not *nil*, then set it to *nil*. In the CIW, enter

```
sstatus( writeProtect nil )
```

You can redefine any SKILL function you define while *writeProtect* is *nil*.

Examining the SKILL Path

Check to see that the *~/SKILL/Functions* directory is in the SKILL path.

1. In the CIW, enter

```
getSkillPath()
```

The system displays the current SKILL path.

Why is it important that the ~/SKILL/Functions directory be in the SKILL path?

Editing the Source Code File

1. Check the *editor* variable. In the CIW, enter

```
editor
```

Is this the editor you want?

2. If the editor variable indicates the wrong editor for you, change the value of the *editor* variable. Assuming you prefer the *textedit* editor, in the CIW, enter

```
editor = "textedit"
```

3. Edit the new source code file. In the CIW, enter

```
edit( "~/SKILL/Functions/TrBBoxArea.il" )
```

Writing the Source Code

1. Enter the following skeletal definition in the file. You can omit the comments.

```
procedure( TrBBoxArea( bBox )
  ;;; compute the area of bBox
  ;;; print the message
  printf( "Box %L area is: %n" bBox area )
  area    ;;; the return result
) ; procedure
```

2. Add source code statements to compute *ll*, *ur*, *llx*, *lly*, *urx*, *ury* and finally *area*.
3. Save the file and exit the editor.

Loading Your Function

In the CIW, enter

```
load( "TrBBoxArea.il" )
```

A *t* is displayed in the CIW output pane.

What does the t indicate?

Testing Your Solution

Try the following test cases. The bounding boxes have been chosen so that you can compute their area easily yourself.

1. In the CIW, enter

```
aSquare = list( 100:100 200:200 )  
TrBBoxArea( aSquare )
```

2. In the CIW, enter

```
aHorizontalLine = list( 100:100 250:100 )  
TrBBoxArea( aHorizontalLine )
```

3. In the CIW, enter

```
aVerticalLine = list( 100:100 100:390 )  
TrBBoxArea( aVerticalLine )
```

— Check your work against the solution on the next page.

Test Case Results

■ The square

```
Box ((100 100) (200 200)) area is: 10000
10000
```

■ The horizontal line

```
Box ((100 100) (250 100)) area is: 0
0
```

■ The vertical line

```
Box ((100 100) (100 390)) area is: 0
0
```

How did you do?

— Check your work against the solution on the next page.

Sample Solution

The sample solution in the `~/SKILL/Functions/Solutions/TrBBoxArea.il` file is shown here for your convenience.

```
procedure( TrBBoxArea( bBox )
  ll    = lowerLeft( bBox )
  ur    = upperRight( bBox )
  lly   = yCoord( ll )
  ury   = yCoord( ur )
  llx   = xCoord( ll )
  urx   = xCoord( ur )
  area  = ( urx - llx )*( ury - lly )
  printf( "Box %L area is: %n" bBox area )
  area
) ; procedure
```

How did you do?

To run the sample solution, enter the following line in the CIW:

```
load( "Solutions/TrBBoxArea.il" )
```

Continue with the lab on the next page.

Optional Enhancement for Advanced Students

You compute the area of a bounding box that the user enters with the mouse. You define a *Layout* bindkey that prompts the user for a bounding box and calls the *TrBBoxArea* function to compute its area.

The *enterBox* function prompts the user to enter a box.

- The *?prompts* argument determines the prompts to show to the user.
- The *?doneProc* argument is the name of the function that the *enterBox* function calls when the user has either completed the box or aborted the operation.

1. In the CIW, enter

```
procedure( TrEnterBoxCB( wid completed? box )
  when( completed? TrBBoxArea( box ) )
  ) ; procedure

procedure( TrEnterBox()
  enterBox(
    ?prompts '( "First Corner" "Second Corner" )
    ?doneProc "TrEnterBoxCB"
  )
  ) ; procedure

hiSetBindKey( "Layout" "<Key>F9" "TrEnterBox()" )
```

2. In the CIW, use the **File → Open** command to open *master mux2 layout*.

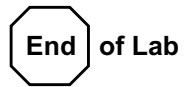
3. Try the bindkey several times.

Verify that your output includes items similar to the following in the CIW:

```
Box ((-0.500000 0.250000) (1.250000 1.000000)) area is: 1.312500
1.312500
Box ((-0.500000 0.500000) (0.0 1.500000)) area is: 0.500000
0.500000
```

Lab Summary

In this lab, you wrote a function that computes the area of a bounding box.



Labs for Module 9

Flow of Control

Lab 9-1 Writing a Database Report Program

Objective: To develop a SKILL function that counts the shapes used in a design.

In this lab, you write a SKILL function that counts the shapes used in a design. During the course, you will make several enhancements to this program.

Requirements

Make *TrShapeReport* accept a single parameter named *wid*. The parameter is the window ID of the window displaying the design.

Make the *TrShapeReport* function display a report in the CIW output pane that includes the following items:

- *libName*, *cellName* and *viewName* of the design
- *rectangle* count
- *polygon* count
- *path* count
- count of all other shapes

Here is an example report.

```
master nand2 layout contains:
Rectangles 12
Polygons   0
Paths      3
Misc       0
```

Continue with the lab on the next page.

Suggestions

Maintain the source code in the `~/SKILL/FlowOfControl/TrShapeReport.il` file.

Use the following variables, functions, and operators:

- The `geGetWindowCellView` function to retrieve the `cellView` database object from the window ID.
- The variable `cv` to store the `cellView` database object.
- The variables `rectCount`, `polygonCount`, `pathCount`, and `miscCount` to store the counts.
- The `printf` function to produce the lines of the report.
- The `~>` operator to retrieve the `libName`, `cellName`, and `viewName` attributes from `cv` to produce the first line of the report.
- A `foreach` loop to process the shapes in the design.
- The `~>` operator to retrieve the `shapes` attribute.
- The `case` function to process each shape's `objType` attribute.
- The `++` postincrement operator to increment the variables.

Continue with the lab on the next page.

Testing Your Solution

First Test

1. In the CIW, enter

```
nand2wid = geOpen( ?lib "master" ?cell "nand2" ?view "layout" ?mode "r" )
```

2. In the CIW, enter

```
TrShapeReport( nand2wid )
```

The CIW displays the following:

```
master nand2 layout contains:
Rectangles 10
Polygons   0
Paths      3
Misc       0
```

Second Test

3. In the CIW, enter

```
mux2wid = geOpen( ?lib "master" ?cell "mux2" ?view "layout" ?mode "r" )
```

4. In the CIW, enter

```
TrShapeReport( mux2wid )
```

The CIW displays the following:

```
master mux2 layout contains:
Rectangles 8
Polygons   0
Paths      7
Misc       6
```

— Check your work against the solution on the next page.

Sample Solution

The following solution does not use local variables. You can enhance this solution to make *rectCount*, *polygonCount*, *pathCount*, and *miscCount* local.

```
procedure( TrShapeReport( wid )
  rectCount = polygonCount = pathCount = miscCount = 0
  cv = geGetWindowCellView( wid )
  printf(
    "%s %s %s contains:"
    cv~>libName cv~>cellName cv~>viewName )
  foreach( shape cv~>shapes
    case( shape~>objType
      ( "rect" rectCount++ )
      ( "polygon" polygonCount++ )
      ( "path" pathCount++ )
      ( t miscCount++ )
    ) ; case
  ) ; foreach
  printf( "\n%-10s %-10d" "Rectangles" rectCount )
  printf( "\n%-10s %-10d" "Polygons" polygonCount )
  printf( "\n%-10s %-10d" "Paths" pathCount )
  printf( "\n%-10s %-10d" "Misc" miscCount )
) ; procedure
```

Did you remember to initialize your numeric variables?

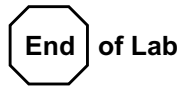
The `~/SKILL/FlowOfControl/Solutions/TrShapeReport.il` file contains the solution.

Lab Summary

In this lab, you developed a SKILL function using the following:

- the *geGetWindowCellView* function
- `~>` operator with various attributes
- the *printf* function
- a *foreach* loop
- the *case* function

You use the *load*, *edit*, and *view* functions to develop your code and to test it.



Lab 9-2 Exploring Flow of Control

Objective: To validate data.

The flow of control in a program that validates data can be complex. In this lab, you write a SKILL function to validate a data item according to the following definition.

Term	Definition
<i>point</i>	A data item is a <i>point</i> only if it is a list of two elements, each of which is either an integer or a floating-point number.

You use the *TrValidatePoint* function in a subsequent lab.

Requirements

Make the *TrValidatePoint* function do the following:

- Accept one argument, *exp*.
- Return *nil*, if *exp* is not a list of two numeric arguments.
- Return *exp*, if *exp* is a list of two numeric arguments.

Recommendations

- Use either the *cond* function or nested *if-then-else* expressions.
- Use the following predicate functions to test the data type.

Predicate Function	Data Type Test
<i>numberp</i>	returns <i>t/nil</i> , depending on whether its arguments is numeric.
<i>listp</i>	return <i>t/nil</i> , depending on whether its arguments is a list.

Testing Your Solution

Use the following test cases in addition to other test cases of your own choice.

Function Call	Expected Result
<i>TrValidatePoint('(5 6))</i>	<i>(5 6)</i>
<i>TrValidatePoint(list(5:0))</i>	<i>nil</i>
<i>TrValidatePoint(list(5 "abc"))</i>	<i>nil</i>
<i>TrValidatePoint(list(5 7 8))</i>	<i>nil</i>

— Check your work against the solution on the next page.

Sample Solution

The `~/SKILL/FlowOfControl/Solutions/TrMakeBBox.il` file contains the solution.

```
;;; TrValidatePoint
;;; returns t/nil whether or not exp is a valid point
;;; test: non-nil list of two numeric arguments

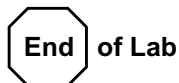
procedure( TrValidatePoint( exp )
  cond(
    ( !exp      nil )
    ( !listp( exp ) nil )
    ( length( exp ) != 2 nil )
    ( !numberp( xCoord( exp ) ) nil )
    ( !numberp( yCoord( exp ) ) nil )
    ( t      exp )
  ) ; cond
) ; procedure
```

Did you use the following functions:

- The *listp* and *numberp* functions?
- The *length* function?
- The *cond* function or nested *if-then-else* expressions?

Lab Summary

In this lab, you wrote a function to validate a point.



Lab 9-3 More Flow of Control

Objective: To compare 2-dimensional points.

In this lab, you write a SKILL function that tests whether one point is lower and to the left of a second point, according to the following definition.

Term	Definition
lower-left	A point P is lower-left of another point Q only if (1) $xCoord(P) \leq xCoord(Q)$ (2) $yCoord(P) \leq yCoord(Q)$

Requirements

Make the *TrLLp* function do the following:

- Accept two arguments *pt1* and *pt2*.
- Return *t* if *pt1* is lower-left of *pt2*.
- Return *nil* otherwise.

Assume that *pt1* and *pt2* are valid points.

Use the following:

- The *xCoord* and *yCoord* functions to retrieve the coordinates of *pt1* and *pt2* as needed.
- The \leq operator to compare numeric values.
- The $\&\&$ operator to combine the results of numeric comparisons.

Continue with the lab on the next page.

Testing Your Solution

Use the following test cases in addition to other test cases of your own choice.

pt1	pt2	result
<i>0:0</i>	<i>10:10</i>	<i>t</i>
<i>10:0</i>	<i>-1:-2</i>	<i>nil</i>
<i>-1:-2</i>	<i>10:0</i>	<i>t</i>

Did you need to explicitly control the precedence of the \leq and $\&\&$ operators?

— Check your work against the solution on the next page.

Solution

The `~/SKILL/FlowOfControl/Solutions/TrMakeBBox.il` file contains the solution.

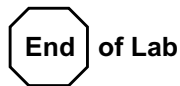
```
;;; TrLLp
;;; returns t/nil indicating pt1 is to the lower-left of pt2

procedure( TrLLp( pt1 pt2 )
  xCoord( pt1 ) <= xCoord( pt2 ) && yCoord( pt1 ) <= yCoord( pt2 )
) ; procedure
```

Lab Summary

In this lab, you wrote a SKILL expression to test whether one point was lower and to the left of another point. You used the

- `<=` and `&&` operators.
- `xCoord` and `yCoord` functions.



Lab 9-4 Controlling Complex Flow

Objective: To build a bounding box.

In this lab, you write two SKILL functions to build a bounding box according to the following definition.

Term	Definition
bounding box	A list of two points. The first point in the list is below and to the left of the second point.

Requirements

Make the *TriMakeBBox* function do these tasks:

- Accept two valid points as arguments *pt1* and *pt2*.
- Return a bounding box derived from the two valid points.

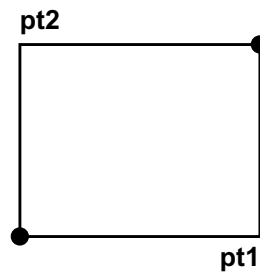
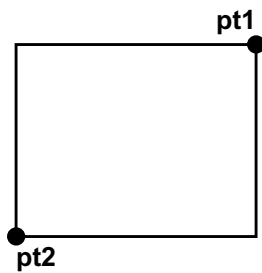
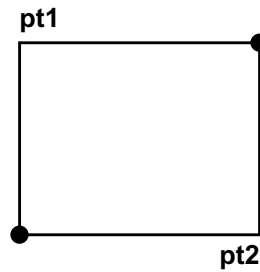
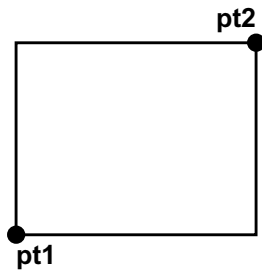
Note: An "i" occurring in a function's prefix usually indicates that the function is an internal or private function. The SKILL language does not allow you to hide internal or private functions so you need to rely on naming conventions. In this case, *TriMakeBBox* is an internal function because it does not do any argument validity checking.

Make the *TrMakeBBox* function do these tasks:

- Accept one argument, *exp*.
- Use your *TrValidatePoint* function to return *nil* if *exp* is not a non-*nil* list of two valid points.
- If *exp* is a list of two valid points, use your *TriMakeBBox* function to return the bounding box.

Suggestions

Notice that *pt1* and *pt2* can occupy the following corners of the bounding box. In each of the four cases, you need to compute the coordinates of the indicated lower-left and upper-right points. Consider using the *min* and *max* functions to treat the four cases together.



Testing Your Solution

Test your code on these test cases.

Function Call	Expected Result
<i>TrMakeBBox('(5 6))</i>	<i>nil</i>
<i>TrMakeBBox(list(5:0 6))</i>	<i>nil</i>
<i>TrMakeBBox(list(6 5:0))</i>	<i>nil</i>
<i>TrMakeBBox(list(list("z") 5:6))</i>	<i>nil</i>
<i>TrMakeBBox(list(0:0 5:6))</i>	<i>((0 0) (5 6))</i>
<i>TrMakeBBox(list(5:6 0:0))</i>	<i>((0 0) (5 6))</i>
<i>TrMakeBBox(list(-5:-6 0:0))</i>	<i>((-5 -6) (0 0))</i>
<i>TrMakeBBox(list(0:6 5:0))</i>	<i>((0 0) (5 6))</i>
<i>TrMakeBBox(list(5:0 0:6))</i>	<i>((0 0) (5 6))</i>

— Check your work against the solution on the next page.

