

Cadence® SKILL Language Programming

Version 5.0

Lecture 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**

Introduction to SKILL Overview.....	1-3
SKILL Programming Overview	1-5
Lab Exercises	1-7
Using the Cadence Online Documentation.....	1-9
Searching Cadence Documents.....	1-11
Search Operators.....	1-13
The SKILL Documentation Set	1-17
The Finder	1-19
Lab Overview.....	1-21
Module Summary.....	1-23

Module 2 **Beginning to Use SKILL**

What Is SKILL?	2-3
What Can SKILL Functions Do?.....	2-5
Starting the Design Framework II Environment.....	2-7
Initializing the Design Framework II Environment.....	2-9
Command Interpreter Window	2-11
Design Framework II User Interface	2-13
The CDS.log File	2-15
The CDS.log File Code.....	2-17
Setting the Log Filter	2-19
SKILL Syntax Summary.....	2-21
Data	2-23
Variables	2-25
Function Calls	2-27
Multiple Lines.....	2-29
Understanding Function Arguments.....	2-31
Operators.....	2-33
Tracing Operator Evaluation.....	2-35
Lab Overview.....	2-37
Displaying Data in the CIW.....	2-39
Displaying Data with Format Control.....	2-41
Solving Common Problems	2-43
What If the CIW Doesn't Respond?	2-45

White Space Sometimes Causes Errors	2-47
Passing Incorrect Arguments to a Function	2-49
Lab Overview.....	2-51
Module Summary.....	2-53

Module 3 SKILL Lists

What Is a SKILL List?	3-3
How SKILL Displays a List	3-5
Creating New Lists	3-7
Adding Elements to an Existing List	3-9
Points of Confusion	3-11
Working with Existing Lists	3-13
Traversing a list with <i>car</i> and <i>cdr</i>	3-15
Frequently Asked Questions	3-17
Two-Dimensional Points	3-19
Computing Points	3-21
Bounding Boxes.....	3-23
Creating a Bounding Box.....	3-25
Retrieving Elements from Bounding Boxes	3-27
Lecture Exercises	3-29
Offsetting a Box.....	3-31
Finding the Smallest Bounding Box.....	3-33
Finding the Intersection of Two Bounding Boxes.....	3-35
Combinations <i>car</i> and <i>cdr</i> Functions.....	3-37
Lab Overview.....	3-39
Module Summary.....	3-41

Module 4 Windows

Design Framework II Windows.....	4-3
Opening a Design Window.....	4-5
Using Bindkeys.....	4-7
Defining Bindkeys	4-9
Describing Events	4-11
Displaying Bindkeys.....	4-13
Standard Bindkey Definitions.....	4-15
Opening a Text Window.....	4-17
Manipulating Windows.....	4-19
Resizing Windows	4-21
Iconifying Windows.....	4-23

Lab Overview.....	4-25
Module Summary.....	4-27

Module 5 Database Queries

Database Objects.....	5-3
Querying a Design	5-5
The ~> Operator	5-7
Querying Designs with the ~> Operator.....	5-9
The <i>cellView</i> Object Type.....	5-11
Instances.....	5-13
The <i>inst</i> Object Type.....	5-15
Nets	5-17
The <i>net</i> Object Type.....	5-19
Instance Terminals.....	5-21
The <i>instTerm</i> Object Type	5-23
Terminals	5-25
The <i>term</i> Object Type	5-27
Figures and Shapes	5-29
Shape Attributes.....	5-31
Lab Overview.....	5-33
Module Summary.....	5-35

Module 6 Menus

Creating a Pop-Up Menu	6-3
Displaying a Pop-Up Menu	6-5
Creating a Pull-Down Menu	6-7
Inserting a Pull-Down Menu.....	6-9
Deleting a Pull-Down Menu	6-11
Lab Overview.....	6-13
Module Summary.....	6-15

Module 7 Customization

Design Framework II Initialization Sequence	7-3
The <i>.cdsinit</i> File	7-5
The Basic <i>.cdsinit</i> Tasks	7-7
Loading SKILL Source Code	7-9
Other Useful <i>.cdsinit</i> Functions	7-11
Determining the Installation Path	7-13

Interfacing with UNIX Files and Directories.....	7-15
Reading Shell Environment Variables.....	7-17
Manipulating a List of Strings	7-19
Manipulating the CIW in the <i>.cdsinit</i> File	7-21
SKILL Development <i>.cdsinit</i> Tasks.....	7-23
Lab Preview	7-25
Lab Overview.....	7-27
Module Summary.....	7-29

Module 8 Developing a SKILL Function

Grouping SKILL Expressions Together	8-3
Grouping Expressions with Local Variables	8-5
Two Common <i>let</i> Errors	8-7
Defining SKILL Functions	8-9
Three Common <i>procedure</i> Errors	8-11
Defining Required Function Parameters.....	8-13
Defining Optional Function Parameters	8-15
Defining Keyword Function Parameters	8-17
Collecting Function Parameters into a List.....	8-19
SKILL Development Cycle	8-21
Loading Source Code.....	8-23
Pasting Source Code into the CIW	8-25
Redefining a SKILL Function	8-27
Lab Overview.....	8-29
Module Summary.....	8-31

Module 9 Flow of Control

Relational Operators	9-3
Logical Operators.....	9-5
Using the <i>&&</i> and <i>//</i> Operators to Control Flow.....	9-7
Branching.....	9-9
The <i>if</i> Function.....	9-11
Two Common <i>if</i> Errors	9-13
Nested <i>if-then-else</i> Expressions	9-15
The <i>when</i> and <i>unless</i> Functions	9-17
The <i>case</i> Function	9-19
The <i>cond</i> Function	9-21
Iteration	9-23
The <i>for</i> Function	9-25

The <i>foreach</i> Function	9-27
The <i>while</i> Function	9-29
The <i>prog</i> and <i>return</i> Functions	9-31
Lab Overview.....	9-33
Module Summary.....	9-35

Module 10 File I/O

Writing Data to a File	10-3
Reading Data from a File.....	10-7
The <i>fscanf</i> Function.....	10-9
Opening a Text Window	10-11
Lab Overview.....	10-13
Module Summary.....	10-15

Module 11 Using the SKILL Development Environment

SKILL Development Tips.....	11-3
Common Bugs	11-5
Using Print Statements.....	11-7
Checking for Common Errors.....	11-9
Check the Running Source Code	11-11
Check All Attribute and Property Names	11-13
Verify ~> Data Types	11-15
Programming Defensively	11-17
Include Data Type Checks	11-19
An Example Type Template	11-21
The <i>error</i> Function.....	11-23
Development Tip Summary	11-25
SKILL Development Environment.....	11-27
SKILL Development Window	11-29
SKILL Debugger Toolbox	11-31
Using Single-Step Tracing to Analyze an Error	11-33
A Single-Step Trace Example.....	11-35
Another Single-Step Trace Example	11-37
Using the SKILL Debugger to Trap an Error	11-39
Installing the SKILL Debugger	11-41
Displaying the SKILL Stack.....	11-43
Analyzing the SKILL Stack.....	11-45
Debugging Example.....	11-47
Suspending and Resuming Your Program.....	11-51

Setting Breakpoints	11-53
Clearing Breakpoints	11-55
Hitting a Breakpoint.....	11-57
Resuming Your Program	11-59
The <i>continue</i> Function	11-61
Single-Stepping Your Program.....	11-63
SKILL Debugger Sessions.....	11-65
Quitting a Debugger Session	11-67
Nesting SKILL Debugger Sessions	11-69
SKILL Debugger Summary	11-71
Frequently Asked Questions	11-73
Other SKILL Debugger Benefits.....	11-75
Lab Overview.....	11-77
SKILL Lint.....	11-79
SKILL Profiler	11-81
Lab Overview.....	11-83
Module Summary.....	11-85

Module 12 List Construction

List Review	12-3
The <i>foreach mapcar</i> Function.....	12-5
Extended <i>foreach mapcar</i> Example.....	12-7
The <i>mapcar</i> Function.....	12-9
Filtering Lists	12-11
The <i>setof</i> Function.....	12-13
A <i>setof</i> Example.....	12-15
Another <i>setof</i> Example.....	12-17
Transforming Elements of a Filtered List.....	12-19
Lab Overview.....	12-21
Module Summary.....	12-23

Module 13 Cellview Data Model

Database Object Review	13-3
Design Framework II Database Technology	13-5
Database Access SKILL Documentation.....	13-7
Description of Database Objects.....	13-9
Database Object Classes	13-11
The Cellview Data Model Road Map	13-13
Opening Cellviews.....	13-15

Shapes	13-17
Creating Shapes	13-19
Updating Shapes	13-21
Saving and Closing a Cellview	13-23
The <i>dbClose</i> Function	13-25
Lab Overview.....	13-27
Hierarchy.....	13-29
The <i>cellView</i> and <i>inst</i> Object Types.....	13-31
The <i>TrFormatInstance</i> Function.....	13-33
A Hierarchical Design.....	13-35
Traversing a Hierarchical Layout Design	13-37
Using <i>dbOpenCellViewByType</i> to Switch Views	13-39
A List of Schematic Views	13-41
Traversing a Hierarchical Schematic Design.....	13-43
Lab Overview.....	13-45
Connectivity	13-47
The <i>term</i> , <i>net</i> , and <i>instTerm</i> Object Types.....	13-49
Connectivity Between Instances and Masters.....	13-51
Figures and Terminals.....	13-53
Lab Overview.....	13-55
Generic Database Object.....	13-57
User-Defined Properties.....	13-59
Examining User-Defined Properties	13-61
Creating User-Defined Properties.....	13-63
Hierarchical User-Defined Properties	13-65
Other Kinds of User-Defined Properties.....	13-67
Retrieving All Attributes of a Database Object	13-69
The Show File Browser	13-71
Lab Overview.....	13-73
Module Summary.....	13-75

Module 14 User Interface

Context Sensitive Pop-up Menus.....	14-3
Fixed Menus.....	14-5
Creating Fixed Menus.....	14-7
Displaying Fixed Menus	14-9
Attaching Fixed Menus to Application Windows	14-11
Lab Overview.....	14-13
Dialog Boxes.....	14-15
The <i>hiDisplayAppDBox</i> Function	14-17

The Dialog Box Blocking Behavior	14-19
The Dialog Box Callback and Button Layout.....	14-21
Lab Overview.....	14-23
List Boxes	14-25
The <i>hiShowListBox</i> Function	14-27
The List Box Callback	14-29
Lab Overview.....	14-31
Standard Forms	14-33
Analyzing a Design Framework II Command.....	14-35
Creating and Displaying a Form	14-37
Example File Form	14-39
Creating the File Name Field.....	14-41
Validating the File Name Field.....	14-43
Creating the File Form.....	14-45
The File Form OK/Apply Callback	14-47
Displaying the File Form	14-49
The <i>?unmapAfterCB</i> Argument	14-51
Controlling the Cursor Icon	14-53
Frequently Asked Questions	14-55
Form Fields	14-57
Lab Overview.....	14-59
Module Summary.....	14-61

Module 15 Advanced Customization

Module Scope	15-3
Design Editor Applications.....	15-5
Design Editor User Trigger Functions.....	15-7
The Design Editor User Postinstall Trigger.....	15-9
Registering a User Postinstall Trigger Function.....	15-11
Developing a User Postinstall Trigger Function.....	15-13
Debugging a User Postinstall Trigger Function.....	15-15
Redefining a User Postinstall Trigger Function.....	15-17
What You Put in the <i>.cdsinit</i> File.....	15-19
Three User Postinstall Trigger Examples	15-21
Adding a Menu Item to the Composer Edit Menu.....	15-23
Finding the Edit Menu Data Structure	15-25
Avoiding Errors and Warnings	15-27
What You Put into the <i>.cdsinit</i> File.....	15-29
Adding a Pull-Down Menu to Composer Windows	15-31
What You Put into the <i>.cdsinit</i> File.....	15-33

Reordering Layout Editor Pull-Down Menus.....	15-35
What You Put into the <i>.cdsinit</i> File.....	15-37
Customizing Forms.....	15-39
Customizing the Default Value of a Form Field.....	15-41
Design Framework II Initialization Sequence	15-43
The <i>defUserInitProc</i> Function	15-45
Manage the Layer Selection Window (LSW).....	15-47
Lab Overview.....	15-49
Module Summary.....	15-51

Module 16 Interprocess Communication

The Scope of SKILL IPC.....	16-3
The <i>toUpperCase.c</i> Source Code.....	16-5
Comparing the Two Communication Styles.....	16-7
Spawning a Child Process.....	16-9
Synchronizing with the Child Process	16-11
Communicating with the Child Process.....	16-13
Synchronous Communication.....	16-15
Synchronous Communication Example.....	16-17
Asynchronous Communication.....	16-19
Asynchronous Communication Example	16-21
Launching the Child Process	16-23
Writing Data to the Child.....	16-25
Receiving Data from the Child Process	16-27
The <i>TrUpperCase_PostFunction</i> Function.....	16-29
Controlling the Child Process	16-31
Lab Overview.....	16-33
Module Summary.....	16-35

Module 17 Data Structures

Disembodied Property Lists.....	17-3
SKILL Structures	17-5
Accessing and Updating Structures	17-7
Arrays.....	17-9
2-Dimensional Arrays.....	17-11
Association Tables	17-13
Using Association Tables as Dynamic Arrays.....	17-15
Associating Shapes with Layer Purpose Pairs	17-17
Traversing Association Tables.....	17-19

Association Lists	17-21
Lab Overview.....	17-23
Module Summary.....	17-25

Module 18 OpenAccess

OpenAccess and SKILL	18-3
What Is OpenAccess?	18-5
OpenAccess Database.....	18-7
Database-Centric Flows	18-9
Code Examples	18-11
What is Si2?	18-13
What Is the OpenAccess Coalition?	18-15
What Does OpenAccess Mean for IC 5.0?	18-17
How Does OpenAccess Affect DFII?.....	18-19
How Will DFII on OA Be Supported?	18-21
How to Obtain OpenAccess.....	18-23
Where to Go for Information	18-25

App. A Cellview Data Model Map

App. B Case Study Source Code

App. C OCEAN Example

OCEAN SKILL Script to Control Simulation.....	C-3
SKILL Script Output from Session->Save Script.....	C-5
Running the SKILL Script in Batch Mode	C-7
Modifying the SKILL Script.....	C-9
OCEAN Demonstration.....	C-11

Module 1: About This Course

Objectives

- Itemize the course modules.
- Outline the schedule.
- Explain what is expected and supplied in the lab exercises.
- Describe SKILL documentation.
- Learn about the Cadence online documentation utilities.
- Use the Cadence Finder to identify the available SKILL functions.
- Itemize the libraries used in the course.

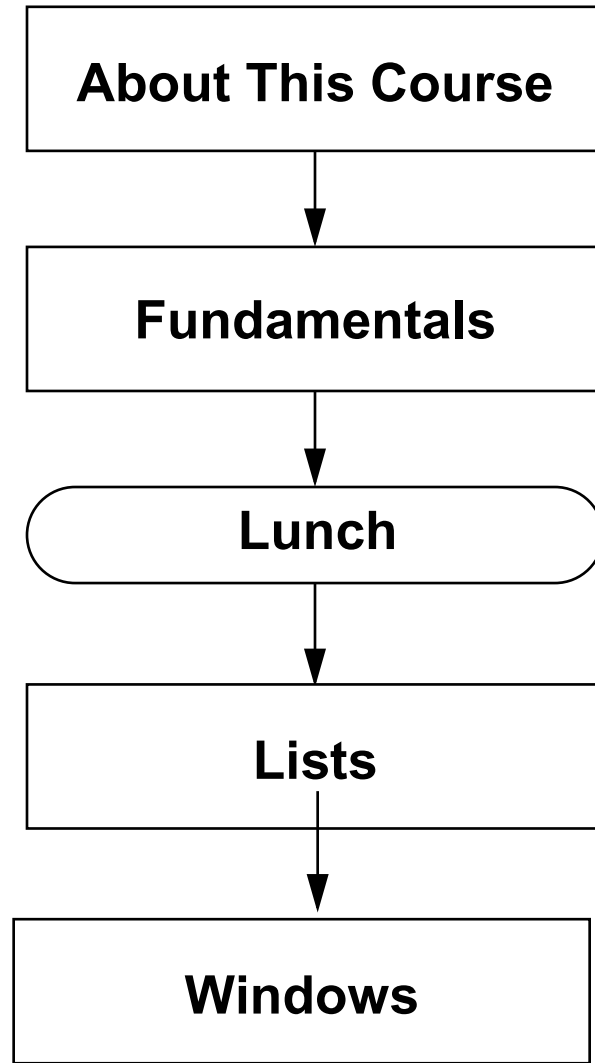
Terms and Definitions

SKILL

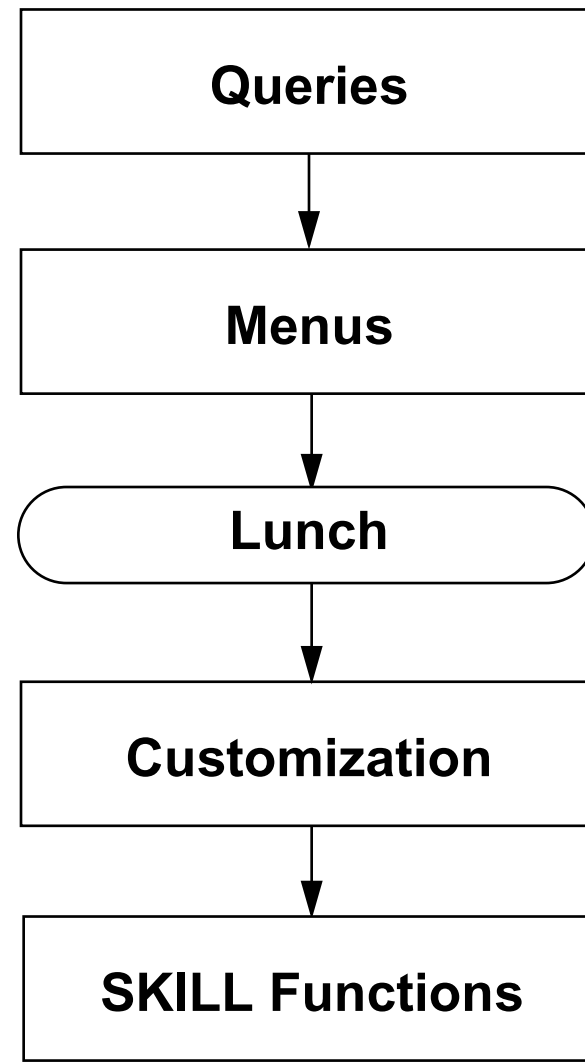
SKILL is a LISP like language with C IO structures. SKILL extends and enhances the functionality of Cadence tools.

Introduction to SKILL Overview

Day 1



Day 2

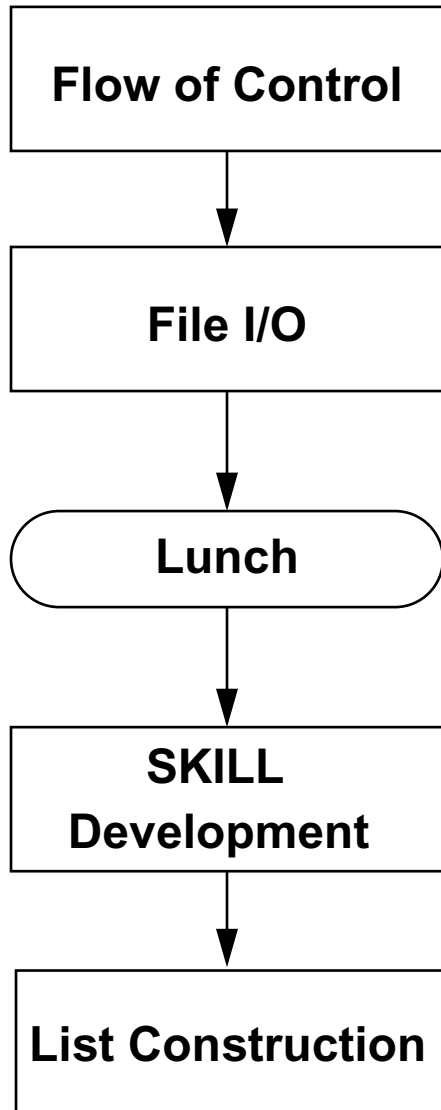


Introduction to SKILL Overview

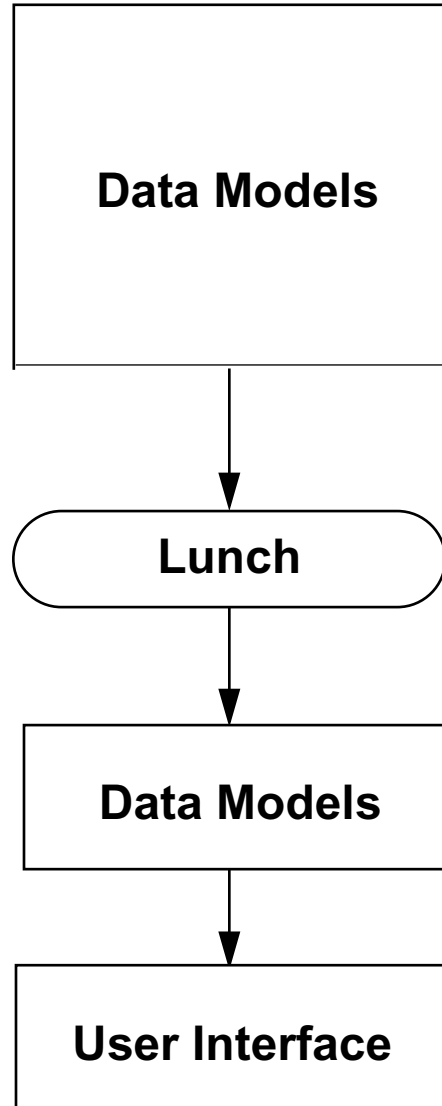
#	Module	Purpose or Objective
1	About This Course	Orientation to the course and SKILL documentation.
2	Beginning to Use SKILL	Survey the role of the SKILL Evaluator in the Design Framework II environment. Survey SKILL data types and syntax.
3	Lists	Build SKILL lists. Retrieve data from SKILL lists.
4	Windows	Manipulate Design Framework II windows. Open Design Framework II application windows. Define application bindkeys.
5	Queries	Query Design Framework II cellviews.
6	Menus	Build pop-up and pull-down menus.
7	Customization	Survey the <i>.cdsinit</i> file.
8	Defining SKILL Functions	Learn to define a SKILL function. Define global and local variables.

SKILL Programming Overview

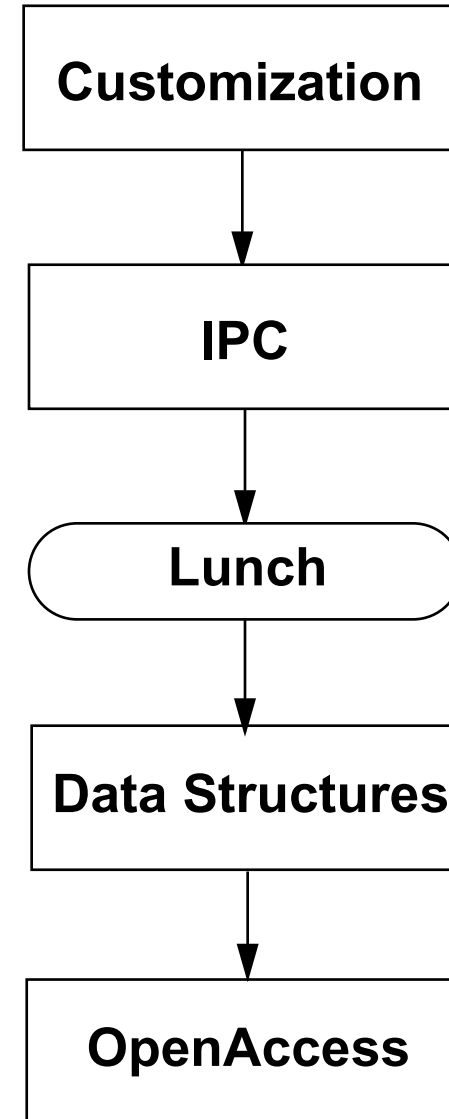
Day 3



Day 4



Day 5



SKILL Programming Overview

#	Module	Purpose or Objective
9	Flow of Control	Use the branching capabilities of the SKILL language.
10	File I/O	Read and write UNIX text files.
11	SKILL Development	Survey the utilities in the SKILL Development Environment.
12	List Construction	Use advanced techniques to build SKILL lists.
13	Data Models	Survey the cellview database model.
14	User Interface	Build dialog boxes, list boxes, and forms.
15	Customization	Customize application user interfaces.
16	Interprocess Communication (IPC)	Launch a child UNIX process.
17	Data Structures	Learn advanced data structures, such as arrays, associative arrays, and SKILL structures.

Lab Exercises

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

- Operational exercises

- Enter an example SKILL expression into the Command Interpreter and observe the results.
Modify the example.
- Examine and run source code.

Solutions to questions are usually on the next page.

- Programming exercises

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

Lab Exercises

Using the Cadence Online Documentation

The new Cadence online documentation system consists of

- A window providing access to the books shipped with your products
- Web-ready manuals in HTML format
- A print-ready PDF (Portable Document Format) file for each manual

The Cadence documentation window lets you:

- Choose and open manuals
- List only those manuals installed on your system
- List books by product name, product family, or document type (manuals, product notes, known problems and solutions)
- Open books from different installed hierarchies
- Open manuals in Netscape or Internet Explorer, automatically starting the web browser if needed
- Open a full-text search page to search for specific text
- Add your own documents into the documentation hierarchy.

Using the Cadence Online Documentation

You invoke the online help system either from UNIX using the *cdsdoc* command or from within the tool using the **Help** menu in the upper right corner of a Cadence application window.

Searching Cadence Documents

You can:

- Search all Cadence documents
- Search selected documents
- Search one document
- Narrow you search with a larger variety of operators
- Search with a combination of operators
- Search for special characters

Searching Cadence Documents

Search Operators

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

Search Operators

Search Operators (continued)

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

Search Operators (continued)

The SKILL Documentation Set

The following manuals cover the SKILL language independently of the design environment.

- SKILL Language Reference Manual
- SKILL User Guide

These manuals cover SKILL applications programming interfaces to the Design Framework II environment.

- User Interface SKILL Functions
- Design Framework II SKILL Functions

These manuals cover SKILL applications programming interfaces to design tools.

- Custom Layout/Physical Verification SKILL Functions
- IC Design Entry & Simulation Environment SKILL Functions

Use the Cadence online documentation to view the SKILL documentation set.

The SKILL Documentation Set

SKILL Language Reference Manual

SKILL User Guide

User Interface SKILL Functions

Design Framework II SKILL Functions

Custom Layout/Physical Verification SKILL Functions

IC Design Entry & Simulation Environment SKILL Functions

The Finder

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

Example description of the *strcat* function:

```
strcat( t_string1 [t_string2 t_string3 ...] ) => t_result
```

Takes input strings and concatenates them.

You can access the Finder in three ways:

- In an xterm window, enter
`cdsFinder &`
- In the SKILL Development window, click the Finder button.
- In the CIW, enter
`startFinder()`

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

The Finder

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

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

To use the Finder, follow these steps:

1. Specify a name pattern in the Search String pane.
2. Click the Search button.
3. Select a function in the Matches window pane.

The Finder displays the abbreviated documentation for the selected function.

Lab Overview

There are no labs for this module.

This rest of this course uses libraries from the *Cell Design Tutorial*.

If you're interested, you can use the Cadence Online Documentation to access the *Cell Design Tutorial*.

Lab Overview

The file `~/SKILL/cds.lib` for this course contains the following information:

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

Module Summary

In this module, we

- Itemized the course modules.
- Outlined the schedule.
- Examined the format of the lab exercises.
- Described SKILL documentation.
- Identified the role of Cadence online documentation in accessing documentation.
- Previewed the Cadence Finder.
- Discussed the libraries used in the course.

Module Summary

Module 2: Beginning to Use SKILL

Objectives

- Start the Design Framework II environment.
- Examine the Command Interpreter Window (CIW).
- Narrate the role of the SKILL Evaluator.
- Examine the CDS.log file.
- Summarize SKILL syntax.
- Display data in the CIW output pane.
- Get the most out of SKILL error messages.

Terms and Definitions

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

What Is SKILL?

SKILL is a high-level, interactive programming language.

SKILL is the command language of the Design Framework II environment.

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

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

What Is SKILL?

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

The IO used in SKILL is that of C. Those of you familiar with Fortran will also see a strong resemblance.

What Can SKILL Functions Do?

The SKILL programming language acts as an extension to the Design Framework II environment.

Some SKILL functions control the Design Framework II environment or perform tasks in design tools. For example, SKILL functions can:

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

Other SKILL functions compute or retrieve data from the Design Framework II environment or from designs. For example, SKILL functions can

- Retrieve the bounding box of the current window.
- Retrieve a list of all the shapes on a given layer purpose pair.

What Can SKILL Functions Do?

The Return Value of a SKILL Function

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

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

Any SKILL data can become a return value.

Starting the Design Framework II Environment

You can choose from two types of sessions:

- Graphic
- Nongraphic

You can replay a Design Framework II session.

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

Note: In the table above `<cds>` represents the name of your executable.

Starting the Design Framework II Environment

Graphic Sessions

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

Nongraphic Sessions

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

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

During a nongraphic session, the Cadence® Design Framework II environment suppresses any graphic output. It does not create any windows.

Replaying Sessions

When replaying a session, the Design Framework II environment evaluates all the SKILL expressions contained in the *-replay* transcript file.

Initializing the Design Framework II Environment

During startup, the Design Framework II environment searches the following directories for a *.cdsinit* file:

- <install_dir>/tools/dfII/local
- the current directory "."
- the home directory

When the Design Framework II environment finds a *.cdsinit* file, it stops searching and loads the *.cdsinit* file.

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

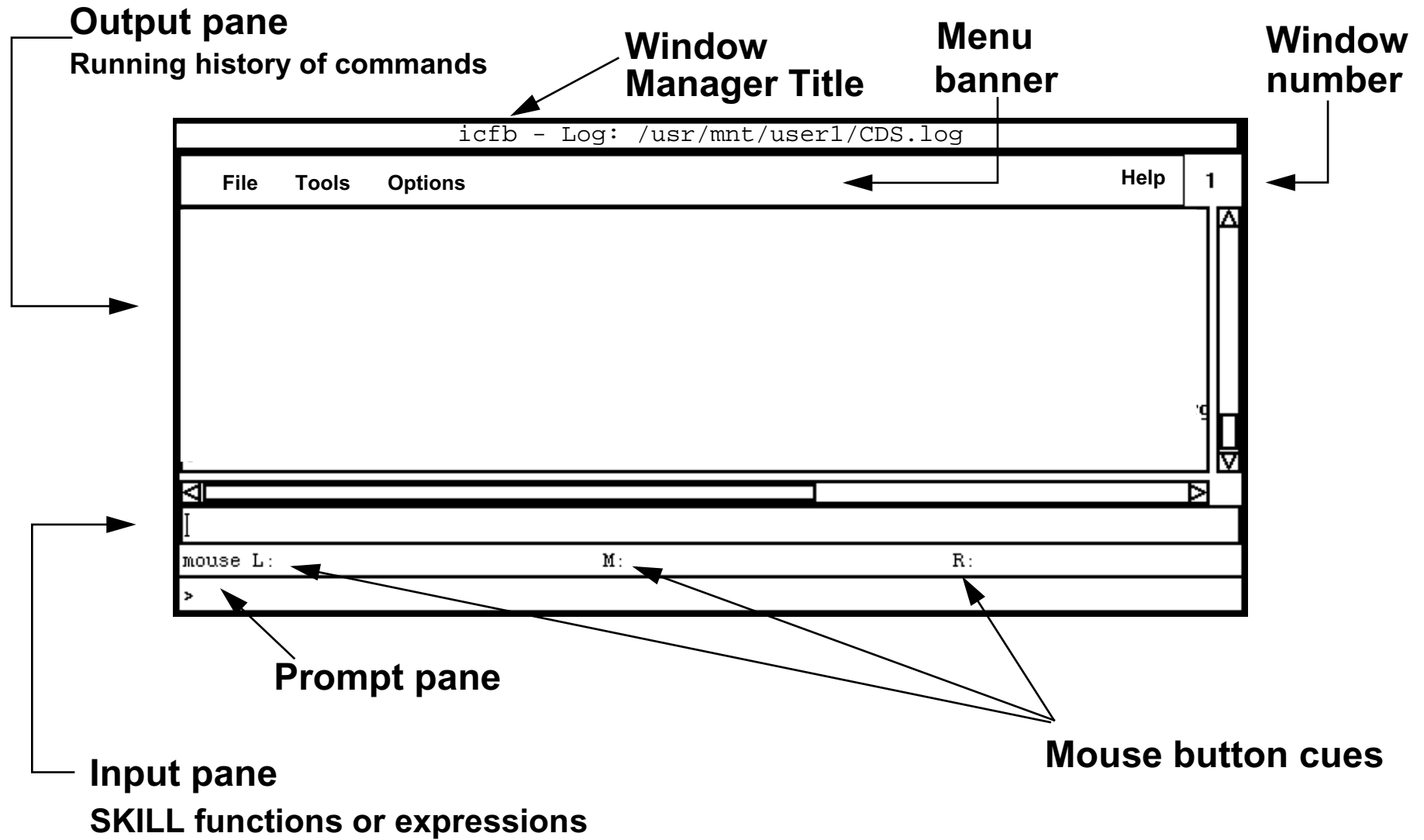
Initializing the Design Framework II Environment

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

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

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

Command Interpreter Window

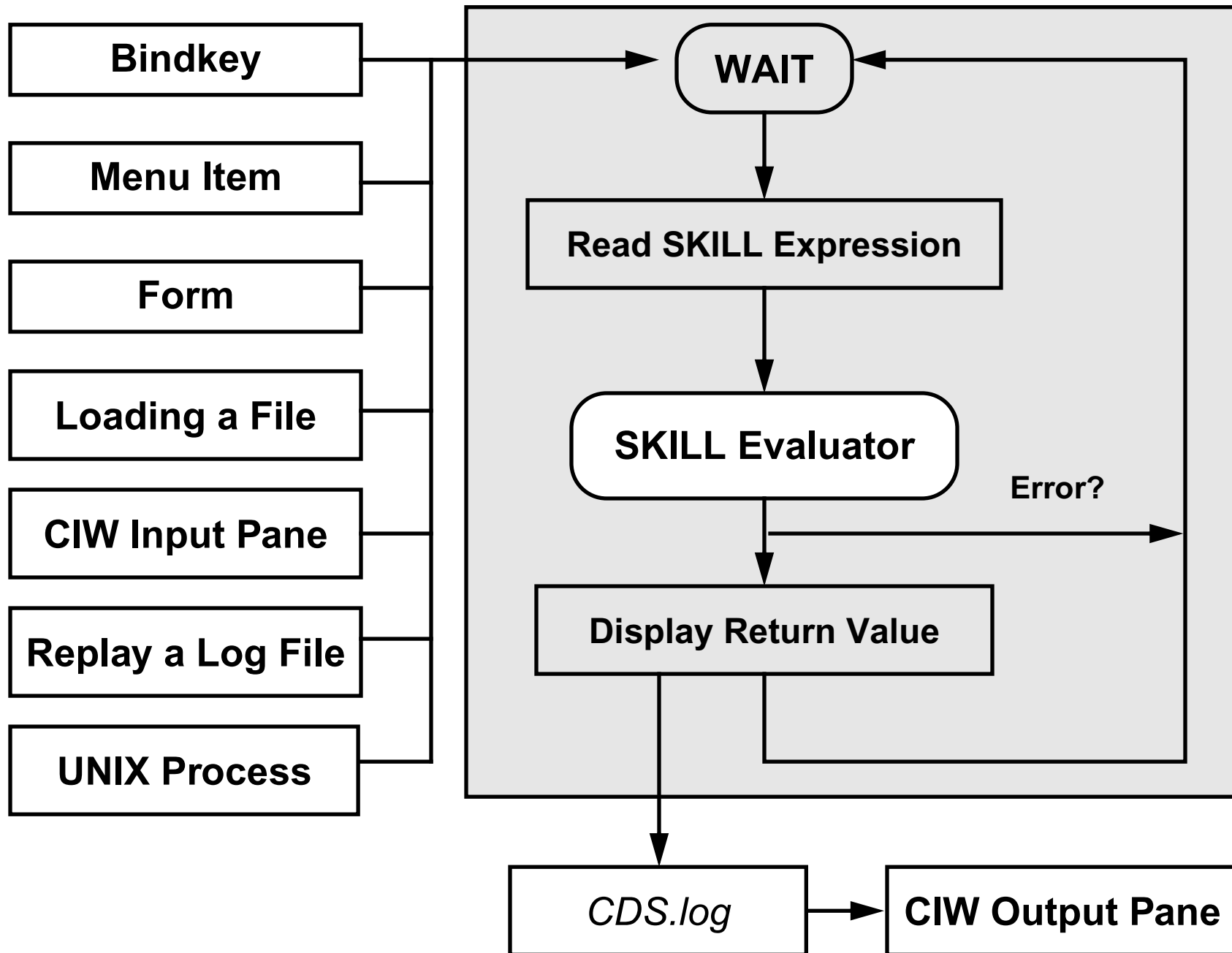


Command Interpreter Window

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

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

Design Framework II User Interface



Design Framework II User Interface

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

Bindkeys and Menu Items

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

Loading SKILL Source Code

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

Replaying a Session File

You can replay a session file. The Design Framework II environment successively sends each SKILL expression in the session file to the SKILL Evaluator. Only the input lines (prefixed by “\i”) and the accelerated input lines (prefixed by “\a”) are processed.