Sample Solution

The `~/SKILL/FlowOfControl/Solutions/TrMakeBBox.il` file contains the full solution for the three labs.

```

procedure( TriMakeBBox( pt1 pt2 )
  let( ( llx lly urx ury pt1x pt2x pt1y pt2y )
    pt1x = xCoord( pt1 )
    pt2x = xCoord( pt2 )
    pt1y = yCoord( pt1 )
    pt2y = yCoord( pt2 )
    llx   = min( pt1x pt2x )
    lly   = min( pt1y pt2y )
    urx   = max( pt1x pt2x )
    ury   = max( pt1y pt2y )
    list( llx:lly urx:ury )
  ) ; let
) ; procedure

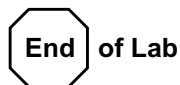
procedure( TrMakeBBox( exp )
  let( ( p r )
    cond(
      ( !exp nil )
      ( !listp( exp ) nil )
      ( length( exp ) != 2 nil )
      ( !TrValidatePoint( p = car( exp ) ) nil )
      ( !TrValidatePoint( r = cadr( exp ) ) nil )
      ( t TriMakeBBox( p r ) )
    ) ; cond
  ) ; let
) ; procedure

```

Did you use the cond function or nested if-then-else expressions?

Lab Summary

In this lab, you wrote two SKILL functions to build a bounding box.



Labs for Module 10

File I/O

Lab 10-1 Writing Data to a File

Objective: Use the *infile*, *fprintf* and *close* functions to write data to a file.

In this lab, you write data to a new file `~/SKILL/FileIO/MyFile.text`.

Obtaining an Output Port on a File

1. In the CIW, enter

```
myPort = outfile( "~/SKILL/FileIO/MyFile.text" )
myPort
```

The system displays the print representation of the port:

```
port: "~/SKILL/FileIO/MyFile.text "
```

2. Verify that the value of *myPort* is an output port by invoking the *outportp* function. In the CIW, enter

```
outportp( myPort )
```

What is the return result?

The return result indicates that the value is an output port.

Writing Data to the File

1. In the CIW, enter

```
lineCount = 0
fprintf( myPort "This is line # %d\n" lineCount++ )
fprintf( myPort "This is line # %d\n" lineCount++ )
fprintf( myPort "This is line # %d\n" lineCount++ )
```

2. Use the *fprintf* function to write the following line to the port:

```
No more lines
```

— Check your work against the solution on the next page.

Answers

1. In the CIW, enter

```
fprintf( myPort "No more lines" )
```

Continue with the lab.

Closing the File

1. In the CIW, enter

```
close( myPort )
```

The port is no longer open.

2. Verify that the port is no longer open by attempting to write data to the port. In the CIW, enter

```
fprintf( myPort "This is line # %d\n" lineCount++ )
```

The CIW displays the following error:

```
*Error* printf/fprintf: cannot send output to a closed port -  
port: "~/SKILL/FileIO/MyFile.text"
```

3. Since *myPort* has been closed, it is a good idea to set *myPort* to *nil*. In the CIW, enter

```
myPort = nil  
outportp( myPort )
```

Why is it a good idea to set myPort to nil?

In general, once you return a resource to the system, such as a file port, you should make sure no SKILL variable contains a reference to the resource. Otherwise, your SKILL application might generate warnings or even errors if it attempts to access the resource.

Viewing the File

1. Verify the contents of the file by starting the *view* function. In the CIW, enter

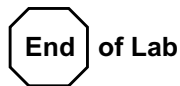
```
view( "~/SKILL/FileIO/MyFile.text" )
```

A view file window appears.

Lab Summary

In this lab, you used the

- *outfile* function to obtain an output port.
- *fprintf* function to write data to the file.
- *close* function to close the file.
- *view* function to verify the contents of the file.



Lab 10-2 Reading Data from a Text File

Objective: Use various functions to read data from a text file.

In this lab, you read data from the file you created in the preceding lab. You can use the solution file.

Obtaining an Input Port on a File

Before you read data from a file, you need to obtain an input port.

1. In the CIW, enter

```
myPort = infile( "~/SKILL/FileIO/Solutions/MyFile.text" )  
myPort
```

The system displays the print representation of the port:

```
port: "~/SKILL/FileIO/MyFile.text "
```

Continue with the lab on the next page.

Reading the File a Line at a Time

Read a line from the file. Be aware of the order of the arguments to the *gets* function.

1. In the CIW, enter

```
nextLine = nil
gets( nextLine myPort )
nextLine
```

The *gets* function returns the next line as a SKILL string and also updates the value of *nextLine*. Notice that the string contains a `"\n"`.

2. Read the next three lines from the file.

You are positioned at the end of the file.

3. Attempt to read the next line from the file.

What does the gets function return in this case?

Is the value of nextLine changed?

— Check your work against the solution on the next page.

Answers

1. To read the next three lines from the file, enter the following in the CIW:

```
gets( nextLine myPort )  
gets( nextLine myPort )  
gets( nextLine myPort )
```

← You can paste the previous line from the log.

2. The *gets* function returns *nil* when positioned at the end of the file. The variable *nextLine* is not automatically updated.

Continue with the lab.

Closing the File

1. Close the port on the file.

Reading the Data From a Text File

Read words from a file and add each word to a list. The following steps guide you.

1. Obtain an input port on the *~/SKILL/FileIO/Solutions/MyFile.text* file.
2. To read the next word from the file, enter these two lines in the CIW:

```
fscanf( myPort "%s" word )  
word
```

The *fscanf* function returns the number of items read according to the conversion specifications.

3. Read the remaining words of the file. Initialize *wordList* to *nil*. Use the *cons* function to add each word to a list, *wordList*.

— Check your work against the solution on the next page.

Answers

1. To read the next word from the file and add it to the list in *wordList*, enter these two lines in the CIW:

```
fscanf( myPort "%s" word )  
wordList = cons( word wordList )
```

Continue with the lab.

Closing the File

1. Close the port on the file.

Reading Numeric Data from a Text File

1. Obtain an input port on the *~/SKILL/FileIO/Solutions/MyFile.text* file.

2. In the CIW, enter

```
fscanf( myPort "%[^0-9]" phrase )  
phrase
```

The first SKILL expression reads the characters up to the next occurrence of *0-9* and stores the SKILL string in the variable *phrase*.

3. In the CIW, enter

```
fscanf( myPort "%d" number )  
number
```

The first SKILL expression reads the next numeric data item and stores its value in the variable *number*.

4. Read the remaining numeric items.

5. Close your input port.

— Check your work against the solution on the next page.

Answers

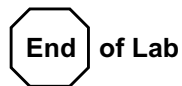
1. The following code reads a single line in the file:

```
fscanf( myPort "%[^0-9]" phrase )  
phrase  
fscanf( myPort "%d" number )  
number
```

Lab Summary

In this module, you used the

- *infile* function to obtain an input port on a file.
- *gets* function to read the file a line at a time.
- *fscanf* function to read data items from the file.
- *close* function to close the file.



Lab 10-3 Writing Output to a File

Objective: To enhance the *TrShapeReport* function to write its output to a file.

In Lab 9-1, you wrote the *TrShapeReport* function. This function displays a summary report in the CIW. In this lab, you enhance this function to send the report to a temporary file and to display this file for the user in a Design Framework II text window.

Requirements

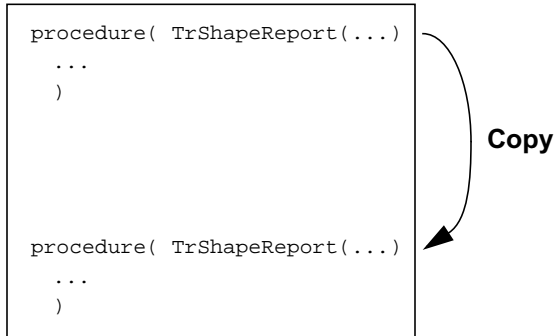
Enhance the *TrShapeReport* function to

- Send its output to a file instead of the CIW.
Send the program output to the */tmp/ShapeReport.txt* file.
- Display the file in a Show File window.
Use the *view* function to display the file in the window titled *"Shape Report"*.

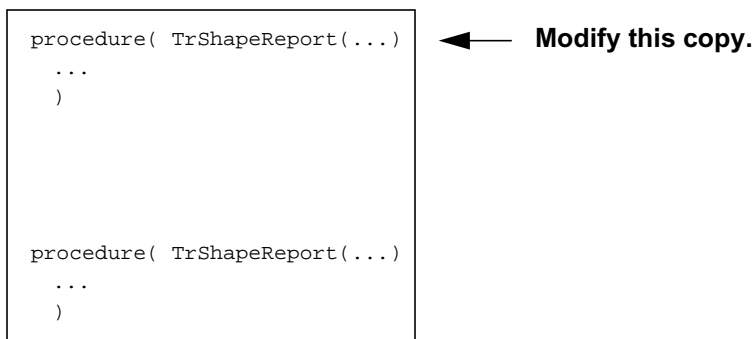
Continue with the lab on the next page.

Recommendations

1. Using the text editor, copy the declaration of the *TrShapeReport* function to a different place in the file.



2. Modify the original *TrShapeReport* function to call a new function named *TrShapeReportToPort*.



Specifically, modify the source code for the *TrShapeReport* function so that it does the following:

- Call the *outfile* function to obtain a port on the file.

```
thePort = outfile( "/tmp/ShapeReport.txt" )
```

- Call the *TrShapeReportToPort* function you write in the next step.

```
TrShapeReportToPort( wid thePort )
```

- Call the *close* function to close the port.

- Call the *view* function as follows. The second parameter, *nil*, asks for the default bounding box.

```
view( "/tmp/ShapeReport.txt" nil "Shape Report" )
```


3. Modify the copy of the *TrShapeReport* function that you made previously to create the *TrShapeReportToPort* function.

```

procedure( TrShapeReport(...)
...
)

procedure( TrShapeReport(...)
...
)

```

← **Modify this copy.**

Specifically, you need to do the following:

- Change the name from *TrShapeReport* to *TrShapeReportToPort* and add an *outport* parameter in addition to the *wid* parameter.

```

procedure( TrShapeReportToPort( wid outport )
...
) ; procedure

```

- Replace each call to the *printf* function with a call to the *fprintf* function. Be sure you include *outport* as the first argument.

Testing Your Solution

First Test

1. In the CIW, enter

```
nand2wid = geOpen( ?lib "master" ?cell "nand2" ?view "layout" ?mode "r" )
```

2. In the CIW, enter

```
TrShapeReport( nand2wid )
```

The CIW displays the following:

```
master nand2 layout contains:
Rectangles 10
Polygons   0
Paths      3
Misc       0
```

Second Test

3. In the CIW, enter

```
mux2wid = geOpen( ?lib "master" ?cell "mux2" ?view "layout" ?mode "r" )
```

4. In the CIW, enter

```
TrShapeReport( mux2wid )
```

The CIW displays the following:

```
master mux2 layout contains:
Rectangles 8
Polygons   0
Paths      7
Misc       6
```

— Check your work against the solution on the next page.

Solutions

The `~/SKILL/FileIO/Solutions/TrShapeReport.il` file contains the following solution:

```

procedure( TrShapeReportToPort( wid output )
  let( ( rectCount polygonCount pathCount miscCount )
    rectCount = polygonCount = pathCount = miscCount = 0
    cv = geGetWindowCellView( wid )
    fprintf(
      output
      "%s %s %s contains:"
      cv~>libName cv~>cellName cv~>viewName
    )
    foreach( shape cv~>shapes
      case( shape~>objType
        ( "rect" rectCount++ )
        ( "polygon" polygonCount++ )
        ( "path" pathCount++ )
        ( t miscCount++ )
      ) ; case
    ) ; foreach
    fprintf( output "\n%-10s %-10d" "Rectangles" rectCount )
    fprintf( output "\n%-10s %-10d" "Polygons" polygonCount )
    fprintf( output "\n%-10s %-10d" "Paths" pathCount )
    fprintf( output "\n%-10s %-10d" "Misc" miscCount )
    list( cv~>libName cv~>cellName cv~>viewName )
  ) ; let
) ; procedure

TrReportBBox = '((120 729) (372 876))

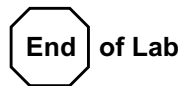
procedure( TrShapeReport( wid )
  thePort = outfile( "/tmp/ShapeReport.txt" )
  when( thePort
    TrShapeReportToPort( wid thePort )
    close( thePort )
  ) ; when
  view( "/tmp/ShapeReport.txt" TrReportBBox "Shape Report" )
) ; procedure

```

Lab Summary

In this lab, you extended your *TrShapeReport* function to write its output to a file and to display the file in a Show File window. You used these functions:

- *fprintf* function
- *outfile* function
- *close* function
- *view* function



Labs for Module 11

SKILL Development Environment

Lab 11-1 Analyzing an Error

Objective: Use the SKILL Debugger to determine the cause of an error.

In this lab, you

- Install the SKILL Debugger before loading SKILL source code.
- Load and run a version of the *TrShapeReport* function with bugs.
- Use the SKILL Debugger Toolbox to track down the problem.

Installing the SKILL Debugger

1. In the CIW, select **Tools** → **SKILL Development**.

The SKILL Development Toolbox appears.

2. Click **SKILL Debugger**.

The SKILL Debugger Toolbox appears. The SKILL Debugger **Enter New Debug Toplevel on Error** radio button indicates that the SKILL Debugger is installed.

Adjusting the *tracelevel* and *tracelength* Variables

1. Display the *tracelevel* variable. In the CIW, enter
`tracelevel`

The system displays *4*, which is the default value.

2. Set *tracelevel* to 2. In the CIW, enter
`tracelevel = 2`

3. Display the *tracelength* variable. In the CIW, enter
`tracelength`

The system displays *5*, which is the default value.

4. Set *tracelength* to 2. In the CIW, enter

```
tracelength = 2
```

Loading the Program

1. In the CIW, enter

```
load( "~/SKILL/DevEnv/DebugError/TrShapeReport.il" )
```

This file defines the *TriShapeReport* function and the *TrShapeReport* function. There is a bug in this program.

Opening an Example Design

1. In the CIW, use the **File** → **Open** command to open *master mux2 layout* with **Read**.

A Layout application window displays *master mux2 layout*.

2. Note the window number of the Layout application window. In the CIW, enter the following:

```
dwid = window( # )
```



Use the window number of the Layout application window in place of the #.

Generating the Error

1. In the CIW, enter

```
TrShapeReport( dwid )
```

The system displays the following message:

```
*** Error in routine postincrement:
Message: *Error* postincrement: can't handle add1(nil)
SKILL Debugger: type 'help debug' for a list of commands or debugQuit
to leave.
```


Displaying the Stack

1. In the SKILL Debugger Toolbox, click on the **Stacktrace** command.

The system displays the stack:

```
<<< Stack Trace >>>
errorHandler("postincrement" 0 ... )
(miscCount = miscCount)
miscCount++
case((shape~>objType) ("rect" &) ... )
foreach(shape (cv~>shapes) ... )
let((rectCount polygonCount ... ) (rectCount = &) ... )
TriShapeReport(wid thePort)
TrShapeReport(window:6)
8
```

Determining the Cause of the Error

1. Display the value of the *rectCount* variable. In the CIW, enter

```
miscCount
```

The system displays *nil*. Attempting to increment the value *nil* caused the error. In fact, the initial value for all the counters should be 0.

Quitting the Debugger

Quit the debugger before redefining the *TriShapeReport* function. Quitting the SKILL Debugger session removes both the *TrShapeReport* function and the *TriShapeFunction* from the stack.

1. In the SKILL Debugger, click **Quit Debugger**.
2. In the SKILL Debugger Toolbox, click the **Stacktrace** command.

The system displays the stack:

```
<<< Stack Trace >>>
0
```

The stack is empty.

Editing and Loading the Fixed Program

You loaded the source code for the *TrShapeReport* function when the SKILL Debugger was installed. Therefore, the SKILL Development Environment remembers the full pathname of the *TrShapeReport* source code file.

1. In the CIW, enter

```
edit( TrShapeReport t )
```

An editor window appears displaying the
~/SKILL/DevEnv/Debugger/TrShapeReport.il file.

2. Change the *nil* to *0* in this line.

```
rectCount = polygonCount = pathCount = miscCount = nil
```

3. Save the file and quit the editor.

The system redefines the two functions and displays these two messages:

```
function TriShapeReport redefined  
function TrShapeReport redefined
```

Testing the Fixed Program

1. In the CIW, enter

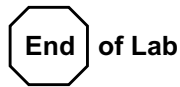
```
TrShapeReport( dwid )
```

The system displays the Shape Report window as expected. You have fixed the bug.

Lab Summary

In this lab, you

- Installed the SKILL Debugger.
- Controlled the amount of stack information displayed by setting the variables *tracelength* and *tracelevel*.
- Loaded and ran a SKILL program which contained an error.
- Used the SKILL Debugger Toolbox **Stacktrace** command to display the stack.
- Used SKILL Debugger Toolbox **Quit Debugger** to allow you to redefine two functions.
- Fixed the bug.



Lab 11-2 Debugger Sessions

Objective: To conduct a SKILL Debugger Session.

In this lab, you

- Load another version of the Shape Report program.
- Set a breakpoint at a function in the Shape Report program.
- Run the Shape Report program.
- Resume execution of the Shape Report program using the **Continue**, **Step**, **Next**, and **Step Out** SKILL Debugger Toolbox commands.

Verifying the Window ID of the Design Window

In the previous lab, you assigned the window id of the design window to the *dwid* variable. The rest of the lab assumes the window number is 7.

1. In the CIW, enter

```
dwid
```

Assuming the window number of the design window is 7, the system displays the following:

```
window:7
```

Verifying that the SKILL Debugger Is Installed

In the previous lab, you installed the SKILL Debugger. Verify that it is still installed.

1. Locate the SKILL Debugger Toolbox.
2. Click the SKILL Debugger **Enter New Debug Toplevel on Error** radio button.

Loading the Application

1. In the CIW, enter

```
load( "~/SKILL/DevEnv/DebugSession/TrShapeReport.il" )
```

SKILL redefines the *TriShapeReport* function and the *TrShapeReport* function. This version of the Shape Report program does not have any bugs.

Setting a Breakpoint

1. In the SKILL Debugger Toolbox, click **Set Breakpoints**.

The Set Breakpoint form appears.

2. In the **Function Names** field, enter

```
TriShapeReport
```

The *TrShapeReport* function calls the *TriShapeReport* function.

3. Click **OK** on the form.

The system puts a breakpoint at the entry to the *TriShapeReport* function. SKILL initiates a Debugger Session whenever the user or an application invokes the *TriShapeReport* function.

4. Verify that the breakpoint exists. In the SKILL Debugger Toolbox, click **Debug Status**.

5. The system displays the following:

```
Traced functions nil
Tracef functions nil
Traced variables nil
Breakpoints (TriShapeReport)
Counted functions nil
```

Running the Shape Report Program

1. In the CIW, enter

```
TrShapeReport( dwid )
```

The system displays the following:

```
<<< Break >>> on entering TriShapeReport
SKILL Debugger: type 'help debug' for a list of commands or debugQuit to
leave.
Entering new debug toplevel due to breakpoint:
Debug 2>
```

SKILL will execute this *TriShapeReport* function call if and when you resume execution with any of the **Step**, **Step Out**, **Next**, **Continue** commands.

Examining Function Call Arguments

The system suspended execution after evaluating the actual argument expressions to the *TriShapeReport* function, and after binding the evaluated arguments to the formal arguments, but before evaluating the body of the *TriShapeReport* function.

1. Display the value of the *wid* argument. In the CIW, enter

```
wid
```

The system displays:

```
window:7
```

2. Display the value of the *thePort* argument. In the CIW, enter

```
thePort
```

The system displays:

```
port: "/tmp/ShapeReport.txt "
```

Continuing Execution

1. In the SKILL Debugger Toolbox, click **Continue**.

The system resumes execution of the function call. The system displays the Shape Report window. The *TrShapeReport* returns the window ID.

Running the Shape Report Program Again

You run the Shape Report program again. This time you use the **Step** command to resume execution from the breakpoint at the entry to the *TriShapeReport* program.

1. In the CIW, enter.

```
TrShapeReport( dwid )
```

As before, the system displays:

```
<<< Break >>> on entering TriShapeReport
SKILL Debugger: type 'help debug' for a list of commands or debugQuit
to leave.
Entering new debug toplevel due to breakpoint:
Debug 2>
```

Using the Step Command

Each time you click **Step**, the system displays the SKILL expression that will be executed as soon as you resume execution.

1. In the SKILL Debugger Toolbox, click **Step** several times, until the system reaches the first call to the *fprintf* function.

Eventually, the system displays:

```
Debug 2>
ilStepCB()
stopped before evaluating fprintf(outport "%s %s %s contains:"
  (cv~>libName)(cv~>cellName) (cv~>viewName))
```

Using the Next Command

To skip over the evaluation of the call to *fprintf* altogether, you can use the **Next** command. The Debugger session continues with evaluation of the next function invocation at the same depth.

Note: *In this case, the **Step** command would have skipped over the evaluation of *fprintf* because it is a built-in function.*

1. Click the **Next** command.

The system displays:

```
ilNextCB()  
stopped before evaluating foreach(shape (cv~>shapes)  
  case((shape~>objType)...  
Debug 2>
```

Stepping into a *foreach* Loop

You can use the **Step** command to step into the evaluation of the *foreach* loop.

1. Click the **Step** command.

The system displays:

```
ilStepCB()  
stopped before evaluating (cv~>shapes)
```

2. Click the **Step** command several more times until the system displays the following:

```
ilStepCB()  
stopped before evaluating miscCount++
```

Using the Step Out Command

You can use the **Step Out** command to skip the remaining evaluation of a function.

1. Click the **Step Out** command.

The system displays:

```
ilStepoutCB()  
stopped before evaluating close(thePort)
```

Notice that the call to the *close* function is at the same level as the call to the *TriShapeReport* function. You reenter the same Debugger Session when the flow of control invokes another function at the same level.

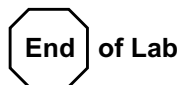
2. Click the **Continue** command.

The systems displays the Shape Report window. The *TrShapeReport* function returns the window ID.

Lab Summary

In this lab, in the SKILL Debugger toolbox you used the

- **Set Breakpoints** command to set a breakpoint at the entry to a function.
- **Continue** command to resume execution.
- **Step** command to observe successive function invocations.
- **Next** command to skip over the evaluation of a function.
- **Step Out** command to skip over the remaining part of a *foreach* loop.



Lab 11-3 Using SKILL Lint

Objective: Uncover potential source problems in *TrShapeReport*.

Running SKILL Lint

1. In the SKILL Development Toolbox, click **SKILL Lint**.

The SKILL Lint form appears.

2. In the Input File field, enter

`~/SKILL/DevEnv/Lint/TrShapeReport.il`

3. Click **OK**.

SKILL Lint analyzes the `~/SKILL/DevEnv/Lint/TrShapeReport.il` file and displays the SKILL Lint Output window.

Examining the SKILL Lint Output

SKILL Lint warns you about global variables in your program.

1. In the SKILL Lint Output window, locate this line:

```
INFO( PREFIXES): Using prefixes: "none".
```

SKILL Lint does not know about the *Tr* prefix that this file uses to indicate its global variables.

2. In the SKILL Lint Output window, locate the statements concerning these variables:

■ the *cv* variable

■ the *thePort* variable

■ the *TrReportBBox* variable

Resolving SKILL Lint Problems

Resolve the following three problems. For each problem, edit and save the source code file if necessary and run SKILL Lint again. Enter appropriate package prefixes, such as *Tr* into the SKILL Lint form.

1. Resolve the problem concerning the *TrReportBBox* variable. You can choose one of the following actions:
 - To advise SKILL Lint about the *Tr* package prefix.
 - To rewrite the *TrShapeReport* program so that it does not rely on the *TrReportBBox* global variable.
 2. Resolve the problems concerning the *aPort* and *thePort* variables.
 3. Resolve the problem concerning the *cv* variable.
- ⇒ Check your work against the solution on the next page.

Answers

1. In the SKILL Lint form, you can list *Tr* as a package prefix. This informs SKILL Lint that you own all global variables that start with *Tr*.

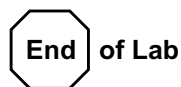
Alternatively, you may get rid of the *TrReportBBox* variable since it is used only once as an argument to the *view* function.

```
procedure( TrShapeReport( wid )
  let( ( aPort )
    thePort = outfile( "/tmp/ShapeReport.txt"
  )
  when( thePort
    TriShapeReport( wid thePort )
    close( thePort )
  ) ; when
  view( "/tmp/ShapeReport.txt"
    '( (120 729) (372 876) )
    "Shape Report" )
  ) ; let
) ; procedure
```

2. In the *TrShapeReport* function, *aPort* is incorrect. Use *thePort* instead.
3. In the *TriShapeReport* function, make *cv* a local variable.

Lab Summary

In this lab, you ran SKILL Lint and resolved several problems that it uncovered.



Labs for Module 12

List Construction

Lab 12-1 Revising the Layer Shape Report

Objective: Use the *setof* and *length* functions to count shapes in a design.

In this lab, you use the *setof* and *length* functions to revise the *TrShapeReport* function that you developed earlier.

If you prefer, use the sample solution:

```
~/SKILL/FlowOfControl/Solutions/TrShapeReport.il
```

Requirements

- Maintain the source code for your revised *TrShapeReport* function in the *~/SKILL/ListConstruction/TrShapeReport.il* file.
- To count the number of rectangles, use the following lines of code:

```
rectCount = length(  
    setof( shape cv~>shapes  
        shape~>objType == "rect" )  
    )
```
- To compute *miscCount*, sum the other counts and subtract them from the number of shapes in the design.

Testing Your Solution

1. Using the same designs as in Lab 9-1, run the old version of the *TrShapeReport* function.

The CIW displays the report.

2. Redefine the *TrShapeReport* function and run against the same designs.

The CIW displays the report.

Are the two reports identical?

— Check your work against the solution on the next page.

Sample Solution

The `~/SKILL/ListConstruction/Solutions/TrShapeReport.il` file contains the solution shown here.

```
procedure( TrShapeReport( wid )
  let( ( rectCount polygonCount pathCount miscCount cv )
    cv = geGetWindowCellView( wid )
    printf(
      "%s %s %s contains:"
      cv~>libName cv~>cellName cv~>viewName )

    rectCount    = length(
      setof( shape cv~>shapes shape~>objType == "rect" ))
    polygonCount = length(
      setof( shape cv~>shapes shape~>objType == "polygon" ))
    pathCount    = length(
      setof( shape cv~>shapes shape~>objType == "path" ))

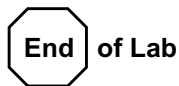
    miscCount =
      length( cv~>shapes ) -
      ( rectCount + polygonCount + pathCount )

    printf( "\n%-10s %-10d" "Rectangles" rectCount )
    printf( "\n%-10s %-10d" "Polygons" polygonCount )
    printf( "\n%-10s %-10d" "Paths" pathCount )
    printf( "\n%-10s %-10d" "Misc" miscCount )
  ) ; let
) ; procedure
```

Lab Summary

In this lab, you used the *length* and *setof* functions in a new version of the *TrShapeReport* function.

Which version do you prefer? Why?



Lab 12-2 Describing the Shapes in a Design

Objective: Use the *foreach mapcar* function to build a list of shape descriptions.

In this lab, you write a function that returns a list of descriptions of the shapes in a design.

For the purposes of this lab, a shape description is a three element list that identifies the object type, the layer name, and the layer purpose of a shape as in this example:

```
( "rect" "metall" "drawing" )
```

Requirements

Make the *TrShapeList* function do the following tasks:

- Accept the window ID of the design window. Name the argument *wid*.
- Return a list of shape descriptions for all the shapes in a design.

Recommendations

Use the following:

- *geGetWindowCellView* function to determine the *cellView* database object for the design.
- *~>* operator to retrieve the *shapes* attribute of this database object.
- *foreach mapcar* function to traverse the list of shapes in the design.

For each shape, do the following:

- Retrieve the *lpp* attribute of the shape.
- Use the *cons* function to add the *objType* attribute onto the front of the *lpp* list.

Testing Your Solution

Test your function on the same designs you used in previous labs.

— Check your work against the solution on the next page.

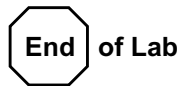
Sample Solution

```
procedure( TrShapeList( wid )
  let( ( cv )
    cv = geGetWindowCellView( wid )
    foreach( mapcar shape cv~>shapes
      cons( shape~>objType shape~>lpp )
    ) ; foreach
  ) ; let
) ; procedure
```

The `~/SKILL/ListConstruction/Solutions/TrShapeList.il` file contains the solution.

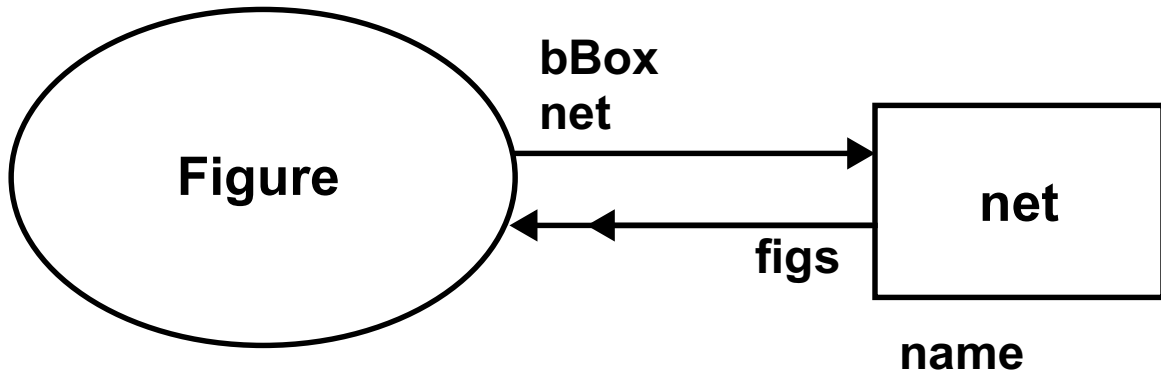
Lab Summary

In this lab, you used the *foreach mapcar* function to process the list of shapes in a design.



Labs for Module 13

Data Models



Lab 13-1 Enhancing the Layer Shape Report SKILL Program

Objective: Develop a SKILL function that itemizes the shapes in a design by layer.

Requirements

Develop a SKILL function named *TrLayerShapeReport* that produces a report tallying the number of each shape on each layer used in a design.

- Each layer purpose pair will occupy a row in the report.
- Each column will represent a **Shape** object type.

Specifically, make the *TrLayerShapeReport* function do the following:

- Accept a single parameter, *cv*, which is the cellview database object.
- Count all the rectangles, lines and labels in the design.
Count all other shapes as miscellaneous.
- Print a report heading followed by a single line for each layer purpose pair in the design on which there are shapes.
- Print *Complete* after the report body.

Continue with the lab on the next page.