Sending a SKILL Expression from a UNIX Process

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

The CDS.log File

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

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

The following text illustrates an example transcript file:

```
\p 1>
\i x = 0
\t 0
\p 1>
\i TrBump()
\o Old value: 0 New Value: 1
\t 1
\p 1>
\a TrBump()
\o Old value: 1 New Value: 2
\r 2
\i TrBump()
\o Old value: 2 New Value: 3
\t 3
```

```
prompt
user type-in SKILL expression
SKILL expression's return value
prompt
user type-in SKILL expression
SKILL expression output
SKILL expression's return value
Prompt
Bindkey SKILL expression
SKILL expression output
SKILL expression's return value
user type-in SKILL expression
SKILL expression output
SKILL expression's return value
```

The CDS.log File

The definition of the *TrBump* function is:

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

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

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

The CDS.log File Code

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

The CDS.log File Code

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

Setting the Log Filter

You can control the kinds of log file data that the CIW output pane displays.

You can set the seven toggle options of the log filter in several ways.

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

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

hiSetFilterOptions argument positions from Cadence Online Documentation:

- 1) **inputMenuCommands** 2) **inputPrompts** 3) **outputProgramResults**
- 4) **outputMenuCommands** 5) **outputUser** 6) **messageErrors** 7) **messageWarnings**

- From the CIW, use the **Options—Log Filter** command to display the Set Log File Display Filter form.

Log File Category	Toggle Options		
Show Input	menu commands	prompts	
Show Output	user	menu commands	program results
Show Messages	errors	warning	

Setting the Log Filter

The arguments to the *hiSetFilterOptions* function correspond to the form as shown. Note the wording on the form.

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

SKILL Syntax Summary

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

SKILL Syntax Summary

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

Data

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

- You use the input syntax to specify data as an argument or to assign a value to a variable.
- SKILL uses the print representation as the default format when displaying data, unless you specify another format.

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

The following table summarizes several common data types.

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

Data

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

The *type* Function

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

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

Examples:

```
type( 4 ) => fixnum /* an integer */  
type( 5.3 ) => flonum /* a floating point number */  
type( "mary had a little lamb" ) => string /* a string */
```

Variables

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

Variable names can contain

- alphanumeric characters
- underscores (_)
- question marks

The first character of a variable cannot be a digit or a question mark.

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

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

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

Variables

SKILL allows both global and local variables. In Module 8 see, *Grouping Expressions with Local Variables*.

SKILL Symbols

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

- Data value. For example $x = 4$ stores the value 4 in the symbol x .
- Function definition. For example, *procedure(x(a b) a+b)* associates a function definition with the symbol x . The function takes two arguments and returns their sum.
- Property list. For example, $x.raiseTime = .5$ stores the name-value pair *raiseTime .5* on the property list for the symbol x .

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

Function Calls

Function names are case sensitive.

SKILL syntax accepts function calls in three ways:

- State the function name first, followed by the arguments in a pair of matching parentheses.

No spaces are allowed between the function name and the left parenthesis.

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

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

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

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

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

Use white space to separate function arguments.

Function Calls

You can use all three syntax forms together.

Multiple Lines

A literal text string cannot span multiple lines.

Function calls

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

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

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

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

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

- The composite SKILL expression returns the return value of the last SKILL expression.
- All preceding return values are ignored.

Multiple Lines

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

Example:

`a = 2`

`a + 2 * (3 + a) => 12`

however,

`a + 2`

`* (3 + 2) =>`

`4`

`* error * - wrong number of arguments: mult expects 2 arguments`

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

Example:

`myString = "this string spans \`

`two lines using a backslash at the end of the first line"`

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

Understanding Function Arguments

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

The documentation for each argument tells you

- The expected data type of the argument
- Whether the argument is required, optional, or a keyword argument

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

- Required arguments with no optional arguments
- Keyword arguments with no required and no optional arguments

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

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

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

Understanding Function Arguments

Required Arguments

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

Optional Arguments

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

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

Keyword Arguments

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

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

Operators

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

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

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

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

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

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

Operators

Operator Precedence

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

Each operator corresponds to a SKILL function.

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

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

Tracing Operator Evaluation

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

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

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

To turn off tracing you use:

```
untrace()
```

Tracing Operator Evaluation

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

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

Lab Overview

Lab 2-1 Using the Command Interpreter Window

Lab 2-2 Exploring SKILL Numeric Data Types

Lab 2-3 Exploring SKILL Variables

Lab Overview

Displaying Data in the CIW

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

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

SKILL displays a return value with its print representation.

SKILL functions often display data before they return a value.

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

Displaying Data in the CIW

The *print* and *println* Functions

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

This nongraphic session transcript illustrates *println*.

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

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

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

Displaying Data with Format Control

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

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

The first argument is a conversion control string containing directives.

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

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

Displaying Data with Format Control

The *printf* Function

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

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

The %L Directive

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

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

Solving Common Problems

These are common problems you might encounter:

- The CIW does not respond.
- The CIW displays inexplicable error messages.
- You pass arguments incorrectly to a function.

Solving Common Problems

What If the CIW Doesn't Respond?

Situation:

- You typed in a SKILL function.
- You pressed Return.
- Nothing happens.

You have one of these problems:

- Unbalanced parentheses
- Unbalanced string quotes

Solution:

The following steps trigger a system response in most cases.

- You might have entered more left parentheses than right parentheses.
- Enter a] character (a closing right square bracket). This character closes all outstanding right parentheses.
- If still nothing happens, enter the " character followed by the] character.

What If the CIW Doesn't Respond?

White Space Sometimes Causes Errors

White space can cause error messages.

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

The error messages:

- do not identify the white space as the cause of the problem.
- vary depending on the surrounding context.

Examples

A SKILL function to concatenate several strings

```
strcat ( "mary" " had" " a" " little" " lamb")  
*** Error in routine eval:  
Message: *Error* eval: illegal function mary
```

An assignment to a variable

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

White Space Sometimes Causes Errors

Passing Incorrect Arguments to a Function

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

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

Examples

The *strcat* function does not accept numeric data.

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

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

The *strlen* function expects at least one argument.

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

Passing Incorrect Arguments to a Function

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

- The number of arguments that the function expects.
- The expected data type of each argument.

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

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

Lab Overview

Lab 2-4 Displaying Data in the CIW

Lab 2-5 Solving Common Errors

Lab Overview

Module Summary

This module

- Introduced SKILL, the command language for the Design Framework II environment.
- Defined what user interface action sends SKILL expressions to the SKILL Evaluator.
- Explored SKILL data, function calls, variables, and operators.
- Showed you ways to solve common problems.

Module Summary

Module 3: SKILL Lists

Objectives

- Build lists
- Retrieve list elements
- Use lists to represent points and bounding boxes

Terms and Definitions

What Is a SKILL List?

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

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

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

SKILL functions commonly return lists you can display or process.

Design database list examples:

- Shapes in a design
- Points in a path or polygon
- Instances in a design

User interface list examples:

- Design Framework II windows currently open
- Pull-down menus in a window
- Menu items in a menu

What Is a SKILL List?

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

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

How SKILL Displays a List

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

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

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

- contains sublists.
- has more than *_itemsperline* number of items.

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

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

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

How SKILL Displays a List

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


Creating New Lists

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

- Specify all the elements literally. Apply the `'` operator to the list.

Expressions	Return Result
<code>'(1 2 3)</code>	<code>(1 2 3)</code>
<code>'("one" 1)</code>	<code>("one" 1)</code>
<code>'(("one" 1) ("two" 2))</code>	<code>(("one" 1) ("two" 2))</code>

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

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

Creating New Lists

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

The ' Operator

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

- Include sublists as elements with a single set of parentheses.
- Do not use the ' operator in front of the sublists.
- Separate successive sublists with white space.

The *list* Function

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

Adding Elements to an Existing List

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

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

Expressions	Return Result
<i>result = '(2 3)</i>	<i>(2 3)</i>
<i>result = cons(1 result)</i>	<i>(1 2 3)</i>

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

Expressions	Return Result
<i>oneList = '(4 5 6)</i>	<i>(4 5 6)</i>
<i>aList = '(1 2 3)</i>	<i>(1 2 3)</i>
<i>bList = append(oneList aList)</i>	<i>(4 5 6 1 2 3)</i>

Adding Elements to an Existing List

The *cons* Function

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

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

The *append* Function

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

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

Points of Confusion

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

Question

Answer

What is the difference between *nil* and *'(nil)*?

nil is a list containing nothing. Its length is 0.
'(nil) builds a list containing the single element *nil*. The length is 1.

How can I add an element to the end of a list?

Use the *append* and *list* functions.
aList = *'(1 2)*
aList = *append(aList list(3))* => *(1 2 3)*

Can I reverse the order of the arguments to the *cons* function? Will the results be the same?

You either get different results or an error.
cons('(1 2) '(3 4)) => *((1 2) 3 4)*
cons('(3 4) '(1 2)) => *((3 4) 1 2)*
cons(3 '(1 2)) => *(3 1 2)*
cons('(1 2) 3) =>
*** Error in routine cons
Error cons: argument #2 should be a list

What is the difference between *cons* and *append*?

cons('(1 2) '(3 4)) => *((1 2) 3 4)*
append('(1 2) '(3 4)) => *(1 2 3 4)*

Points of Confusion

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

Working with Existing Lists

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

Working with Existing Lists

The *nth* Function

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

The *member* Function

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

The *setof* Function

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

- The first argument is the variable that stands for an element of the list.
- The second argument is the list.
- The third argument is one or more expressions that you write in terms of the variable. The final expression is the test. It determines if the element is included in the new list.

Traversing a list with car and cdr

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

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

2: `car(alist) => 1`

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

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

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

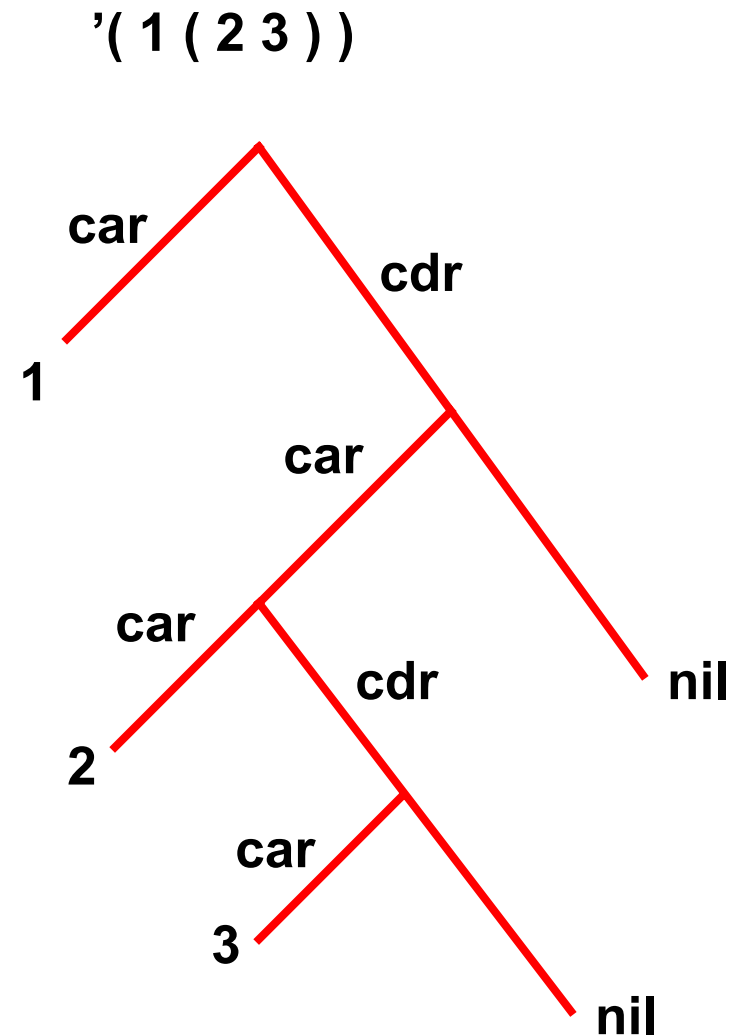
5: `car(car(cdr(alist))) => 2`

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

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

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

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



Traversing a list with car and cdr

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

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

Frequently Asked Questions

Students often ask these questions:

- Why are such critical functions as *car* and *cdr* called such weird names?
- What is the purpose of the *car* and *cdr* functions?
- Can the *member* function search all levels in a hierarchical list?
- How does the *setof* function work? What is the variable *x* for?

Frequently Asked Questions

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

Two-Dimensional Points

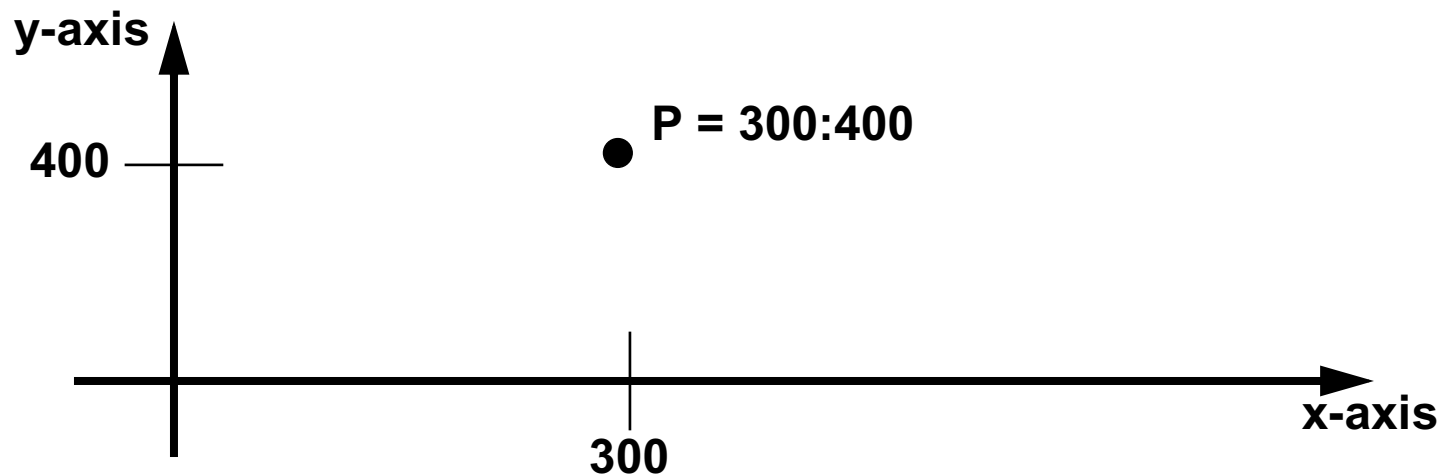
SKILL represents a two-dimensional point as a two-element list.

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

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

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

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



Two-Dimensional Points

The : Operator

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

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

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

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

The *xCoord* and *yCoord* Functions

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

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

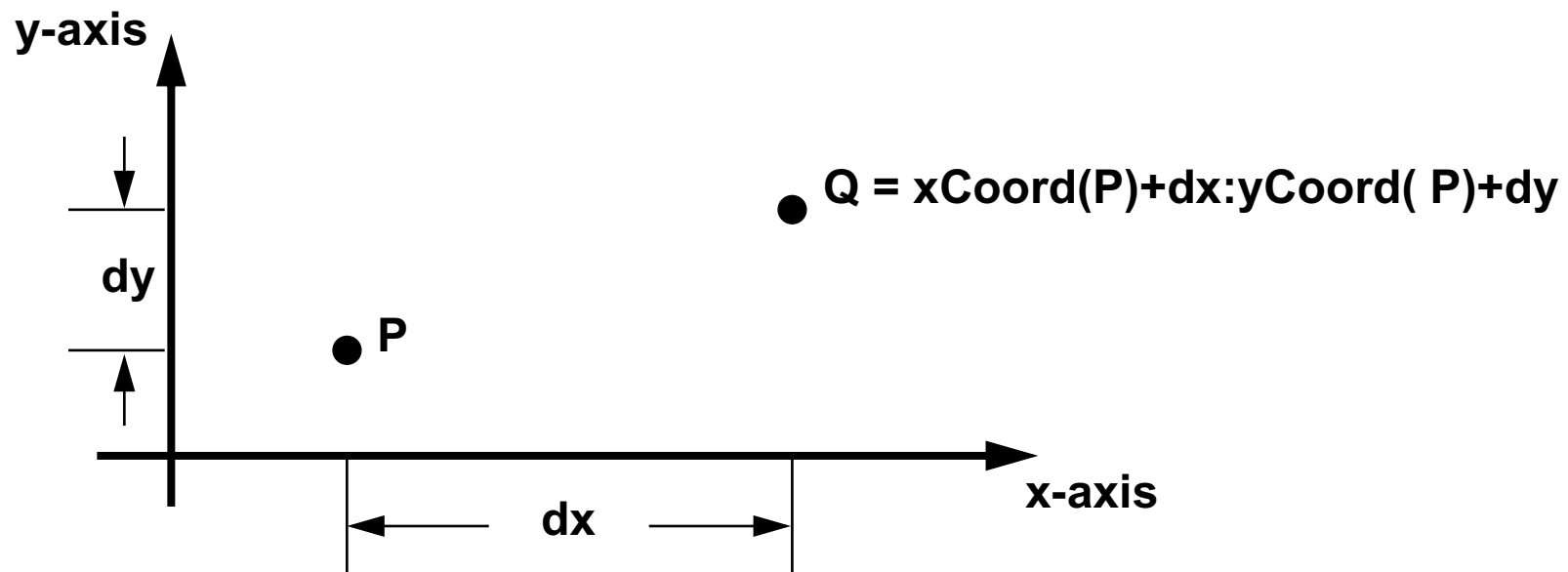
Computing Points

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

$$3+4*5:4+7*8 \Rightarrow (23 \ 60)$$

Computing a point from another point is easy.

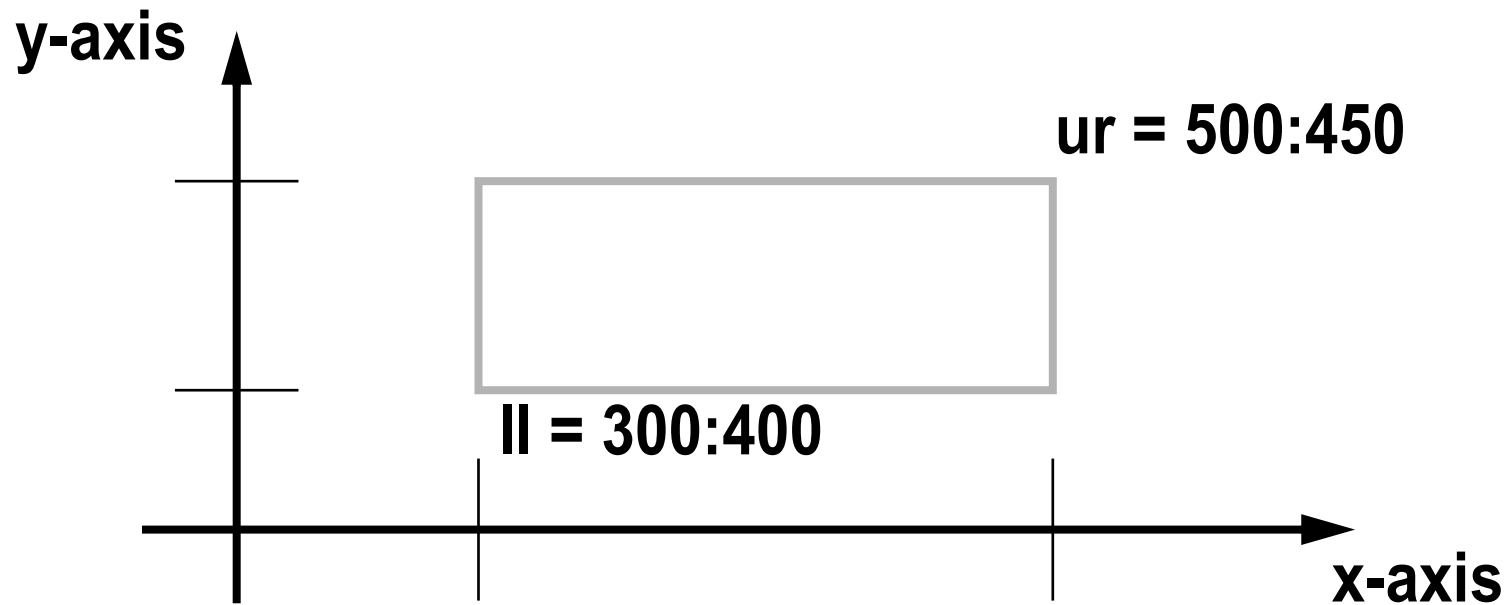
For example, given a point P , apply an offset dx and dy in both directions.



Computing Points

Bounding Boxes

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



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

Bounding Boxes

Remember that `:` is a point operator.

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

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

Creating a Bounding Box

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

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

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

Creating a Bounding Box

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

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

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

Retrieving Elements from Bounding Boxes

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

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

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

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

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

Retrieving Elements from Bounding Boxes

Lecture Exercises

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

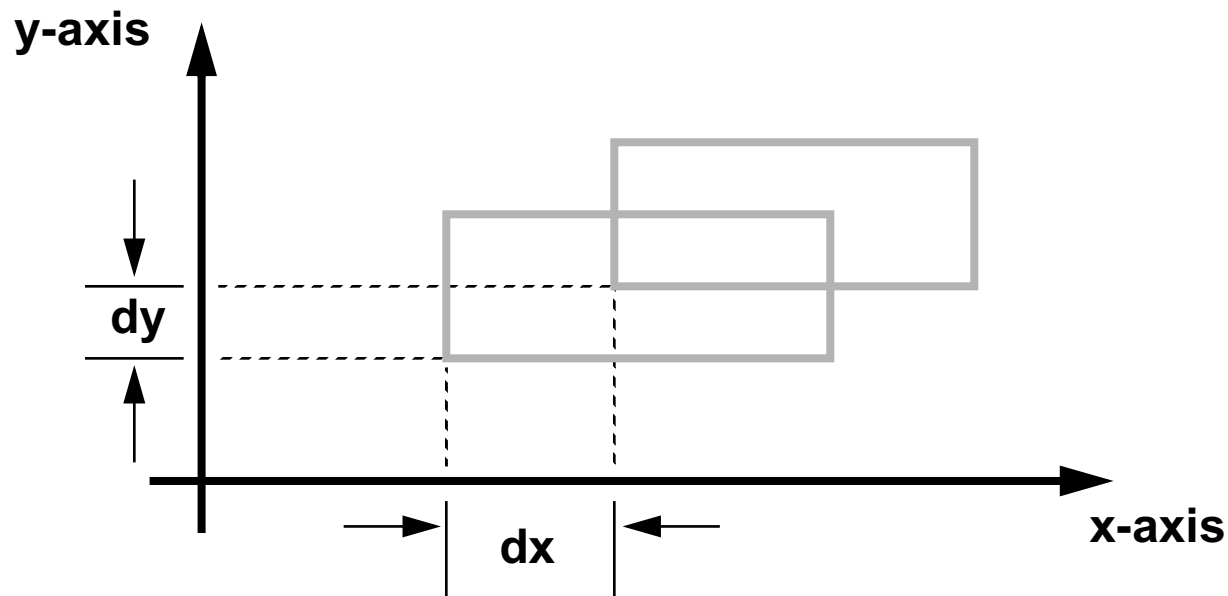
- Offsetting a box
- Finding the smallest bounding box
- Finding the intersection of two bounding boxes

Lecture Exercises

Offsetting a Box

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

Using variables, write an expression for the upper-right bounding box.



Use this template as a starting point for writing the expression.

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

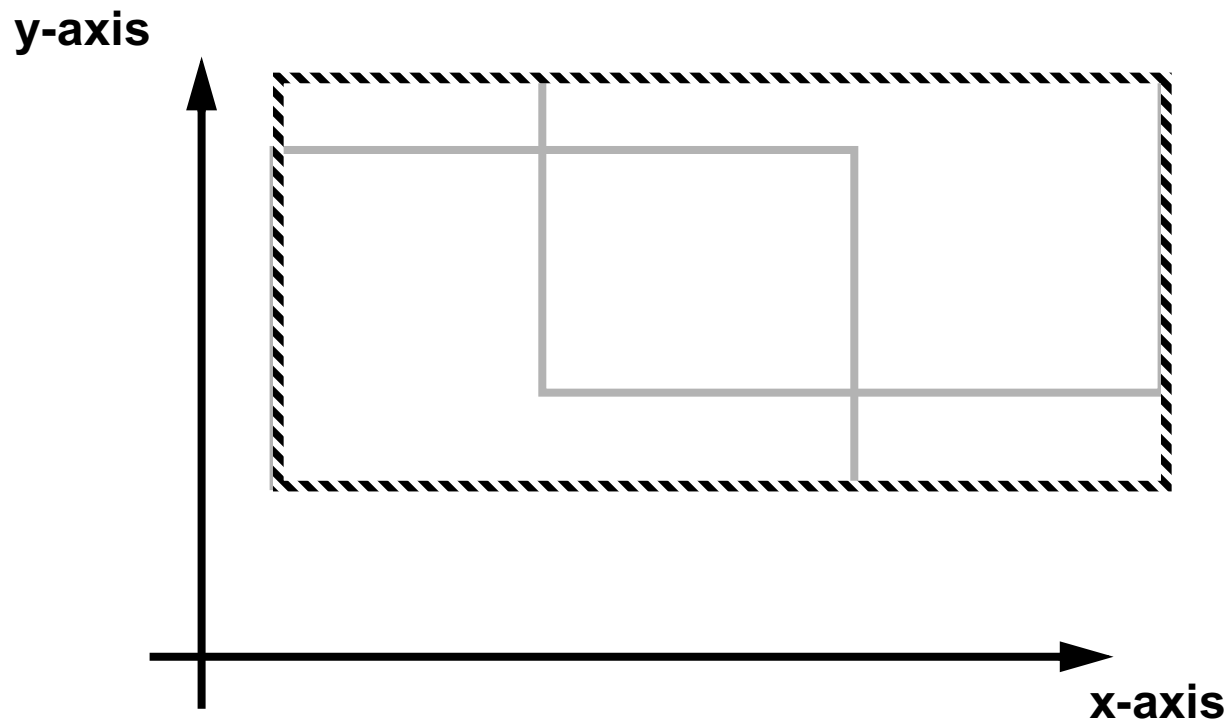
Offsetting a Box

To view the solution, turn the page upside down.

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

Finding the Smallest Bounding Box

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



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

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

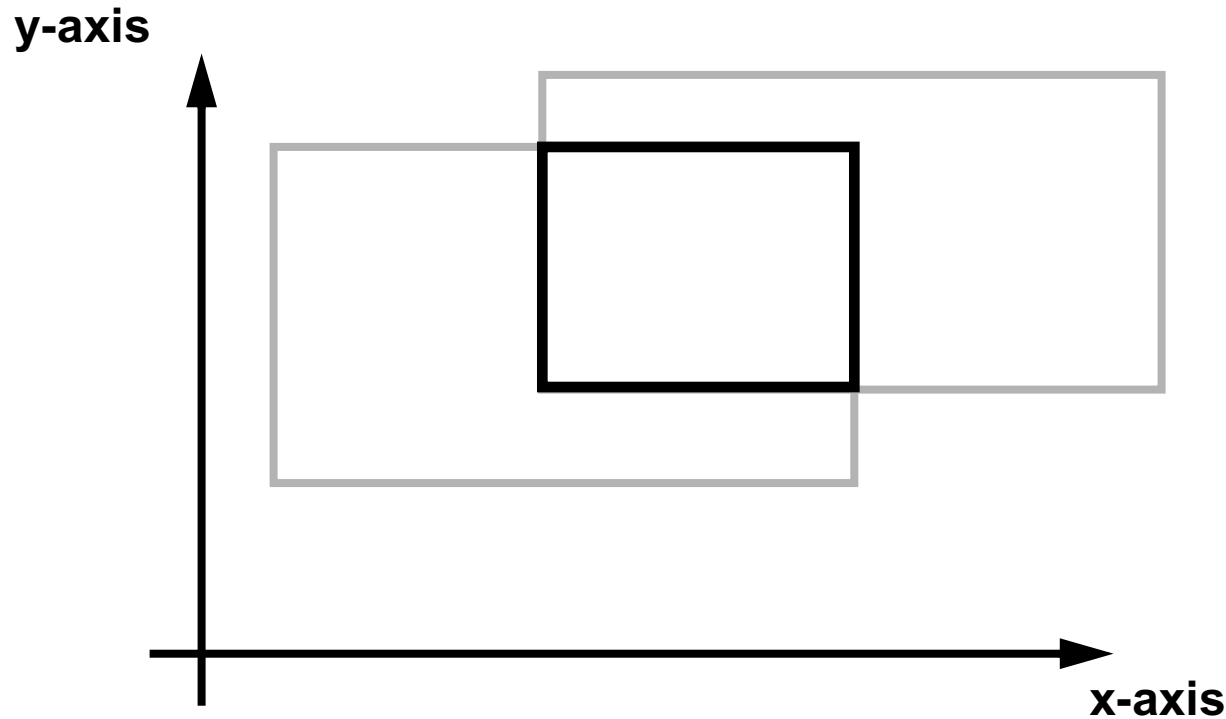
Finding the Smallest Bounding Boxes

To view the solution, turn the page upside down.

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

Finding the Intersection of Two Bounding Boxes

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



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

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

Finding the Intersection of Two Bounding Boxes

To view the solution, turn the page upside down.

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

Combinations *car* and *cdr* Functions

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

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

Functions	Combination	Bounding box examples	Expression
<i>car</i>	<i>car</i> (...)	lower left corner	<i>ll</i> = <i>car</i> (<i>bBox</i>)
<i>cadr</i>	<i>car</i> (<i>cdr</i> (...))	upper right corner	<i>ur</i> = <i>cadr</i> (<i>bBox</i>)
<i>caar</i>	<i>car</i> (<i>car</i> (...))	x-coord of lower left corner	<i>llx</i> = <i>caar</i> (<i>bBox</i>)
<i>cadar</i>	<i>car</i> (<i>cdr</i> (<i>car</i> (...)))	y-coord of lower left corner	<i>lly</i> = <i>cadar</i> (<i>bBox</i>)
<i>caadr</i>	<i>car</i> (<i>car</i> (<i>cdr</i> (...)))	x-coord of upper right corner	<i>urx</i> = <i>caadr</i> (<i>bBox</i>)
<i>cadadr</i>	<i>car</i> (<i>cdr</i> (<i>car</i> (<i>cdr</i> (...]	y-coord of upper right corner	<i>ury</i> = <i>cadadr</i> (<i>bBox</i>)

Combinations *car* and *cdr* Functions

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

The *cadr*, *caar*, *cadar* Functions

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

Lab Overview

Lab 3-1 Building Lists

Lab Overview

Module Summary

This module covered the following:

- SKILL lists can contain any type of SKILL data. *nil* is the empty list.
- Use the ' operator and the *list* function to build lists.
- Use the *cons* and *append* functions to build lists from existing lists.
- Use the *length* function to count the number of elements in a list.
- Use the *member* function to find an element in an existing list.
- Use the *setof* function to filter a list according to a condition.
- Two-dimensional points are represented by two element lists.
- Bounding boxes are also represented by two element lists.

Module Summary

Module 4: Windows

Objectives

- Understand the window ID data type.
- Open design windows.
- Define application bindkeys.
- Open read-only text windows.
- Manage windows.

Terms and Definitions

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

Design Framework II Windows

Most Design Framework II windows have a window number.

- CIW
- Application windows
- The Layer Selection Window (LSW) is not one of these windows.

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

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

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

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

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

The CIW is never the current window.

SKILL functions act on the current window by default.

Design Framework II Windows

The *hiGetWindowList* Function

The *hiGetWindowList* function returns a list of the window IDs of the existing Design Framework II windows.

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

The *hiGetCurrentWindow* Function

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

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


Opening a Design Window

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

The *geOpen* function

- Loads the design into virtual memory.
- Creates a window displaying the design.
- Returns the window ID.

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

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

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

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

Opening a Design Window

The *geOpen* Function

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

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

Using Bindkeys

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

There are several different uses for bindkeys:

- To initiate a command, such as the Zoom In command.
- To use the mouse during a command to enter points.
- To perform subsidiary actions, such as panning the window, during a command.