Suggestions

- Use this skeleton:

```

;;; print report heading
foreach( LP cv~>layerPurposePairs
  ;;; initialize counters
  foreach( shape LP~>shapes
    case( shape~>objType
      ;;; increment counters
    ) ; case
  ) ; foreach
  ;;; print line in report
) ; foreach

```

- Use the symbols *rectCount*, *labelCount*, *lineCount*, and *miscCount* as local variables to maintain the various counts.
- Use the following *printf* statements for the report heading and for each line in the report:

```

;;; for the report heading
printf(
  "%-15s %-15s %-10s %-10s %-10s %-10s"
  "Layer Name" "Layer Purpose"
  "Rectangle" "Label" "Line" "Misc"
)

;;; for each line of the report

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

```

Testing Your Solution

1. Test your solutions on the *master nand2 layout* design. Be careful to pass a *cellView* database object to your function instead of a window ID.
2. Verify the results by invoking the *leHiSummary* function. Pass it the window ID of your layout application window.

— Check your work against the solution on the next page.

Sample Solution

A suggested solution is in the

~/SKILL/DataModels/Solutions/LayerShapeReport.il file shown here.

```

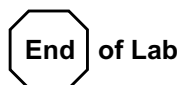
procedure( TrLayerShapeReport( cv )
  let( ( rectCount labelCount lineCount miscCount )
    printf(
      "%-15s %-15s %-10s %-10s %-10s %-10s"
      "Layer Name" "Layer Purpose"
      "Rectangle" "Label" "Line" "Misc"
    )
    foreach( LP cv~>layerPurposePairs
      rectCount = 0
      labelCount = 0
      lineCount = 0
      miscCount = 0
      foreach( shape LP~>shapes

        case( shape~>objType
          ( "rect" ++rectCount )
          ( "line" ++lineCount )
          ( "label" ++labelCount )
          ( t ++miscCount )
        ) ; case
      ) ; foreach
      printf( "\n%-15s %-15s %-10d %-10d %-10d %-10d"
        LP~>layerName LP~>purpose
        rectCount labelCount lineCount miscCount
      )
    ) ; foreach
    printf( "\nComplete" )
  ) ; let
) ; procedure

```

Lab Summary

In this lab, you developed a SKILL function that produces a report that tallies the number of shapes on each layer purpose pair in a design.



Lab 13-2 Creating a Cellview

Objective: To use SKILL functions to create a cellview containing several rectangles.

In this lab, you write a SKILL function to do these tasks:

- Open a cellview.
- Add rectangles to the cellview.
- Save the cellview.

Requirements

Make the *TrRectangles* function accept these four arguments:

- *libName*, *cellName* and *viewName* specify the design.
- *rectList* is a list of rectangle descriptions, such as
("metal1" 0:0 .5:.5)

Make the *TrRectangles* function do the following tasks:

- Open the cellview.
- Add rectangles to the specified design with the corresponding layer name and bounding box.
- Return a list of the "*rect*" database objects.

You can assume that the layer names and bounding boxes are valid.

Getting Started

Before you write the SKILL function, try the following steps to familiarize yourself with the actions your function performs.

1. In the CIW, enter

```
newCV = dbOpenCellViewByType( "master" "new" "layout"
    "maskLayout" "w" )
```

What does the newCV variable now contain?

2. In the CIW, enter

```
newRect = dbCreateRect( newCV "metal1" list( 0:0 .5:.5 ) )
```

What does the newRect variable now contain?

3. In the CIW, enter

```
dbSave( newCV )
dbClose( newCV )
```

4. Use the *geOpen* function to open the *Cells new layout*.

Are you satisfied with what you see?

Writing Your Function

1. Write the *TrRectangles* function based on the requirements.

Testing Your Solution

1. In the CIW, enter

```
geOpen(
    ?lib "master"
    ?cell "new"
    ?view "layout" )
```

2. Delete all the shapes in the design. In the CIW, enter

```
geSelectAll()
foreach( dbObject geGetSelSet()
    dbDeleteObject( dbObject))
```

3. In the CIW, enter

```
TrRectangles( "master" "new" "layout"  
  list(  
    list( "metall" 0:0 1.0:1.5 )  
    list( "poly1" 1:1 2:3 )) )
```

— Check your work against the solution on the next page.

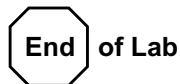
Sample Solution

The `~/SKILL/DataModels/Solutions/Rectangles.il` file contains the solution.

```
procedure( TrRectangles( libName cellName viewName rectList )
  let( ( cv )
    cv = dbOpenCellViewByType( libName cellName viewName
      "maskLayout" "w" )
    when( cv
      foreach( mapcar rectDescription rectList
        dbCreateRect(
          cv
          car( rectDescription )
          cdr( rectDescription )
        )
      ) ; foreach
    ) ; when
  ) ; let
) ; procedure
```

Lab Summary

In this lab, you wrote a function that adds several rectangles to a cellview using the `dbCreateRect` function.



Lab 13-3 Aligning Rectangles

Objective: Use the `~>` operator to update the `bBox` attribute of a rectangle.

In this lab, you develop a function to align several rectangles with a target rectangle. The function aligns the top side of the rectangles.

Requirements

Make the *TrAlignRectangles* function do these tasks:

- Accept a list of rectangles as the single argument *rectList*.
- Move the rectangles so that their top edges are collinear with the top edge of the first rectangle.

Suggestions

The target rectangle is *car(rectList)*.

Use the *foreach* function to process each rectangle in the *cdr(rectList)* list as follows:

- Compute the appropriate *deltaY* distance required to align the rectangle with the target rectangle.
- Use the *geTransformUserBBox* function to compute the new bounding box for the rectangle.

```
newBBox = geTransformUserBBox( rect~>bBox
  list( 0:deltaY "R0" )
)
```
- Use the `~>` operator to update the bounding box of the rectangle.

```
rect~>bBox = newBBox
```

Testing Your Solution

Define a Layout bindkey to make your testing easier.

1. In the CIW, enter

```
hiSetBindKey( "Layout" "<Key>F9"  
  "TrAlignRectangles( geGetSelSet() )" )
```

2. Use the *geOpen* function to put the *master new layout* cellview into a window.

Repeat the following steps to test your function until you are satisfied.

3. Enter several rectangles.
4. Select all the rectangles.
5. Press **F9** to run your function.
6. Undo the effects of your command. In the CIW, enter
 hiUndo()

— Check your work against the solution on the next page.

Sample Solution

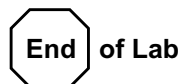
A suggested solution is in the
~/SKILL/DataModels/Solutions/AlignRectangles.il file.

```
procedure( TrAlignRectangles( rectList )
  let( ( targetRect targetY delta )
    targetRect = car( rectList )
    targetY = yCoord( upperRight( targetRect~>bBox ) )
    foreach( rect cdr( rectList )
      deltaY = targetY - yCoord( upperRight( rect~>bBox ) )
      rect~>bBox = getTransformUserBBox(
        rect~>bBox
        list( 0:deltaY "R0" )
      )
    ) ; foreach
  ) ; let
) ; procedure

hiSetBindKey( "Layout" "<Key>F9" "TrAlignRectangles( geGetSelSet() )" )
```

Lab Summary

In this lab, you wrote a SKILL function to align several rectangles with a target rectangle.



Lab 13-4 Exploring Hierarchy

Objective: Use ~> expressions to explore the hierarchy of a design.

Opening a Design

1. In the CIW, select **File** → **Open**.
2. Fill out the Open File form with these values.

Library Name	master
Cell Name	mux2
View Name	schematic
Mode	read

3. Click **OK**.

A Composer™ window displays the *master mux2 schematic* design.

Retrieving a Cellview Database Object from a Window

1. In the CIW, enter

```
cv = geGetWindowCellView( )
```

cv contains a database object for the design appearing in the window.
2. Verify that the cellview type is "*schematic*". In the CIW, enter

```
cv~>cellViewType
```

The system displays the cellview type of the design.

Examining the Instance Masters

1. In the CIW, enter

```
cv~>instanceMasters
```

The system displays a list of database objects. Each is a master that is instantiated in the design.

2. In the CIW, enter

```
cv~>instanceMasters~>cellName
```

The system displays the list of the cell names of the instance masters.

```
( "Inv" "gnd" "vdd" "nand2" "opin" "ipin" )
```

3. In the CIW, enter

```
cv~>instanceMasters~>libName
```

The system displays the list of the library names containing the instance masters.

```
("master" "basic" "basic" "master" "basic" "basic")
```

4. In the CIW, enter

```
cv~>instanceMasters~>viewName
```

The system displays the list of the view names of the instance masters.

```
("symbol" "symbol" "symbol" "symbol" "symbol" "symbol" )
```

Examining an Instance

1. Use the mouse to select the inverter instance named *"I6"*.

2. In the CIW, enter

```
I6 = car( geGetSelSet( ) )
```

I6 contains the first (and only) element of the selected set.

3. Verify that the instance name is *"I6"*. In the CIW, enter

```
I6~>name
```

The system displays the name of the instance.

Querying the Master

1. In the CIW, enter

```
I6~>master
```

The system displays the database object of the master.

2. Verify that the cellview type is *"schematicSymbol"*. In the CIW, enter

```
I6~>master~>cellViewType
```

The system displays the cellview type of the master.

3. In the CIW, enter

```
I6~>cellName
```

The system displays the cell name of the instance master.

4. In the CIW, enter

```
I6~>master~>cellName
```

The system displays the cell name of the instance master.

5. In the CIW, enter

```
I6~>master~>viewName
```

The system displays the view name of instance master.

Viewing the Master

1. In the CIW, enter

```
geOpen( ?lib I6~>libName ?cell I6~>cellName ?view I6~>viewName ?mode "r")
```

The window appears with the following title:

```
Virtuoso-Symbol Reading: master inv symbol
```

Lab Summary

In this lab, you

- Retrieved the list of master cellviews instantiated in a top-level cellview.
- Determined the *cellViewType* and the *viewName*, *libName*, *cellName* attributes of all of the master cellviews.
- Selected an instance and queried its master cellview.



End of Lab

Lab 13-5 Traversing a Hierarchical Design

Objective: Run the *TrHierarchyTraversal* function.

Opening a Design

1. Open the design *master mux2 layout*. In the CIW, enter

```
mux2Layout = dbOpenCellViewByType( "master" "mux2" "layout" )
```

Loading the *TrHierarchyTraversal* Function

1. In the CIW, enter

```
load( "HierarchyTraversal.il" )
```

The source code defines the following functions:

- the *TrSwitch* function
- the *TrHierarchyTraversal* function
- the *TrReferenceFromDBObject* function

Continue with the lab on the next page.

Running the *TrHierarchyTraversal* Function

1. In the CIW, enter

```
expandedHierarchy = TrHierarchyTraversal( mux2Layout nil )
```

The variable *expandedHierarchy* now contains a list of database objects. Each database object in the list is a master cellview instantiated in the hierarchy beneath the top-level cellview *mux2Layout*.

2. Determine the libraries that each of the masters came from. In the CIW, enter

```
expandedHierarchy~>libName
```

SKILL displays:

```
( "pCells" "pCells" "master" "pCells" "pCells"
  "master" "cellTechLib" "cellTechLib"
)
```

3. Determine the cell names of each of the database objects. In the CIW, enter

```
expandedHierarchy~>cellName
```

SKILL displays:

```
( "ptranA" "ntran" "nand2" "ntransistor" "ptransistor"
  "Inv" "M1_POLY1" "M2_M1"
)
```

4. Determine the view names of each of the database objects. In the CIW, enter

```
expandedHierarchy~>viewName
```

SKILL displays:

```
( "layout" "layout" "layout" "layout" "layout"
  "layout" "symbolic" "symbolic"
)
```

Closing All the Master Cellviews

The next SKILL expression you enter uses the *dbClose* function to close all the master cellviews that the *TrHierarchyTraversal* function opened. To understand the rationale for the expression, consider the following points:

- When you call the *dbClose* function, CDBA does not necessarily purge the *cellView* database object from memory.
- CDBA only purges it when no other *cellView* database object refers to it.
- On the other hand, if you pass a purged database object to *dbClose*, the function raises an error and returns *nil*.
- When an expression in a *foreach* loop raises an error, SKILL aborts the *foreach* loop.
- You can trap a SKILL error within an expression by enclosing it with the *errset* function.

1. In the CIW, enter

```
foreach( master expandedHierarchy errset( dbClose( master )))
```

The system displays the return result of the *foreach* function.
It is the value of the *expandedHierarchy* variable.

Checking for Purged Cellview Database Objects

To check whether CDBA has purged a *cellView* database object, retrieve one of the object attributes, such as the *libName* attribute.

1. In the CIW, enter

```
expandedHierarchy~>libName
```

SKILL does the following tasks:

- It displays a warning for each of the master cellviews in the *expandedHierarchy* list that CDBA has purged from memory.
- It returns a list with *nil* or the value of the *libName* attribute for each *cellView* database object.

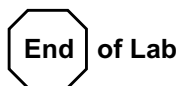
```
dbGetq: Object is a purged/freed object. - db:41192492
dbGetq: Object is a purged/freed object. - db:41193516
dbGetq: Object is a purged/freed object. - db:40007724
dbGetq: Object is a purged/freed object. - db:40009772
dbGetq: Object is a purged/freed object. - db:40010796
dbGetq: Object is a purged/freed object. - db:40016940
(nil nil nil nil nil
 nil nil nil
)
```

Because each element of the list is *nil*, you can conclude that CDBA has purged all the master database objects in the *expandedHierarchy* list.

Lab Summary

In this lab, you used the

- *TrHierarchyTraversal* function on a top-level layout cellview.
- *~>* operator to determine the library name, cell name and view name of each master cellview in the expanded hierarchy.
- *dbClose* to close all the masters in the expanded hierarchy.



Lab 13-6 Developing a Netlister

Objective: To develop a simple netlister.

In this lab, you develop a simple net-based netlister.

Structure your netlister as two functions: the *TrNetList* function and the *TriNetList* function. The prefix *Tri* indicates that the *TriNetList* function is an internal or private *Tr* function.

Requirements for the *TriNetList* Function

1. Make the *TriNetList* function accept a single argument: *cv* is the *cellView* database object of the design to netlist.
2. Make the *TriNetList* function use the following format for the netlist:

```
NET: <net name> TERM: <terminal name or "none" >
    INST_TERM: <instTerm name1> INST: <instance name1>
    INST_TERM: <instTerm name2> INST: <instance name2>
    INST_TERM: <instTerm name3> INST: <instance name3>
    ...
```

Requirements for the *TrNetList* Function

1. Make the *TrNetList* function accept three arguments:
 - *libName* is the library name of the design to netlist.
 - *cellName* is the cell name.
 - *viewName* is the view name.
2. Make the *TrNetList* function perform these tasks:
 - Call the *dbOpenCellViewByType* function to open the design in memory.
 - Call the *TriNetList* function.
 - Call the *dbClose* function.

Testing Your Solution

1. In the CIW, enter

```
TrNetList( "master" "mux2" "schematic" )
```

The system displays:

```
NET: B TERM: B
INST_TERM: A          INST I1
NET: net7 TERM: None
INST_TERM: A          INST I3
INST_TERM: Y          INST I6
NET: net6 TERM: None
INST_TERM: Y          INST I3
INST_TERM: B          INST I0
NET: Y TERM: Y
INST_TERM: Y          INST I0
NET: A TERM: A
INST_TERM: B          INST I3
NET: vdd! TERM: None
INST_TERM: vdd!       INST I8
NET: gnd! TERM: None
INST_TERM: gnd!       INST I9
NET: SEL TERM: SEL
INST_TERM: B          INST I1
INST_TERM: A          INST I6
NET: net4 TERM: None
INST_TERM: Y          INST I1
INST_TERM: A          INST I0
```

2. In the CIW, enter

```
TrNetList( "master" "mux2" "layout" )
```

The system displays:

```
NET: SEL TERM: SEL
NET: B TERM: B
NET: A TERM: A
NET: gnd! TERM: gnd!
NET: Y TERM: Y
NET: vdd! TERM: vdd!
```

3. In the CIW, enter

```
TrNetList( "master" "mux2" "extracted" )
```

The system displays the following output.

```
NET: A                TERM: A
INST_TERM: G          INST +0
INST_TERM: G          INST +7
NET: 2                TERM: None
INST_TERM: S          INST +1
INST_TERM: D          INST +0
NET: 3                TERM: None
INST_TERM: S          INST +3
INST_TERM: D          INST +2
NET: B                TERM: B
INST_TERM: G          INST +10
INST_TERM: G          INST +3
NET: 5                TERM: None
INST_TERM: S          INST +11
INST_TERM: G          INST +8
INST_TERM: S          INST +4
INST_TERM: G          INST +1
NET: SEL              TERM: SEL
INST_TERM: G          INST +11
INST_TERM: G          INST +9
INST_TERM: G          INST +4
INST_TERM: G          INST +2
...
```

The output has been truncated to fit the page.

— Check your work against the solution on the next page.

Sample Solution

1. The following solution is in the
~/SKILL/DataModels/Solutions/NetBasedNetlister.il file.

```

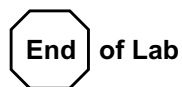
procedure( TriNetList( cv )
  foreach( net cv~>nets
    printf( "NET: %-15s TERM: %s\n"
      net~>name (net~>term~>name || "None")
    )
    foreach( instTerm net~>instTerms
      printf( " INST_TERM: %-10s INST %s\n"
        instTerm~>name instTerm~>inst~>name
      )
    ) ; foreach
  ) ; foreach
t
) ; procedure

procedure( TrNetList( libName cellname viewName )
  let( ( cv )
    cv = dbOpenCellViewByType( libName cellname viewName )
    when( cv
      TriNetList( cv )
      dbClose( cv )
    ) ; when
  ) ; let
) ; procedure

```

Lab Summary

In this lab, you traced the connectivity in a schematic.



Lab 13-7 Developing an Instance-based Netlister

Objective: Develop a simple instance-based netlister.

In this lab, you develop an instance-based netlister.

Requirements for the *TriInstNetList* Function

Make the *TriInstNetList* function accept a single argument:

cv is the *cellView* database object of the design to netlist.

Make the *TriInstNetList* function use the following format for the netlist:

```
INST: <inst name>
  INST_TERM: <instTerm name1> NET: <net name1>
  INST_TERM: <instTerm name2> NET: <net name2>
  INST_TERM: <instTerm name3> NET: <net name3>
...
or
INST: <inst name> TERM: <terminal name> NET: <net name>
```

Requirements for the *TriInstNetList* Function

Make the *TriInstNetList* function accept three arguments:

- *libName* is the library name of the design to netlist.
- *cellName* is the cell name.
- *viewName* is the view name.

Make the *TriInstNetList* function do these tasks:

- Call the *dbOpenCellViewByType* function to open the design into memory.
- Call the *TriInstNetList* function.
- Call the *dbClose* function.

Testing Your Solution

1. In the CIW, enter

```
TrInstNetList( "master" "mux2" "schematic" )
```

The system displays:

```
INST: I6
  INST_TERM: A      NET SEL
  INST_TERM: Y      NET net7
INST: I9
  INST_TERM: gnd!   NET gnd!
INST: I8
  INST_TERM: vdd!   NET vdd!
INST: I3
  INST_TERM: B      NET A
  INST_TERM: Y      NET net6
  INST_TERM: A      NET net7
INST: I1
  INST_TERM: B      NET SEL
  INST_TERM: Y      NET net4
  INST_TERM: A      NET B
INST: I0
  INST_TERM: B      NET net6
  INST_TERM: Y      NET Y
  INST_TERM: A      NET net4
INST: I5 TERM: Y      NET: Y
INST: I7 TERM: SEL    NET: SEL
INST: I4 TERM: B      NET: B
INST: I2 TERM: A      NET: A
```

2. In the CIW, enter

```
TrInstNetList( "master" "mux2" "layout" )
```

The system displays:

```
INST: I35  
INST: I31  
INST: I30  
INST: I32  
INST: I36  
INST: I4  
INST: I5  
INST: I34  
INST: I41  
INST: I40  
INST: I37  
INST: I38  
INST: I6  
INST: I3  
INST: I18  
INST: I1  
INST: I2
```

3. In the CIW, enter

```
TrInstNetList( "master" "mux2" "extracted" )
```

The system displays the following output.

```
INST: +13
  INST_TERM: B      NET vdd!
  INST_TERM: G      NET 10
  INST_TERM: D      NET vdd!
  INST_TERM: S      NET Y
INST: +12
  INST_TERM: B      NET vdd!
  INST_TERM: G      NET 7
  INST_TERM: D      NET Y
  INST_TERM: S      NET vdd!
INST: +11
  INST_TERM: B      NET vdd!
  INST_TERM: G      NET SEL
  INST_TERM: D      NET vdd!
  INST_TERM: S      NET 5
INST: +10
  INST_TERM: B      NET vdd!
  INST_TERM: G      NET B
  INST_TERM: D      NET vdd!
  INST_TERM: S      NET 10
INST: +9
  INST_TERM: B      NET vdd!
  INST_TERM: G      NET SEL
  INST_TERM: D      NET 10
  INST_TERM: S      NET vdd!
...
```

The output has been truncated to fit the page.

— Check your work against the solution on the next page.

Sample Solution

1. The following solution is in the
~/*SKILL*/DataModels/Solutions/InstBasedNetlister.il file.

```

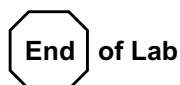
procedure( TriInstNetList( cv )
  foreach( inst cv~>instances
    if( inst~>net
      then
        printf( "INST: %s TERM: %-10s NET: %-10s\n"
          inst~>name inst~>net~>term~>name inst~>net~>name
        )
      else
        printf( "INST: %s\n" inst~>name )
        foreach( instTerm inst~>instTerms
          printf( " INST_TERM: %-10s NET %s\n"
            instTerm~>name instTerm~>net~>name
          )
        ) ; foreach
      ) ; if
    ) ; foreach
  t
) ; procedure

procedure( TrInstNetList( libName cellname viewName )
  let( ( cv )
    cv = dbOpenCellViewByType( libName cellname viewName )
    when( cv
      TriInstNetList( cv )
      dbClose( cv )
    ) ; when
  ) ; let
) ; procedure

```

Lab Summary

In this lab, you traced the connectivity in a cellview.



Lab 13-8 Exploring User-Defined Properties

Objective: To display the user-defined properties on a cellview.

Opening a Design

1. In the CIW, select **File** → **Open**. Open a cellview of interest to you from the *master* library.
2. Note the application window number in the upper right corner for future use. In the CIW, enter the following but replace the # character with the window number of the application window:

```
designWid = window( # )
topLevelCV = geGetWindowCellView( designWid )
```

Displaying Property Names and Values

1. Create a file called *TrShowProp.il*. Enter the following into the CIW:

```
edit( "~/SKILL/DataModels/TrShowProp.il" t )
```

2. Enter the following code:

```
procedure( TrShowProp( cv )
  foreach( item cv~>prop
    printf("%s " item~>name)
    print( item~>value)
  ) ; foreach
  t
) ; procedure
```

Why do we use the print function instead of the printf?

The *print* and *println* functions handle any data type. You cannot predict the data type of *item~>value*.

3. Save your edits and exit the editor. In the CIW, enter

```
TrShowProp( topLevelCV )
```

The user-defined property names and values appear on a single line in the CIW.

Make each name and value appear on a separate line.

4. Use the editor again to change the line from

```
print(item~>value)
```

to

```
println(item~>value)
```

5. Save your edits and then exit the editor again. In the CIW, enter

```
TrShowProp( topLevelCV )
```

Building a List of Property Name and Value Pairs

Sometimes you need a SKILL list of the name and value pairs. In the CIW, enter

```
foreach( mapcar prop topLevelCV~>prop  
list( prop~>name prop~>value ))
```

The system displays a return result similar to the following:

```
(( "segSnapMode" "anyAngle")
  ("snapMode" "diagonal")
  ("ySnapSpacing" 0.5)
  ("xSnapSpacing" 0.5)
  ("gridMultiple" 5)
  ("gridSpacing" 1.0)
  ("drawDottedGridOn" "TRUE")
  ("drawGridOn" "TRUE")
  ("stopLevel" 20)
  ("drcSignature" 145386014)
  ("drcChecked" "Jul 11 14:14:39 1992")
  ("borderCols" 1)
  ("startLevel" 0)
  ("borderOnly" "FALSE")
  ("iconsOn" "FALSE")
  ("originMarkersOn" "TRUE")
  ("drawInstancePins" "FALSE")
  ("pathCL" "yes")
  ("accessEdgesOn" "FALSE")
  ("drawAxesOn" "TRUE")
  ("netsOn" "FALSE")
  ("abstractViewName" "abstract")
  ("drawSurroundingOn" "TRUE")
  ("pin#" 15)
  ("textJustificationOn" "FALSE")
  ("instLabel" "master")
  ("borderRows" 1)
  ("instance#" 18)
  ("instancesLastChanged"
    "Aug 18 12:19:47 1992")
  ))
```

How does this compare with what the `TrShowProp` function does?

— Check your work against the solution on the next page.

Answer

The *foreach(mapcar ...)* expression behaves as follows:

- Does not print anything.
- Returns a SKILL list of property name value pairs.

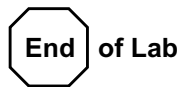
The *TrShowProp* function behaves as follows:

- Prints formatted output to the CIW.
- Returns the *t* value.

Lab Summary

In this lab exercise, you

- Displayed the user-defined properties of a design in the CIW.
- Built a list of the user-defined property names and values.



Lab 13-9 Dumping Database Objects

Objective: Use the ~> operator and the Show File Browser to dump database objects.

Opening a Design

1. Open the design *master mux2 layout*. In the CIW, enter

```
mux2Layout = dbOpenCellViewByType( "master" "mux2" "layout" )
```

The system allocates a database object in virtual memory and assigns it to the variable *mux2Layout*.

Using the ~> Operator to Dump Database Objects

1. List the attribute names on *mux2Layout*. In the CIW, enter

```
mux2Layout~>?
```

The system displays the following list:

```
( cellView objType prop bBox lib
  libName cellName cell cellViewType conns
  DBUPerUU fileName groupMembers groups instHeaders
  instHeaderRefs instRefs instanceMasters instances isParamCell
  layerPurposePairs lpps memInsts mode modifiedButNotSaved
  mosaics nets physConns shapes signals
  sigNames subMasters superMaster terminals userUnits
  viewName view textDisplays assocTextDisplays needRefresh
  netCount anyInstCount termCount
)
```

Each element in the list is an attribute on *mux2Layout*.

2. List the instances in *mux2Layout*. In the CIW, enter

```
mux2Layout~>instances
```

The system displays a list of database objects each of which represents an instance in *mux2Layout*.

3. Verify the object type of each element of the list. In the CIW enter

```
mux2Layout~>instances~>objType
```

The system displays the following list:

```
( "inst" "inst" "inst" "inst" "inst"
  "inst" "inst" "inst" "inst" "inst"
  "inst" "inst" "inst" "inst" "inst"
  "inst" "inst"
)
```

4. List the attribute names for the first instance. In the CIW, enter

```
car( mux2Layout~>instances )~>?
```

The system displays the following list in the CIW:

```
(cellView objType prop bBox children
groupMembers isAnyInst isShape matchPoints net
parent physConns pin purpose textDisplays
assocTextDisplays baseName cellName instHeader instTerms
libName master name numInst viewName
conns mag orient status transform
xy
)
```

Each element of the list is an attribute name for the *inst* object type.

Using the Show File Browser

The Show File Browser writes all the attribute names and values for a database object to a file and displays the file in a Show File window.

1. In the CIW, enter

```
load( "~/SKILL/CaseStudy/ShowFileBrowser.il" )
TrShowFileBrowser( mux2Layout )
```

The system opens a Show File Browser window displaying information similar to the following:

```
( "cellView32521260"
  ( cellView "cellView32521260" )
  ( objType "cellView" )
  ( prop
    ( "prop33510272" "prop33510200" "prop33510148" "prop33510096" "prop33510044"
      "prop33509996" "prop33509940" "prop33509892" "prop33509844" "prop33509692"
      "prop33509644" "prop33509620" "prop33509420" "prop33466340" "prop33466296"
      "prop33466244" "prop33466188" "prop33466128" "prop33466076" "prop33466028"
      "prop33465968" "prop33465936" "prop33477388" "prop33477324" "prop33477236"
      "prop33477192" "prop33477112" "prop32521872" "prop32521384"
    )
  )
  . . .
)
```

The Show File Browser displays database objects, such as *db:24203004* as *inst24203004*. It uses the same digits but replaces the *db:* with the object type of the database object.

Browsing the First Instance

To browse the first instance in this design, follow these steps:

1. Scroll the Show File Browser window until a portion resembling the following lines becomes visible:

```
( instances
  ( "inst33513212" ...
    ...
  )
)
```

Remember that the database object references you see are probably different.

2. Select the database object reference by double clicking the left button over the *"inst33513212"*.

The text between the quotes is highlighted.



3. Pop up the Browser menu and select **Browse Selection** with the middle mouse button.

A Show File Browser appears for the instance.

Navigating Among Show File Browser Windows

You can retrace your steps through a sequence of Show File Browser windows. Successive Show File Browser windows are linked in a *parent/child* relationship.

1. In the current Show File Browser window, pop up the Browser menu and select **Raise Parent Browser**.

The initial Show File Browser for the cellview database object becomes the top window.

2. In your initial Show File Browser, pop up the Browser menu and select **Raise Most Recent Child Browser**.

The Show File Browser for the *inst* database object becomes the top window.

Raising the CIW

You can enter SKILL expressions into the CIW to aid to your exploration. Another Browser menu option allows you to raise the CIW.

1. In either Show File Browser window, pop up the Browser menu and select **Raise CIW**.

The CIW becomes the top window.

Clearing the Selection

You can clear all the selected text in a Show File Browser window with the Clear Selection command.

1. In either Show File Browser, select several database object references.



2. Pop up the Browse menu and select **Clear Selection**.

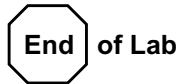
The selected text is cleared.



Lab Summary

In this lab, you used the

- *dbOpenCellViewByType* function to load a design into virtual memory.
- *~>?* expression to retrieve a list of the attribute names on a database object.
- Show File Browser to browse a sequence of database objects.



Lab 13-10 Building the Cellview Data Model Road Map

Objective: Execute a SKILL function to build the Cellview Database Model Road Map

In this lab, you execute the *TrRoadMap* SKILL function to build the Cellview Data Model Road Map.

The *TrRoadMap* function accepts a single argument named *wid*. When you call the *TrRoadMap* function, pass the window ID of a window that contains an empty layout cellview. The *TrRoadMap* function calls several *dbCreate** functions to create the shapes that comprise the Cellview Data Model Road Map.

1. In the CIW, enter

```
load( "~/SKILL/CaseStudy/RoadMap.il" )
```

The software defines the *TrRoadMap* SKILL function.

2. Use the **File** → **New** → **Cellview** command to create the cell *master RoadMap layout*.

A window appears. It is the current window.

3. In the CIW, enter

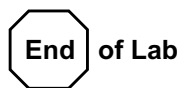
```
TrRoadMap( hiGetCurrentWindow() )
```

The *TrRoadMap* function creates the road map. You might have to use the **Fit** → **All** command so you can view the road map.

4. Save the design.

Lab Summary

You built the Cellview Database Model Road Map as a layout cellview.