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

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

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

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

Using Bindkeys

This example illustrates the different uses of bindkeys.

1. With the mouse cursor in a Layout Editor window, the user presses the **z** key.
The Zoom In command starts.
2. The user clicks the left mouse button to indicate the first corner of the region to zoom.
3. The user presses the **Tab** key.
The Pan command starts.
4. The user clicks the left mouse button to indicate the center point of the pan command.
The Pan command finishes.
5. Finally, to indicate the second corner of the region, the user clicks the left mouse button again.
The Zoom In command finishes.

Defining Bindkeys

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

- The application type, which identifies the application by means of a text string. Typical application types include the following:
 - *"Command Interpreter"*
 - *"Layout"*
 - *"Schematics"*
 - *"Graphics Browser"*
- The keyboard or mouse event that triggers the SKILL expression. Typical events include the following:
 - Press the **a** key.
 - Press the left mouse button.
 - Draw through with the left mouse button.
- The mode that governs the bindkey. The bindkey is either modeless or is in effect only when the user enters points.
- The SKILL expression that the bindkey triggers.

Defining Bindkeys

The *hiGetAppType* Function

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

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

The *hiSetBindKey* Function

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

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

Use the curly braces, { }, to group several SKILL expressions together.

Describing Events

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

- Study the Cadence online documentation.
- Display the bindkeys for an application that uses the event.

Examples

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

To limit the event to entering points, append EF to the event syntax as in this example:

"<Btn1Down>EF"

Describing Events

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

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

Enterfunctions

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

The list of enterfunctions that collect points are:

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

Additional enterfunctions are *enterNumber* and *enterString*.

Displaying Bindkeys

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

1. In the CIW, use the **Options—Bind Key** command to display the Key or Mouse Bindings form.
2. In the Application Type Prefix cyclic field, choose the name of the application.
3. Click the Show Bind Keys button.

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

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

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

Displaying Bindkeys

The *hiGetBindKey* Function

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

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

The *hiSetBindKeys* Function

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

Standard Bindkey Definitions

The following files contain the standard bindkey definitions:

- `<install_dir>/tools/dfll/samples/local/schBindKeys.il`
- `<install_dir>/tools/dfll/samples/local/leBindKeys.il`

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

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

Standard Bindkey Definitions

The *alias* Function

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

```
alias( bk hiSetBindKey )
```

Opening a Text Window

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

Example

This example displays the Composer bindkey file.

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

Use the *prependInstallPath* function to make a pathname relative to the Design Framework II installation directory. This function prepends *<install_dir>/tools/dfl* to the path name.

Example

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

```
view(  
    prependInstallPath("samples/local/schBindKeys.il" )  
    ;; path to file  
    '((406 506) (1032 806)) ;; window bounding box  
    "Schematics Bindkeys" ;; window title  
) => window:6
```

Opening a Text Window

The *view* Function

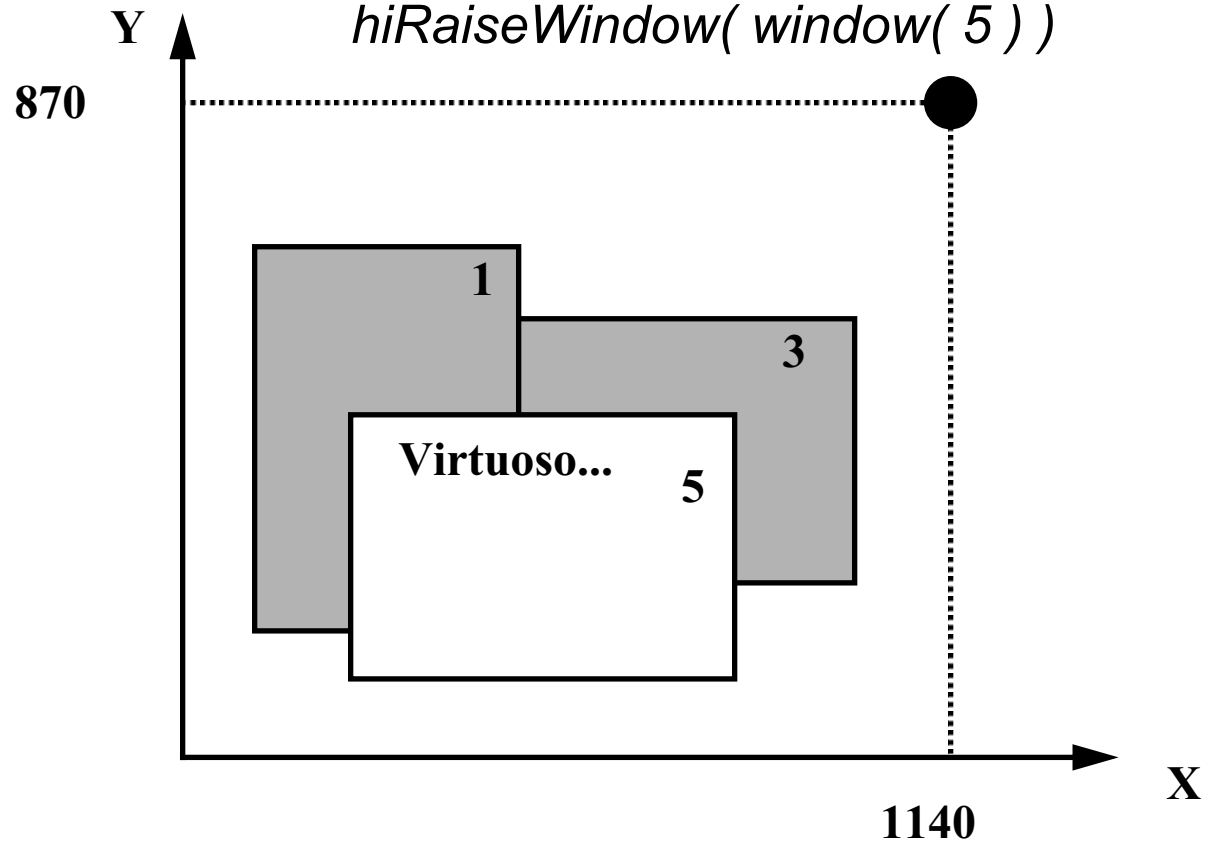
The *view* function takes several optional arguments.

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

Manipulating Windows

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

hiRaiseWindow(window(5))



Manipulating Windows

Naming Windows

■ The **hiGetWindowName** Function

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

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

■ The **hiSetWindowName** Function

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

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

Raising and Lowering Windows

■ The **hiRaiseWindow** Function

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

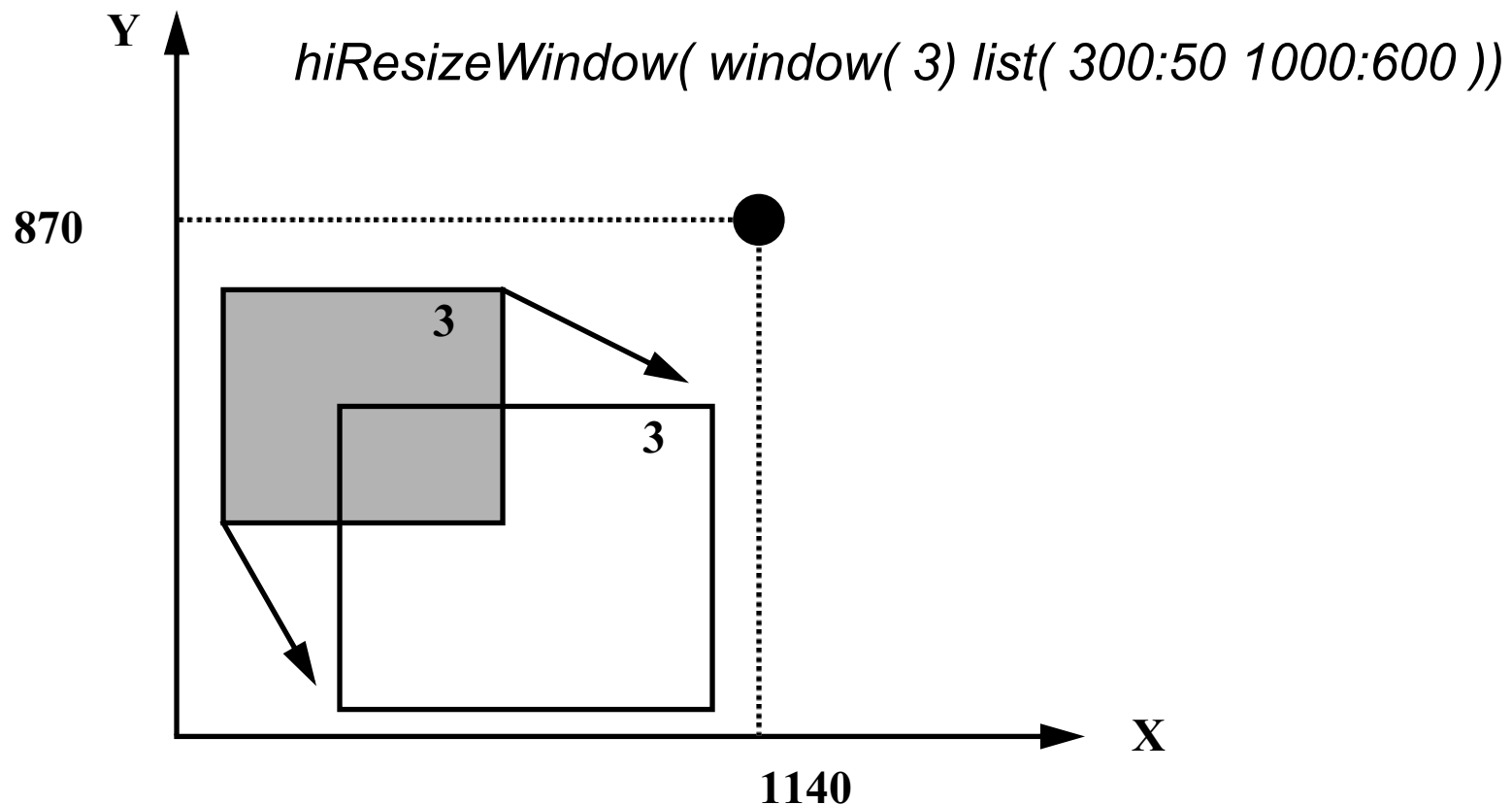
```
hiRaiseWindow( window(5) ) => t
```

■ The **hiLowerWindow** Function

Resizing Windows

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

The unit of measurement for screen coordinates is a pixel.



Resizing Windows

The *hiGetMaxScreenCoords* Function

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

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

The *hiGetAbsWindowScreenBBox* Function

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

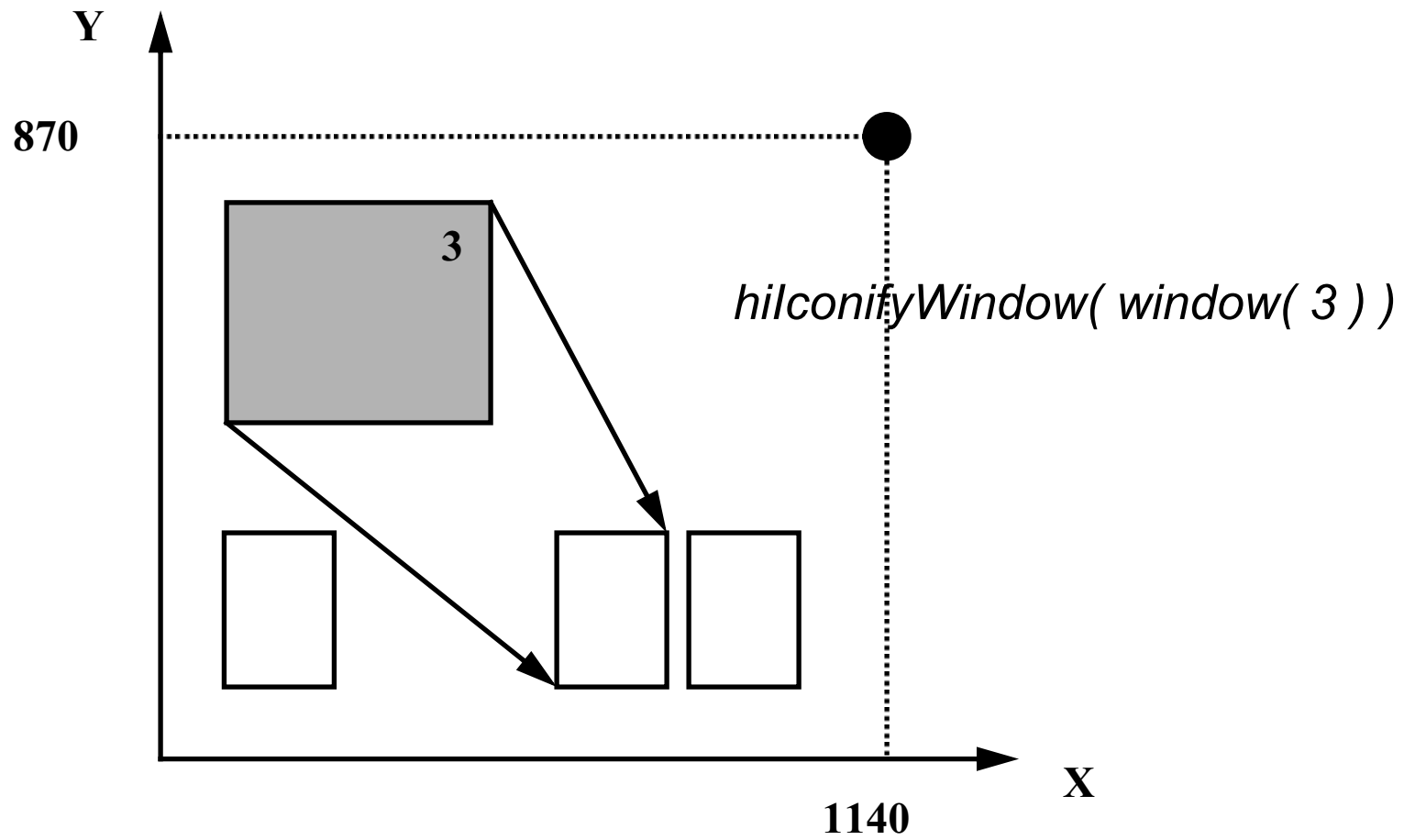
```
hiGetAbsWindowScreenBBox( window(1) t ) =>  
((200 300) (650 700))
```

The *hiResizeWindow* Function

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

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

Iconifying Windows



Iconifying Windows

The *hiIconifyWindow* Function

The *hiIconifyWindow* function iconifies an open window.

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

The *hiGetWindowIconifyState* Function

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

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

The *hiDeiconifyWindow* Function

The *hiDeiconifyWindow* function opens an iconified window.

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

Lab Overview

Lab 4-1 Opening Windows

Lab 4-2 Resizing Windows

Lab 4-3 Storing and Retrieving Bindkeys

Lab 4-4 Defining a Show File Bindkey

Lab Overview

Module Summary

In this module, we discussed

- The window ID data type.
- Opening design windows.
- Defining application bindkeys.
- Opening read-only text windows.
- Managing windows.

Module Summary

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

Module 5: Database Queries

Objectives

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

Terms and Definitions

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

Database Objects

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

- Physical information (rectangles, lines, and paths)
- Logical information (nets and terminals)

SKILL organizes design data in virtual memory in terms of database objects.

Each database object belongs to an object type that defines a set of common attributes that describe the object.

The set of object types and their attributes is fixed by Cadence.

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

- the *cellView* object type
- the *inst* object type
- the *net* object type
- the Figure object types with their common attributes
- the Shape object types with their common attributes

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

Database Objects

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

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

- a string
- an integer
- a floating-point number
- a database object
- a list, possibly of database objects

Querying a Design

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

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

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

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

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

Querying a Design

The *geGetWindowCellView* Function

See the Cadence online documentation to read about this function.

The ~> Operator

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

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

Summary of ~> Syntax

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

The ~> Operator

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

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

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


Querying Designs with the ~> Operator

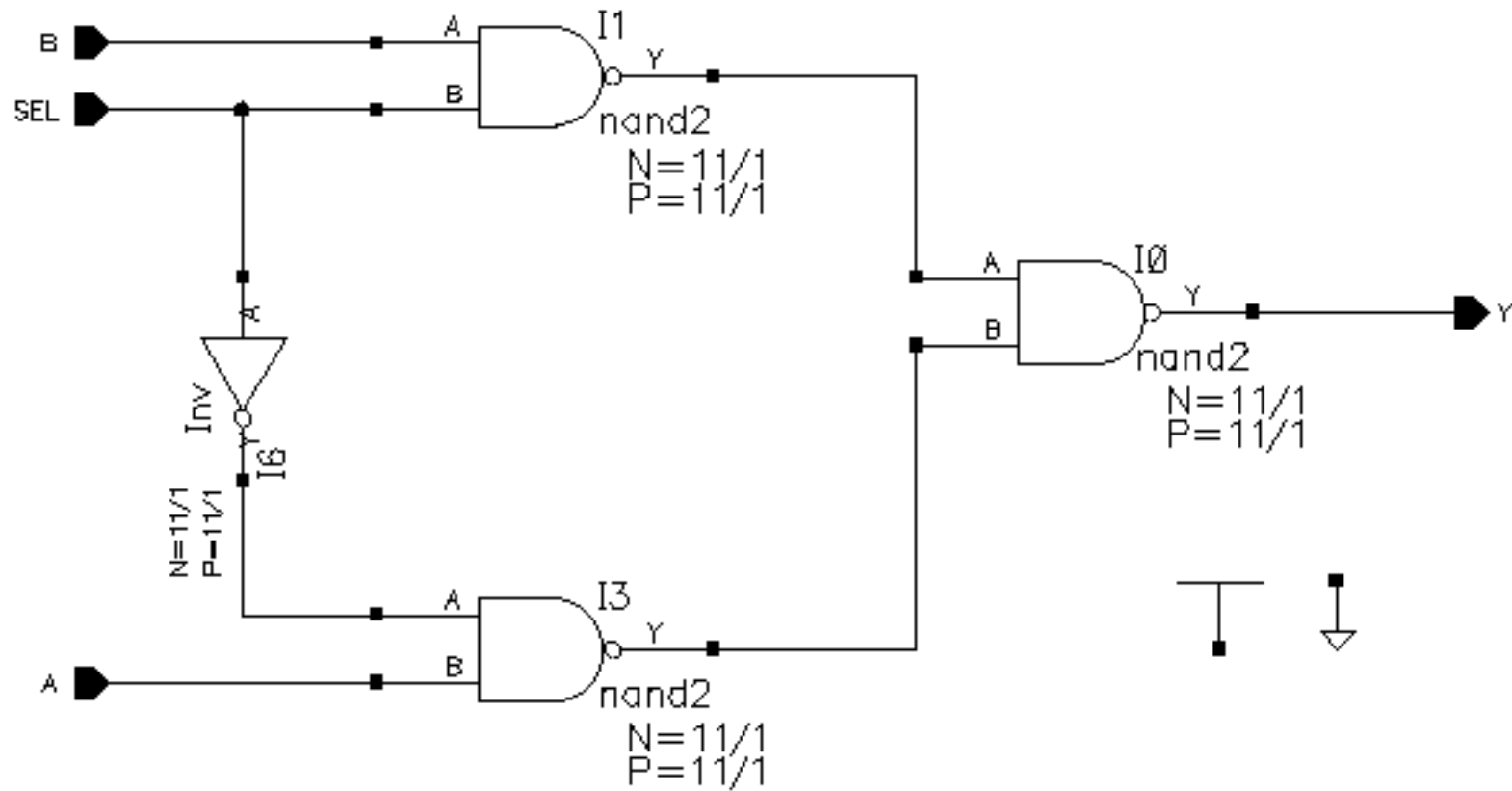
The queries stated in the objectives for this module are previewed below.

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

Querying Designs with the $\sim>$ Operator

The *cellView* Object Type

Example: *master mux2* schematic

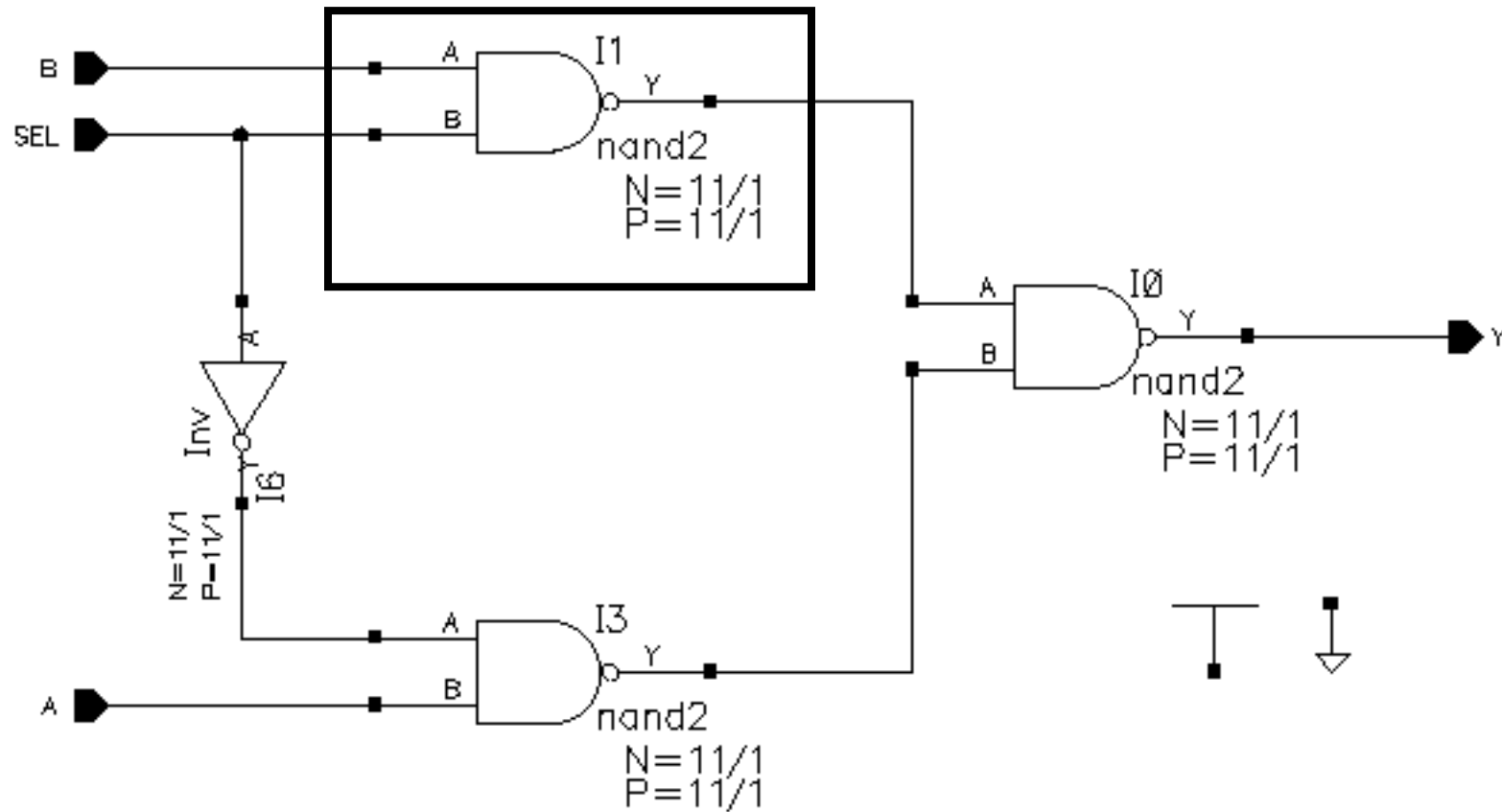


The *cellView* Object Type

The *cellView* object type includes the following attributes among others:

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

Example: Instance *I1* of *master mux2* schematic



In this design, each of the instances has user-defined properties that describe the physical sizes of the transistors.

Instances

The *inst* Object Type

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

- The list of the instances in the design

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

- The list of instance names

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

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

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

- The list of master cell names

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

The *inst* Object Type

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

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

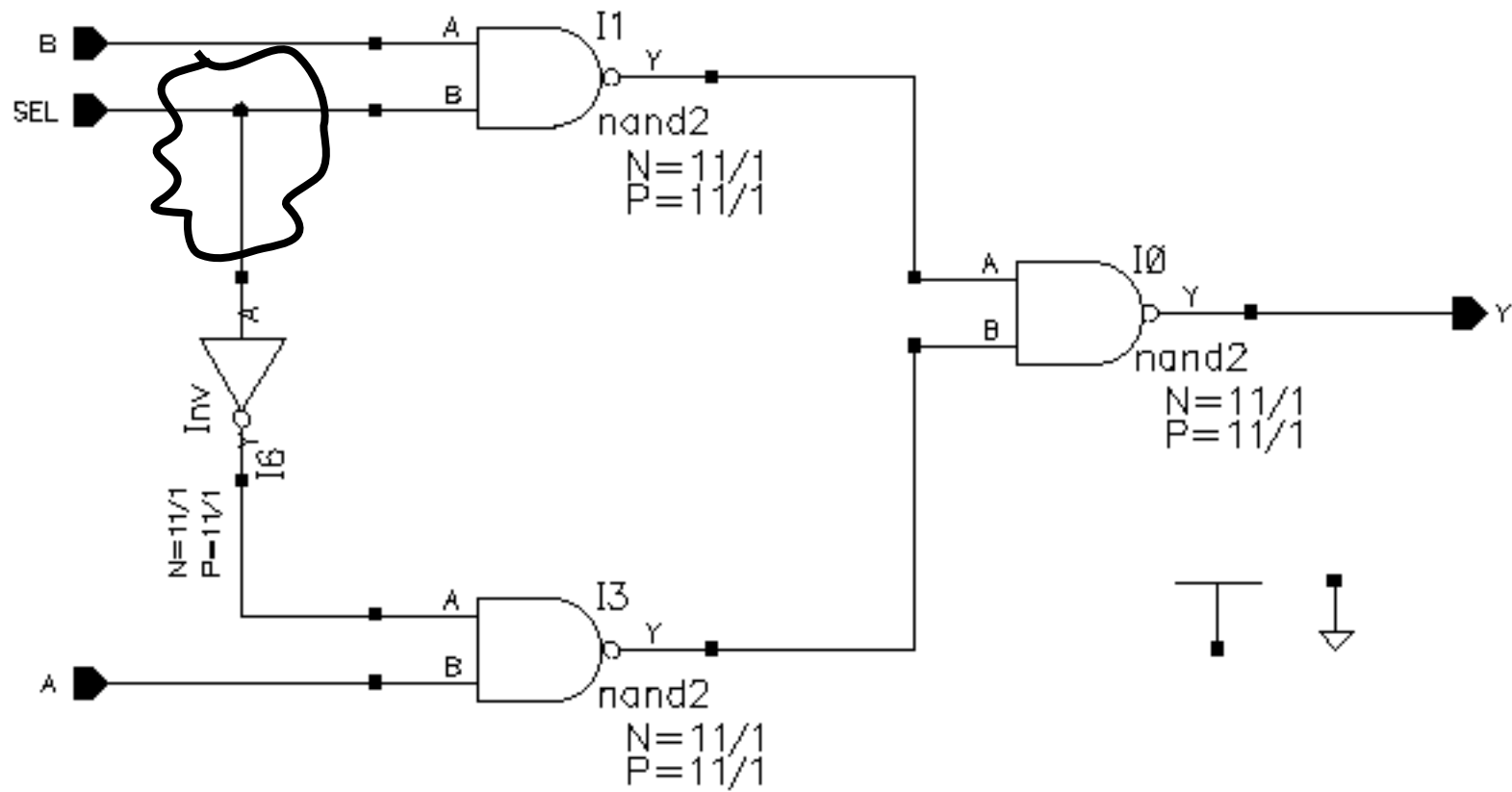
The *dbFindAnyInstByName* Function

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

See the Cadence online documentation to read about this function.

Nets

Example: net *SEL* in *master mux2* schematic



Nets

The *net* Object Type

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

- The nets in the design

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

- The number of nets

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

- The names of the nets

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

The *net* Object Type

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

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

The *dbFindNetByName* Function

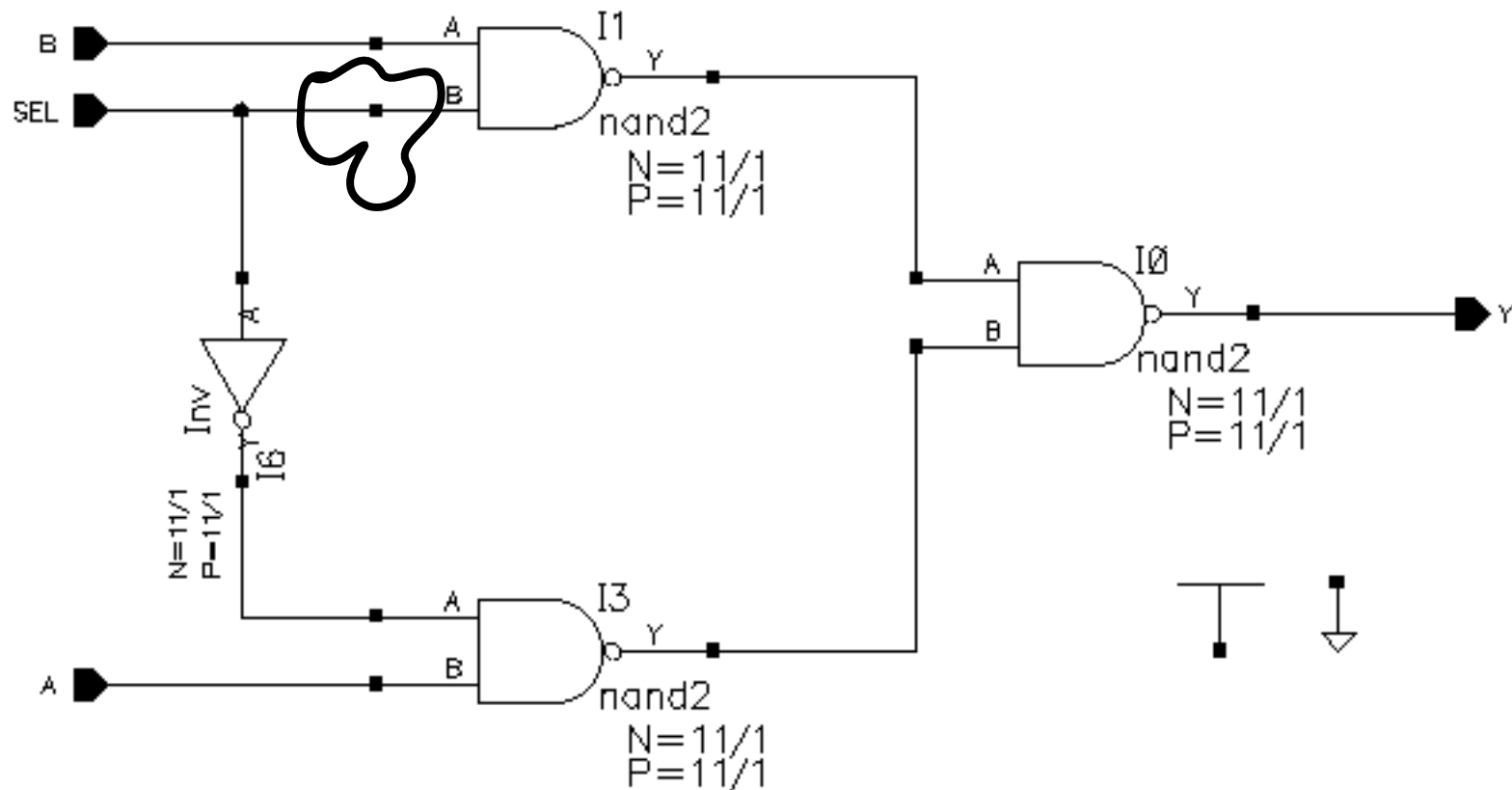
See the Cadence online documentation to read about this function.

Instance Terminals

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

- Each instance contains one instance terminal at least.
- Each net connects to one instance terminal at least.

Example: The *B* instance terminal on the *I1* instance.



Instance Terminals

The *instTerm* Object Type

You can retrieve the following data:

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

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

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

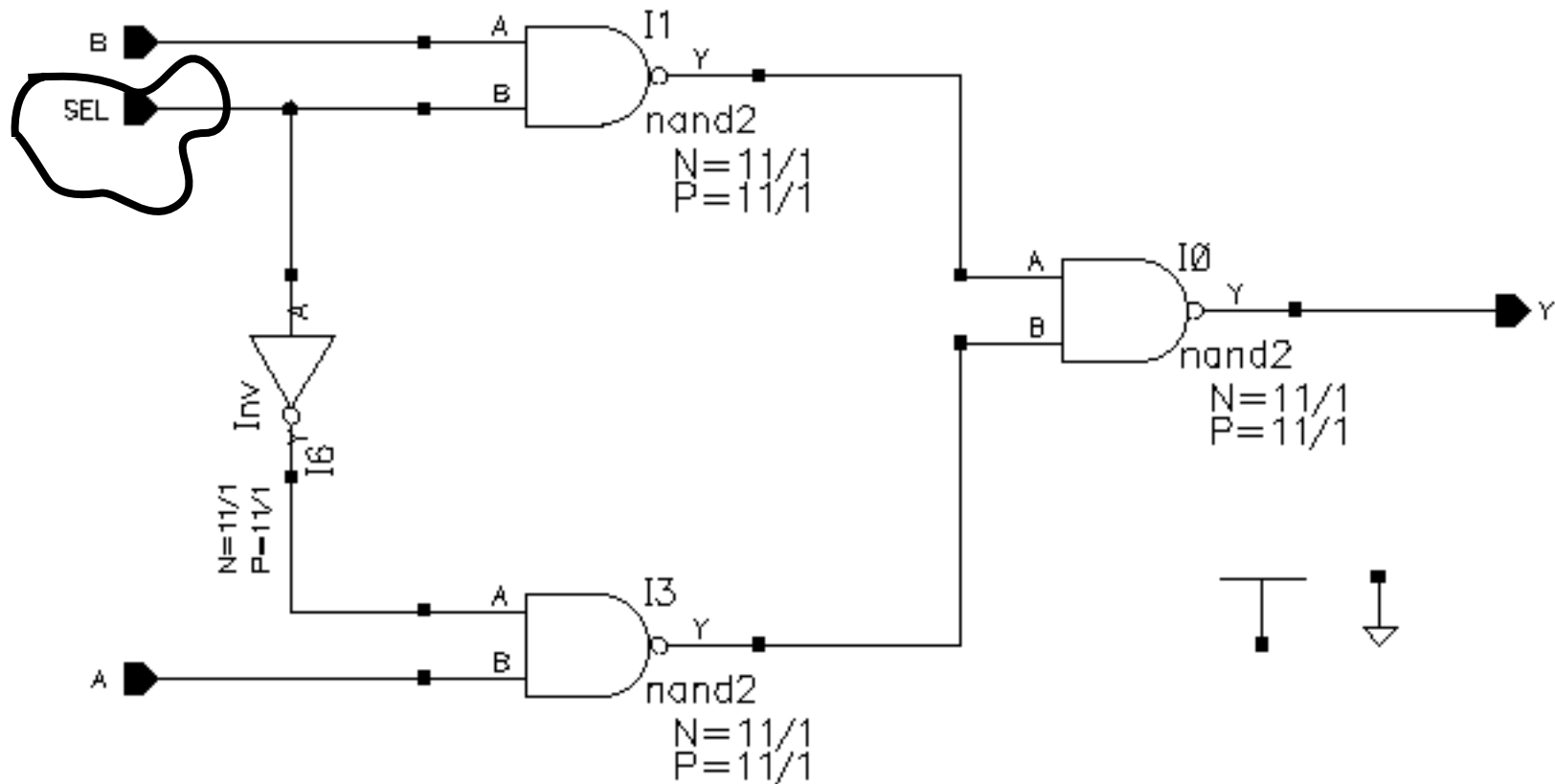
```
dbFindAnyInstByName( mux2 "I1" )~>instTerms~>net~>name =>  
  ( "SEL" "net4" "B" )
```

The *instTerm* Object Type

Terminals

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

Internal nets do not have terminals.