Labs for Module 14

User Interface

Lab 14-1 Exploring Fixed Menus

Objective: Create and display a fixed menu and attach it to an application window.

Building a Sample Fixed Menu

Build a vertical fixed menu using the *hiCreateVerticalFixedMenu* function. Make each menu item display its *?name* in the CIW when you select the menu item.

1. Edit the `~/SKILL/UserInterface/FixedMenu.il` file.
2. In the editor, select the definition of the *TrCreateMenuItem* function. Paste it into the CIW and press **Return**.

```
procedure( TrCreateMenuItem( theMenuSymbol )
  set(
    theMenuSymbol
    hiCreateMenuItem(
      ?name      theMenuSymbol
      ?itemText  get_pname( theMenuSymbol )
      ?callback  sprintf( nil "println( '%L )" theMenuSymbol )
    )
  ) ; set
) ; procedure
```

The *TrCreateMenuItem* function uses two useful functions.

- The *get_pname* function returns the name of the symbol as a text string. Thus, the *?itemText* of the menu item is the name of symbol contained in the *theMenuSymbol* variable.
- The *set* function assigns the data structure of the menu item to the symbol contained in the *theMenuSymbol* variable. The *set* function is related to the *setq* function that implements the = operator.

3. In the editor, select the six invocations of *TrCreateMenuItem* and paste them in the CIW.

```
TrCreateMenuItem( 'item1 ' )
TrCreateMenuItem( 'item2 ' )
TrCreateMenuItem( 'item3 ' )
TrCreateMenuItem( 'item4 ' )
TrCreateMenuItem( 'item5 ' )
TrCreateMenuItem( 'item6 ' )
```

The variables, *item1*, *item2*, and so forth, now contain the data structures for menu items.

4. In the editor, select the invocation of the *hiCreateVerticalFixedMenu* function and paste it into the CIW.

```
hiCreateVerticalFixedMenu(
    'TrExampleVerticalFixedMenu
    list( item1 item2 item3 item4 item5 item6 )
    6    ;;; number of rows
    1    ;;; number of columns
)
```

Displaying the Fixed Menu

1. In the editor, select the invocation of the *hiDisplayFixedMenu* function and paste it in the CIW.

```
hiDisplayFixedMenu(
    TrExampleVerticalFixedMenu
    "left" ;;; placement must be one of "top", "bottom", "right", or
    "left"
)
```

2. Click on the menu items to confirm their operation.

Each menu item displays its *?name* in the CIW.

3. Click the **Done** button.

The fixed menu disappears.

In the next step, you attach the fixed menu to a design window.

Opening a Design Window

1. In the CIW, select **File** → **Open**.
2. Fill out the Open File form with these values.

Library Name	master
Cell Name	mux2
View Name	layout
Mode	read

3. Click **OK**.

A Virtuoso window displays the *master mux2 layout* design. The window has a fixed menu attached to its left side.

4. Ensure that it is the current window by pulling down one of its menus and sliding off of it.

Retrieving the Attached Fixed Menu

1. In the CIW, enter

```
attachedMenu = hiGetWindowFixedMenu( )
```

The system stores the data structure for the attached fixed menu in the *attachedMenu* variable. You will refer to *attachedMenu* later.

Removing the Fixed Menu

1. In the CIW, enter

```
hiRemoveFixedMenu( )
```

The attached fixed menu disappears from the side of the application window.

What is the return value?

Attaching the Sample Fixed Menu

1. In the CIW, enter

```
hiAddFixedMenu( ?fixedMenu TrExampleVerticalFixedMenu )
```

What is the return value?

Restoring the Original Fixed Menu

You do not need to remove a fixed menu before adding another one.

1. Reattach the original fixed menu to the window. In the CIW, enter

```
hiAddFixedMenu( ?fixedMenu attachedMenu )
```

The original fixed menu is attached again.

Lab Summary

In this lab, you

- Built and displayed a fixed menu.
- Retrieved the fixed menu.
- Removed the fixed menu.
- Attached a fixed menu.
- Replaced an attached fixed menu with another one.

Lab 14-2 Exploring Dialog Boxes

Objective: Create and Display Various Dialog Boxes.

Using a Modeless Dialog Box

1. In a text editor of your choice, enter:

```
hiDisplayAppDBox(
    ?name          gensym( 'TrDBox )
    ?dboxBanner    "Example"
    ?dboxText      "First\nSecond Line\nThird Line"
    ?dialogType    hicInformationDialog
    ?callback      "printf( \"\nExecuting Callback ..done\");"
    ?dialogStyle   'modeless
)
```

Specifying *?name gensym('TrDBox)* ensures that each dialog box has a unique variable to hold the data structure of the dialog box. This is crucial for modeless dialog boxes that might be on the screen indefinitely.

2. Paste the expression into the CIW.

A modeless dialog box appears.

3. Observe the icon in the dialog box.

4. Move the cursor to the CIW to verify that you can pull down a menu.

A modeless dialog box does not prevent you from interacting with the Design Framework II environment.

5. Repeat the above steps with the following values for *?dialogType*:

```
hicErrorDialog hicInformationDialog
hicMessageDialog hicQuestionDialog
hicWarningDialog hicWorkingDialog
```

Using a Modal Dialog Box

1. In a text editor of your choice, enter

```
hiDisplayAppDBox(  
    ?name          gensym( 'TrDBox )  
    ?dboxBanner    "Example"  
    ?dboxText       "First\nSecond Line\nThird Line"  
    ?dialogType     hicInformationDialog  
    ?callback       "printf( \"\nExecuting Callback ..done\");"  
    ?dialogStyle    'modal  
)
```

2. Paste the expression into the CIW.

A modal dialog box appears.

3. Move the cursor to the CIW to verify that you CANNOT pull down a menu.

A modal dialog box prevents you from interacting with the Design Framework II environment. Yet you can still interact with applications outside the Design Framework II environment.

Using a System Modal Dialog Box

1. In a text editor of your choice, enter

```
hiDisplayAppDBox(  
    ?name          gensym( 'TrDBox )  
    ?dboxBanner    "Example"  
    ?dboxText       "First\nSecond Line\nThird Line"  
    ?dialogType     hicInformationDialog  
    ?callback       "printf( \"\nExecuting Callback ..done\");"  
    ?dialogStyle    'systemModal  
)
```

2. Paste the expression into the CIW.

A system modal dialog box appears.

3. Move the cursor to the CIW to verify that you CANNOT pull down a menu.

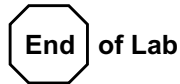
4. Move the mouse off the dialog box to verify you CANNOT interact with any other window in the environment.

A system modal dialog box prevents you from interacting with all applications, even those outside the Design Framework II environment.

Lab Summary

In this lab, you

- Observed the dialog box icons you can use.
- Experienced interaction constraints for application modal, system modal, and modeless dialog boxes.



Lab 14-3 Exploring List Boxes

Objective: Load and examine sample source code that invokes *hiShowListBox*.

Examining a Simple List Box Application

1. In the CIW, enter

```
load( "~/SKILL/UserInterface/ListBox.il" )
```

This code defines several functions, including the *TrShowListBox* function and the *TrExampleListBoxCB* function.

2. Examine the source code. In the CIW, enter

```
view( "~/SKILL/UserInterface/ListBox.il" )
```

3. Display a list box. In the CIW, enter

```
TrShowListBox( '( "apple" "orange" "kiwi" ) )
```

A list box appears.

4. Make a selection and click on **OK**.

Can you describe what happens?

5. Display a second list box. In the CIW, enter

```
TrShowListBox( '( "wood" "water" "metal" ) )
```

A second list box appears.

6. Verify that the two list boxes behave independently.

*How does the *TrShowListBox* function guarantee an independent list box each time you invoke it?*

Exploring an Advanced List Box Application

1. In the CIW, enter

```
load( "~/SKILL/UserInterface/FunctionListBox.il" )
```

This code defines the

- *TrFunctionListBox* function
- *TrListFunctions* function
- *TrFunctionListBoxCB* function

2. Examine the source code. In the CIW, enter

```
view( "~/SKILL/UserInterface/FunctionListBox.il" )
```

3. Display a list box. In the CIW, enter

```
TrFunctionListBox( "^hiGet" )
```

A list box appears. It includes choices for all functions whose names begin with *hi* and include *Box*.

4. Several times, make a selection and click **Apply**.

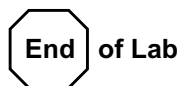
Each time you click **Apply**, the system displays the arguments list for the function.

5. Click **Cancel**.

The list box disappears.

Lab Summary

In this lab, you loaded and examined two sample List Box applications.



Lab 14-4 Exploring Forms

Objective: Examining source code for sample form fields.

Examining the File Form Application

1. In the CIW, enter

```
view( "~/SKILL/UserInterface/FileForm.il" )
```

2. Examine the source code.

3. In the CIW, enter

```
load( "~/SKILL/UserInterface/FileForm.il" )
```

4. Interact with the form and compare the results with the code.

Sampling Various Form Fields

1. In the CIW, enter

```
view( "~/SKILL/UserInterface/FormFields.il" )
```

What fields are not used in the form?

2. In the CIW, enter

```
load( "~/SKILL/UserInterface/FormFields.il" )
```

3. Interact with the form and compare the results with the code.

Lab 14-5 Writing a Form Application

Objective: To build a form that resizes and renames a window.

Requirements

Build a form to allow the user to change the name or size of a window.

Make the form have the title "*window:1*" where the appropriate window number is substituted for *1*.

Make *OK* and *Apply* use current field values to set the name of window and to resize the window.

Make the form have the following fields:

- Name field.

A string field with prompt "*Name*".

The initial and default value should be the window's name.

- Height Field

A scale field with prompt "*Height*".

The initial and default value should be the current height of the window.

The value range should be from 0 to the maximum allowable *y* value.

- Width Field

A scale field with prompt "*Width*".

The initial and default value should be the current width of the window.

The value range should be from 0 to the maximum allowable *x* value.

Assumptions

Assume that there will be only one instance of your form active at any single time. This assumption is removed in a subsequent optional portion of this lab.

Suggestions

Structure your solution into two functions.

- The *TrCreateWindowForm* function

This function builds the form.

- The *TrWindowFormCB* function

This function is the callback for the form and handles the renaming and resizing of the window.

You can use the utility functions, such as *TrResizeWindow*, in *~/SKILL/UserInterface/WindowUtilities.il*.

1. In the CIW, enter

```
view( "~/SKILL/UserInterface/WindowUtilities.il" )
```

2. In the CIW, enter

```
load( "~/SKILL/UserInterface/WindowUtilities.il" )
```

Detailed suggestions follow on the next page.

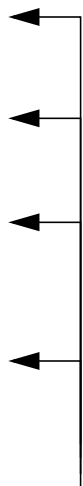
Developing the *TrCreateWindowForm* Function

Edit the `~/SKILL/UserInterface/WindowForm.il` template file. Provide the indicated parameters to complete the program.

```

procedure( TrCreateWindowForm( wid )
  let( ( nameField heightField widthField theFormSymbol theForm )
    nameField =
      hiCreateStringField( ;;; you provide the parameters
        )
    heightField =
      hiCreateScaleField( ;;; you provide the parameters
        )
    widthField =
      hiCreateScaleField( ;;; you provide the parameters
        )
    theFormSymbol = 'WindowForm
    theForm =
      hiCreateAppForm( ;;; you provide parameters
        )
    theForm->wid = wid
    theForm;;; my return value
  ) ; let
) ; procedure

```



Provide the Parameters

- Use the *hiCreateAppForm* function to build the data structure of the form.
- Pass *?name 'WindowForm* to the *hiCreateAppForm* function.
- Your form callback needs to determine which window to act on. You cannot use a global variable. Instead, store the window ID as a user-defined slot. Use the `->` operator as follows.

```
theForm->wid = wid
```

Developing the *TrWindowFormCB* Function

Edit the `~/SKILL/UserInterface/WindowForm.il` template file. Provide the code to complete the function definition.

```
procedure( TrWindowFormCB( theForm )
  let( ( wid newName newHeight newWidth )
    ;; pick up current form values
    ;; set the window name
    ;; resize the window
    t    ;; my return value
  ) ; let
) ; procedure
```

- The *TrWindowFormCB* procedure takes the data structure of the form as its only parameter.
- Use the `->` operator with the various field symbols to access the current value of each field. For example, access the current value of *widthField* as follows:

```
theForm->widthField->value
```

- Use the `->` operator to determine the window ID as follows:

```
theForm->wid
```

- Use *hiSetWindowName* to set the name of the window.
- Use *TrResizeWindow* to resize the window.

Testing Your Solution

1. Create a form that refers to the CIW. In the CIW, enter

```
wf = TrCreateWindowForm( window(1) )
```

2. Display the form. In the CIW, enter

```
hiDisplayForm( wf )
```

3. Choose new values for the fields and click **Apply**. Do this several times, observing the obvious changes to the size and name of the CIW.

4. Click **Defaults** to establish the original field values and then click **Apply**. Observe how the CIW reacts.
5. Cancel the form.

Optional Enhancements

Make *TrCreateWindowForm* allow multiple instances of the form to be active at the same time.

Write *TrCreateWindowForm* so that the user can manipulate several windows with simultaneously active forms.

Generating Form Symbols

For your solution to differentiate between multiple form instances, it is sufficient to specify a unique symbol when you create a form. Instead of using the symbol of *WindowForm*, draw from the sequence *WindowForm0*, *WindowForm1*, *WindowForm2* as needed.

1. Redefine your *TrCreateWindowForm* function to use the *gensym* function as follows:

```
theFormSymbol = gensym( 'WindowForm '
```

The *gensym* function generates new symbols on the fly, deriving their common base name from the symbol you provide. To see this, enter the following line in the CIW:

```
formSymbol = gensym( 'WindowForm ')  
formSymbol = gensym( 'WindowForm ')  
formSymbol = gensym( 'WindowForm ')
```

Testing Your Solution

1. Choose any three windows. Assuming that they are *window(1)*, *window(2)* and *window(3)*, enter the following lines into the CIW:

```
wf1 = TrCreateWindowForm( window(1) )  
hiDisplayForm( wf1 )  
wf2 = TrCreateWindowForm( window(2) )  
hiDisplayForm( wf2 )  
wf3 = TrCreateWindowForm( window(3) )  
hiDisplayForm( wf3 )
```

Three forms appear.

2. Change field values in the three forms and apply the changes.

The windows revert to their original sizes and titles.

3. Cancel the forms.

Sample Solution

This is the first portion of the sample solution.

```

procedure( TrCreateWindowForm( wid )
  let( ( nameField heightField widthField theFormSymbol theForm )
    nameField =
      hiCreateStringField(
        ?prompt      "Name"
        ?name        'nameField
        ?value        hiGetWindowName( wid )
        ?defValue     hiGetWindowName( wid )
        ?callback     "println( 'nameField )"
      )
    heightField =
      hiCreateScaleField(
        ?prompt      "Height"
        ?name        'heightField
        ?value        TrgetWindowheight( wid )
        ?defValue     TrgetWindowheight( wid )
        ?range        list( 0 yCoord( hiGetMaxScreenCoords() ) )
        ?callback     "println( 'heightField )"
      )
    widthField =
      hiCreateScaleField(
        ?prompt      "Width"
        ?name        'widthField
        ?value        TrgetWindowwidth( wid )
        ?defValue     TrgetWindowwidth( wid )
        ?range        list( 0 xCoord( hiGetMaxScreenCoords() ) )
        ?callback     "println( 'widthField )"
      )
    theFormSymbol = gensym( 'WindowForm );;; advanced version
                    ;;; otherwise use 'WindowForm
    theForm =
      hiCreateAppForm(
        ?name          theFormSymbol
        ?formTitle     sprintf( nil "%L" wid )
        ?dontBlock     t
        ?callback      "TrWindowFormCB( hiGetCurrentForm() )"
        ?fields        list( nameField heightField widthField )
        ?unmapAfterCB  t
      )
    theForm->wid = wid ;;; assign wid as user-defined prop.
    theForm ;;; my return value
  ); let
) ; procedure

```

The sample solution continues on the next page.

Sample Solution (*continued*)

This is the final portion of the sample solution.

```
procedure( TrWindowFormCB( theForm )
  let( ( wid newName newHeight newWidth )
    ;;; pick up current form values
    wid      = theForm->wid
    ;;; user specified slot linking
    ;;; the form to the window

    newName  = theForm->nameField->value
    newHeight = theForm->heightField->value
    newWidth  = theForm->widthField->value

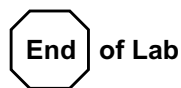
    hiSetWindowName( wid newName ) ;;; set the window name

    TrResizeWindow( ;;; resize the window
      ?id      wid
      ?height  newHeight
      ?width   newWidth
    ) ; TrResizeWindow

    t ;;; my return value
  ) ; let
) ; procedure
```

Lab Summary

In this lab, you built a form that allows the user to change the name or size of a window.



Labs for Module 15

Advanced Customization

Lab 15-1 Adding a Menu Item to the Composer Edit Menu

Objective: To study and run a lecture example.

Studying the SKILL Source Code

1. In the CIW, enter

```
view( "~/SKILL/Customization/AddItemToSchematicPulldown.il" )
```

2. Study the declaration of the *TrUserPostInstallTrigger* function.
Locate the calls to the following functions:

- The *boundp* function

- The *hiCreateMenuItem* function

- The *hiAddMenuItem* function

Does the TrUserPostInstallTrigger function reference the args parameter?

3. Study the call to the *deRegUserTriggers* function.

What view type is passed to the deRegUserTriggers function?

Loading the SKILL Source Code

1. In the CIW, enter

```
load( "~/SKILL/Customization/AddItemToSchematicPulldown.il" )
```

The *TrUserPostInstallTrigger* function customizes a Composer™ window before the menu banner appears.

Checking the Trigger Function

Use the *deGetAppInfo* function to check whether there is a user postinstall trigger function registered for the *schematic* view type.

1. In the CIW, enter

```
deGetAppInfo( "schematic" )->userPostInstallTrigger
```

SKILL displays the non-*nil* return result:

```
TrUserPostInstallTrigger
```

Opening a Design

1. Select **File** → **Open**.
2. Fill out the form with these values.

Library Name	master
Cell Name	mux2
View Name	schematic
Mode	read

3. Click **OK**.

A Composer window displays the *master mux2 schematic* design.

4. Verify that the **Edit** pull-down menu contains a menu item named **Example**.

Descending the Hierarchy

1. Select the instance named *I1* in the design. The instance cell name is *nand2*.
2. Descend the hierarchy into the symbol view of the instance master by selecting **Design** → **Hierarchy** → **Descend Read**.
The **Descend** form appears.
3. Set the **View name** field to **symbol**.

4. Click **OK**.

The window displays the *master nand2 symbol* cellview.

Is this a Composer window?

Do you expect the Edit pull-down menu to contain the Example menu item?

5. Verify that the **Edit** pull-down menu does not contain a menu item named **Example**.

Ascending the Hierarchy

1. Ascend the hierarchy to the original design.
Select **Design** → **Hierarchy** → **Return**.

Is this a Composer window?

Do you expect the Edit pull-down menu to contain the Example menu item?

2. Verify that the **Edit** pull-down menu contains a menu item named **Example**.

Unregistering the Trigger Function

In the next lab exercise, you register another trigger for the *schematic* view type. To prepare for the next exercise, unregister the current *schematic* trigger function.

1. In the CIW, enter

```
deUnRegUserTriggers( "schematic" )
```

SKILL displays the return result:

```
t
```

2. In the CIW, enter

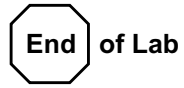
```
deGetAppInfo( "schematic" )->userPostInstallTrigger
```

SKILL displays the return result:

```
nil
```

Lab Summary

In this lab, you studied and ran a lecture example.



Lab 15-2 Adding a Pull-Down Menu to a Composer Window

Objective: To study and run a lecture example.

Studying the SKILL Source Code

1. In the CIW, enter

```
view( "~/SKILL/Customization/AddPulldownToSchematicBanner.il" )
```

2. Study the declaration of the *TrUserPostInstallTrigger* function.
Locate the calls to the following functions:

- The *hiInsertBannerMenu* function

- The *hiGetBannerMenus* function

- The *TrCreateSchematicPulldownMenu* function

Does the TrUserPostInstallTrigger function reference the args parameter?

3. Study the call to the *deRegUserTriggers* function.

What view type is passed to the deRegUserTriggers function?

Loading the SKILL Source Code

1. In the CIW, enter

```
load( "~/SKILL/Customization/AddPulldownToSchematicBanner.il" )
```

The *TrUserPostInstallTrigger* function customizes a Composer window before the menu banner appears.

Checking the User Postinstall Trigger

Use the *deGetAppInfo* function to check whether there is a user postinstall function registered for the *schematic* view type.

1. In the CIW, enter

```
deGetAppInfo( "schematic" )->userPostInstallTrigger
```

SKILL displays the non-*nil* return result:

```
TrUserPostInstallTrigger
```

Opening a Design

1. Select **File** → **Open**.
2. Fill out the form with these values.

Library Name	master
Cell Name	mux2
View Name	schematic
Mode	read

3. Click **OK**.

A Composer window displays the *master mux2 schematic* design.

4. Verify that the Composer window contains the pull-down **Example Menu** in the rightmost position. You might need to enlarge the window to make the pull-down menu visible.

Descending the Hierarchy

1. Select the instance named *I1* in the design. The instance cell name is *nand2*.
2. Descend the hierarchy into the symbol view of the instance master by selecting **Design** → **Hierarchy** → **Descend**.

The **Descend** form appears.

3. Set the **View name** field to **symbol**.

4. Click **OK**.

The window displays the *master nand2 symbol* cellview.

Is this a Composer window?

Do you expect the graphics window to contain the pull-down Example Menu?

5. Verify that the Composer window contains the pull-down **Example Menu**.

Ascending the Hierarchy

1. Return to the original design by ascending the hierarchy.
Select **Design** → **Hierarchy** → **Return**.

Is this a Composer window?

Do you expect the graphics window to contain the pull-down Example Menu?

2. Verify that the graphics window contains the pull-down **Example Menu**.

Unregistering the User Postinstall Trigger

Unregister the current user postinstall trigger.

1. In the CIW, enter

```
deUnRegUserTriggers( "schematic" )
```

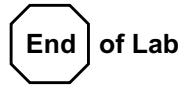
SKILL displays the return result:

```
t
```

How can you verify that there is not user postinstall trigger function registered?

Lab Summary

In this lab, you studied and ran a lecture example.



Lab 15-3 Reversing the Layout Editor Pull-Down Menus

Objective: To study and run a lecture example.

Studying the SKILL Source Code

1. In the CIW, enter

```
view( "~/SKILL/Customization/ReverseLayoutEditorMenus.il" )
```

2. Study the declaration of the *TrUserPostInstallTrigger* function.
Locate the calls to the following functions:

- The *hiGetBannerMenus* function.

- The *hiDeleteBannerMenus* function

- The *hiInsertBannerMenu* function

Does the TrUserPostInstallTrigger function reference the args parameter?

3. Study the call to the *deRegUserTriggers* function.

What view type is passed to the deRegUserTriggers function?

Loading the SKILL Source Code

1. In the CIW, enter

```
load( "~/SKILL/Customization/ReverseLayoutEditorMenus.il" )
```

The *TrUserPostInstallTrigger* function customizes a Layout Editor window before the menu banner appears.

Opening a Design

1. Select **File—Open**.

2. Fill out the form with these values.

Library Name	master
Cell Name	mux2
View Name	layout
Mode	read

3. Click **OK**.

A Virtuoso window displays the *master mux2 layout* design.

4. Verify that the pull-down menus in the Layout Editor window are in reverse order.

Descending the Hierarchy

1. Select the left-most instance in the design. The master cell name is *nand2*.
2. Descend the hierarchy into the symbol view of the instance master by selecting **Design** → **Hierarchy** → **Descend Read**.

The **Descend** form appears.

The window displays the *master nand2 layout* cellview.

Do you expect the graphics window to contain pull-down menus in reverse order?

3. Verify that the pull-down menus in the *master nand2 layout* design window are in reverse order.

Ascending the Hierarchy

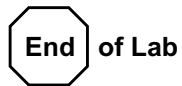
1. Ascend the hierarchy back to the original design.
Select **Design**—**Hierarchy**—**Return**.

Do you expect the graphics window to contain pull-down menus in reverse order?

2. Verify that the pull-down menus in the Layout Editor window are in reverse order.

Lab Summary

In this lab, you studied and ran a lecture example.



Lab 15-4 Customizing the Initial Window Placement

Objective: To develop a user postinstall trigger function.

Requirements

Customize the Layout Editor software to display the design windows in a given bounding box specified in a global variable.

The user's `.cdsinit` file specifies the bounding box in the global variable `Trmasklayoutiwp`. For example,

```
Trmasklayoutiwp = '((0 0) (500 500))'
```



The user must invoke the **Fit All** command after the Layout Editor window appears.

Planning Your Solution

Plan your solution to reflect your answers to these questions:

Which hi function can your trigger function call to resize the Layout Editor window?*

What arguments does this hi function expect?*

How can your trigger function determine the window ID of the Layout Editor window?

Which view type must you pass to the `deRegUserTriggers` function?

— Check your work against the solution on the next page.

Answers

Your trigger function can call the *hiResizeWindow* function to resize the Layout Editor window.

The *hiResizeWindow* function expects two arguments. The first argument is the window ID. The second argument is the bounding box.

Your trigger function can use the *args->window* syntax to determine the Layout Editor window ID.

You must pass *"maskLayout"* as the first argument to the *deRegUserTriggers* function.

Developing Your Trigger Function

1. Develop your trigger function and register it.
 2. Verify that it works.
- Check your work against the solution on the next page.

Answers

The *.cdsinit* file must contain the following:

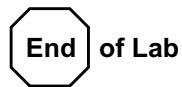
```
Trmasklayoutiwp = '((0 0) (500 500))

procedure( TrUserPostInstallTrigger( args
)
  hiResizeWindow(
    args->window Trmasklayoutiwp )
)

deRegUserTriggers( "maskLayout"
  nil
  nil
  'TrUserPostInstallTrigger
)
```

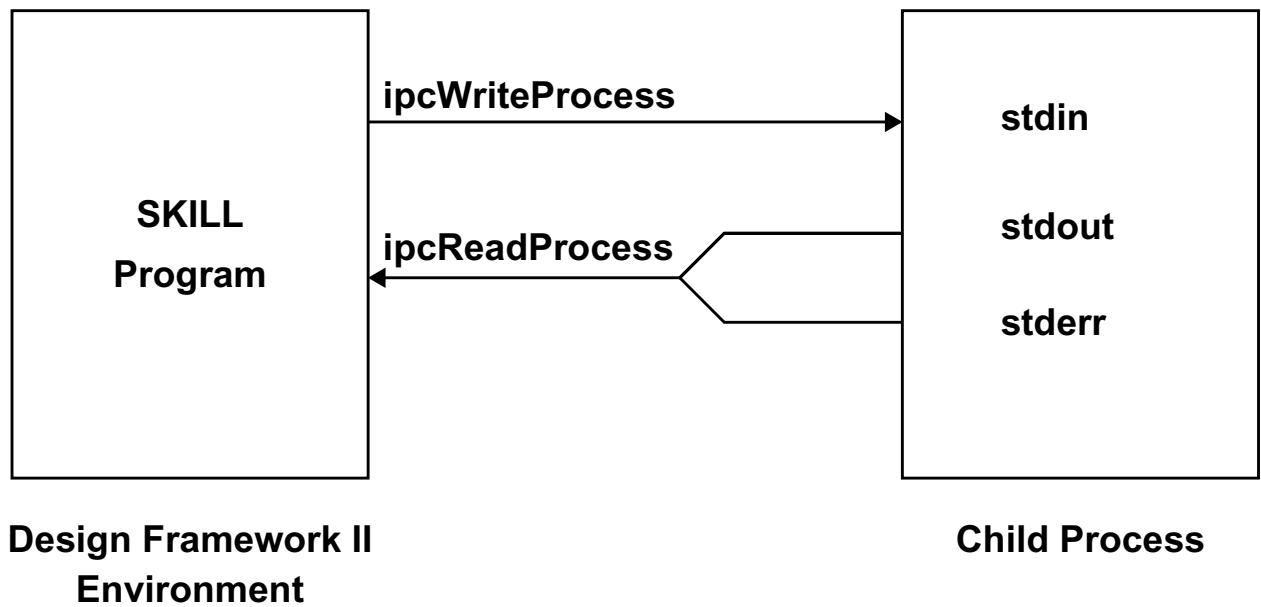
Lab Summary

You developed a user postinstall trigger function that resizes a Layout Editor window.



Labs for Module 16

Interprocess Communication



Lab 16-1 Compiling a C Program

Objective: To compile and run the C program from the lecture.

Complete this lab before doing subsequent lab exercises. In subsequent lab exercises, you launch this program as a separate UNIX child process.

Studying the *IPC/toUpperCase.c* Program

1. In the CIW, enter

```
view( "IPC/toUpperCase.c" )
```

A text window displays the C source code.

2. Locate the following function calls in the source code:

■ *gets(p)*

■ *toupper(*p)*

■ *printf("%s \n", s)*

Compiling the *IPC/toUpperCase.c* Program

cc is the C compiler. It translates programs written in the C programming language into executable load modules.

1. In an xterm window, enter

```
cc IPC/toUpperCase.c -o IPC/toUpperCase.out
```

The *IPC/toUpperCase.out* file is the executable file you pass to the *ipcBeginProcess* function in subsequent labs.

Running the *IPC/toUpperCase.out* Program

Verify that *toUpperCase* works.

1. Enter the following lines in an xterm window:

```
IPC/toUpperCase.out  
the first line
```

End each line with a **Return**.

The program displays the following line:

```
THE FIRST LINE
```

2. Enter more lines of text to capitalize. End each line with a **Return**.

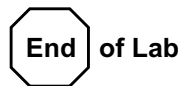
The program displays each line capitalized.

3. Terminate the program by pressing

Control - **c**

Lab Summary

In this lab, you compiled and ran the *C* program that you use in subsequent lab exercises.



Lab 16-2 Exploring Synchronous Communication

Objective: To run the synchronous communication example.

In this lab, you

- Run the synchronous communication example from the lecture.
- Use the SKILL trace facility to observe the flow of control.

Use the C program you compiled in the previous lab.



If you have not completed Lab 16-1, please do so before proceeding with this lab.

Studying the SKILL Sample Source Code

1. In the CIW, enter

```
view( "IPC/TrUpperCaseList.il" )
```

A text window displays the SKILL source code.

2. Locate and study function calls for the following functions:

- *ipcBeginProcess*
- *ipcWaitForProcess*
- *ipcWriteProcess*
- *ipcReadProcess*
- *ipcKillProcess*

Loading the SKILL Sample Source Code

1. In the CIW, enter

```
load( "IPC/TrUpperCaseList.il" )
```

SKILL declares the *TrUpperCaseList* function.

Running the Example

1. In the CIW, enter

```
TrUpperCaseList( '( "a" "bc" "def" ) )
```

SKILL displays the return result in the CIW output pane.

```
( "A \n" "BC \n" "DEF \n" )
```

Tracing the Example

Use the SKILL trace facility to observe the flow of control.