Terminals

The *term* Object Type

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

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

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

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

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

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

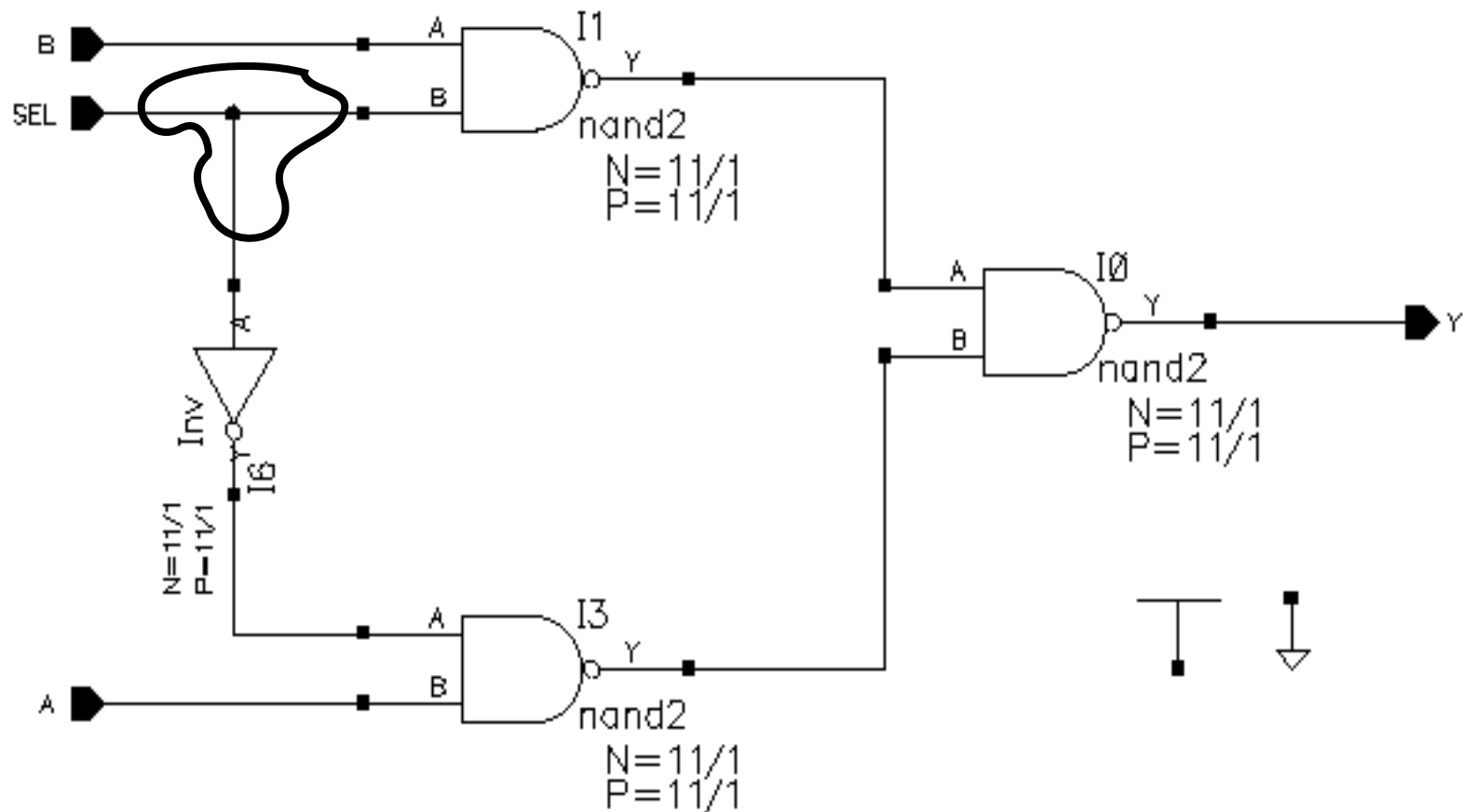
The *term* Object Type

The *dbFindTermByName* Function

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

See the Cadence online documentation to read about this function.

Figures and Shapes



- *geGetSel/Set* - function used to retrieve the currently selected set
- *geSelectFig* - adds an object to the selected set if it passes the selection filter.

Figures and Shapes

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

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

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

The *geGetSelSet* Function

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

The *geSelectFig* Function

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

Shape Attributes

All shapes have several common attributes.

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

Shape Attributes

Lab Overview

Lab 5-1 Querying Design Databases

In this lab, you open these designs:

- master mux2 schematic
- master mux2 layout
- master mux2 extracted
- master mux2 symbol

You enter SKILL expressions to answer the following questions:

- How many nets are in the design?
- What are the net names?
- Are there any instances in the design? If so, how many?
- Are there any shapes in the design? If so, how many?

Lab Overview

Module Summary

In this module, we covered

- Database object concepts
- Several specific object types
 - the *cellView* object type
 - the *inst* object type
 - the various shape object types
 - the net object type
- The *geGetWindowCellView* function
- The *geGetSel/Set* function
- Using the *~>* operator to retrieve attributes and user-defined properties.

Module Summary

Module 6: Menus

Objectives

- Create and display pop-up menus.
- Create pull-down menus.
- Install and remove pull-down menus from window banners.

Terms and Definitions

Callback

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

Creating a Pop-Up Menu

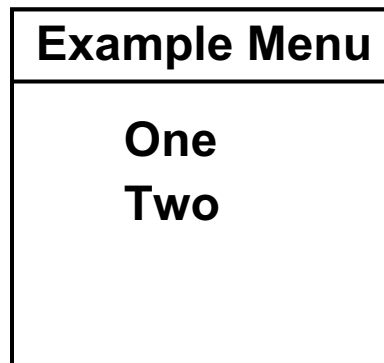
Use the *hiCreateSimpleMenu* function to build a pop-up menu with choices.

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

```
hiCreateSimpleMenu(  
  'TrExampleMenu  
  "Example Menu"  
  ' ( "One" "Two" )  
  ' ( "println( \"One\" )" "println( \"Two\" )" )  
) => array[7]:27015368
```

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

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



Creating a Pop-Up Menu

The *hiCreateSimpleMenu* Function

Example Arguments	Meaning
<i>'TrExampleMenu</i>	The menu variable. <i>hiCreateSimpleMenu</i> stores the data structure of the menu in this variable. <i>TrExampleMenu => array[7]:27015368</i>
<i>"Example Menu"</i>	The title of the menu.
<i>'("One" "Two")</i>	The list of the choices. No repetitions.
<i>'("println(\"One\")" "println(\"Two\")")</i>	The list of callbacks. Each callback is a string representing a single expression. Use \" to embed a single quote. Use curly braces, { }, to group multiple expressions into a single expression.

Displaying a Pop-Up Menu

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

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

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

```
hiSetBindKey( "Schematics"  
  "Shift Ctrl<Btn2Down>(2)"  
  "hiDisplayMenu( TrExampleMenu )" )
```



hiDisplayMenu

Displaying a Pop-Up Menu

The *hiDisplayMenu* Function

Example Argument	Meaning
<i>TrExampleMenu</i>	Use the variable you passed to <i>hiCreateSimpleMenu</i> . Do not apply the ' operator to the <i>TrExampleMenu</i> variable because you want to refer to the value of the variable.

Creating a Pull-Down Menu

General characteristics of a pull-down menu include the following:

- You can insert a pull-down menu in one or more menu banners or in slider menus.
- These menus can contain textual, iconic, or slider items.
- All items in pull-down menus are active.

Use the *hiCreatePulldownMenu* function to create a pull-down menu.

Use the *hiCreateMenuItem* function to create the menu items.

```
TrMenuItemOne = hiCreateMenuItem(  
    ?name          'TrMenuItemOne  
    ?itemText      "One"  
    ?callback      "println( \"One\" )" )
```

Pass the list of menu items to the *hiCreatePulldownMenu* function.

```
hiCreatePulldownMenu(  
    'TrPulldownMenu      ;;; the menu variable  
    "Example Menu"       ;;; menu title  
    list( TrMenuItemOne TrMenuItemTwo )  
)
```

Creating a Pull-Down Menu

The *hiCreateMenuItem* Function

Formal Argument	Actual Argument	Meaning
<i>?name</i>	<i>'TrMenuItemOne</i>	The symbolic name of the item. Quote this variable. Make sure that you also store the return result in this variable.
<i>?itemText</i>	<i>"One"</i>	The text that appears on the menu.
<i>?callback</i>	<i>"println(\"One\")"</i>	The action triggered by this menu item.

The *hiCreatePulldownMenu* Function

Example Argument	Meaning
<i>'TrPulldownMenu</i>	The menu variable. Quote this variable. The function stores the pull-down data structure menu in this variable.
<i>"Example Menu"</i>	The menu title.
<i>list(TrMenuItemOne TrMenuItemTwo)</i>	The list of menu items.

Inserting a Pull-Down Menu

Assume that a single window is already open.

Use the *hiInsertBannerMenu* function to insert a pull-down menu in the window menu banner.

The following line inserts a pull-down menu in the leftmost position of the CIW:

```
hiInsertBannerMenu( window( 1 ) TrPulldownMenu 0 )
```

If you attempt to insert a pull-down menu that is already present in a window banner, the following actions occur:

- You get a warning message.
- Your request is ignored.

To replace a pull-down menu, first delete the menu you want to replace.

Inserting a Pull-Down Menu

The *hiInsertBannerMenu* Function

Example Argument	Meaning
<i>window(1)</i>	The window ID
<i>TrPulldownMenu</i>	The data structure for the pull-down menu. Use the variable you passed to <i>hiCreatePulldownMenu</i> . Do not apply the ' operator to the <i>hiCreatePulldownMenu</i> variable because you want to refer to its contents.
<i>0</i>	The index of the menu after it has been inserted. 0 is the leftmost.

The *hiInsertBannerMenu* function only works on a single window.

Deleting a Pull-Down Menu

Use the *hiDeleteBannerMenu* function to remove a pull-down menu from the window banner.

The pull-down menus on the right move one position to the left.

This example removes the leftmost menu in the CIW.

```
hiDeleteBannerMenu( window( 1 ) 0 )
```

Notice that the *hiDeleteBannerMenu* function requires the index of the pull-down menu. You can easily determine the index by counting.

Deleting a Pull-Down Menu

The *hiDeleteBannerMenu* Function

Example Argument	Meaning
<i>window(1)</i>	The window ID.
<i>0</i>	The index of the menu. 0 is the leftmost.

Lab Overview

Lab 6-1 Exploring Menus

- Exploring Pop-Up Menus
- Exploring Pull-Down Menus

Lab Overview

Module Summary

In this module, we covered

- Creating and displaying a pop-up menu.
- Creating a pull-down menu.
- Installing a pull-down menu in a window banner.
- Deleting a pull-down menu from a window banner.

Module Summary

Function	Description
<i>hiCreateSimpleMenu</i>	Builds a pop-up menu with text choices only.
<i>hiCreateMenu</i>	Builds a pop-up menu text or icon choices.
<i>hiDisplayMenu</i>	Displays a pop-up menu built by <i>hiCreateSimpleMenu</i> or <i>hiCreateMenu</i> .
<i>hiCreateMenuItem</i>	Builds a menu item. Several menus can share this item simultaneously.
<i>hiCreatePulldownMenu</i>	Builds a pull-down menu with text or icon choices.
<i>hiInsertBannerMenu</i>	Installs a pull-down menu in a window banner.
<i>hiDeleteBannerMenu</i>	Removes a pull-down menu from the banner.

Module 7: Customization

Objectives

- Provide an overview of the Design Framework II Initialization sequence.
- Basic tasks to accomplish in your *.cdsinit* file.
- Survey several useful SKILL functions.
- Optional tasks to accomplish in your *.cdsinit* file.
- Preview the laboratory exercise.

Terms and Definitions

Context

A SKILL context is a binary file containing compiled SKILL code. Each SKILL application is associated with one or more contexts.

Design Framework II Initialization Sequence

Cadence partitions applications into one or more SKILL contexts. A SKILL context is a binary file containing compiled SKILL code and SKILL data structures.

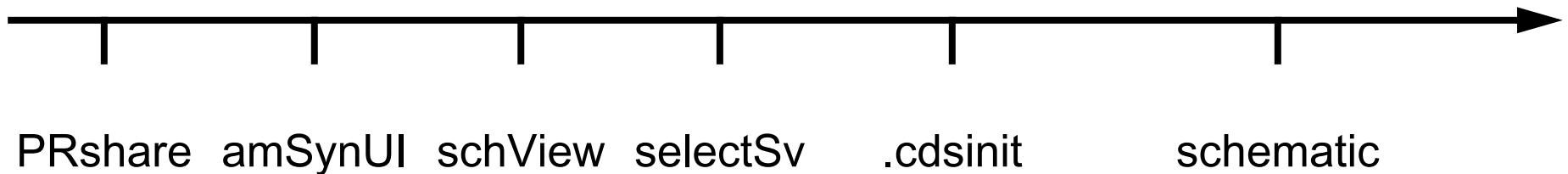
When you start the Design Framework II environment, it loads these items:

1. Loads one or more SKILL contexts.
2. Loads all *.cdsenv* files.
3. Loads the *.cdsinit* customization file.

The search order for the *.cdsinit* file is <install_dir>/tools/dfII/local, the current directory, and the home directory.

Once the environment finds a *.cdsinit* file, the search stops. However, a *.cdsinit* can load other *.cdsinit* files.

4. Responds to user activity by loading other SKILL contexts.



Design Framework II Initialization Sequence

The context files are in the `<install_dir>/tools.dfII/etc/context` directory.

Each executable loads different contexts.

- Study the log file to determine the contexts that your executable loads prior to the *.cdsinit* file.
- The *icfb* executable loads these contexts, as the following excerpt for the *CDS.logs* indicates.

```
Loading PRshare.cxt
Loading amSynUI.cxt
Loading schView.cxt
Loading selectSv.cxt
```


The *.cdsinit* File

The site administrator has three strategies to control the user customization.

Policy	Customization Strategy
1. The site administrator does all the customization.	The <code><install_dir>/tools/dfll/local/.cdsinit</code> contains all customization commands. There are no <i>./cdsinit</i> or <i>~/.cdsinit</i> files involved.
2. The site administrator does some customization. Each user can customize further.	The <code><install_dir>/tools/dfll/local/.cdsinit</code> file contains a command to load the <i>./cdsinit</i> or <i>~/.cdsinit</i> files
3. Each user does all his/her own customization.	The <code><install_dir>/tools/dfll/local/.cdsinit</code> file does not exist. All customization handled by either the <i>./cdsinit</i> or <i>~/.cdsinit</i> files.

Study the `<install_dir>/tools/dfll/cdsuser/.cdsinit` file for sample user customizations.

The *.cdsinit* File

The Basic *.cdsinit* Tasks

Make the *.cdsinit* file accomplish these three tasks:

- Set the SKILL path.
The SKILL path is a list of directories containing SKILL source code and data files. The *load*, *loadi*, *view*, and other file system interface functions refer to the SKILL path to resolve relative pathnames.
- Load your SKILL utilities.
Use the *load* function or the *loadi* functions.
- Define application bindkeys.
Use the either the *hiSetBindKey* function or the *hiSetBindKeys* functions.

For maximum flexibility, implement your *.cdsinit* file so that another *.cdsinit* file can load it as in these examples:

- Another *.cdsinit* file can set the SKILL path before loading your *.cdsinit* file.
- Set the SKILL path to the current SKILL path plus the directories that contain your SKILL source code files.

The Basic *.cdsinit* Tasks

The SKILL Path

The *getSkillPath* Function

The *getSkillPath* function returns a list of directory pathnames in search order.

The *setSkillPath* Function

The *setSkillPath* function sets the path to a list of directory pathnames.

Loading SKILL Source Code

Use the *load* or *loadi* functions to load SKILL source code.

- The *load* function returns *t* if all SKILL expressions run without errors. Any error causes the *load* function to abort.
- The *loadi* function executes every SKILL expression in the file, regardless of errors.

Example

Load standard Composer bindkey definitions.

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

Loading SKILL Source Code

The *load* Function

The *loadi* Function

Standard Bindkey Definitions

These files contain standard bindkey definitions for the Composer and Virtuoso software.

Application	Pathname
Composer Schematic	<install_dir>/tools/dfII/samples/local/schBindKeys.il
Virtuoso Layout	<install_dir>/tools/dfII/samples/local/leBindKeys.il

Other Useful *.cdsinit* Functions

SKILL functions can answer the following questions:

- What is the installation path?
- What is the value of a particular shell environment variable?
- Does a particular file exist?
- What are all the files in a directory?

Use the Cadence online documentation to study the functions on the next few pages.

Other Useful *.cdsinit* Functions

Determining the Installation Path

Use the *prependInstallPath* function to make a pathname relative to the Design Framework II installation directory. The function prepends *<install_dir>/tools/dfll* to the path name.

Example

This example code adds three directories to the front of the current SKILL path. The three directories are in the installation hierarchy.

```
TrSamplesPath = list(  
    prependInstallPath( "etc/context" )  
    prependInstallPath( "local" )  
    prependInstallPath( "samples/local" )  
)  
  
setSkillPath(          ;;; requires a list of strings  
    append(  
        TrSamplesPath ;;; a list of strings  
        getSkillPath() ;;; the current SKILL path  
    ) ; append  
    ) ; setSkillPath
```

Determining the Installation Path

The *prependInstallPath* Function

Interfacing with UNIX Files and Directories

Use the *simplifyFilename* function to determine the full pathname to a file. It correctly deals with the following items:

- the user's home directory
- the current working directory
- the SKILL path

Example

```
simplifyFilename( "~/SKILL" ) => "/usr1/mnt/skilldev/SKILL"  
simplifyFilename( "../cdsinit" )=>  
    "/usr1/mnt/skilldev/SKILL/.cdsinit"  
simplifyFilename( "Menus" ) =>  
    "/usr1/mnt/skilldev/SKILL/Menus"
```

Use the *getDirFiles* function to retrieve the files and subdirectories in a directory.

Example

```
getDirFiles( "Menus" ) =>  
    ( "." ".." "Solutions" "PullDown.il" "PopUp.il"  
      "MenuItemUtilities.il" "SimpleMenuExample.il" )
```

Interfacing with UNIX Files and Directories

The *simplifyFilename* Function

The *simplifyFilename* function expands the specified path to a full path. The tilde, /, and ./ are resolved, and redundant backslashes are removed. This shows exactly where the files are accessed for reporting information or error reporting by functions that require paths.

The *isFile* Function

The *isFile* function returns *t* if the file exists and returns *nil* otherwise. Use the *isFile* function to test for the existence of the file in the SKILL search path before attempting to load the file.

```
when( isFile( "SomeUtilities.il" )  
      loadi( "SomeUtilities.il" ) ) ; when
```

The *when* Function

The *when* function evaluates the first expression to determine whether or not to evaluate the subsequent expressions in its body.

The *isDir* Function

The *isDir* function returns *t* if the directory exists and returns *nil* otherwise.

The *getDirFiles* Function

The *getDirFiles* function returns the list of files in the directory or subdirectory.

Reading Shell Environment Variables

Use the *getShellEnvVar* function to determine the value of a shell environment variable.

For example, the following line returns the value of the USER variable.

```
getShellEnvVar( "USER" )=> "user1"
```

Reading Shell Environment Variables

The *getShellEnvVar* Function

Manipulating a List of Strings

Use the *buildString* function to make a single string from a list of strings. The *buildString* function provides separating space by default.

Example

```
buildString( '( "a" "b" "c" "d" ) ) => "a b c d"  
buildString( '( "a" "b" "c" "d" ) "/" ) => "a/b/c/d"
```

Use the *parseString* function to parse a single string into a substring based on delimiter characters.

Example

```
parseString( "a b c d" ) => ( "a" "b" "c" "d" )  
parseString( "a/b/c/d" "/" ) => ( "a" "b" "c" "d" )
```

Manipulating a List of Strings

The *buildString* Function

The *parseString* Function

Manipulating the CIW in the *.cdsinit* File

You can accomplish the following optional tasks in your *.cdsinit* file:

- Positioning the CIW.

```
hiResizeWindow(window(1) list( 10:10 750:200))
```

- Adding pull-down menus to the CIW. Consider the following example from the previous module.

```
hiInsertBannerMenu( window( 1 ) TrPulldownMenu 0 )
```

- Removing pull-down menus from the CIW menu banner. Consider the following example from the previous module.

```
hiDeleteBannerMenu( window( 1 ) 0 )
```

Manipulating the CIW in the *.cdsinit* File

SKILL Development *.cdsinit* Tasks

SKILL programmers can accomplish the following optional tasks in the *.cdsinit* file:

- Clear SKILL function write protection.
You can redefine your SKILL functions during the session.

```
sstatus( writeProtect nil )
```

- Set the log filter options.
The CIW will display the maximum amount of SKILL information.

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

- Install the SKILL debugger.
The SKILL debugger will be available to trap errors.

```
installDebugger()  
alias( q debugQuit )  
alias( c continue )
```

SKILL Development *.cdsinit* Tasks

Lab Preview

In the lab, you make several bindkey definitions immediately available when the Design Framework II environment starts.

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

You define a bindkey for the *Command Interpreter* application to raise the current window.

With a single key stroke, you can toggle back and forth between the current window and the CIW. This feature has the following benefits:

- You can keep the CIW large without losing track of your current window.
- You can bury the CIW until you need it.

Lab Preview

Lab Overview

Lab 7-1 Setting Bindkeys in the *.cdsinit* File

Lab Overview

Module Summary

In this module, we covered

- The Design Framework II Initialization sequence.
- Basic tasks to accomplish in your *.cdsinit* file.
- Several useful SKILL functions.
- Optional tasks to accomplish in your *.cdsinit* file.
- A preview of the laboratory exercise.

Module Summary

Module 8: Developing a SKILL Function

Objectives

- Grouping several SKILL expressions into a single SKILL expression.
- Declaring local variables.
- Declaring a SKILL function.
- Understanding the SKILL software development cycle.
 - Loading your SKILL source code.
 - Redefining a SKILL function.

Terms and Definitions

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

Grouping SKILL Expressions Together

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

Use the curly braces, { }, to group a collection of SKILL statements into a single SKILL statement.

The return value of the single statement is the return value of the last SKILL statement in the group. You can assign this return value to a variable.

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

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

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

Grouping SKILL Expressions Together

Curly Braces

The following statements refer to the example above:

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

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

Grouping Expressions with Local Variables

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

- Include a list of the local variables followed by one or more SKILL expressions.
- These variables will be initialized to *nil*.

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

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

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

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

Grouping Expressions with Local Variables

The *let* Function

You can freely nest *let* statements.

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

- The current value of a variable is simply the top of the stack.
- Assigning a value to a variable changes only the top of the stack.

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

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

Two Common *let* Errors

The two most common *let* errors are:

- Including whitespace after *let*.

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

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

- Omitting the list of local variables.

The error messages vary and can be obscure and hard to decipher.

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

Two Common *let* Errors

Defining SKILL Functions

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

- The name is known as the function name.
- The group of statements is the function body.

Example

```
bBox = list( 100:200 350:450 )
```

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

```
bBoxHeight = TrBBoxHeight()
```

Write global variable

Define function

Local variables

Read global variable

Read global variable

Return value

Invoke function

Defining SKILL Functions

The *procedure* Function

Local Variables

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

Global Variable

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

Return Value

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

Three Common *procedure* Errors

The three most common *procedure* errors are:

- Including whitespace after the *procedure* function.

```
procedure ( TrBBoxHeight( )
```

```
    ...
```

```
) ; procedure
```

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

- Including whitespace after your function name.

```
procedure( TrBBoxHeight ( )
```

```
    ...
```

```
) ; procedure
```

```
*Error* procedure: illegal formal list - TrBBoxHeight
```

- Omitting the argument list.

```
procedure( TrBBoxHeight
```

```
    ...
```

```
) ; procedure
```

```
*Error* procedure: illegal formal list - TrBBoxHeight
```

Notice that the error message refers to the *procedure* function.

The arguments to *procedure* are the function name, the argument list of the function, and the expression composing the body of the function.

Three Common *procedure* Errors

- Including White Space After the *procedure* Function
- Including White Space After the Function Name
- Omitting the Argument List

Defining Required Function Parameters

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

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

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

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

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

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

Defining Required Function Parameters

You can use the *let* function to declare the variables *ll*, *ur*, *ury*, and *lly* to be local variables. In the example, the variable *bBox* occurs both as a global variable and as a formal parameter. Here's how SKILL keeps track. When you call the *TrBBoxHeight* function, the SKILL Evaluator:

- Saves the current value of *bBox*.
- Evaluates *list(50:150 200:300)* and temporarily assigns it to *bBox*.
- Restores the saved value of *bBox* when the *TrBBoxHeight* function returns.

Defining Optional Function Parameters

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

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

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

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

- *TrOffsetBBox(someBBox)*
dx and *dy* default to 0.
- *TrOffsetBBox(someBBox 0.5)*
dy defaults to 0.
- *TrOffsetBBox(someBBox 0.5 0.3)*.

Defining Optional Function Parameters

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

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

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

Defining Keyword Function Parameters

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

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

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

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

- *TrOffsetBBox(someBBox ?dx 0.5 ?dy 0.4)*
- *TrOffsetBBox(someBBox ?dx 0.5)*
dy defaults to 0.
- *TrOffsetBBox(someBBox ?dy 0.4)*
dx defaults to 0.

Defining Keyword Function Parameters

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

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

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

Collecting Function Parameters into a List

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

Example

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

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

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

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

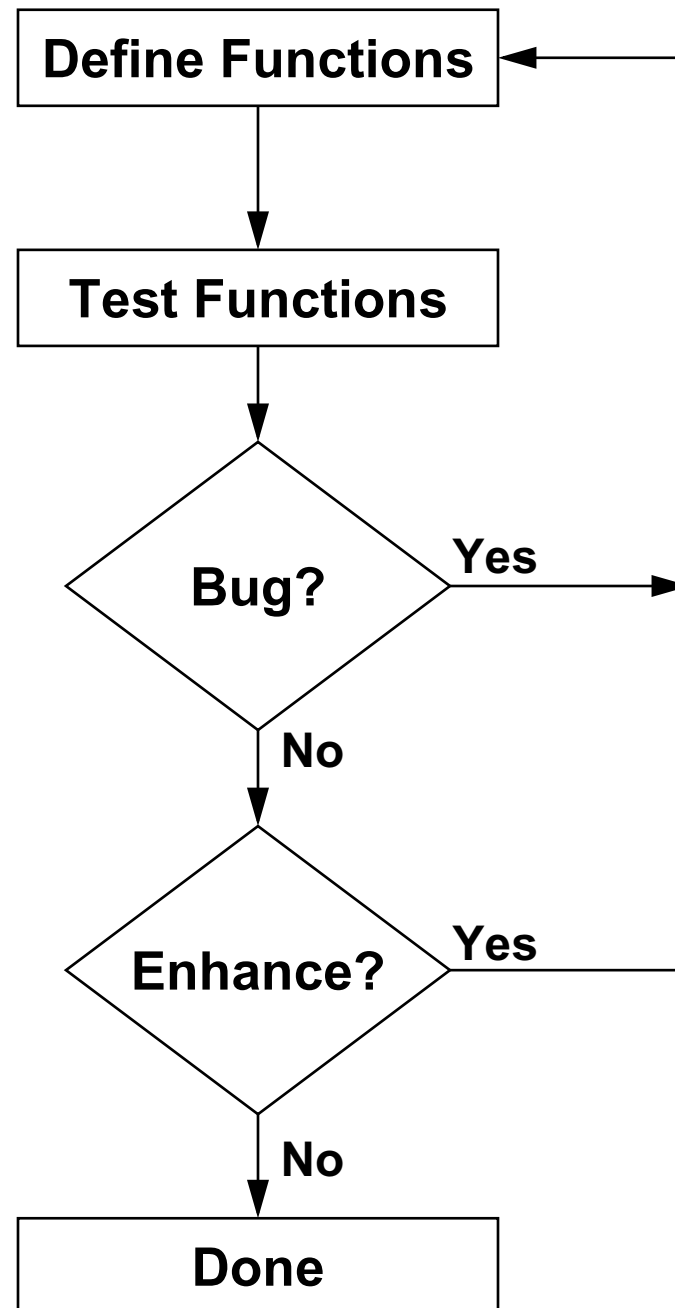
- `TrCreatePath(3:0 0:4)`
- `TrCreatePath(3:0 0:4 0:0)`

Collecting Function Parameters into a List

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

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

SKILL Development Cycle



SKILL Development Cycle

SKILL allows you to redefine functions without exiting the Design Framework II environment. A rapid edit and run cycle facilitates bottom-up software development.

Defining SKILL Functions

Launching an Editor from the Design Framework II Environment

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

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

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

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

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

```
edit( TrExampleFunction )
```

Testing SKILL Functions

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

Loading Source Code

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

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

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

- Syntax problems with a *procedure* definition.
- Attempts to *load* a file that does not exist.

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

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

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

Loading Source Code

The *loadi* Function

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

The SKILL Path Functions

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

Pasting Source Code into the CIW

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

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

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

1. Use the mouse in your editor to select the source code.
2. Move the cursor over the CIW input pane.
3. Click the middle mouse button.
Your selection is displayed in the CIW input pane in a single line.
4. Press Return.
Your entire selection is displayed in the CIW output pane.

Pasting Source Code into the CIW

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

```
:set nonumber
```

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

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

Why is that?

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

Redefining a SKILL Function

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

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

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

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

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

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

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

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

Redefining a SKILL Function

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

Lab Overview

Lab 8-1 Developing a SKILL Function

- Develop a SKILL function, *TrBBoxArea*, to compute the area of a bounding box. The bounding box is a parameter.
- Model your SKILL function after the example SKILL function, *TrBBoxHeight*.

Lab Overview

Module Summary

In this module, we covered

- Using curly braces, { }, to group SKILL statements together
- Using the *let* function to declare local variables
- Declaring SKILL functions with the procedure function
 - Name
 - Arguments
 - Body
- The SKILL development cycle
- Maintaining SKILL code
 - The *edit* function and *editor* variable
 - The *load* function
 - The *writeProtect* switch
- The lab exercise

Module Summary

Module 9: Flow of Control

Objectives

- Review relational operators
- Describe logical operators
- Examine branching statements
- Discuss several methods of iteration

Terms and Definitions

Iteration To repeatedly execute a collection of SKILL expressions.

Relational Operators

Use the following operators to compare data values.

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

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

Relational Operators

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

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

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

```
1 > "abc"
```

```
*** Error in routine greaterp:
```

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

Logical Operators

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

SKILL provides generalized boolean operators.

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

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

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

Logical Operators

The && Operator

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

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

The // Operator

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

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

Using the && and || Operators to Control Flow

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

Example of Using the || Operator

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

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

Using the `&&` and `//` Operators to Control Flow

Branching

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

Branching

The *if* Function

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

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

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

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

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

The *if* Function

Two Common *if* Errors

Two common *if* errors are:

- Including whitespace after *if*

```
if ( shapeType == "rect"
    ...
)
```

*** Error in routine if
Message: *Error* if: too few arguments ...

- Including a right parenthesis after the conditional expression

```
if( shapeType == "rect" )
```

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

Two Common *if* Errors

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

Remember

- To avoid whitespace immediately after the *if* syntax function.
- To place parentheses as follows:
if(... then ... else ...)

Nested *if-then-else* Expressions

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

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

Nested *if-then-else* Expressions

The *when* and *unless* Functions

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

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

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

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

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

The *when* and *unless* Functions

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

The case Function

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

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

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

Candidate value
Target

Target

Target

Catch all

The *case* Function

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

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

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

The *cond* Function

Use the *cond* function when your logic involves multiway branching.

```
cond(  
  ( condition1 exp11 exp12 ... )  
  ( condition2 exp21 exp22 ... )  
  ( condition3 exp31 exp32 ... )  
  ( t expN1 expN2 ... )  
) ; cond
```

The *cond* function

- Sequentially evaluates the conditions in each arm, until it finds one that is non-*nil*. It then executes all the expressions in the arm and exits.
- Returns the last value computed in the arm it executes.

The *cond* function is equivalent to

if	condition1	then	exp11exp12 ...
else if	condition2	then	exp21exp22 ...
else if	condition3	then	exp31exp32 ...
...			
else			expN1 expN2

The *cond* Function

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

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

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

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

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

Iteration

Iteration task	Function
Numeric range	for
List of values	foreach
While a condition is non- <i>nil</i>	while

Iteration

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

The *for* Function

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

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

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

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

- Saves the current value of the index variable before evaluating the loop expressions.
- Restores the index variable to its saved value after exiting the loop.
- Returns the value *t*.

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

- the placement of the parentheses: *for(...)*.
- to avoid putting whitespace immediately after the *for* syntax function.

The *for* Function

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

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

The *foreach* Function

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

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

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

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

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

The *foreach* expression returns the list of values over which it iterates.

The *foreach* Function

In the example,

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

The *while* Function

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

Example of a *while* expression.

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

The *while* Function

The *prog* and *return* Functions

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

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

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

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

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

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

The *prog* and *return* Functions

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

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

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

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

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

- The current value of a variable is simply the top of the stack.
- Assigning a value to a variable changes only the top of the stack.

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

When the flow of control exits, the system pops the temporary value off the value stack, restoring the previous value.

Lab Overview

Lab 9-1 Writing a Database Report Program

You write a SKILL function to count the shapes in a design.

You enhance this program in subsequent labs.

Lab 9-2 Exploring Flow of Control

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

Lab 9-3 More Flow of Control

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

Lab 9-4 Controlling Complex Flow

You write a SKILL function to validate a bounding box.

Lab Overview

Module Summary

In this module, we covered

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

Module Summary

Module 10: File I/O

Objectives

- Write UNIX text files.
- Read UNIX text files.
- Open a text window.

Terms and Definitions

Writing Data to a File

Instead of displaying data in the CIW, you can write the data to a file.

Both *print* and *println* accept a second, optional argument that must be an output port associated with the target file.

Use the *outfile* function to obtain an output port for a file. Once you are finished writing data to the file, use the *close* function to release the port.

This example uses the *for* function to execute the *println* function iteratively. The *i* variable is successively set to the values 1,2,3,4, and 5.

```
myPort = outfile( "/tmp/myFile" ) => port:"/tmp/myFile"
for( i 1 5
  println( list( "Number:" i) myPort )
) => t
close( myPort ) => t
myPort = nil
```

After you execute the *close* function, the file */tmp/myFile* contains the following lines:

```
( "Number:" 1)
( "Number:" 2)
( "Number:" 3)
( "Number:" 4)
( "Number:" 5)
```

Writing Data to a File

The *print* and *println* Functions

Notice how SKILL displays a port in the CIW.

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

Use a full pathname with the *outfile* function. Keep in mind that *outfile* returns *nil* if you do not have write access to the file, or if you cannot create the file in the directory you specified in the pathname.

The *print* and *println* functions raise an error if the port argument is *nil*. Observe that the type template uses a *p* character to indicate a port is expected.

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

The *close* Function

The *close* function does not update your port variable after it closes the file. To facilitate robust program logic, set your port variable to *nil* after calling the *close* function.

Writing Data to a File (continued)

Unlike the *print* and *println* functions, the *printf* function does not accept an optional port argument.

Use the *fprintf* function to write formatted data to a file. The first argument must be an output port associated with the file.

Example

```
myPort = outfile( "/tmp/myFile" )  
for( i 1 5  
    fprintf( myPort "\nNumber: %d" i )  
    ) ; for  
close( myPort )
```

The example above writes the following data to */tmp/myFile*:

```
Number: 1  
Number: 2  
Number: 3  
Number: 4  
Number: 5
```

Writing Data to a File (continued)

The *fprintf* Function

Reading Data from a File

Use the *infile* function to obtain an input port on a file.

The *gets* function reads the next line from the file.

This example prints every line in *~/.cshrc* to the CIW.

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

Reading Data from a File

The *infile* Function

The *gets* Function

The *gets* function reads the next line from the file. The arguments of the *gets* function are the:

- variable that receives the next line.
- input port.

The *gets* function returns the text string or returns *nil* when the end of file is reached.

The *when* Function

The first expression within a *when* expression is a condition. SKILL evaluates the condition. If it evaluates to a non-*nil* value, then SKILL evaluates all the other expressions in the *when* body. The *when* function returns either *nil* or the value of the last expression in the body.

The *while* Function

SKILL repeatedly evaluates all the expressions in the *while* body as long as the condition evaluates to a non-*nil* value. The *while* function returns *nil*.

The *fscanf* Function

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

The arguments of *fscanf* are

- the input port
- the conversion control string
- the variable(s) that receive(s) the matching data values.

The *fscanf* function returns the number of data items matched.

This example prints every word in `~/.cshrc` to the CIW.

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

The *fscanf* Function

The format directives commonly found include the ones in this table.

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

The following is an example of output from the *~/CDS.log*.

```
\o "#.cshrc"  
\o "for"  
\o "Solaris"  
\o "#Cadence"  
\o "Training"  
\o "Database"  
\o "setup"  
\o "for"  
\o "the"  
\o "97A"  
\o "release"  
\o "#"
```


Opening a Text Window

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

```
view( "~/SKILL/.cdsinit" ) => window:5
```

The *view* function is very useful for displaying a report file to the user.

The *view* function resolves a relative pathname in terms of the SKILL path. This is a list of directories you can establish in your *.cdsinit*.

See Module 7, *Customization*, for specifics on the SKILL path.

To select text from your *CDS.log* file, try the following:

```
view(  
    "~/CDS.log"   ;;; pathname to CDS.log  
    nil           ;;; default location  
    "Log File"    ;;; window title  
    t             ;;; auto update  
) => window:6
```

Opening a Text Window

The *view* Function

The *view* function takes several optional arguments.

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

Lab Overview

Lab 10-1 Writing Data to a File

Lab 10-2 Reading Data from a Text File

Lab 10-3 Writing Output to a File

Enhance your *TrShapeReport* function to write the output to a file.

Lab Overview

Module Summary

In this module, we covered

- Writing text data to a file by using
 - the *outfile* function to obtain an output port on a file.
 - an optional output port parameter to the *print* and *println* functions.
 - a required port parameter to the *fprintf* function.
 - the *close* function to close the output port.
- Reading a text file by using
 - the *infile* function to obtain an input port.
 - the *gets* function to read the file a line at a time.
 - the *fscanf* function to convert text fields upon input.
 - the *close* function to close the input port.

Module Summary

Module 11: Using the SKILL Development Environment

Objectives

- Discuss common bugs.
- Identify mistakes to avoid.
- Explore the most useful facilities of the SKILL debugger.
 - Single-step tracing
 - Function tracing
 - Trapping SKILL errors
 - Setting and clearing breakpoints
 - Single-step execution
- Learn about SKILL Lint.
- Learn about the SKILL Profiler.

Terms and Definitions

SKILL Development Tips

The Design Framework II environment includes the following:

- The capability to write your own SKILL programs and run the SKILL programs of others as well as your own.
- The SKILL documentation accessible through the Cadence online documentation.

In theory, this is all you need to develop SKILL programs.

SKILL Development Tips

Common Bugs

When a SKILL program does not behave according to its specifications, you need to determine the underlying cause and correct the program accordingly.

Consider a Shape Report Program with a bug in it. Here are some possible outcomes when you run the program.

- No Shape Report window appears.

SKILL displays the following error message:

```
*Error* postincrement: can't handle add1(nil)
```

- No Shape Report window appears.

SKILL displays the following error message:

```
*Error* fprintf/sprintf:  
format spec. incompatible with data - nil
```

- The Shape Report window appears. The *rect* counter is 0.
- The Shape Report window appears. All the counters are 0.

Common Bugs

The cause for each of the above problems are as follows:

1. The program incremented a variable that was initialized to *nil* instead of initialized to 0.
2. The program called the *printf* function to display the value of a variable when the value was *nil*.
3. The program used "*Rect*" instead of "*rect*" in as a target value in a *case* expression. Consequently, none of the rectangles were identified.
4. The program includes a case expression that used the *shape~>ObjType* instead of *shape~>objType* to retrieve the *objType* attribute. The program was not written to handle the value *nil*. None of the shapes were correctly identified.

Using Print Statements

Use *println* or *printf* statements to

- Verify the flow of control.
- Determine the values of function arguments and local variables.

Observe the following guidelines:

- Avoid inserting debugging print statements at the end of a function. Doing so can adversely affect the return of the function.
- Remove debugging print statements before distributing your program to others.
- Use the *println* function or the *%L* format with the *printf* function to avoid introducing errors.

Using Print Statements

The problems with relying on print statements include the following:

- You might not have access to the source code.
- The print statements that you insert might cause their own set of problems.
- You have to remember to remove such *println* and *printf* statements before distributing your program to others.

Checking for Common Errors

Check for the following mistakes:

- Make sure that the source code you examine matches the source code that you are running. You might have forgotten to load the latest version of one of your functions.
- Check all of the `~>` attributes in your program for spelling.
- Check the data types you pass to the `~>` operator.

The next several pages discuss these suggestions in greater detail.

Checking for Common Errors

Check the Running Source Code

Have you ever edited your program and forgotten to reload the new source code?

When debugging your function, use the *pp* function to verify that you have loaded your latest version of the function.

The *pp* function displays the definition of a SKILL function.

The *pp* function does not use the file system. Instead, it accesses the definition of the function in virtual memory.

Example

This example displays the definition of the *TrExample* function that is currently in memory.

```
procedure( TrExample( x y ) x+y )

pp( TrExample )
procedure(TrExample(x y)
  (x + y)
)
nil
```

Check the Running Source Code

The *pp* Function

The *pp* function only works on the SKILL functions that you define. You cannot use *pp* to inspect the following kinds of SKILL functions:

- Built-in SKILL functions defined in C
- Encrypted SKILL functions
- SKILL functions defined in a context

The *pp* function accepts an optional port argument. The following code declares the *TrExample* function and writes its definition to the */tmp/myfun.txt* file.

```
procedure( TrExample( x y )
  x+y
) ; procedure

let( ( FILE )
  FILE = outfile( "/tmp/myfun.txt" )
  pp( TrExample FILE )
  close( FILE )
) ; let
```

Check All Attribute and Property Names

Check the spelling of the database object attribute and window property names that you use in your program.

Remember that the `~>` operator returns *nil* when the object does not have a given attribute or property.

This situation has several symptoms:

- A *printf* error caused by *nil*.
- A *foreach* loop that does nothing.

Example

Can you spot the problem in the following fragment of code?

```
procedure( TrCountRectangles( cv )
  let( ( count )
    count = 0
    foreach( shape cv~>shaps
      ++count
    ) ; foreach
    count
  ) ; let
) ; procedure
```

Check All Attribute and Property Names

Verify ~> Data Types

The ~> operator works on many different kinds of objects. Different types of objects are likely to contain different attributes or properties.

Example

Can you spot the problem in the following fragment of code?

```
hiSetBindKey( "Layout" "<Key>F8"
  "println( TrCountRectangles( hiGetCurrentWindow() ))" )
procedure( TrCountRectangles( cv )
  let( ( count )
    count = 0
    foreach( shape cv~>shapes
      ++count
    ) ; foreach
    count
  ) ; let
) ; procedure
```

What do you need to do to fix the problem?

Verify ~> Data Types

In the example:

The input required for *TrCountRectangles* is the *databaseid* of a cell view.

The input passed to the function is a *windowid*.

Programming Defensively

Consider designing your code to make bugs easier to track down. Bugs that raise an error far away from the root cause are harder to track down.

For example,

- Declare the expected argument types of one or more of your functions.

You can either have SKILL perform the argument type checking or write your own code to check.

- Include assertions in your code that check for a condition that holds if things are in working order.

Write a function to check that a condition holds and raise an error if the condition does not hold.

The next several pages discuss these suggestions in greater detail.

Programming Defensively

Include Data Type Checks

You can have SKILL check the argument type by providing a type template when you declare the function. Using this technique you can find errors at their source.

The type template is a string. Each character of the string encodes the expected data type of the corresponding argument.

Follow these guidelines:

- Place the type template at the end of the argument list.
- Use the same character to encode a data type that the Cadence Finder uses. Here are some examples:

Character	Data Type	Comment
w	window ID	
d	data base object	
g	general	Indicates more than one data type works.
l	list	
x	integer	

Include Data Type Checks

Using type templates has the following limitations:

- You cannot control the wording of the error message.
- You can only check the SKILL data type of arguments. For example, you cannot check for database objects with a specific *objType* attribute value.
- You cannot specify that the SKILL data type be one of several possible types. You have to use the "g" character for this situation.
- You can use type templates only for required arguments.

An Example Type Template

The *TrCellNameMatch* function returns the *cellView* database object if the name contains *phrase*; otherwise it returns *nil*. The expected types of the two arguments are:

- *cv*, a database object
- *phrase*, a text string

The "*dt*" is called a type template. It encodes the expected argument types.

```
procedure( TrCellNameMatch( cv phrase "dt" )
  let( ((name cv~>cellName))
    when( rexMatchp( phrase name ) cv )
  ) ; let
) ; procedure
```

If you call *TrCellNameMatch* with a window ID instead of a database object, SKILL raises an error:

```
\i TrCellNameMatch( window(6) "nand2" )
\o *** Error in routine TrCellNameMatch:
\o Message: *Error* TrCellNameMatch:
      argument #1 should be dbobject (type template = "dt") - window:6
```

An Example Type Template

If you do not declare the *TrCellNameMatch* function with a type template, then the error message is more obscure since it comes from the *rexMatchp* function.

```
\i TrCellNameMatch( window(6) "nand2" )  
\o *** Error in routine rexMatchp:  
\o Message: *Error* rexMatchp:  
    argument #2 should be either  
    a string or a symbol (type template = "tS")
```

When you use type templates, you cannot control the wording of the error message.

Also, type templates only specify the SKILL type of arguments. You cannot use a type template to check that an argument is a database object with a specific *objType* attribute value.

The *error* Function

You can call the *error* function when your code determines that further evaluation is impossible or pointless.

Pass the following two arguments to the *error* function:

- A format string which contains the error message.
- Arguments to be substituted into the format string.

Example

The following version of the *TrCellNameMatch* function checks specifically that the first argument *obj* is a *cellView* database object.

```
procedure( TrCellNameMatch( obj phrase )
  cond(
    ( !dbobjectp( obj ) || obj~>objType != "cellView"
      error(
        " TrCellNameMatch: obj is not cellView - %L" obj )
      )
    ( ... )    ;;; Other checks
    ( t ... ) ;;; This alternative contains the rest of the function
  ) ; cond
) ; procedure
```

The *error* Function

Use the *%L* format string to specify the default print representation of the argument.

Development Tip Summary

The following techniques are often very effective for isolating or minimizing problems:

- Develop working programs incrementally, adding functionality in stages.
- Become familiar with common problems and their symptoms.
- Examine your source code for common problems.
- Document your source code. Often, as you annotate your code, you uncover problems.
- Use *println* or *printf* statements to clarify the flow of control or display the values of local variables.
- Program defensively to identify problems as soon as they occur.

Development Tip Summary

SKILL Development Environment

The SKILL Development Environment is a separate product from the Design Framework II environment. It contains several utilities to help you perform a variety of SKILL development tasks.

Utility	Task
SKILL Debugger	Trap errors.
	Suspend execution.
	Monitor execution.
SKILL Lint	Identify potential errors, oversights, and inefficient constructs.
SKILL Profiler	Measure performance and memory usage.

SKILL Development Environment

The SKILL Development Environment tools help you to

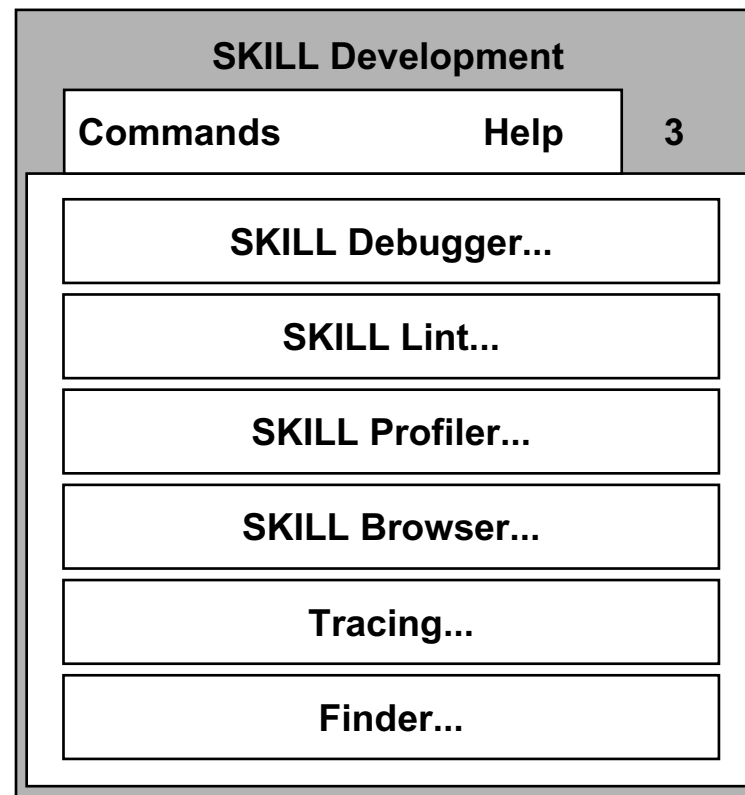
- Reduce the time it takes to develop your SKILL program.
- Improve the quality and efficiency of your SKILL program.

SKILL Development Window

You can access all of the capabilities of the SKILL Development Environment through SKILL functions.

Alternatively, the SKILL Development window is an interface to many of the facilities of the SKILL Development Environment.

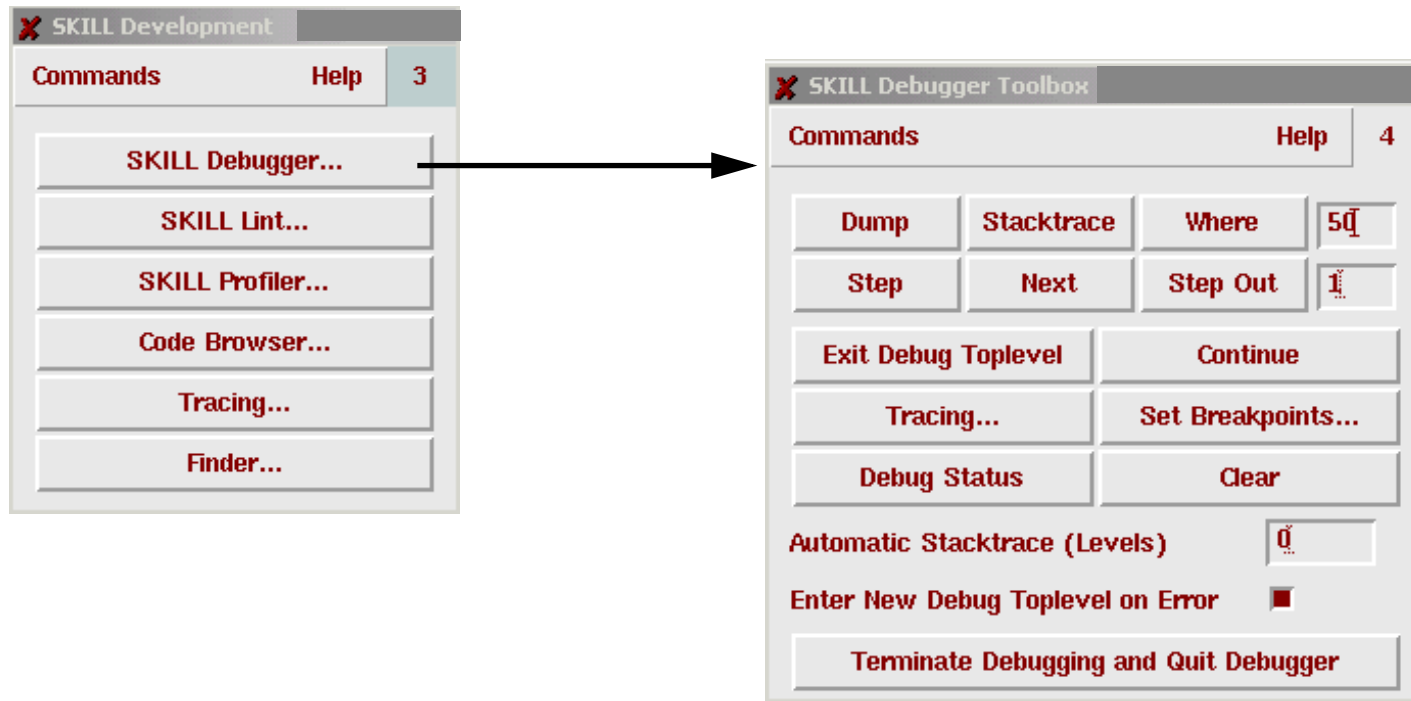
In the CIW, the **Open** → **SKILL Development** command opens the SKILL Development window.



SKILL Development Window

SKILL Debugger Toolbox

The SKILL Debugger button in the SKILL Development window opens the SKILL Debugger Toolbox.



The SKILL Debugger Toolbox provides a command button interface to most of the SKILL Debugger commands.

SKILL Debugger Toolbox

In your *.cdsinit* file, you can define a bindkey for the **F4** key to open the SKILL Debugger Toolbox.

```
hiSetBindKey( "Command Interpreter"  
  "<Key>F4" "ilDebugToolBox()" )
```

Using Single-Step Tracing to Analyze an Error

When your program causes a SKILL error, the following happens:

- Your program aborts before completing.
- The CIW displays an error message.

Often, single-step tracing provides enough information for you to determine the cause of the error.

To use single step tracing follow these steps:

1. In the CIW, turn on single-step tracing.

```
tracef( t )
```

2. Run your application.

The SKILL Debugger displays the arguments and the return value of each function call.

SKILL displays an error message.

3. In the CIW, turn off single-step tracing.

```
untrace( )
```

Using Single-Step Tracing to Analyze an Error

Single-step tracing can be very effective but it has the following drawbacks:

- Single-step tracing produces a lot of output. If the SKILL error does not occur early in the run, you can easily get confused or lost looking at the output.
- The output shows each function call with evaluated arguments. It can be hard to correlate the output with your source code.

The Tracing Form

The **Tracing** command is in both the SKILL Debugger Toolbox and the SKILL Development Toolbox. It brings up the Tracing form. You can use this form to control tracing instead of using the *tracef* and *untrace* functions.

A Single-Step Trace Example

Examine the lines preceding the error message for evidence.

```
\i TrShapeReport( window(3))
\o |window(3)
\o |window --> window:3
\o |TrShapeReport(window:3)
\o ||outfile("/tmp/ShapeReport.txt")
\o ||outfile --> port:"/tmp/ShapeReport.txt"
\o ||TriShapeReport(window:3 port:"/tmp/ShapeReport.txt")
\o |||(window:3~>libName)
\o |||getSGq --> nil
\o |||(window:3~>cellName)
\o |||getSGq --> nil
\o |||(window:3~>viewName)
\o |||getSGq --> nil
\o |||fprintf(port:"/tmp/ShapeReport.txt" "%s %s %s contains:" nil nil nil)
\o |||fprintf/sprintf: format spec. incompatible with data - nil
\o |||
```

Can you spot what causes the fprintf/sprintf error?

A Single-Step Trace Example

In this example, the trace output indicates that the code expects a cellview database object but is dealing with a window ID.

Can you determine the cause of the error from the following trace?

Sometimes trace output yields no evidence.

How can you find out more about this error?

Another Single-Step Trace Example

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

procedure( TriShapeReport( wid output )
    let( ( rectCount polygonCount pathCount miscCount )
        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
```

Using the SKILL Debugger to Trap an Error

Normally, when your program causes a SKILL error, your program is aborted. SKILL discards the knowledge about the program state at the time of the error. This knowledge includes the following:

- The values of local variables
- The sequence of function calls under evaluation at the time of the SKILL error

This information can help you determine the cause of the error. To examine this information before SKILL discards it, you must trap the error. To do this:

1. Install the SKILL Debugger.

This enables the SKILL Debugger to trap any untrapped error.

2. Run your application to reproduce the error.

The SKILL Debugger traps the error.

3. Display the SKILL stack and dump the local variables.

Analyze the flow of control.

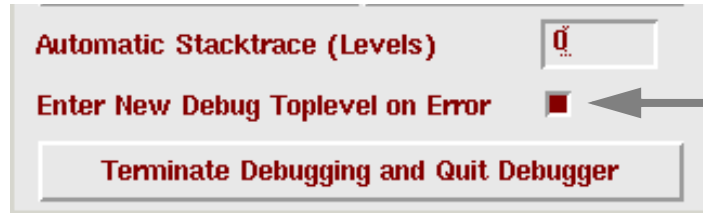
The next several pages cover more details of this debugging approach.

Using the SKILL Debugger to Trap an Error

Installing the SKILL Debugger

There are several ways to install the SKILL Debugger.

- Use the *installDebugger* function in your *.cdsinit* file to install the SKILL Debugger at your session's startup.
- Use the SKILL Debugger Toolbox to install the debugger.



Click here to install the debugger.

When you install the SKILL Debugger, the prompt in the lower left corner of the CIW changes from `>` to `1>`. The prompt in the *CDS.log* file also changes from `>` to `1>`.

Installing the SKILL Debugger

The *installDebugger* Function

You can use the *installDebugger* function in your *.cdsinit* file to install the SKILL Debugger. The *installDebugger* function does the following tasks:

- Installs the SKILL Debugger as the default error handler.
- Turns on the *debugMode* switch to clear the write protection on all functions. While the SKILL Debugger is installed, you can redefine any SKILL function.

The *uninstallDebugger* Function

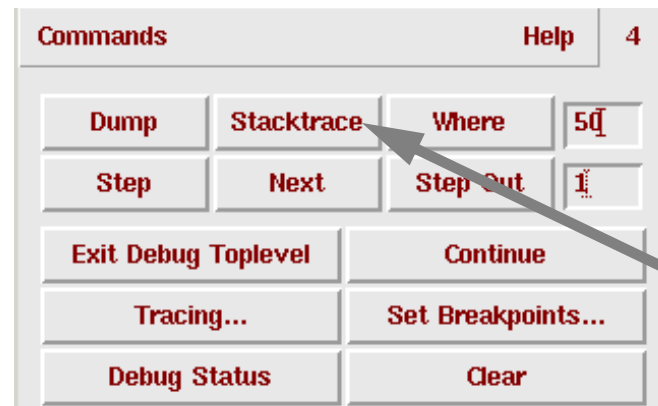
The *uninstallDebugger* function does the following tasks:

- Restores the normal system error handler.
- Turns off the *debugMode* switch to restore the write protection on all functions.

Displaying the SKILL Stack

To display the SKILL stack, use one of these two alternatives:

- Use the **Stacktrace** command on the SKILL Debugger Toolbox.



- Use the *stacktrace* function.

Displaying the SKILL Stack

The Dump Command

Prints the current value of all the local variables on the stack.

The Stacktrace Command

Prints the functions on the stack and their arguments.

The Where Command

This command displays the SKILL stack and the values of all the local variables that you can reference.

Analyzing the SKILL Stack

A SKILL stack trace displays the sequence of function calls under evaluation, using one line per function call. For the most recent function call, it displays all the expressions under evaluation.

```
\p Debug 2>
\i stacktrace()
\e <<< Stack Trace >>>
\e errorHandler("postincrement" 0 t ... )
\e (miscCount = miscCount)
\e miscCount++
\e case((shape~>objType) ("rect" rectCount++) ("polygon" polygonCount++) ... )
\e foreach(shape (cv~>shapes) case((shape~>objType) ("rect" &) ("polygon" &) ... ))
\e let((rectCount polygonCount pathCount ... ) (rectCount = (polygonCount = &))
      (cv = geGetWindowCellView(wid)) ... )
\e TriShapeReport(wid thePort)
\e TrShapeReport(window(4))
\t 8
\p Debug 2>
\i miscCount
\t nil
\p Debug 2>
```

SKILL abbreviates each line with these notations:

- (...) indicates arguments that have been clipped.
- (&) indicates function invocations that have been clipped.

You control the clipping with the *tracelevel* and *tracelength* variables.

Analyzing the SKILL Stack

tracelevel

This environment variable controls the depth of elements in a list that are printed during tracing.

tracelength

This environment variable controls the number of elements in a list that are printed during tracing.

Debugging Example

The next several pages show an excerpt from the `~/CDS.log` file.

This page shows the following steps:

1. Installing the SKILL Debugger
2. Running your program

The SKILL Debugger traps the error.

```
\p >
\i installDebugger()
\t t
\p 1>
\i TrShapeReport( window( 4 ))
\o *** Error in routine postincrement:
\o Message: *Error* postincrement: can't handle add1(nil)
\o SKILL Debugger: run GUI-based tools/bin/cdsFinder
    on search string `debug' for a list of commands or debugQuit to leave.
\p Debug 2>
```

Debugging Example

Debugging Example (continued)

This excerpt from the `~/CDS.log` shows these next two steps:

1. Dumping the SKILL stack at the time of the error.
2. Examining the value of the *miscCount* variable.

```
\p Debug 2>
\i stacktrace()
\e <<< Stack Trace >>>
\e errorHandler("postincrement" 0 t ... )
\e (miscCount = miscCount)
\e miscCount++
\e case((shape~>objType) ("rect" rectCount++) ("polygon" polygonCount++) ... )
\e foreach(shape (cv~>shapes) case((shape~>objType) ("rect" &) ("polygon" &) ... ))
\e let((rectCount polygonCount pathCount ... ) (rectCount = (polygonCount = &))
      (cv = geGetWindowCellView(wid)) ... )
\e TriShapeReport(wid thePort)
\e TrShapeReport(window(4))
\t 8
\p Debug 2>
\i miscCount
\t nil
\p Debug 2>
```

What caused the error?

How can you fix the error?

Debugging Example (continued)

Suspending and Resuming Your Program

Sometimes the bug you are investigating does not cause a SKILL error. When your program runs to completion, you need to validate the output. Using the CIW, you can interactively verify expressions from your program.

The SKILL expressions in your program typically depend upon function arguments and local variables.

Before verifying the SKILL expressions in the CIW, you need to set the variables in the expressions to appropriate values. But, doing so is time consuming and error prone. It is better to run your program and suspend the execution at the expression you want to evaluate.

To have SKILL suspend your program, you have two alternatives:

- You can edit the source code for your program to call the *break* function. SKILL suspends your program when your program calls the *break* function.
- You can set a breakpoint at a function boundary. SKILL suspends your program when the flow of control reaches the function boundary.

After examining local variables and verifying expressions, you resume the execution of your program.

Suspending and Resuming Your Program

A function boundary consists of four points. Each point is an event that occurs during a function call. The order of events is call, entry, exit, and return. In the following table, the caller refers to the code that contains the function call and the callee refers to the function itself:

Point	Description
Call	In the caller, after SKILL evaluates the argument expressions
Entry	In the callee, after binding the caller's actual arguments to the callee's formal arguments
Exit	In the callee, after computing the return result
Return	In the caller, after running to the caller

You can set a breakpoint at all four points for any function you define while the SKILL Debugger is installed.

For other SKILL functions, you can set a breakpoint at only the call point or the return point.

Setting Breakpoints

To set a breakpoint at a function, use one of these two alternatives:

- Use the **Set Breakpoints** command available on the SKILL Debugger Toolbox. This command lets you
 - Set a conditional breakpoint at the entry point of a function.
 - Set a conditional breakpoint at the exit point of a function.
 - Clear all breakpoints that are currently set.

This command does not let you set more than one breakpoint on a function at a time.

- Use the *breakpt* function.

Example

If you want to set a breakpoint both at the entry point and at the exit point of a function, you must use the *breakpt* function.

The following example sets an unconditional breakpoint at the entry and at the exit of the *TriShapeReport* function.

```
breakpt( TrShapeReport ( ( entry t ) ( exit t ) ) )
```

Setting Breakpoints

The *breakpt* Function

```
breakpt( [s_function [break_condition]])=> g_result
```

This function sets breakpoints on one or more functions. The SKILL debugger is the default break handler and is entered automatically when a breakpoint is encountered. The functions *breakpt* and *unbreakpt* set and clear breakpoints on the functions given.

If no *break_conditions* are given, the breakpoint is called "unconditional." The behavior of an unconditional breakpoint is as follows:

- if the function is read-protected or not yet defined under *debugMode*, the breakpoint is assumed to be set at the "call" point.
- Otherwise, it is assumed to be set at the "entry" point of the function.

Value returned:

List of functions whose breakpoints have been set.

Clearing Breakpoints

To clear one or more breakpoints, use one of these two alternatives:

- Use the **Set Breakpoints** command available on the SKILL Debugger Toolbox. This command allows you to
 - Clear a breakpoint at a function.
 - Clear all breakpoints that are currently set.
- Use the *unbreakpt* function.

Clearing Breakpoints

The *unbreakpt* Function

```
unbreakpt( [s_function... | t] ) => g_result
```

Clears breakpoints for the functions listed. The return value is the names of the functions for which the breakpoints were removed.

Hitting a Breakpoint

The following excerpt from the `~/CDS.log` shows these steps:

1. Setting a breakpoint at the entry of the *TriShapeReport* function.
2. Calling the *TrShapeReport* function.
3. Displaying the stack at the breakpoint.
4. Examining the *wid* and *thePort* function arguments.

```
\i breakpoint TriShapeReport
\t (TriShapeReport)
\p 1>
\i TrShapeReport( window( 4 ))
\o <<< Break >>> on entering TriShapeReport
\o SKILL Debugger: ...
\p Debug 2>
\i stacktrace
\o <<< Stack Trace >>>
\o breakHandler(nil)
\o entry of TriShapeReport
\o TriShapeReport(wid thePort)
\o TrShapeReport(window(4))
\t 3
\p Debug 2>
\i wid
\t window:4
\p Debug 2>
\i thePort
\t port: "/tmp/ShapeReport.txt"
\p Debug 2>
```

Hitting a Breakpoint

Resuming Your Program

To resume your suspended program from a breakpoint, you have these options:

- Use the **Continue** command on the SKILL Debugger Toolbox.



- Use the *continue* function.

Your program continues to execute. If the flow of control reaches another breakpoint, your program suspends execution again.

Resuming Your Program

The Continue Command

Continues execution from a breakpoint.

The Continue Function

```
cont( )  
continue( )
```

Continues execution from a breakpoint. The cont and continue functions are identical.

```
cont( ) => exits the break handler
```

The *continue* Function

Use the *continue* function to resume your program from a breakpoint.

Example

```
\i breakpt( outfile TriShapeReport TrShapeReport )
\t (TrShapeReport TriShapeReport outfile)
\p 1>
\i TrShapeReport( window( 4 ))
\o <<< Break >>> on entering TrShapeReport
\o SKILL Debugger: ...
\p Debug 2>
\i continue
\o <<< Break >>> on calling outfile with args ("/tmp/ShapeReport.txt")
\o SKILL Debugger: ...
\p Debug 2>
\i continue
\o <<< Break >>> on entering TriShapeReport
\o SKILL Debugger: ...
\p Debug 2>
\i stacktrace
\o <<< Stack Trace >>>
\o breakHandler(nil)
\o entry of TriShapeReport
\o TriShapeReport(wid thePort)
\o TrShapeReport(window(4))
\t 3
\p Debug 2>
```

The *continue* Function

The excerpt from the *~/CDS.log* shows these steps:

1. Setting a breakpoint at the entry of the *outfile*, *TriShapeReport*, and *TrShapeReport* functions.
2. Calling the *TrShapeReport* function.
3. Continuing from the breakpoint at the entry of the *TrShapeReport* function.
4. Continuing from the breakpoint at the call to the *outfile* function.
5. Displaying the stack.

Single-Stepping Your Program

From a breakpoint, you can execute your program one function call at a time.

To run your program in single-step mode, you have the following options:

- Use the **Step** command on the SKILL Debugger Toolbox.



- Use the *step* function.

SKILL suspends your program after executing the current function call.

Single-Stepping Your Program

This example shows single-stepping from a breakpoint at the *TrShapeReport* function.