1. Trace the *ipcWriteProcess* and *ipcReadProcess* functions. In the CIW, enter

```
tracef( ipcReadProcess ipcWriteProcess )
```

SKILL displays the list.

```
(ipcWriteProcess ipcReadProcess)
```

2. In the CIW, enter

```
TrUpperCaseList( '( "a" "bc" "def" ) )
```

SKILL displays output similar to the following:

```
TrUpperCaseList( '( "a" "bc" "def" ) )
|||ipcWriteProcess(ipc:1 "a\n")
|||ipcWriteProcess --> t
|||ipcReadProcess(ipc:1 10)
|||ipcReadProcess --> "A \n"
|||ipcWriteProcess(ipc:1 "bc\n")
|||ipcWriteProcess --> t
|||ipcReadProcess(ipc:1 10)
|||ipcReadProcess --> "BC \n"
|||ipcWriteProcess(ipc:1 "def\n")
|||ipcWriteProcess --> t
|||ipcReadProcess(ipc:1 10)
|||ipcReadProcess --> "DEF \n"
("A \n" "BC \n" "DEF \n")
```

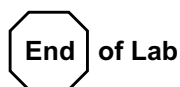
3. Turn off tracing. In the CIW, enter

```
untrace( )
```

Lab Summary

In this lab, you

- Studied the SKILL source code for the synchronous communication example.
- Loaded the SKILL source code.
- Ran the example.
- Used the SKILL trace facility to observe the flow of control.



Lab 16-3 Exploring Asynchronous Communication

Objective: To run the asynchronous communication example.

In this lab, you

- Run the asynchronous communication example from the lecture.
- Use the SKILL tracing facility to observe the flow of control.



If you have not completed Lab 16-1, please do so before proceeding with this lab.

Studying the SKILL Sample Source Code

1. In the CIW, enter

```
view( "IPC/TrUpperCase_Launch.il" )
```

2. Locate and study the definitions for the following functions:

- *TrUpperCase_Launch*
- *TrUpperCase_Write*
- *TrUpperCase_DataHandler*
- *TrUpperCase_PostFunction*

Loading the SKILL Sample Source Code

1. In the CIW, enter

```
load( "IPC/TrUpperCase_Launch.il" )
```


Running the Example

1. In the CIW, enter

```
TrUpperCase_Launch( '( "a" "bc" "def" ) )
```

SKILL displays output similar to the following:

```
TrUpperCase_Launch( '( "a" "bc" "def" ) )  
Launched child ipc:1  
t  
Child ipc:1 result: ( "A \n" "BC \n" "DEF \n" )
```

The child ID might be different.

Tracing the Example

In the steps that follow, use the SKILL trace facility to observe the flow of control involving these functions:

■ *TrUpperCase_Write*

■ *TrUpperCase_DataHandler*

■ *TrUpperCase_PostFunction*

1. In the CIW, enter

```
tracef(  
    TrUpperCase_Write TrUpperCase_DataHandler TrUpperCase_PostFunction )
```

2. In the CIW, enter

```
TrUpperCase_Launch( '( "a" "bc" "def" ) )
```

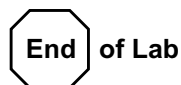
SKILL displays output similar to the following:

```
TrUpperCase_Launch( '( "a" "bc" "def" ) )
||TrUpperCase_Write(ipc:1)
||TrUpperCase_Write --> ("bc" "def")
Launched child ipc:1
t
>
|TrUpperCase_DataHandler(ipc:1 "A \n")
||TrUpperCase_Write(ipc:1)
||TrUpperCase_Write --> ("def")
|TrUpperCase_DataHandler --> ("def")
|TrUpperCase_DataHandler(ipc:1 "BC \n")
||TrUpperCase_Write(ipc:1)
||TrUpperCase_Write --> nil
|TrUpperCase_DataHandler --> nil
|TrUpperCase_DataHandler(ipc:1 "DEF \n")
|||TrUpperCase_PostFunction(ipc:1 0 )
Child 38 result: ("A \n" "BC \n" "DEF \n")
|||TrUpperCase_PostFunction --> t
|TrUpperCase_DataHandler --> t
```

Notice the prompt, >, separates the output of the *TrUpperCase_Launch* function from the trace output of the data handler. You can conclude that the *TrUpperCase_Launch* function returned before the *TrUpperCase_DataHandler* function ran.

3. Turn off tracing. In the CIW, enter

```
untrace( )
```



Labs for Module 17

Data Structures

Introduction

In Lab 9-1, you wrote the *TrShapeReport* function to count shapes in a design. The function uses variables to count shapes. The function uses a miscellaneous count.

In these labs, you enhance the *TrShapeReport* function to count every kind of shape in a design. You do not use a miscellaneous count. Instead of variables, you use a dynamic data structure.

The three data structures are

- Association Table

Using an association table is the syntactically the easiest of the three approaches.

- Association List

Using an association is just about as straight forward as using an association table, provided you choose the right format for each entry.

- Disembodied Property List

Using a disembodied property list for this application is tricky.

Lab 17-1 Exploring Associative Tables

Objective: Use an association table to maintain a list of counts.

In this lab, you revise your original *TrShapeReport* function to count every kind of shape in a design.

You can use the sample solution for the *TrShapeReport* function. The sample solution is in the *~/SKILL/FlowOfControl/Solutions/TrShapeReport.il* file.

Requirements

Make the *TrShapeReportAssocTable* function do the following:

- Store its counters in an association table instead of variables.
- Use the *objType* of a shape to label the associated count in the report displayed in the CIW.

Otherwise, make *TrShapeReportAssocTable* behave like the original *TrShapeReport* as in this example:

```
master mux2 layout contains:
rect          8
path          7
textDisplay 6
```

Suggestions

Use the *makeTable* function to create the association table.

```
shapeTable = makeTable( "Shape Table" 0 )
```

Use an expression like the following to update a count.

```
shapeTable[ shape~>objType ] =
  shapeTable[ shape~>objType ] + 1
```

After you traverse the design counting shapes, use the *foreach* function to traverse the association table, displaying the counts in the CIW.

```
foreach( key shapeTable
  ...
) ; foreach
```

Continue with the lab after you have finished writing your *TrShapeReportAssocTable* function.

Testing Your Solution

First Test

1. If you have not already done so, in the CIW, enter

```
nand2wid = geOpen(
  ?lib "master"
  ?cell "nand2"
  ?view "layout"
  ?mode "r" )
```

2. In the CIW, enter

```
TrShapeReportAssocTable( nand2wid )
```

The CIW displays:

```
master nand2 layout contains:
rect      12
path      3
```

Second Test

3. If you have not already done so, in the CIW, enter

```
mux2wid = geOpen(
  ?lib "master"
  ?cell "mux2"
  ?view "layout"
  ?mode "r" )
```

4. In the CIW, enter

```
TrShapeReportAssocTable( mux2wid )
```

The CIW displays:

```
master mux2 layout contains:
rect          8
path          7
textDisplay 6
```

— Check your work against the solution on the next page.

Sample Solution

```

procedure( TrShapeReportAssocTable( wid )
  let( ( cv shapeTable shapeList )
    cv = geGetWindowCellView( wid )
    shapeList = cv~>shapes
    when( shapeList ;;; cell view has shapes
      shapeTable = makeTable( "Shape Table" 0 )

      foreach( shape cv~>shapes
        shapeTable[ shape~>objType ] = shapeTable[ shape~>objType ] + 1
      ) ; foreach

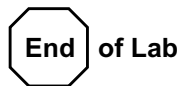
      printf(
        "%s %s %s contains:"
        cv~>libName cv~>cellName cv~>viewName )
      foreach( key shapeTable
        printf( "\n%-10s %-10d"
          key ;;; objType
          shapeTable[ key ] ;;; count
        )
      ) ; foreach
    t
  ) ; when
) ; let
) ; procedure

```

The `~/SKILL/DataStructures/Solutions/TrShapeReport.il` file contains the solution.

Lab Summary

In this lab, you wrote the *TrShapeReportAssocTable* function to count all the shapes in a design.



Lab 17-2 Exploring Association Lists

Objective: Use an association list to maintain a list of counts.

In this lab, you revise your original *TrShapeReport* function to count every kind of shape in a design.

You can use the sample solution for the *TrShapeReport* function. The sample solution is in the *~/SKILL/FlowOfControl/Solutions/TrShapeReport.il* file.

Requirements

Make the *TrShapeReportAssocList* function do the following:

- Store its counters in an association list instead of in variables.
- Use the *objType* of a shape to label the associated count in the report displayed in the CIW.

Otherwise, make *TrShapeReportAssocList* behave like *TrShapeReport* as in this example:

```
master mux2 layout contains:
polygon    3
label      6
path       7
rect       9
```

Suggestions

Use a *foreach* loop to traverse the design. Each time around the loop, you either increment an existing count in the association list or add an entry to the association list.

For each shape in the design, follow these steps:

1. Access the object type of the shape.

2. Use the *assoc* function to access the entry for a shape.

```
shapeEntry = assoc(  
  shape~>objType shapeAssocList )
```

If the *shapeEntry* is non-*nil*, then increment the count.

Otherwise, add an entry to the *shapeAssocList* function.

The crucial decision is how to structure the entries in the association list.

Make the entries in the association list initially resemble the following:

```
( "rect" count 0 )
```

Why not make each entry resemble the following?

```
( "rect" 0 )
```

— Check your work against the solution on the next page.

Answer

You can use the `->` operator to access and update the count, since the example is a disembodied property list!

1. For example, in the CIW, enter

```
example = '( "rect" count 0 )
example->count = example->count+1
```

2. Compare the above syntax with the syntax of the *rplaca* function to change the 0 to 1 in the following:

```
example2 = '( "rect" 0 )
rplaca( cdr( example2 ) cadr( example2)+1 )
```

Continue with the lab when you have finished writing the *TrShapeReportAssocList* function.

Testing Your Solution**First Test**

1. If you have not already done so, in the CIW, enter

```
nand2wid = geOpen(
  ?lib "master"
  ?cell "nand2"
  ?view "layout"
  ?mode "r" )
```

2. In the CIW, enter

```
TrShapeReportAssocList( nand2wid )
```

The CIW displays:

```
master nand2 layout contains:
paths      3
rect       12
```

Second Test

3. If you have not already done so, in the CIW, enter

```
mux2wid = geOpen( ?lib "master" ?cell "mux2" ?view "layout" ?mode "r" )
```

4. In the CIW, enter

```
TrShapeReportAssocList( mux2wid )
```

The CIW displays:

```
master mux2 layout contains:
polygon 3
label    6
path     7
rect     9
```

— Check your work against the solution on the next page.

Sample Solution

```

procedure( TrShapeReportAssocList( wid )
  let( ( cv shapeAssocList shapeList shapeEntry )
    cv = geGetWindowCellView( wid )
    shapeList = cv~>shapes
    when( shapeList ;;; cell view has shapes
      shapeAssocList = nil

    foreach( shape cv~>shapes
      shapeEntry = assoc( shape~>objType shapeAssocList )
      if( shapeEntry
        then ;;; there is an entry already
          shapeEntry->count = shapeEntry->count+1
        else;;; no entry yet so add one at front of shapeAssocList
          ;;; entry looks like
          ;;; ( <obj type> count <number> )
          ;;; entry is a disembodied property list
          shapeAssocList = cons(
            list( shape~>objType 'count 1 )
            shapeAssocList
          )
        ) ; if
      ) ; foreach

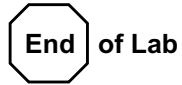
    printf(
      "%s %s %s contains:"
      cv~>libName cv~>cellName cv~>viewName )
    foreach( entry shapeAssocList
      printf( "\n%-10s %-10d"
        car( entry );;; objType
        entry->count;;; count
      )
    ) ; foreach
  t
  ) ; when
) ; let
) ; procedure

```

The `~/SKILL/DataStructures/Solutions/TrShapeReport.il` file contains the solution.

Lab Summary

In this lab, you wrote the *TrShapeReportAssocList* function to count all the shapes in a design.



Lab 17-3 Exploring Disembodied Property Lists

Objective: Use a disembodied property list to maintain a list of counts

In this lab, you revise your *TrShapeReport* function to count every kind of shape in a design.

You can use the sample solution for the *TrShapeReport* function. The sample solution is in the `~/SKILL/FlowOfControl/Solutions/TrShapeReport.il` file.

Requirements

Make the *TrShapeReportDPL* function do the following:

- Store its counters in a disembodied property list instead of variables.
- Use the *objType* to label the associated count.

Otherwise, make *TrShapeReportDPL* behave like *TrShapeReport* as in this example:

```
master mux2 layout contains:
polygon 3
label 6
path 7
rect 9
```

Suggestions

Use the following logic to build the disembodied property list.

For each shape in the design, follow these steps:

1. Access the object type of the shape.
2. Increment the corresponding property value in the disembodied property list.

To implement the above logic, you need to use:

3. The following expression to initialize the disembodied property list.

```
list( nil )
```

4. The *concat* function to create property name from a shape object type.
5. The *get* function to retrieve a property value from the disembodied property list.
6. The *putprop* function to store a property value in a disembodied property list.

Print the counters using the *while* function to traverse the disembodied property list.

Why can't you use a foreach loop?

Continue with the lab on the next page.

Testing Your Solution

First Test

1. In the CIW, enter

```
nand2wid = geOpen(  
  ?lib "master"  
  ?cell "nand2"  
  ?view "layout"  
  ?mode "r" )
```

2. In the CIW, enter

```
TrShapeReportDPL( nand2wid )
```

The CIW displays:

```
master nand2 layout contains:  
paths      3  
rect       12
```

Second Test

3. In the CIW, enter

```
mux2wid = geOpen(  
  ?lib "master"  
  ?cell "mux2"  
  ?view "layout"  
  ?mode "r" )
```

4. In the CIW, enter

```
TrShapeReportDPL( mux2wid )
```

The CIW displays:

```
master mux2 layout contains:  
polygon 3  
label   6  
path    7  
rect    9
```

— Check your work against the solution on the next page.

Sample Solution

```

procedure( TrShapeReportDPL( wid )
  let( ( shapeTypePropList shapeType shapeTypeSymbol shapeTypeCount cv )
    shapeTypePropList = list( nil )
    cv = geGetWindowCellView( wid )
    printf(
      "%s %s %s contains:"
      cv~>libName cv~>cellName cv~>viewName )
    foreach( shape cv~>shapes
      shapeType = shape~>objType
      shapeTypeSymbol = concat( shapeType )
      shapeTypeCount = get( shapeTypePropList shapeTypeSymbol )
      if( shapeTypeCount
        then
          putprop( shapeTypePropList shapeTypeCount+1 shapeTypeSymbol )
        else
          putprop( shapeTypePropList 1 shapeTypeSymbol )
      ) ; if
    ) ; foreach

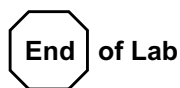
    shapeTypePropList = cdr( shapeTypePropList ) ;;; skip the nil
    while( shapeTypePropList
      printf( "\n%-10s %-10d"
        car( shapeTypePropList ) ;;; the object type
        cadr( shapeTypePropList ) ;;; the count
      )
      shapeTypePropList = cddr( shapeTypePropList )
    ) ; while
  ) ; foreach
) ; procedure

```

The `~/SKILL/DataStructures/Solutions/TrShapeReport.il` file contains the solution.

Lab Summary

In this lab, you wrote the *TrShapeReportDPL* function to count all the shapes in a design.



Labs for Module 18

OpenAccess

No Labs for This Module

Lab 18-2 No Labs for this Module