```
\i breakpt TrShapeReport
\t (TrShapeReport)
\p 1>
\i TrShapeReport( window( 4 ))
\o <<< Break >>> on entering TrShapeReport
\o SKILL Debugger: ...
\p Debug 2>
\i step
\o stopped before evaluating let((thePort)
    (thePort = outfile("/tmp/ShapeReport.txt")) ... )
\p Debug 2>
\i step
\o stopped before evaluating (thePort = outfile("/tmp/ShapeReport.txt"))
\p Debug 2>
\i step
\o stopped before evaluating outfile("/tmp/ShapeReport.txt")
\p Debug 2>
\i step
\o stopped before evaluating when(thePort TriShapeReport(wid thePort) ... )
\p Debug 2>
\i step
\o stopped before evaluating TriShapeReport(wid thePort)
\p Debug 2>
\i step
\o <<< Entering TriShapeReport >>>
\o stopped before evaluating let((rectCount polygonCount pathCount ... ) ...
\p Debug 2>
```

SKILL Debugger Sessions

When the SKILL Debugger suspends your program because of a breakpoint or an error trap, it creates a debugger session.

In a debugger session, you can examine the program state as follows:

- Use the CIW to examine local variables or evaluate an expression that refers to local variables.
- Display the stack to clarify the flow of control.

Breakpoint

If the program is suspended due to a breakpoint, you have these choices:

- Resume the program.
- Abort the program by quitting the debugger session.

Error Trap

If the program is suspended due to an error trap, you can only abort the program.

SKILL Debugger Sessions

After you have fixed a problem, make sure that you abort the program before rerunning it.

Quitting a Debugger Session

When you are at a breakpoint, you can abort your program by quitting the debugger session.

When the SKILL Debugger traps an error, you cannot resume your program. After examining local variables and displaying the stack, you need to quit the debugger session.

You can choose either of these ways to quit a debugger session:

- Use the **Exit Debug Toplevel** command in the SKILL Debugger Toolbox.



- Use the *debugQuit* function.

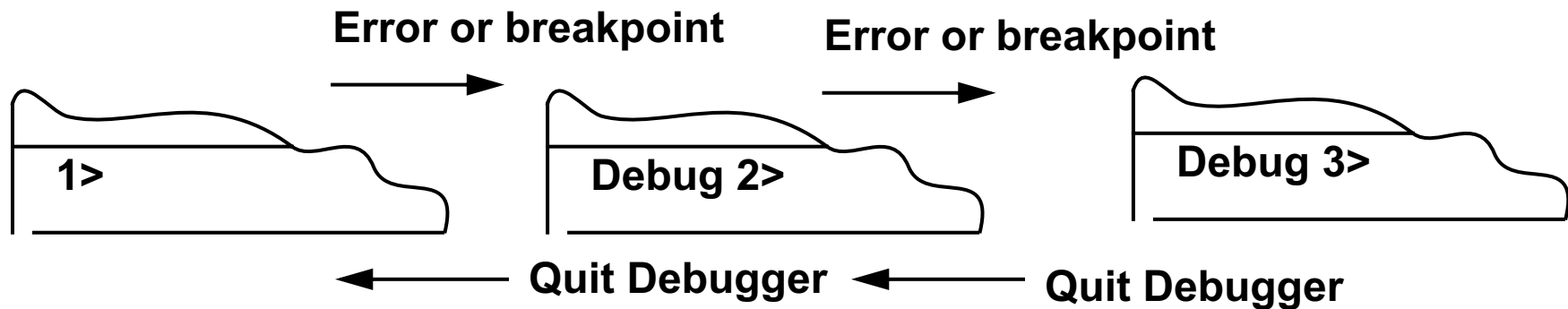
In fact, the SKILL Debugger does not restrict your access to other activities in the Design Framework II environment.

But it's a good idea to exit the debugger session as soon as you have finished your investigation of the program state.

Quitting a Debugger Session

Nesting SKILL Debugger Sessions

The SKILL Debugger nests debugger sessions. The prompt in the CIW indicates a number that is one higher than the number of nested SKILL Debugger sessions.



Why does the SKILL Debugger nest debugger sessions?

Nesting SKILL Debugger Sessions

The SKILL Debugger does not restrict your access to the Design Framework II environment. During a debugger session, you can still do the following through the user interface:

- Trigger the evaluation of other SKILL expressions.
- Use the CIW to evaluate a SKILL expression.

One of these SKILL expressions can generate an error or call a function.

SKILL Debugger Summary

To set an error trap, install the SKILL Debugger. This error trap only traps errors that are not otherwise trapped.

You do not have to install the SKILL Debugger if you just want to set breakpoints.

It is a good idea to install the SKILL Debugger and define several aliases in your *.cdsinit* file. Include the following lines in your *.cdsinit* file:

```
installDebugger()  
alias( q debugQuit )  
alias( c continue )
```

SKILL Debugger Summary

Frequently Asked Questions

Question

The CIW prompt says *Debug 23>*. Is there one command that will get me back to a CIW prompt of *1>*?

Answer

No. You have to click Quit Debugger 22 times. The Quit Debugger command terminates the most deeply nested debugger session. This is the most recently created debugger session.

Question

Why can't I just uninstall the SKILL Debugger to get back to a CIW prompt of *1>*?

Answer

Uninstalling the debugger changes the CIW prompt back to the normal *>*. The debugger will not trap subsequent errors. However, the nested debugger sessions remain. Also, breakpoints are still active and most of the debugger commands still work.

Frequently Asked Questions

Other SKILL Debugger Benefits

While the SKILL Debugger is installed, you have the following advantages:

- You can redefine any SKILL function.
- When you call the *edit* function, you can specify just the function name, as in this example:

```
edit( TrShowFileBrowser t )
```

- You can call the *whereIs* function to display information about a function's source code file as in this example.

```
whereIs( TrShowFileBrowser )  
TrShowFileBrowser is defined in file  
"/usr1/mnt/skilldev/SKILL/CaseStudy/ShowFileBrowser.il"  
at line 492.
```

Other SKILL Debugger Benefits

While the SKILL Debugger is installed, the *load* function associates the file being loaded with each function that the file declares.

Lab Overview

Lab 11-1 Analyzing an Error

Lab 11-2 Debugger Sessions

Lab Overview

SKILL Lint

SKILL Lint performs the following tasks:

- Analyzes your source code for many common errors and inefficient techniques.
- Generates hints, suggestions, warnings, and errors.
 - Locates local variables that are never used.
 - Isolates globals that are never referenced.
 - Checks for globals without the designated prefixes (such as *Tr*).
 - Validates procedure calls for the correct number of arguments.
 - Checks for incorrect keyword arguments.
 - Checks for a *return* expression outside of a *prog* expression.
 - Advises the use of more efficient constructs.
- Produces a summary score for your program.

You can read further about SKILL Lint in the *SKILL User Guide*.

SKILL Lint

You access SKILL Lint from the SKILL Development Toolbox.

SKILL Lint can analyze all the source code in a file or in a context.

You direct SKILL Lint output either to a file or to the *CDS.log*.

You can selectively disable SKILL Lint checks.

You need to identify package prefixes that your global variables use.

SKILL Profiler

The SKILL Profiler tells you where your SKILL programs are taking the most time and allocating the most memory. Specifically, it performs the following tasks:

- Measures the time spent in each function that executes longer than 1/60th of a second.
- Shows how much SKILL memory is allocated in each function.
- Displays a tree graph of all functions that your program executed, and the time that those functions spent, or the memory that those functions allocated.

You can measure performance without having to modify function definitions.

You can read further about the SKILL Profiler in the SKILL *User Guide*.

SKILL Profiler

Lab Overview

Lab 11-3 Using SKILL Lint

Lab Overview

Module Summary

In this module, we

- Matched SKILL development utilities with development tasks
- Surveyed the SKILL Debugger
- Learned about SKILL Lint and the SKILL Profiler

Module Summary

Module 12: List Construction

Objectives

- Review the techniques for building a list.
 - The ' operator
 - The *list* function
 - The *cons* function
 - The *append* function
- Use the *foreach mapcar* function to make a list that corresponds one to one with a given list.
- Use the *setof* function to make a filtered copy of a list.
- Use the *foreach mapcar* function and *setof* function together to make a list that corresponds one to one with a filtered list.

Terms and Definitions

List Review

You can make a new list by:

- Specifying all the elements literally.

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

- Computing each element from an expression.

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

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

- Adding an element to the front of a list.

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

- Merging two lists together.

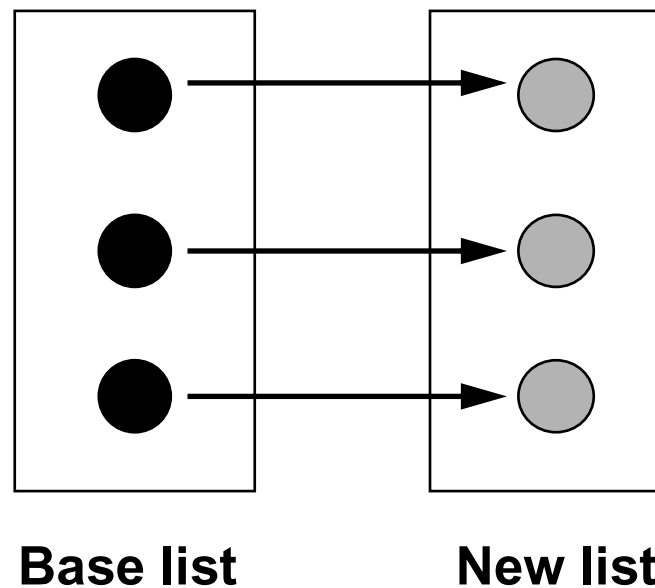
```
oneList = '( 4 5 6 ) => ( 4 5 6 )  
aList = '( 1 2 3 ) => ( 1 2 3 )  
bList = append( oneList aList )  
=> ( 4 5 6 1 2 3 )
```

List Review

The *foreach mapcar* Function

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

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



Example

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

The *foreach mapcar* Function

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

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

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

Extended *foreach mapcar* Example

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

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

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

- However, you are responsible for collecting the return results. Use the **cons** function.
- In addition, the list will be in reverse order. Use the *reverse* function to make a copy of the list in the correct order.

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

Extended *foreach mapcar* Example

The *mapcar* Function

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

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

The *mapcar* function accepts two arguments:

- A function of one argument
- A list

The *mapcar* function

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

Examples

- A *foreach mapcar* expression

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

- An equivalent *mapcar* expression

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

The *mapcar* Function

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

- A trace of a *foreach* expression

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

- A trace of a *foreach mapcar* expression

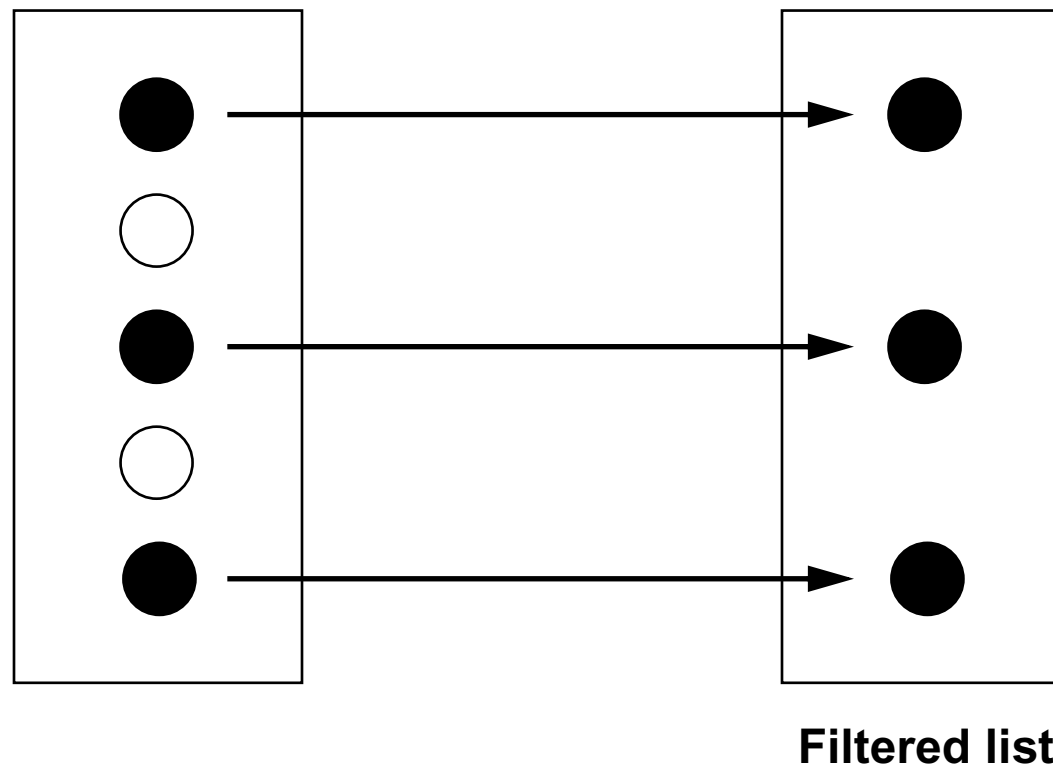
funobj:0x21980a8 represents the function that SKILL dynamically generates to represent $x**2$.

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

Filtering Lists

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

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



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

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

Filtering Lists

The *setof* Function

The *setof* function accepts several arguments:

- A local variable that holds a list element in the filter expressions.
- The list to filter.
- One or more expressions that compose the filter.

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

- Binds each element to the local variable.
- Evaluates the expressions in its body.
- Uses the result of the last expression to determine whether to include the element.

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

The *setof* Function

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

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

A *setof* Example

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

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

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

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

A *set of* Example

Another *setof* Example

Suppose you want to compute the intersection of two lists.

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

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

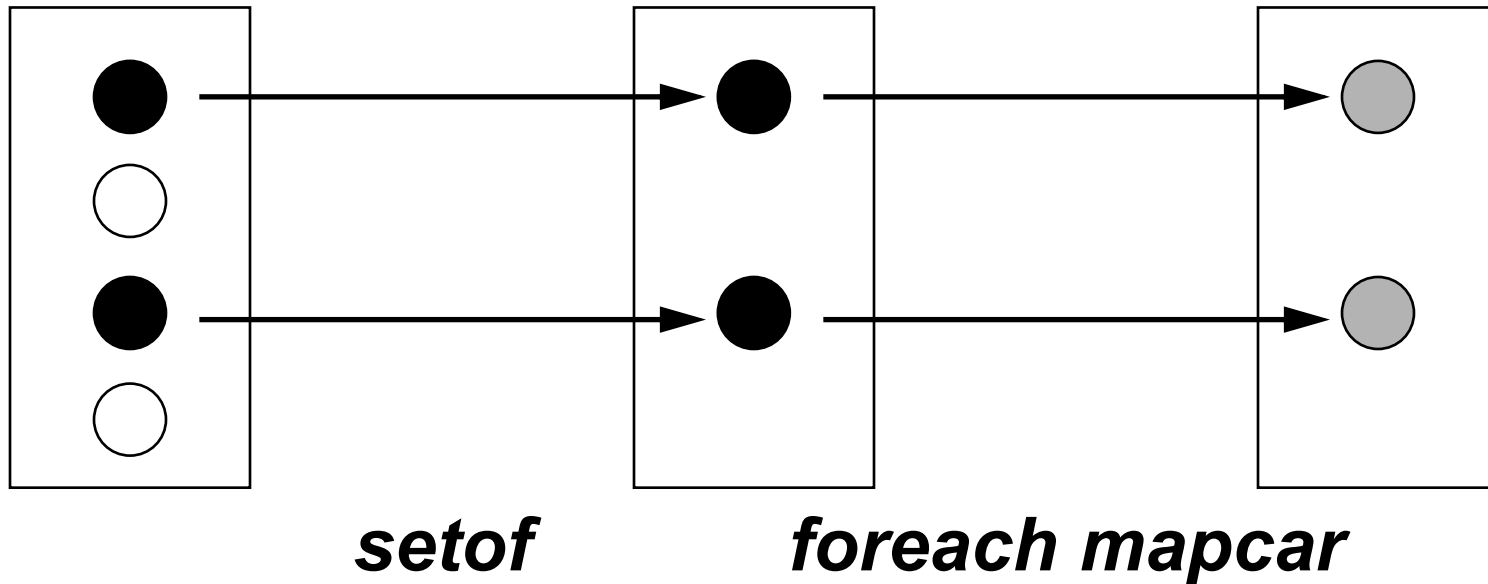
The more efficient way is to use the *setof* function.

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

Another *setof* Example

Transforming Elements of a Filtered List

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



- Use the *setof* function to filter the list.
- Use the *foreach mapcar* function to build the resulting list.

Transforming Elements of a Filtered List

This example computes the list of squares of odd integers.

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

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


Lab Overview

Lab 12-1 Revisiting the Shape Report Program

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

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

Lab 12-2 Describing the Shapes in a Design

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

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

Lab Overview

Module Summary

In this module, we covered

- The *foreach mapcar* function to build lists.
- The *setof* function to make a filtered copy of a list.
- The *foreach mapcar* function and the *setof* function used together to build a list by transforming each element.

Module Summary

Module 13: Cellview Data Model

Objectives

- Review database object concepts.
- Survey the cellview data model documentation.
- Open a cellview nongraphically.
- Create geometry in a design.
- Save and close a cellview.
- Survey how the cellview data model represents
 - Geometry
 - Hierarchy
 - Connectivity
 - User-defined Properties
- Retrieve all the attributes of a database object.

Terms and Definitions

CDBA	Design Framework II database technology.
Library	A collection of design objects that you refer to by a logical name, such as cells, views, and cellviews.
Cell	A component of a design: a collection of different representations of the components implementation, such as its schematic, layout, or symbol.
Cellview	A particular representation of a particular component, such as the layout of a flip–flop or the schematic of a NAND gate.
View	An occurrence of a particular view type identified by its user–defined name, "XYZ". Each "XYZ" view has an associated <i>viewType</i> , such as <i>maskLayout</i> , <i>schematic</i> , or <i>symbolic</i> .
Pin	A physical implementation of a terminal.
Terminal	A logical connection point of a component.

Database Object Review

In the Database Queries module, we covered

- Database object concepts
- Several specific object types
 - The *cellview* object type
 - The *inst* object type
 - The *Shapes* object types
 - The *net* object type
- The *geGetWindowCellView* function
- The *geGetSel/Set* function
- Using the *~>* operator to retrieve attributes and user-defined properties.

Database Object Review

Design Framework II Database Technology

CDBA refers to the Design Framework II database technology. It includes the following:

- A cellview data model
- A SKILL API (Application Programming Interface)

The Cellview Data Model expresses the following design data:

- Geometry
- Hierarchy
- Connectivity

Applications Programs can express user-defined data and relationships by means of

- User-defined properties
- User-defined groups

Design Framework II Database Technology

CDBA

C-level database access (CDBA) consists of procedural access layers on top of the Cadence database. CDBA routines provide nearly complete access to all the data stored in the Cadence design database. You can create, modify, save, retrieve, and maintain data in the Cadence database. CDBA functions are available at the programming level and allow tight and complete control over database operations.

Cellview Data Model

Database Access SKILL Documentation

The “Database Access” chapter in the *Design Framework II SKILL Functions Reference Manual* contains the following sections:

- Introduction
- Data Stored as Objects
- Database Access Functions
- Description of Database Objects

See the Cadence online documentation to access this material by

- Searching for specific functions, such as the *dbOpenCellViewByType* function.
- Using the table of contents for this material.

Database Access SKILL Documentation

Description of Database Objects

The “Description of Database Objects” section has a series of tables that list the attributes for each object type. Do a search in CDSDoc for “Description of Database Objects” to locate the section.

Each table consists of four columns.

- The **Attribute** column lists the attribute name.
- The **Set?** column lists whether you can use the ~> operator together with the = operator to update the attribute value.
- The **Type** column lists the constrained value of the attribute. The value of an attribute is constrained to be one of these SKILL data types:
 - String
 - Integer
 - Float
 - Database object
 - List
 - Boolean (appears as a string, *true* or *false*)
- The **Description** column summarizes the meaning of the attribute.

Description of Database Objects

There are three types of attributes as shown in this table.

Attribute	Description
Mandatory	Must be specified at the time the database object is created. These attributes are essential to the viability of the database object.
Optional	Might or might not be present on a particular database object, in which case its value is <i>nil</i> .
Derived	Are computed dynamically from other attributes. Consequently, you cannot change them.

Database Object Classes

Several object types have similar attributes as in these examples:

- All object types have an *objType* attribute.
- The *inst* and *mosaicInst* object types both have a *master* attribute.
- All shape object types and the *inst* object type have a *bBox* attribute.
- All shape object types have a *layerName* attribute.

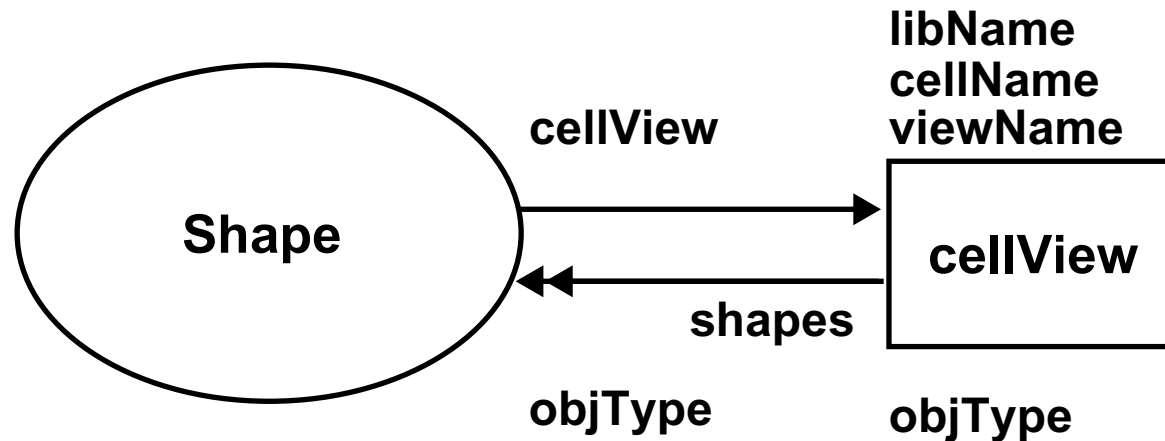
An object *class* is a collection of object types with similar attributes.

Class	Scope
<i>Generic Database Object</i>	All database objects
<i>Figure</i>	Any selectable database object
<i>Instance</i>	The <i>inst</i> and <i>mosaicInst</i> database objects
<i>Shape</i>	All shape database objects

Database Object Classes

The Cellview Data Model Road Map

In this module, diagrams such as this one summarize the Cellview Data Model.



The diagrams use the following conventions:

- An **ellipse** represents a class of object types that share attributes.
- A **rectangle** represents an object type.
- Attribute names are clustered around the ellipses or rectangles.
- Arrows represent relationships between two object types or two object type classes. Each arrow is labelled with an attribute name.
- A single-headed, thin, arrow represents a 1(0) to 1 relationship
- A double-headed, thin, arrow represents a 1(0) to many relationship

The Cellview Data Model Road Map

The table below summarizes the example road map. The road map focuses on attributes that represent relationships between object types.

Object Type or Class	Attribute Name	Attribute Value
<i>Shape</i>	<i>objType</i>	
<i>Shape</i>	<i>cellView</i>	1 <i>cellView</i> database object.
<i>cellView</i>	<i>objType</i>	
<i>cellView</i>	<i>libName</i>	
<i>cellView</i>	<i>cellName</i>	
<i>cellView</i>	<i>viewName</i>	
<i>cellView</i>	<i>shapes</i>	0 or more <i>Shape</i> database objects.

The `~/SKILL/CaseStudy/RoadMap.il` defines a function that builds the complete road map as a layout cellview. See Lab 10, *Building the Cellview Data Model Road Map*, for further details.

Opening Cellviews

The *dbOpenCellViewByType* function opens a cellview nongraphically. If successful, it returns the database object of the cellview.

```
cv = dbOpenCellViewByType(  
    "master"           ;; library name  
    "mux2"             ;; cell name  
    "schematic"        ;; view name  
    "schematic"        ;; view type  
    "r"                ;; access mode  
                        ;; "r" = read, "a" = append, "w" = overwrite  
)  
⇒ db:12345678
```

Use the *dbOpenCellViewByType* function to create a new cellview.

```
cv = dbOpenCellViewByType( "master" "new"  
    "layout" "maskLayout" "w" )  
=> db:37711916
```

Opening Cellviews

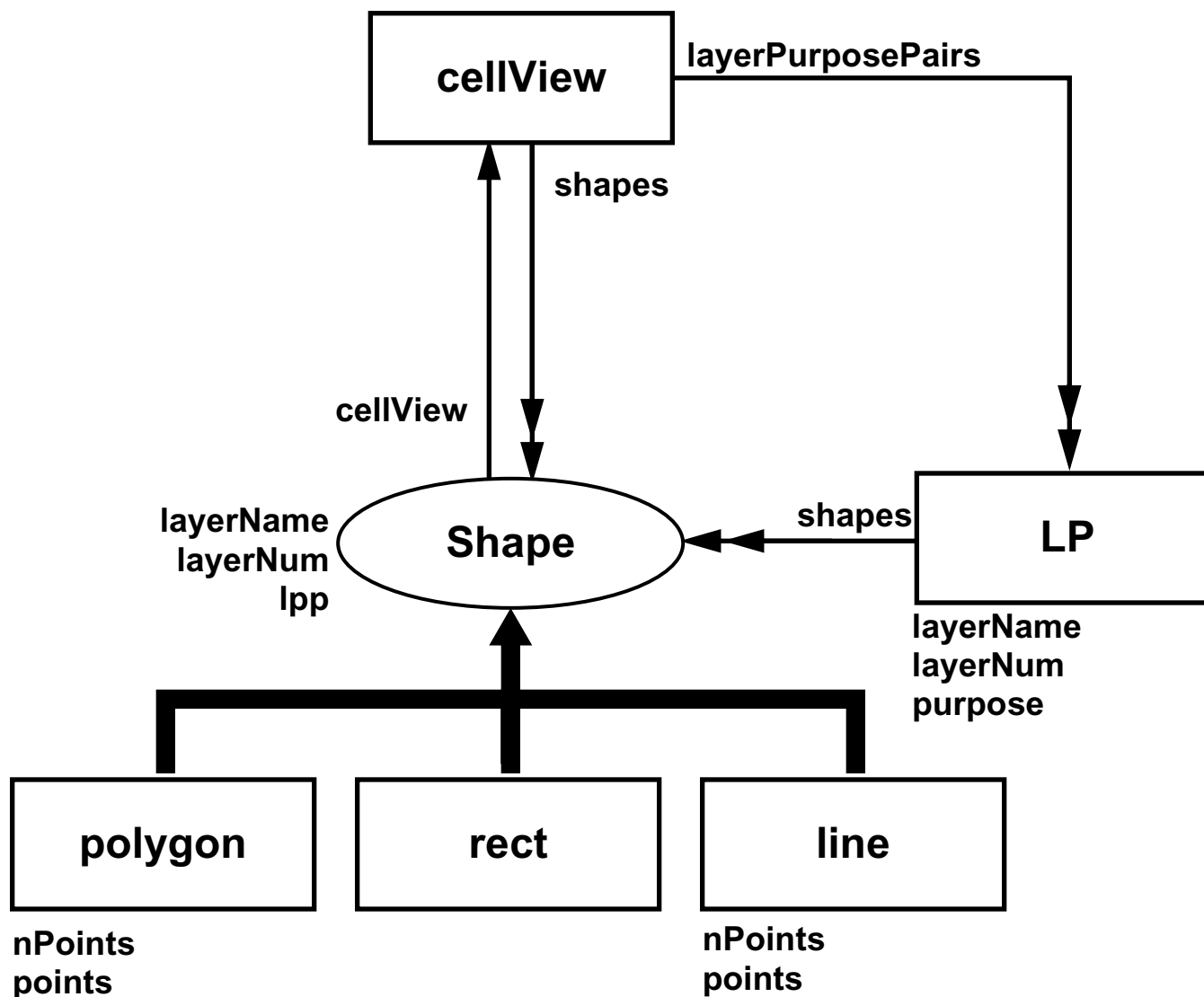
The *dbOpenCellViewByType* Function

If the *dbOpenCellViewByType* function is unsuccessful, it returns *nil*.

```
cv = dbOpenCellViewByType( "master" "boxes"  
    "layout" "maskLayout" )  
*WARNING* failed to open cellview (boxes layout)  
from lib (master) in 'r' mode  
because cellview does not exist.  
nil
```

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

Shapes



Note: Not all shape object types are shown.

➡ is a kind of

Shapes

Every *cellView* database object has a *shapes* attribute. The value of *shapes* is a list of all shapes in the design. If you need to process all the shapes in a design by layer purpose pair, retrieve the *layerPurposePairs* attribute, whose value is a list of *LP* database objects.

Each *LP* database object has:

- A *shapes* attribute, which is a list of all shapes that are on that layer purpose pair.
- *layerName* and *layerNum* attributes, which identify the layer purpose pair.

Each *Shape* database object has:

- *layerName* and *layerNum* attributes, which identify the layer name and number of the layer where the shape is.
- An *lpp* attribute whose value is the list of layer name and layer purpose.

Creating Shapes

There are several SKILL functions that create database objects. These functions:

- Begin with *dbCreate*.
- Require a cellview database object and enough other arguments to describe the object.
- Return the new database object.

Use the *dbCreateRect* function to create a rectangle on a layer with a given bounding box.

```
let( ( cv newRect )
  cv = dbOpenCellViewByType( "master" "new"
    "layout" "maskLayout" "w" )
  newRect = dbCreateRect(
    cv                ;;; cellView database object
    "metal1"          ;;; layer name
    list( 0:0 .5:.5 ) ;;; bounding box
  )
  ...
) ; let
```

Creating Shapes

The *dbCreateRect* Function

```
dbCreateRect( d_cellView tx_layer l_bBox ) => d_rect / nil  
dbCreateRect( d_cellView (tx_layer [t_purpose] ) l_bBox ) => d_rect /  
nil
```

This function creates a rectangle.

Arguments	Interpretation
d_cellView	Specifies the cellview.
tx_layer	Specifies either the layer name or number.
t_purpose	Specifies purpose, defaults to <i>drawing</i> .
l_bBox	Defines the lower left and upper right corners of the bounding box.

Returns the *dbObject* of the rectangle.

Returns *nil* if the rectangle is not created.

Updating Shapes

You can often use the `~>` operator to update a shape.

Expression	Action	Example
<code>~>bBox =</code>	Move a rectangle	<code>aRect~>bBox = newBBox</code>
<code>~>layerName =</code>	Change a shape's layer	<code>aRect~>layerName = "thinox"</code>
<code>~>points =</code>	Change a path's points	<code>aPath~>points = newPoints</code>
<code>~>width =</code>	Change a path's width	<code>aPath~>width = 3.0</code>

There are attributes you are not allowed to set.

```
aPath~>nPoints = 6
Message: *Error* setSGq: dbSetq:
Can not set attribute - nPoints
```

The `getTransformUserBBox` function transforms a bounding box according to a displacement vector, a rotation, and a magnification.

```
newBBox = getTransformUserBBox( aRect~>bBox
  list( 0:deltaY "R0" )
)
```

Updating Shapes

Refer to the reference documentation for further information on these functions.

The *geTransformUserPoint* Function

The *geTransformUserPoint* function transforms a point or list of points by the displacement and rotation passed in.

The *geTransformUserBBox* Function

The *geTransformUserBBox* function is similar to *geTransformUserPoint*, except that it expects a bounding box. It returns the transformed bounding box in the standard format: the first point is the lower-left corner and the second point is the upper-right corner of the bounding box.

Saving and Closing a Cellview

The *dbSave* function saves a modified cellview that has been opened in write or append mode.

```
let( ( cv newRect )
  cv = dbOpenCellViewByType( "master" "new"
    "layout" "maskLayout" "w" )
  newRect = dbCreateRect(
    cv                ;;; cellView database object
    "metal1"          ;;; layer name
    list( 0:0 .5:.5 ) ;;; bounding box
  )
  dbSave( cv )
  ...
) ; let
```

You cannot save or overwrite a cellview that is open in read or scratch mode.

Saving and Closing a Cellview

The *dbSave* Function

See the Cadence online documentation to read more about this function. For instance, you can specify a different cellview to save the data to.

If you are editing a schematic cellview you will also need to do a *schCheck* prior to the *dbSave* function. This corresponds to the *Check and Save* step that you perform interactively when editing or expanding a schematic cellview.

The *dbClose* Function

Use the *dbClose* function when you no longer need a cellview in virtual memory.

```
let( ( cv newRect )
  cv = dbOpenCellViewByType( "master" "new"
    "layout" "maskLayout" "w" )
  newRect = dbCreateRect(
    cv                ;;; cellView database object
    "metal1"          ;;; layer name
    list( 0:0 .5:.5 ) ;;; bounding box
  )
  dbSave( cv )
  dbClose( cv )
) ; let
```

Avoid accessing purged objects. The `~>` operator cannot retrieve attributes from a purged database object that has been removed from virtual memory. In that case, the `~>` operator issues a warning and returns *nil*.

```
cv~>cellName
dbGetq: Object is a purged/freed object. - db:37711916
nil
```

After closing a cellview, set all variables containing the database object to *nil*.

The *dbClose* Function

See the Cadence online documentation to read more about the *dbClose* function.

To manage memory efficiently, CDBA maintains its own virtual memory pool. CDBA uses a simple reference count algorithm to determine when it can purge a cellview from virtual memory.

For a particular cellview, when the number of times you have called *dbClose* equals the number of times you have called *dbOpenCellViewByType*, CDBA returns the memory to the memory pool. CDBA uses the memory for subsequent *dbOpenCellViewByType* calls. Design Framework II returns the CDBA memory pool to UNIX when you exit the Design Framework II environment.

Lab Overview

Lab 13-1 Enhancing the Layer Shape Report SKILL Program

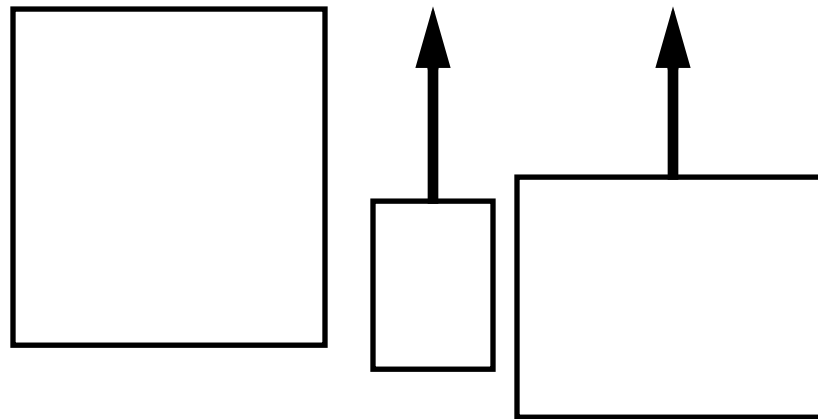
You develop a SKILL function that produces a report to count the number of each kind of shape on each layer in a design.

Lab 13-2 Creating a Cellview

You create a cellview containing various shapes.

Lab 13-3 Aligning Rectangles

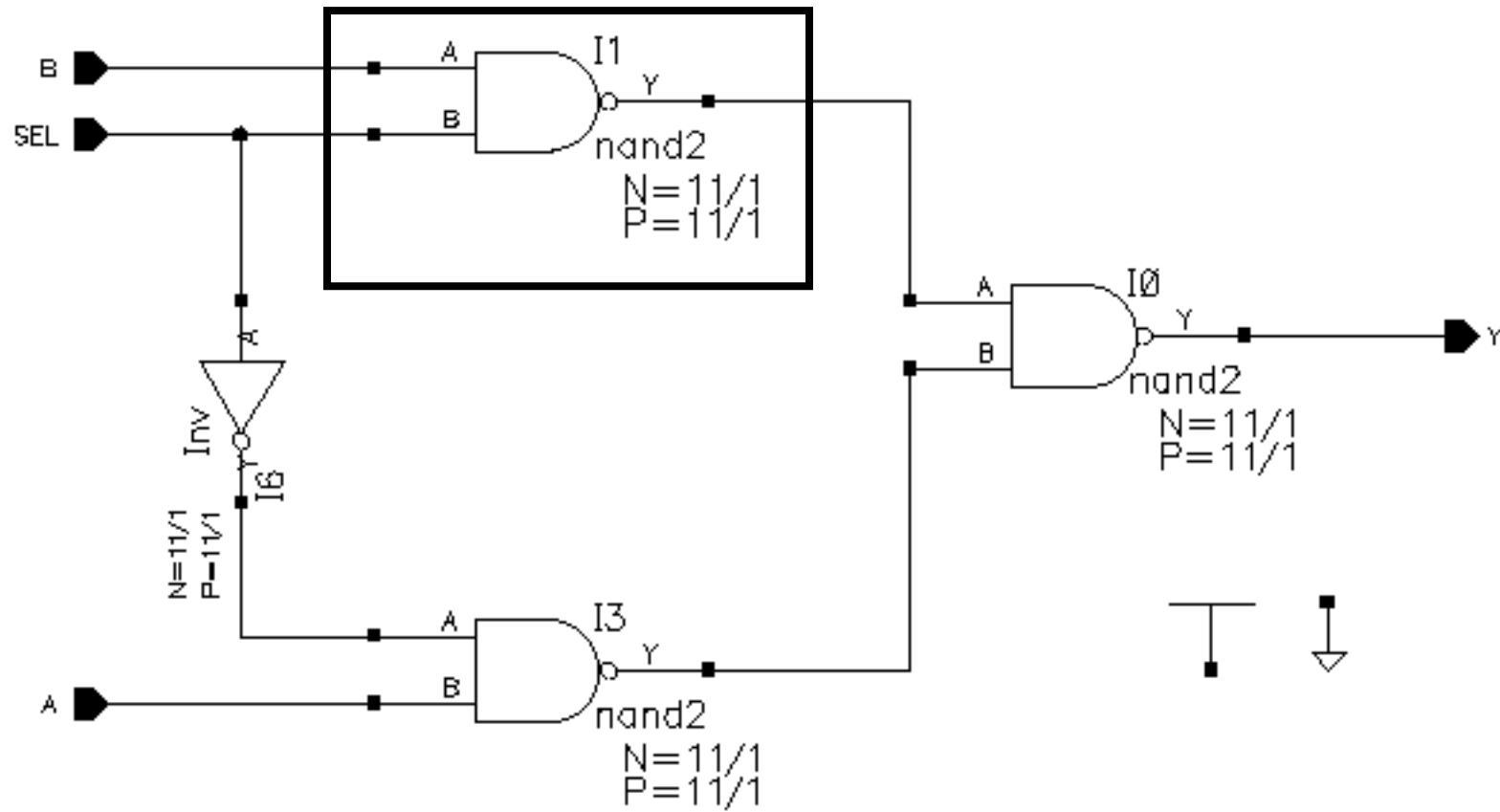
You develop a SKILL function to align the top sides of a collection of rectangles.



Lab Overview

Hierarchy

Example: Instance *I1* of *master mux2* schematic



Hierarchy

The following examples review ~> expressions you have seen previously.

■ The list of the instances in the design

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

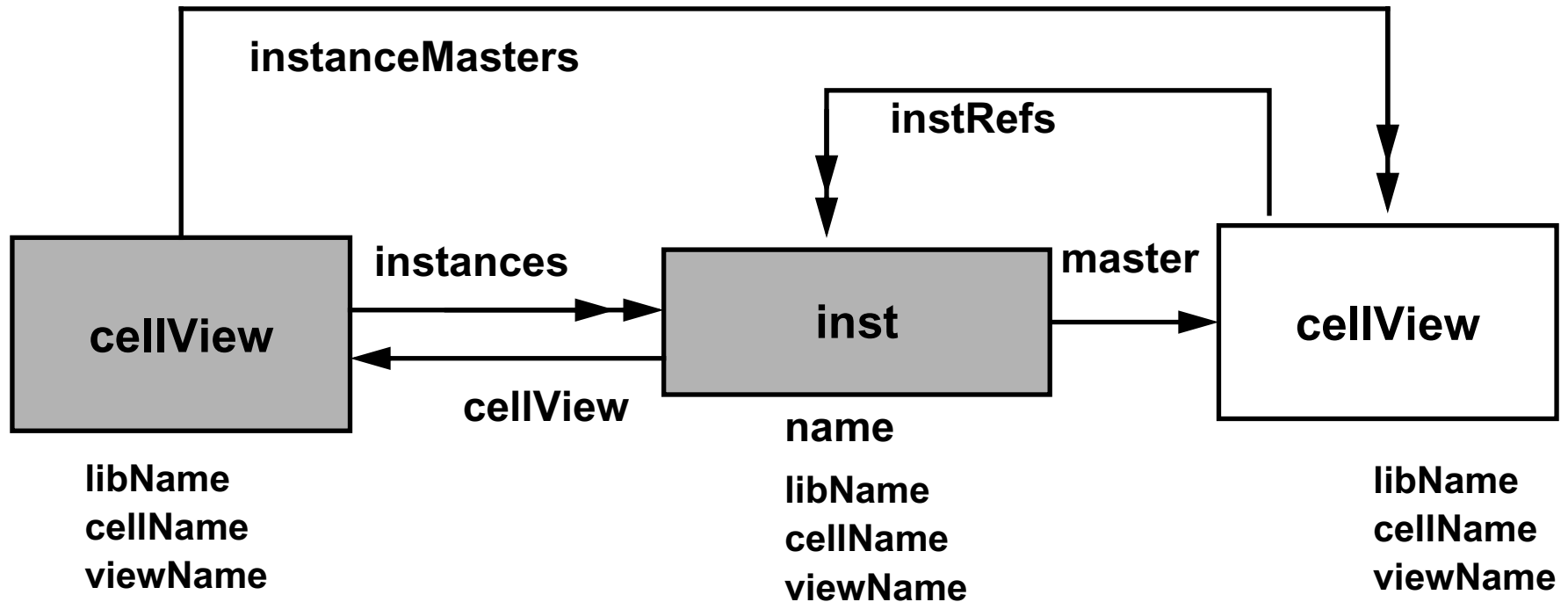
■ The list of master cell names

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

■ The list of instance names

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

The *cellView* and *inst* Object Types



```
mux2 = dbOpenCellViewByType( "master" "mux2" "schematic" )  
mux2~>cellName ==> "mux2"  
nth( 3 mux2~>instances )~>cellName ==> "nand2"
```

The *cellView* and *inst* Object Types

The *cellView* Object Type

Each *cellView* database object has:

- *libName*, *cellName*, and *viewName* attributes, whose values uniquely identify the design.
- an *instanceMasters* attribute, whose value is a list of all *cellView* database objects instantiated in this design.
- an *instRefs* attribute, whose value is a list of all *Instance* database objects with this design as master.

The *inst* Object Type

Each *Instance* database object has:

- the *objType* attribute value equal to "*inst*" or "*mosaicInst*".
- a *name* attribute whose value is the name of the instance.
- the *libName*, *cellName*, and *viewName* attributes, whose values uniquely identify the master.

The *TrFormatInstance* Function

The *TrFormatInstance* function prints information about an instance.

```
procedure( TrFormatInstance( inst )
  let( ( cv format )
    cv = inst~>cellView
    format = "%s %s %s %s %s\n" ;;; your choice
    printf(
      format
      inst~>name
      inst~>cellName ;; the master's cellName
      cv~>libName
      cv~>cellName
      cv~>viewName
    )
  ) ; let
) ; procedure
```

Assuming that *inst* contains an instance database object,

- The *inst~>objType* expression is either "*inst*" or "*mosaicInst*".
- The *inst~>cellView* expression is the *cellView* containing the instance.
- The *inst~>master* expression is the instance's master database object.
- The *inst~>master~>objType* expression is "*cellView*".

The *TrFormatInstance* Function

Sample output from the *CDS.log*.

```
\i TrFormatInstance( car( geGetSelSet()))  
\o I1 nand2 master mux2 schematic  
\t t  
\i TrFormatInstance( car( geGetSelSet()))  
\o I6 Inv master mux2 schematic  
\t t
```

*Can you account for the return value **t** by studying the source code?*

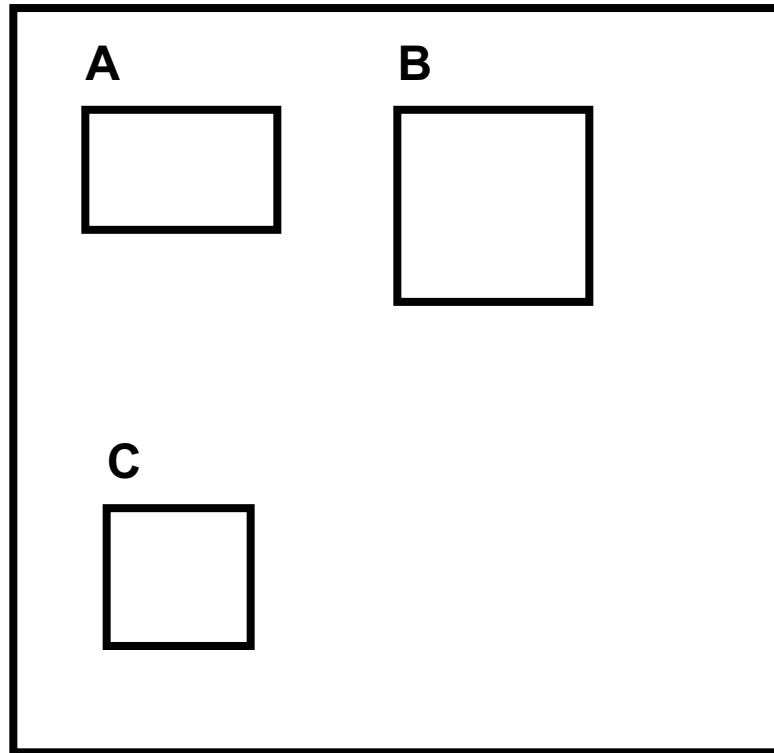
This more realistic format string prints the information in columns.

```
format =  
"Instance: %-10s Master: %-10s within CellView: %-10s %-10s %-10s \n"
```

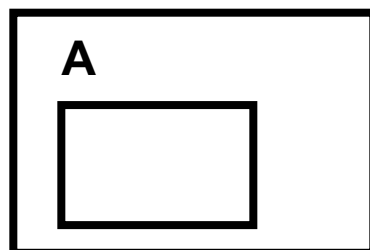
A Hierarchical Design

Assume you have a hierarchical design and you need a list of masters that are instantiated in the hierarchy. In this example, the list is (*A B C D*).

Top



D



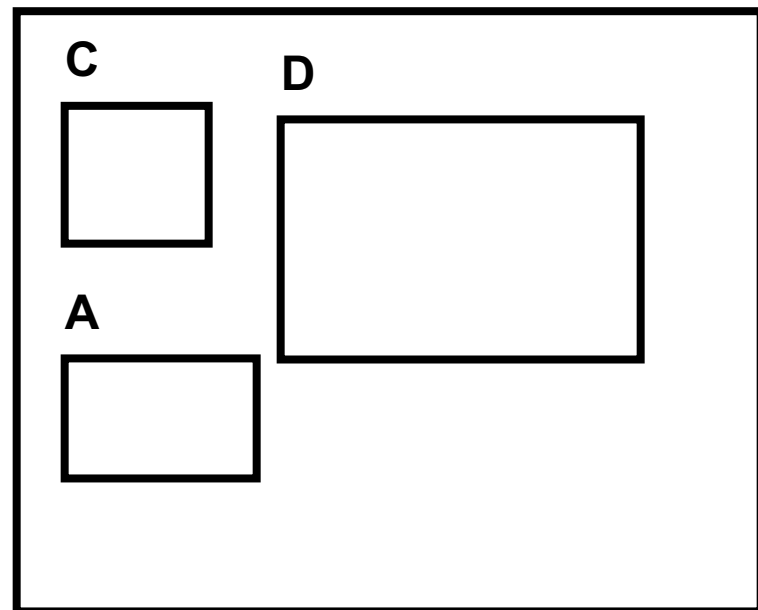
A



C

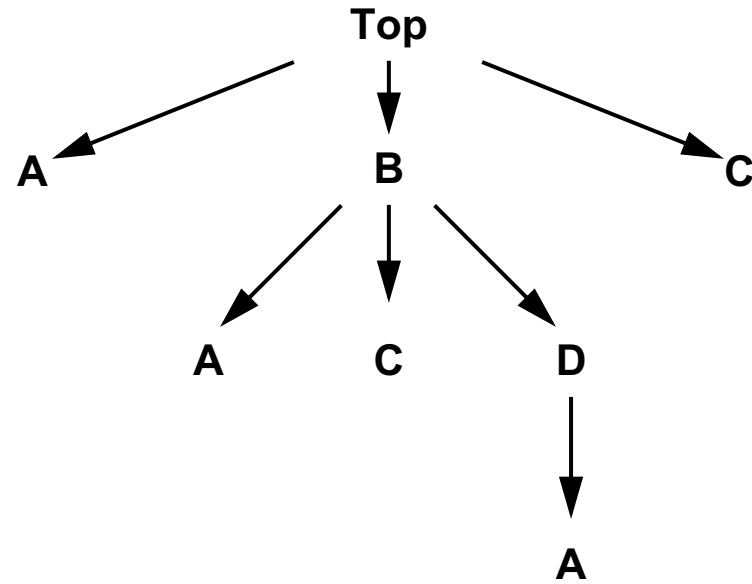


B



A Hierarchical Design

The following tree describes the hierarchy. A and C are leaf cells that contain no instances.



Traversing a Hierarchical Layout Design

```
procedure( TrHierarchyTraversal( cellView listSoFar )
  foreach( master cellView~>instances~>master
    let( ( nextCellView )
      nextCellView = master
      cond(
        ( !nextCellView nil )
        ( member( nextCellView listSoFar ) nil )
        ( t
          listSoFar =
            TrHierarchyTraversal(
              nextCellView
              cons( nextCellView listSoFar ) )
        )
      ) ; cond
    ) ; let
  ) ; foreach
  listSoFar;; return value
) ; procedure
```

The *cellView* parameter identifies the cellview to expand.

The *listSoFar* parameter identifies cellviews to avoid expanding.

Traversing a Hierarchical Layout Design

The *TrHierarchyTraversal* function is recursive.

The *cellView* parameter identifies the cellview to expand. The *listSoFar* parameter identifies cellviews to avoid expanding.

For each master at this level of the hierarchy, you compute the next cellview to expand recursively. There are three outcomes:

- There is no appropriate cellview. This can occur if the master is not in the library. You do nothing.
- The appropriate cellview has already expanded. You can tell by checking if it is an element of the *listSoFar*. You do nothing with this master.
- There is an appropriate cellview and you haven't expanded it yet. You add it to *listSoFar* and recursively expand it.

You accumulate all the expansions underneath each master in *listSoFar* and return *listSoFar*.

When calling *TrHierarchyTraversal*, pass *nil* for the *listSoFar*.

```
expansion = TrHierarchyTraversal(  
    dbOpenCellViewByType( "master" "mux2" "layout" ) nil )
```

Using *dbOpenCellViewByType* to Switch Views

You can pass a list of view names to the *dbOpenCellViewByType* function.

```
dbOpenCellViewByType(  
  "Example"  
  "nand2"  
  '( "silos" "schematic" )  
)  
⇒ db:12345678
```

In the example above, the *dbOpenCellViewByType* function performs one of the following tasks:

- If a *nand2 silos* cellview exists in the *Example* library, then the function opens the cellview and returns the database object.
- If a *nand2 schematic* cellview exists in the *Example* library, then the function opens the cellview and returns the database object.

Otherwise, the function returns *nil*.

Using *dbOpenCellViewByType* to Switch Views

A List of Schematic Views

The *TrSwitch* function passes a switch list to the *dbOpenCellViewByType* function. The *TrSwitch* function returns a *schematic* view if you pass a *symbol* cellview to it.

```
procedure( TrSwitch( masterCellView )
  if( !masterCellView
  then
    nil
  else
    case( masterCellView~>cellViewType
      ( "schematicSymbol"
        dbOpenCellViewByType(
          masterCellView~>libName
          masterCellView~>cellName
          '( "schematic" "cmos.sch" ) ;;; SWITCH LIST
        )
      )
      ( "maskLayout"
        masterCellView )
      ( t nil )
    ) ; case
  ) ; if
) ; procedure
```

A List of Schematic Views

Traversing a Hierarchical Schematic Design

```
procedure( TrHierarchyTraversal( cellView listSoFar )
  foreach( master cellView~>instances~>master
    let( ( nextCellView )
      nextCellView = TrSwitch( master )
      cond(
        ( !nextCellView nil )
        ( member( nextCellView listSoFar ) nil )
        ( t
          listSoFar =
            TrHierarchyTraversal(
              nextCellView
              cons( nextCellView listSoFar ) )
        )
      ) ; cond
    ) ; let
  ) ; foreach
  listSoFar;; return value
) ; procedure
```

The *cellView* parameter identifies the cellview to expand.

The *listSoFar* parameter identifies cellviews to avoid expanding.

Traversing a Hierarchical Schematic Design

Lab Overview

Lab 13-4 Exploring Hierarchy

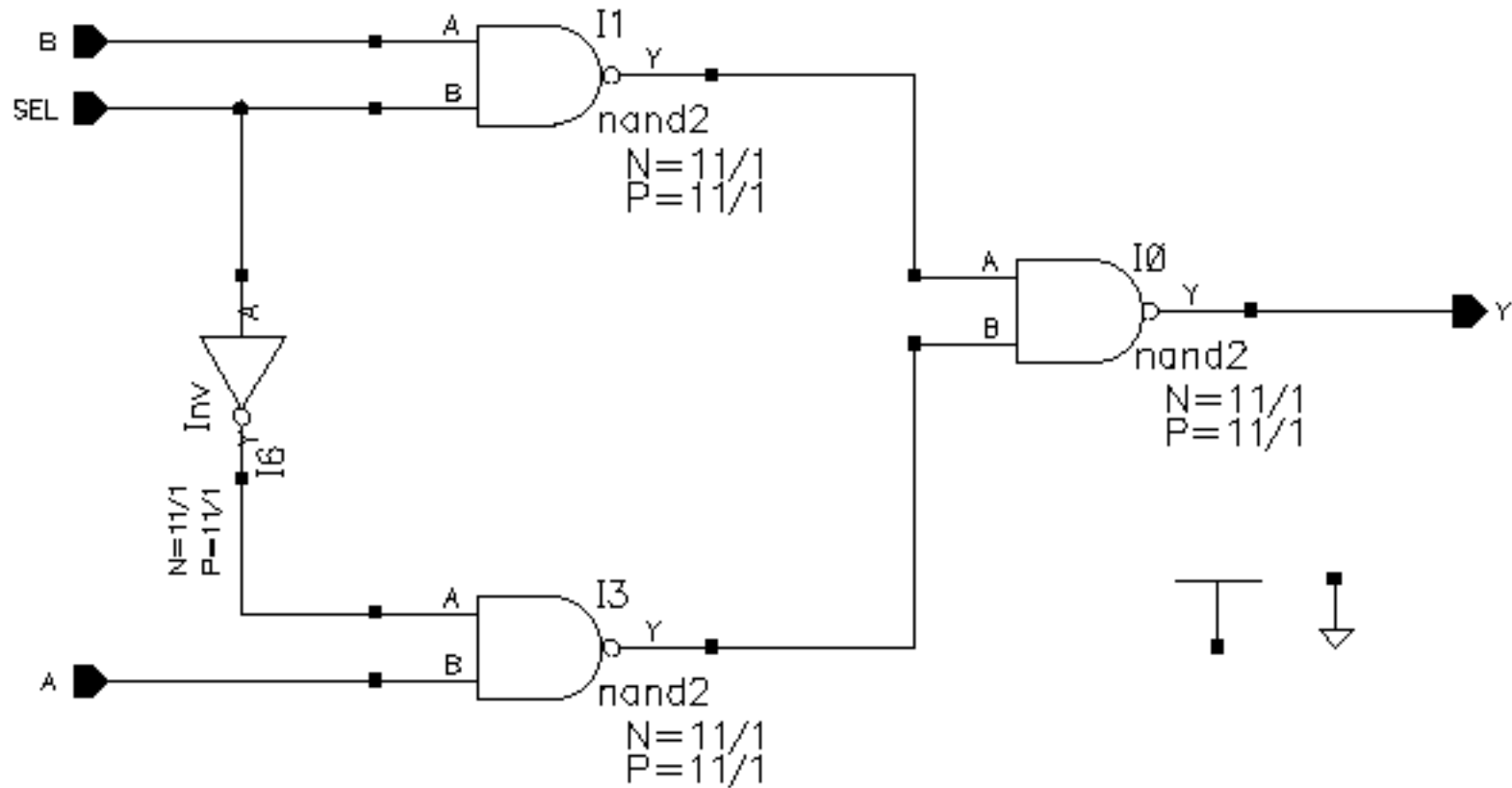
Lab 13-5 Traversing a Hierarchical Design

You run the *TrHierarchyTraversal* function on *master mux2 layout* and turn on tracing to observe the recursion.

Labs

Connectivity

Example: net *SEL* in *master mux2* schematic



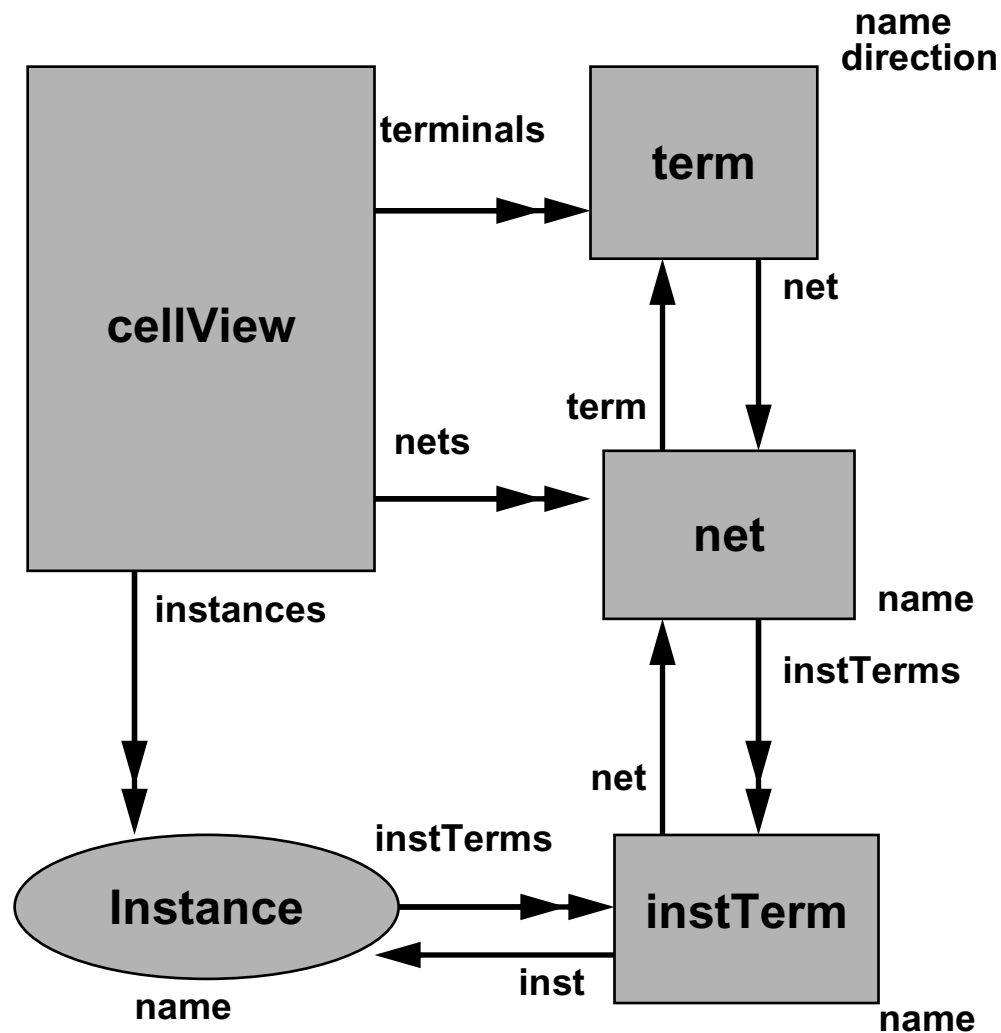
External nets connect a terminal with one or more instance terminals.

Internal nets connect two or more instance terminals.

Each instance terminal is on one instance.

Connectivity

The *term*, *net*, and *instTerm* Object Types



The *term*, *net*, and *instTerm* Object Types

Each *cellView* database object has

- A *terminals* attribute, whose value is a list of all *term* database objects in this design.
- A *nets* attribute, whose value is a list of all *net* database objects in this design.

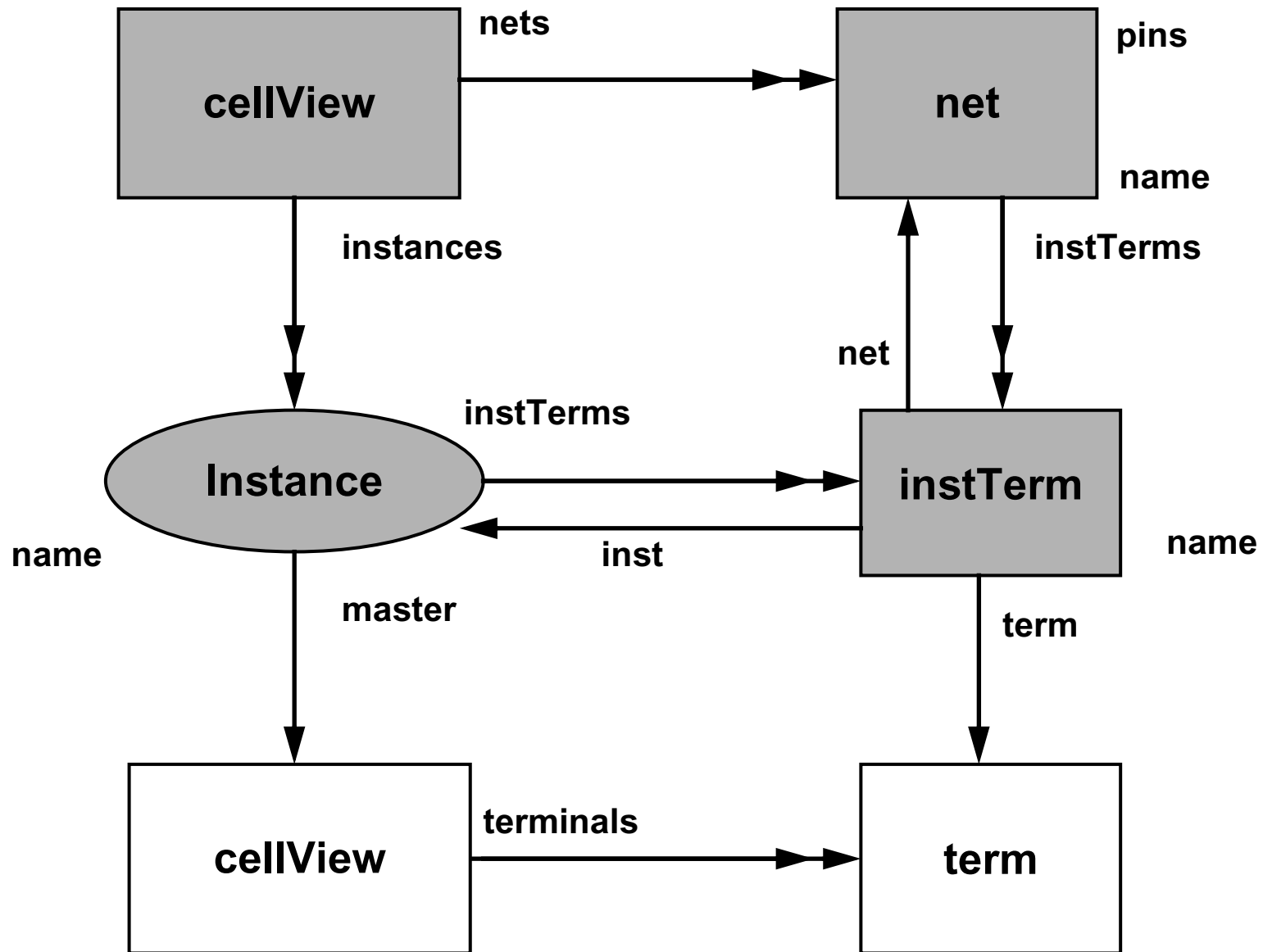
A net runs between one or more internal instance terminals. Each instance terminal is on a unique instance. In addition, a net can connect to the outside world by means of a terminal.

Each *net* database object has

- A *term* attribute, whose value is the unique *term* database object for this net.
- An *instTerms* attribute, whose value is a list of *instTerm* database objects.

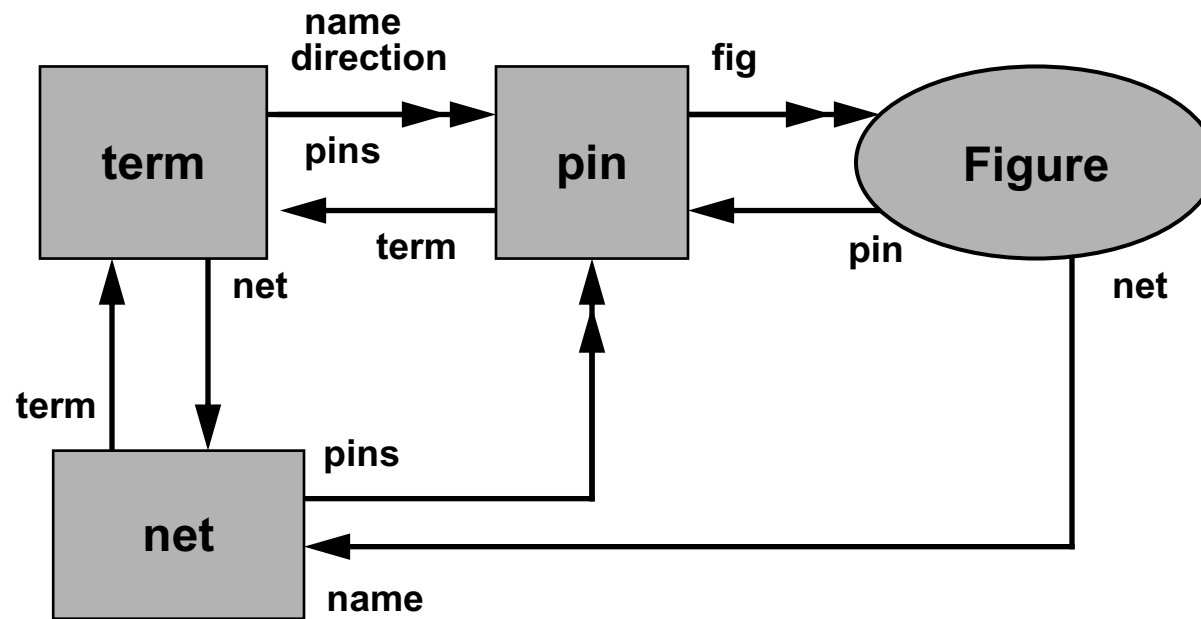
Each *instTerm* object has an *inst* attribute, whose value indicates the instance.

Connectivity Between Instances and Masters



Connectivity Between Instances and Masters

Figures and Terminals



For a *schematic* cellView, *Figures* associated with *term* objects are *Instances* whose *purpose* attribute is "*pin*".

For a *maskLayout* cellView, *Figures* associated with *term* objects are *Shapes*.

This example determines what type of figure is associated with the terminal pin(s).

```
cv~>terminals~>pins~>fig~>objType
```

Figures and Terminals

Every *Instance* database object has a *purpose* attribute. This attribute helps you determine the role of the instance in the design.

Lab Overview

Lab 13-6 Developing a Netlister

You develop a simple net-based netlister.

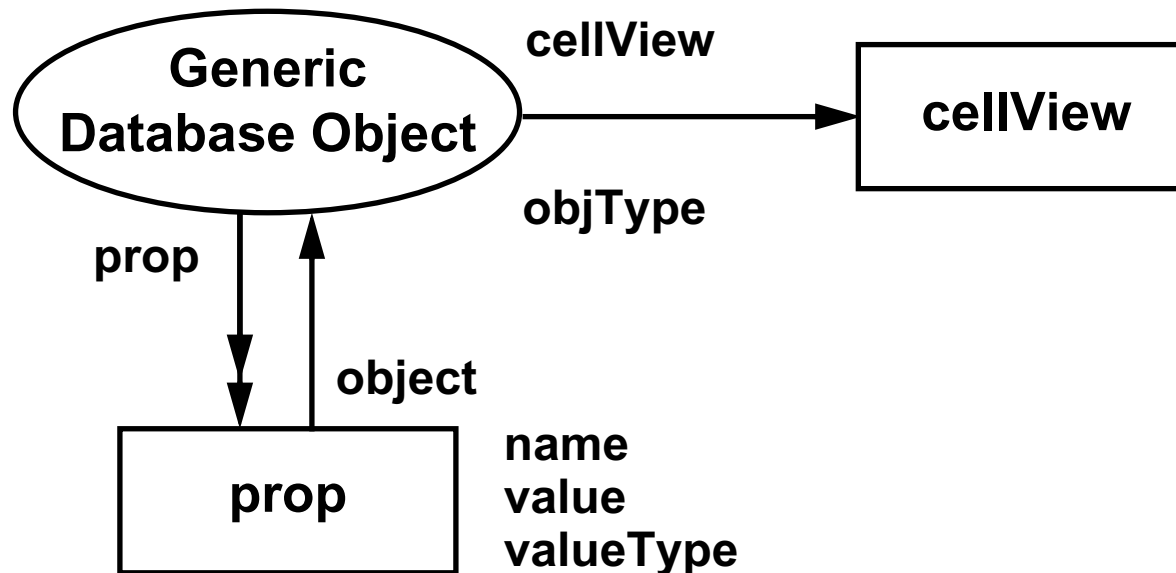
Lab 13-7 Developing an Instance-Based Netlister

You develop a simple instance-based netlister.

Lab Overview

Generic Database Object

Every database object is a kind of *Generic Database Object*. Each specific database object has three attributes: *objType*, *cellView*, and *prop*.



Attribute	Meaning
objType	The specific object type: " <i>net</i> ", " <i>line</i> ", etc.
cellView	The cellView database object containing the database object.
prop	0, 1 or more prop database objects.

Generic Database Object

The value of the *objType* attribute is a text string that indicates the type of the database object.

The value of the *prop* attribute is a list, possibly *nil*, of *prop* database objects. Each such object represents a user-defined property on the database object.

User-Defined Properties

Applications are responsible for creating, updating, and interpreting user-defined properties.

CDBA represents a user-defined property by a database object with object type "*prop*".

Every database object, except *prop* objects themselves, can have one or more user-defined properties.

CDBA stores all the properties on an object as the value of the *prop* attribute.

Example

Assume that window 5 is displaying *master mux2 schematic*.

```
cv = geGetWindowCellView( window(5) ) => db:18253868  
cv~>prop => (db:22856044 ... )  
cv~>prop~>objType => ( "prop" ... )
```

User-Defined Properties

The CDBA manages the *prop* attribute but does not manage any specific user-defined properties. Applications are responsible for managing their own user-defined properties.

Examining User-Defined Properties

Use the `~>prop` syntax to retrieve a list of *prop* database objects.

Use the `~>prop~>name` syntax to retrieve a list of property names.

Use the `~>prop~>value` syntax to retrieve a list of property values.

Example

```
cv~>prop~>name => ( "net#" ... )  
cv~>prop~>value => ( 22 ... )  
cv~>prop~>valueType => ( "int" ... )
```

Use the user-defined property name on the right side of the `~>` operator to retrieve a specific property value. The user-defined property name must satisfy the syntax for a SKILL symbol. Include the name in quotes if the name includes illegal characters.

```
cv~>"net#" => 22
```

Given the property name, use the *dbFindProp* name to retrieve the *prop* database object.

```
dbFindProp( cv "net#" )~>valueType => "int"
```

Examining User-Defined Properties

Use a *foreach mapcar* expression to retrieve a list of property name and property value pairs.

```
foreach( mapcar prop cv~>prop
  list( prop~>name prop~>value )
); foreach
=>

(( "net#" 22)
  ("schGeometryLastRecorded" "Oct  2 08:43:49 1995")
  ("lastSchematicExtraction" "Oct  2 08:43:49 1995")
  ("schXtrVersion" "sch.10.0")
  ("instance#" 10)
  ("instancesLastChanged" "Oct  2 08:43:48 1995")
  ("schGeometryVersion" "sch.ds.gm.1.4")
)
```

Creating User-Defined Properties

You can use the `~>` operator to create a user-defined property. The *valueType* is derived from the SKILL data type of the property value you assign.

```
cv~>TrStringProp = "high"
dbFindProp( cv "TrStringProp" )~>valueType => "string"
cv~>TrFloatProp = .5
dbFindProp( cv "TrFloatProp" )~>valueType => "float"
cv~>TrILListProp = '(1 2)
dbFindProp( cv "TrILListProp" )~>valueType => "ILList"
```

Use the appropriate *dbCreate.*Prop* SKILL function to

- Specify a *valueType* as *"time"*.
- Create enumerated, range, or hierarchical user-defined properties.

Creating User-Defined Properties

See the Cadence online documentation for the list of allowed property *valueTypes*. Refer to the *Description of Database Objects* section of Chapter 2 of the *Design Framework II SKILL Functions Reference Manual*.

Hierarchical User-Defined Properties

Use a hierarchical property to represent structured data.

Use the *dbCreateHierProp* Function to create a hierarchical property.

Use the *~>* operator to create properties associated with the hierarchical property. Each such associated property represents a slot in the structured data.

The *prop* attribute of a hierarchical property is always *nil*. The *value* attribute lists the property objects associated with a hierarchical property.

Example

```
hp = dbCreateHierProp( cv "TrHierProp" ) => db:39320456
hp~>a = 1
hp~>b = 2
hp~>value => (db:39320544 db:39320504)
hp~>value~>object => (db:39320456 db:39320456)
hp~>prop => nil
```

Hierarchical User-Defined Properties

The *dbCreateHierProp* Function

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

The *dbReplaceHierProp* Function

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

Other Kinds of User-Defined Properties

CDBA supports the following kinds of user-defined properties:

- Enumerated user-defined properties
- Range user-defined properties

Other Kinds of User-Defined Properties

See the Cadence online documentation to read more about these functions.

- *dbCreateEnumProp*
- *dbReplaceEnumProp*
- *dbCreateRangeProp*
- *dbReplaceRangeProp*

Retrieving All Attributes of a Database Object

The `~>?` expression returns a list of all the attribute names of a database object.

```
cv~>? =>  
  (cellView objType prop bBox cellName ... )
```

The `~>??` expression returns a list of attribute names and values.

```
cv~>?? =>  
  (db:18253868  
    cellView db:18253868  
    objType "cellView"  
    prop ( db:22972580 db:2297508 ... )  
    bBox ((-3.500000 -0.218750) (2.275000 0.337500))  
    ... )
```

Retrieving All Attributes of a Database Object

The `~>??` expression builds a list that starts with the database object in question, followed by alternating name value pairs.

This list is an example of a disembodied property list. Such lists usually start with *nil*, but the list you build with the `~>??` expression starts with the database object in question.

The Show File Browser

Displays the attribute names and values for a database object in a Show File window.

You can perform the following tasks:

- Browse any displayed database objects.
- Select associated Figures in a design window.
- Probe associated Nets in the design window.
- Raise parent and children browsers.

Example SKILL Function

```
TrShowFileBrowser( geGetWindowCellView() )  
TrShowFileBrowser( car( geGetSelSet() ) )
```

The Show File Browser is only available as part of this course.

The Show File Browser

Here is the code to start the Show File Browser with a bindkey.

```
hiSetBindKey(  
  "Schematics" "Ctrl<Btn2Down>"  
  "{ mouseSingleSelectPt() TrBrowseObject() }" )  
  
hiSetBindKey(  
  "Layout" "Ctrl<Btn2Down>"  
  "{ mouseSingleSelectPt() TrBrowseObject() }" )  
  
procedure( TrBrowseObject( )  
  TrShowFileBrowser(  
    car( geGetSelSet( ) ) || geGetWindowCellView()  
  )  
) ; procedure
```

Both *geGetSelSet* and *geGetWindowCellView* functions default to the current window. We assume that the *TrBrowseObject* function is called with the bindkey mechanism, so that the design window is the current window.

To browse any database object showing in a Browser window, such as "*net12345678*", do the following:

- Double click over "*net12345678*" with the mouse.
- Select the Browse Selection menu item.

Lab Overview

Lab 13-8 Exploring User-Defined Properties

Lab 13-9 Dumping Database Objects

Lab 13-10 Building the Cellview Data Model Road Map

Lab Overview

Module Summary

In this module, we

- Reviewed database object concepts.
- Opened a cellview into virtual memory.
- Created shapes in a design.
- Surveyed the cellview database model.
 - Geometry
 - Hierarchy
 - Connectivity
 - User-defined properties
- Traversed a hierarchical database.
- Retrieved all the attributes on a database object.

Module Summary

Module 14: User Interface

Objectives

- Context sensitive pop-up menus
- Fixed menus
- Dialog boxes
- List boxes
- Forms

Terms and Definitions

Context Sensitive Pop-up Menus

The *TrWindowsPopUp* function creates and displays a pop-up menu that lists the Design Framework II windows currently on the desktop. The menu choice becomes the top window.

```
procedure( TrWindowsPopUp()  
  hiDisplayMenu(  
    hiCreateSimpleMenu(  
      'TrWindowsPopUpMenu  
      "Windows"  
      foreach( mapcar wid hiGetWindowList()  
        hiGetWindowName( wid )  
      ) ; foreach  
      foreach( mapcar wid hiGetWindowList()  
        sprintf(  
          nil  
          "hiRaiseWindow( window( %d ))"  
          wid->windowNum  
        )  
      ) ; foreach  
    ) ; hiCreateSimpleMenu  
  ) ; hiDisplayMenu  
) ; procedure  
hiSetBindKey( "Command Interpreter"  
  "<Key>F6" "TrWindowsPopUp()" )
```

Context Sensitive Pop-Up Menus

You can create menus dynamically at the time you invoke the *hiDisplayMenu* function. Such menus can be context sensitive. The callback for each menu item can refer to

- Literal data
- Global variables

```
hiDisplayMenu( hiCreateSimpleMenu( ... ) )
```

The *sprintf* Function

Use the *sprintf* function to dynamically create the callback for each menu item.

The *sort* Function

You can use the *sort* function to sort the window list. Use *TrGetSortedWindowList* in place of *hiGetWindowList* in the *TrWindowPopUp* function declaration.

```
procedure( TrCompareWindowTitles( w1 w2 )
  alphalessp( hiGetWindowName(w1) hiGetWindowName(w2))
) ; procedure

procedure( TrGetSortedWindowList()
  sort( hiGetWindowList() 'TrCompareWindowTitles' )
) ; procedure
```

Fixed Menus

Fixed menus have the following characteristics:

- They typically occupy the entire top or side of the screen.
- They include a two dimensional array of menu items.
- They include **Done** as the last menu item.

Clicking a menu item in a fixed menu does not set the current window.

Fixed Menus

Creating Fixed Menus

Create the menu items using the *hiCreateMenuItem* function.

```
TrMenuItem1 = hiCreateMenuItem(  
    ?name          'TrMenuItem1  
    ?itemText      "One"  
    ?callback      "println( \"One\" )" )  
)  
TrMenuItem2 = hiCreateMenuItem(  
    ?name          'TrMenuItem2  
    ?itemText      "Two"  
    ?callback      "println( \"Two\" )" )  
)
```

Use the *hiCreateVerticalFixedMenu* or *hiCreateHorizontalFixedMenu* functions to create a fixed menu.

```
hiCreateVerticalFixedMenu(  
    'TrExampleVerticalFixedMenu  
    list( TrMenuItem1 TrMenuItem2 )  
    2      ;; number of rows  
    1      ;; number of columns  
)
```

Creating Fixed Menus

The *hiCreateVerticalFixedMenu* Function

The *hiCreateHorizontalFixedMenu* Function

Displaying Fixed Menus

Use the *hiDisplayFixedMenu* function to display a fixed menu.

```
hiDisplayFixedMenu(  
    TrExampleVerticalFixedMenu  
    "top"    ;;; one of "top", "bottom", "right", or "left"  
)
```

You can use the *hiFixedMenuDown* function to remove a fixed menu from the screen.

```
hiFixedMenuDown( TrExampleVerticalFixedMenu )
```

Displaying Fixed Menus

The *hiDisplayFixedMenu* Function

The *hiFixedMenuDown* Function

Attaching Fixed Menus to Application Windows

You can attach a vertical fixed menu to these types of windows:

- Graphics windows
- Text windows
- Form windows

To attach your vertical fixed menu, use the

- *hiCreateVerticalFixedMenu* function to create your menu.
- *hiAddFixedMenu* function to attach a fixed menu to a window. It replaces any attached fixed menu.

```
hiAddFixedMenu(  
    ?fixedMenu TrExampleVerticalFixedMenu  
    ?window window( 4 )  
)
```

By default, the menu appears on the left side of the application window.

The **Done** menu item does not appear.

Clicking a menu item sets the current window.

Attaching Fixed Menus to Application Windows

Cadence applications attach fixed menus that usually consist of icon items.

On the User Preferences form, the Show Fixed Menu Names field controls whether the *itemText* for an item is dynamically displayed.

Several functions manipulate attached, fixed menus.

The *hiAddFixedMenu* Function

The *hiGetWindowFixedMenu* Function

Use the *hiGetWindowFixedMenu* function to retrieve an attached, fixed menu.

Do not confuse the *hiGetWindowFixedMenu* function with either the *hiGetWindowMenu* function or the *hiGetBannerMenus* function.

```
wid = geOpen(  
    ?lib "master" ?cell "nand2" ?view "layout"  
    ?mode "r")  
attachedMenu = hiGetWindowFixedMenu( wid )  
type( attachedMenu ) => hiFixMenu
```

The *hiRemoveFixedMenu* Function

Use the *hiRemoveFixedMenu* function to remove an attached, fixed menu.

```
hiRemoveFixedMenu( wid ) => t
```

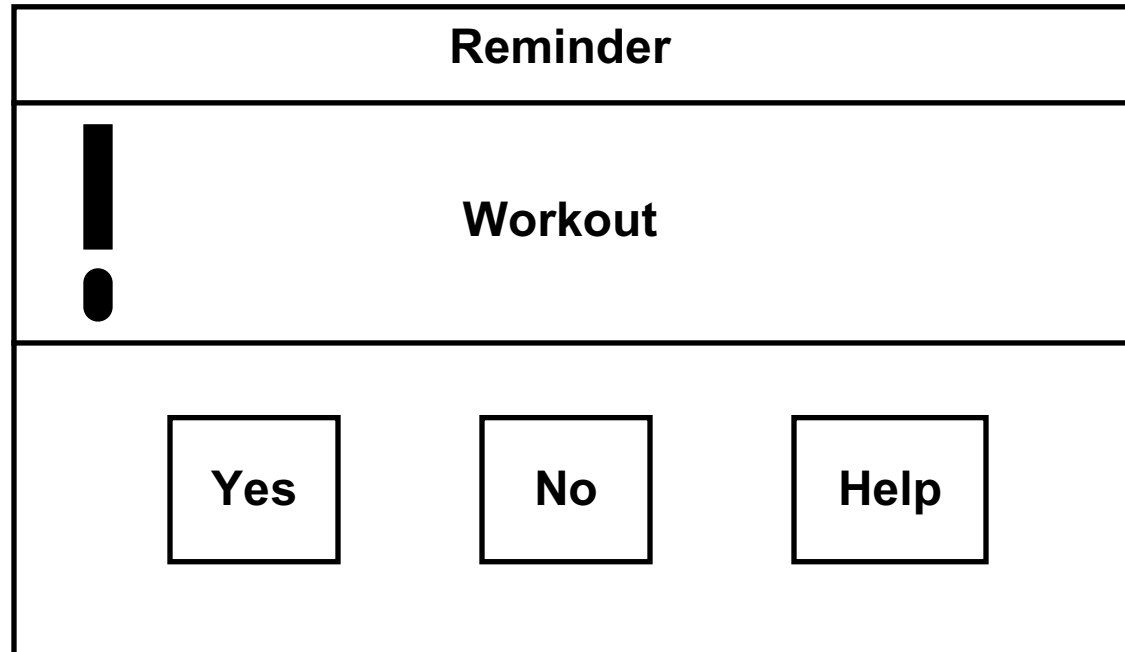
Lab Overview

Lab 14-1 Exploring Fixed Menus

- Create a vertical fixed menu.
- Attach the menu to an application window.

Lab Overview

Dialog Boxes



Clicking the **Yes** button

- Displays *Workout completed* in the CIW.
- Removes the dialog box.

Dialog Boxes

Dialog boxes are pop-up windows that display a message for the user.

Use dialog boxes to

- Ask for confirmation from the user.
- Display status messages bundled with an associated follow-up callback that the user can select.

There are three kinds of dialog boxes.

System Modal	Blocks any further action in any application until response. Use for confirmation.
Modal	Blocks any further action in the Design Framework II environment until response. Use for confirmation.
Modeless	Nonblocking. Useful for status messages or reminders.

Each dialog box has two or three buttons to allow the user to confirm an action or status and to dismiss or close the dialog box.

The *hiDisplayAppDBox* Function

The *hiDisplayAppDBox* function creates and displays a dialog box.

```
hiDisplayAppDBox(  
    ?name          gensym( 'TrReminderDialogBox )  
    ?dboxBanner    "Reminder"  
    ?dboxText       "Shop"  
    ?callback       "TrReminderCB( \"Shop\" )"  
    ?dialogType     hicWarningDialog  
    ?dialogStyle    'modeless  
    ?buttonLayout   'YesNo  
)  
  
procedure( TrReminderCB( reminderText )  
    printf( "%s completed" reminderText )  
) ; procedure
```

Create variable

**Modeless to allow
independent work**

The callback

The *?name* argument must be a unique global variable. The *hiDisplayAppDBox* function stores the data structure of the dialog box in this variable.

Use the *gensym* function to create a new global variable.

The *?callback* argument is a text string that contains the entire function call you want Design Framework II to execute when the user clicks OK or Yes.

The *hiDisplayAppDBox* Function

The *hiDisplayAppDBox* function creates and displays a dialog box. The dialog box is destroyed when it is removed from the screen.

Keyword	Argument	Meaning
---------	----------	---------

<i>?name</i>	The <i>dboxHandle</i> symbol is bound to the data structure of the dialog box and then reset to <i>nil</i> when box is dismissed.
<i>?dboxBanner</i>	Text that appears within the window manager banner text of the dialog box.
<i>?dboxText</i>	Message that appears in the dialog box.
<i>?callback</i>	Callback that executes whenever the user selects <i>OK</i> or <i>Yes</i> .
<i>?dialogType</i>	<i>hicWarningDialog</i> , <i>hicErrorDialog</i> etc. See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?dialogStyle</i>	Blocking behavior of this dialog box. Acceptable choices are <i>'systemModal</i> , <i>'modal</i> , or <i>'modeless</i> .
<i>?buttonLayout</i>	<i>'OKCancel</i> , <i>'YesNo</i> , <i>'YesNoCancel</i> , etc. See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?location</i>	See <i>User Interface SKILL Functions Reference Manual</i> .

The Dialog Box Blocking Behavior

You specify the blocking behavior with the *?dialogStyle* argument. The three choices are listed in this table.

Dialog Style	Blocking Behavior	<i>hiDisplayAppDBox</i> returns
<i>modeless</i>	Does not block any application.	Immediately.
<i>modal</i>	Blocks Design Framework II applications.	After box is dismissed.
<i>systemModal</i>	Blocks all applications in the system until the user responds to the dialog box.	After box is dismissed.

Caution

Do not rely on the delayed return of the *hiDisplayAppDBox* function in the case of *modal* and *systemModal* boxes. In some future release, this function will immediately return regardless of the dialog style.

The Dialog Box Blocking Behavior

The Dialog Box Callback and Button Layout

You specify the button layout with the *?buttonLayout* argument.

You specify a SKILL expression with the *?callback* argument.

Represent the SKILL expression as a text string. It is the entire function call, just like callbacks for menu items and bindkeys.

The following buttons trigger the callback.

Button Layout Argument	Button that Triggers Callback
<i>OKCancel</i>	OK
<i>YesNo</i>	Yes
<i>YesNoCancel</i>	Yes/No
<i>CloseHelp</i>	
<i>Close</i>	

The Dialog Box Callback and Button Layout

The following program displays modeless dialog boxes to verify the circumstances under which the system triggers the callback.

The source code is in the *~/SKILL/DialogBoxes/ButtonLayout.il* file.

```
TrButtonLayouts =
  '( OKCancel YesNo YesNoCancel CloseHelp Close )

procedure( TrDialogBoxes( )
  foreach( buttonLayout TrButtonLayouts
    hiDisplayAppDBox(
      ?name      gensym( 'TrDialogBox )
      ?dboxBanner "Test"
      ?dboxText  get_pname( buttonLayout )
      ?buttonLayout buttonLayout
      ?dialogType hicInformationDialog
      ?dialogStyle 'modeless
      ?callback
        sprintf( nil "TrDialogBoxCB( '%L )" buttonLayout )
    )
  ) ; foreach
) ; procedure

procedure( TrDialogBoxCB( arg )
  printf( "TrDialogBoxCB called: %L\n" arg )
) ; procedure
```

Lab Overview

Lab 14-2 Exploring Dialog Boxes

Lab Overview

List Boxes

Produce		
OK	Cancel	Apply
Apple Orange Kiwi		

Use a list box to display choices to the user.

List boxes display a list of user-defined strings in a scrollable window.

The window has a vertical and horizontal scrollbar and the size of the window is constant. You can select a string (or strings) using the left mouse button.

The application can use the string(s) that you select after the window is dismissed.

Both the **Cancel** and **OK** buttons remove the window from the screen; however, using the **Cancel** button ignores your selections.

List boxes do not block the user.

List Boxes

In this release, the *hiShowListBox* function does not return until the user clicks OK or Cancel. However, in a future release the *hiShowListBox* function might return immediately.

The *hiShowListBox* Function

The *hiShowListBox* function creates and displays a list box.

```
procedure( TrShowListBox( aList )
  hiShowListBox(
    ?name      gensym( 'TrExampleListBox )
    ?choices    aList
    ?callback   'TrExampleListBoxCB
    ?title      "Example List Box"
    ?multipleSelect  t
    ?applyButton    t
  )
) ; procedure

procedure( TrExampleListBoxCB( ok theListBox )
  if( ok then
    printf( "You choose:\n" )
    foreach( choice theListBox->value
      printf( "%15s\n" choice )
    ) ; foreach
  else printf( "You clicked Cancel.\n" )
  ) ; if
) ; procedure

TrShowListBox( '( "apple" "orange" "kiwi" ) )
```

The *hiShowListBox* Function

The *hiShowListBox* function creates and displays a list box containing a group of text strings.

Keyword Argument	Meaning
<i>?name</i>	A global variable in which SKILL stores data structure.
<i>?title</i>	The title.
<i>?choices</i>	List of text string choices.
<i>?callback</i>	Function called when the user clicks OK , Cancel , or Apply .
<i>?multipleSelect</i>	Controls single item selection or multiple item selection.
<i>?applyButton</i>	t => provides Apply button.

The *hiDisplayListBox* function is obsolete and will be removed in a future release. Avoid using it.

The List Box Callback

Write your callback function so that it accepts two arguments.

```
procedure( TrExampleListBoxCB( ok theListBox )
  ...
) ; procedure
```

Design Framework II triggers the callback function when the user clicks **OK**, **Cancel**, or **Apply**.

Design Framework II supplies two arguments.

User Action	Callback Arguments	hiShowListBox return value	List Box Screen Behavior
OK	<i>t</i> the list box data structure	<i>t</i>	Disappears.
Apply	<i>t</i> the list box data structure		Remains on screen.
Cancel	<i>nil</i> <i>nil</i>	<i>nil</i>	Disappears.

The List Box Callback

The callback can use the SKILL expressions in this table.

Expression	Meaning
<i>theListBox->value</i>	Retrieves the list of selected items.
<i>theListBox->choices</i>	Retrieves the list of items that the user can select.
<i>theListBox->choices = '(...)</i>	Changes the list of items that the user can select.

While the box is displayed, other SKILL functions can similarly manipulate the list box data structure through a global variable.

Lab Overview

Lab 14-3 Exploring List Boxes

Lab Overview

Standard Forms

Standard forms solicit data from the user to complete a command.

Forms contain a window banner with generic command buttons—**OK**, **Cancel**, **Defaults**, **Apply**, and **Help**.

You can define other fields and command buttons specific to your application.

The user interacts with a form by mouse and keyboard.

Open File

OK Cancel Defaults Help

Library Name master

Cell Name mux2

View Name layout

Mode ☒ edit ☐ read

Library path file

Cell Names

- mux2**
- mux2_connect
- Inv
- Inv_save
- nand2
- scratch
- test

Forms provide these generic command buttons automatically.

You define the fields and buttons specific to your application.

Standard Forms

Standard forms solicit data from the user.

You create each standard form with a banner containing the name of the form and any of the following buttons:

Button	Function
<i>OK</i>	Completes the command and removes the form from the screen.
<i>Cancel</i>	Cancels any changes or selections made to the form and removes the form from the screen.
<i>Defaults</i>	Sets all values in the form to their default values.
<i>Apply</i>	Completes the command and leaves the form on the screen.
<i>Help</i>	Displays a document containing additional information.

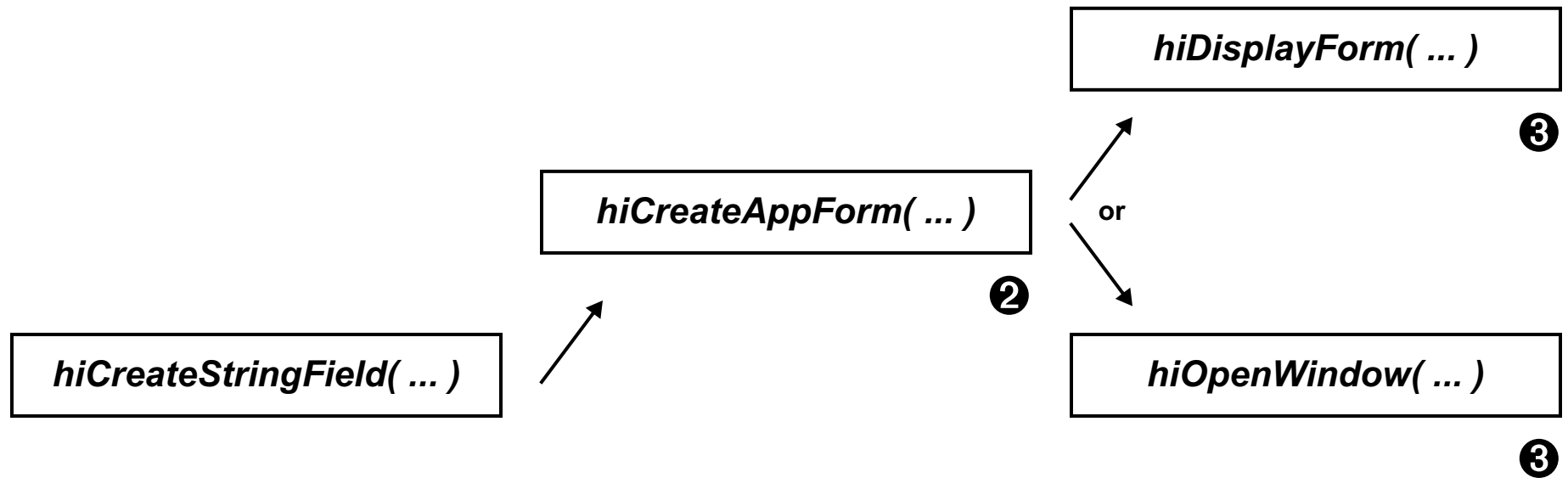
Analyzing a Design Framework II Command

The table describes how Design Framework II and SKILL respond to the user during a typical command that displays a form.

User	Design Framework II	SKILL
Chooses menu item.	Invokes menu item callback.	Menu item callback displays the form.
Fills in data.	Invokes field callback(s).	Field callbacks provide feedback and set other field values.
Clicks OK/Apply.	Invokes form callback.	Form callback completes the command.

Analyzing a Design Framework II Command

Creating and Displaying a Form



Create Forms from the Bottom Up

Follow these steps to create and display a form:

1. Use the *hiCreate** functions to create the fields.
2. Use the *hiCreateAppForm* function to create the form.
3. Use the *hiDisplayForm* function to display the form or use the *hiOpenWindow* function to install the form in a window.

Creating and Displaying a Form

The *hiCreateAppForm* Function

The *hiDisplayForm* Function

The *hiOpenWindow* Function

Example File Form

This example shows a form with a single string field.

File Form				
OK	Cancel	Defaults	Apply	Help
File Name		<input type="text" value=".cshrc"/>		

- The user specifies a file name in the *File Name* field.
- The form callback checks whether the file exists. If the file exists, the form callback displays it in a Show File window.

Example File Form

Creating the File Name Field

Pass the data structure of the File Form to the validation routine of the File Name field.

Thus, the validation routine can access other fields in the form.

Use the *hiGetCurrentForm* function to retrieve the data structure of the File Form.

```
TrFileNameField = hiCreateStringField(  
    ?name           'TrFileNameField           ← TrFileNameField  
    ?prompt         "File Name"  
    ?defValue       ".cshrc"  
    ?callback       "TrDoFieldCheckCB( hiGetCurrentForm() )"  
    ?editable       t  
)  
  
procedure( TrDoFieldCheckCB( theForm )  
    if( isFile( theForm->TrFileNameField->value ) ← TrFileNameField  
        ...  
    ) ; procedure
```

Creating the File Name Field

The *hiCreateStringField* Function

Creates a string field entry for a form.

Keyword Argument	Meaning
<i>?name</i>	Symbolic name of this field
<i>?prompt</i>	Field prompt
<i>?value</i>	Initial value
<i>?help</i>	See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?defValue</i>	Default value
<i>?font</i>	See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?callback</i>	The function call(s) to execute when the user clicks into another field.
<i>?format</i>	See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?editable</i>	<i>nil</i> => read-only <i>t</i> => user can edit this field.

Validating the File Name Field

The *TrDoFieldCheckCB* function returns *t* if the file exists and returns *nil* otherwise.

The *theForm* argument is the data structure that the *hiGetCurrentForm* function returns.

TrFileNameField is the symbol you passed to the *hiCreateStringField* function.

```
procedure( TrDoFieldCheckCB( theForm )
  if( isFile( theForm->TrFileNameField->value ) then
    println("File exists")
    t
  else
    println("File Does Not Exist--Try Again")
    nil
  ) ;if
) ; procedure
```

Validating the File Name Field

User actions trigger the execution of the field validation routines. They execute after the field and form creation routines have returned.

Do not rely on passing local variables to either of the following:

- Your form field checking routines
- Your form callback routine

However, you can use global variables to communicate with the field checking routines as well as with the form callback routine.

Creating the File Form

Use the *hiCreateAppForm* function to create a form data structure.

```
TrFileForm = hiCreateAppForm(  
    ?name           'TrFileForm  
    ?formTitle      "File Form"  
    ?callback       'TrFileFormCB  
    ?fields         list( TrFileNameField )  
    ?help           " "  
    ?unmapAfterCB   t  
)
```

Make the *?name* argument be a global variable. This variable plays these two roles:

- The form data structure contains a reference to the *?name* parameter.
- The *hiCreateAppForm* function stores the data structure as the value of the *?name* parameter.

Creating the File Form

The *hiCreateAppForm* Function

Returns the SKILL data structure of a form with the specified field entries.

Keyword Argument	Meaning
<i>?name</i>	A global variable. SKILL stores the form's data structure in this variable.
<i>?fields</i>	A list of the field data structures.
<i>?formTitle</i>	The form name, which appears in the form's window banner.
<i>?callback</i>	The function to be called when the user selects OK or Apply in the form.
<i>?unmapAfterCB</i>	If <i>t</i> , SKILL removes the form after the callback returns. Otherwise, the form is removed immediately when the user clicks OK . See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?formType</i>	See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?dialogStyle</i>	See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?buttonLayout</i>	See <i>User Interface SKILL Functions Reference Manual</i> .
<i>?initialSize</i>	See <i>User Interface SKILL Functions Reference Manual</i> .

The File Form OK/Apply Callback

Make sure that the form callback does the following:

- Performs all the field checks again.
- Continues the command.

```
procedure( TrFileFormCB( theForm )  
  if( TrDoFieldCheckCB( theForm ) then  
    hiSetCallbackStatus( theForm t )  
    hiHighlightField(  
      theForm  
      'TrFileNameField  
      'background )  
    view(  
      theForm->TrFileNameField->value )  
  else  
    hiSetCallbackStatus( theForm nil )  
    hiHighlightField(  
      theForm 'TrFileNameField 'error )  
  ) ; if  
) ; procedure
```

Validate field

Clear unmap veto

Form data structure

Field symbol

Highlight type

Veto form unmap

The File Form OK/Apply Callback

To facilitate a user-friendly interface, make your callback validate the fields before continuing the command.

If there are any fields with errors, then

- Highlight the fields with errors.
- Prevent the removal of the form from the screen.
- Do not continue the command.

If all fields contain valid data, then

- Remove any field highlighting.
- Continue the command.

The *?unmapAfterCB* Argument

If you create the form with *?unmapAfterCB* set to *t*, the form stays on the screen until the form callback returns.

The *hiSetCallbackStatus* Function

When you set the callback status to *nil*, the form stays on the screen. To permit the form to be removed later, set the callback status to *t*.

The *hiHighlightField* Function

Displaying the File Form

Button	Callback Evaluated	<i>hiDisplayForm</i> Return Value	Form Removed from Screen
OK	Yes	<i>t</i>	Yes
Apply	Yes	<i>does not return</i>	No
Cancel	No	<i>nil</i>	Yes

Use the *hiDisplayForm* function to display a form you create with the *hiCreateAppForm* function.

By default, the *hiDisplayForm* function does not return until the user clicks OK or clicks Cancel.

Displaying the File Form

The *hiDisplayForm* Function

By default, the *hiDisplayForm* function does not return until the user clicks **OK** or clicks **Cancel**.

The *?dontBlock* Argument

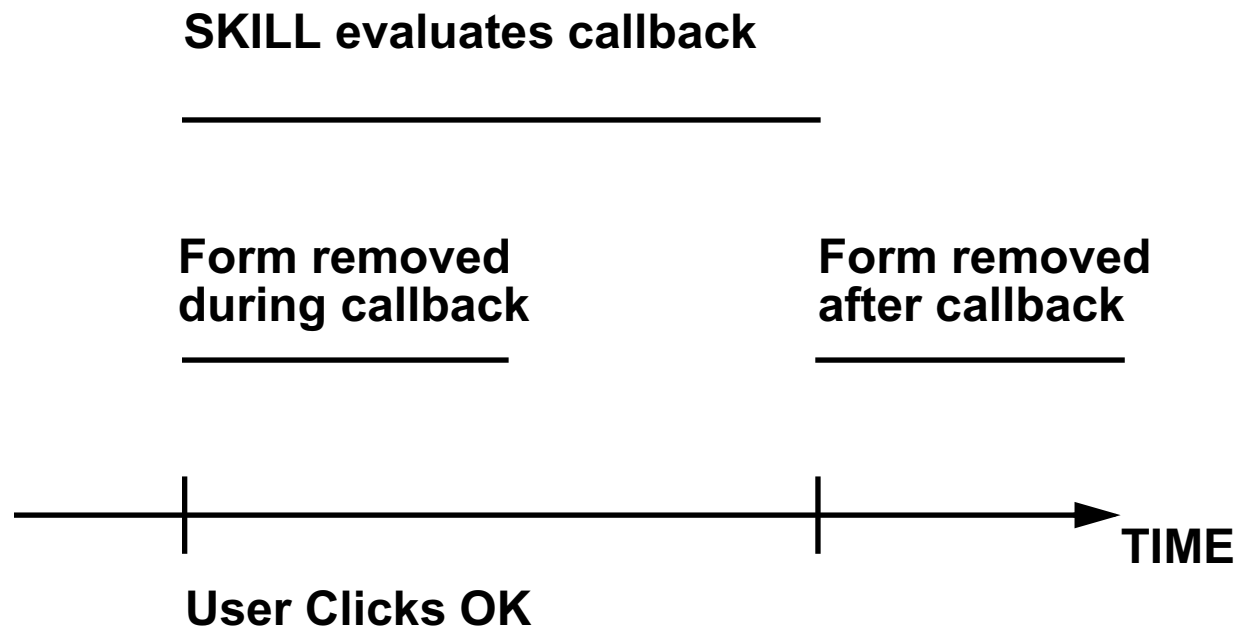
If you pass *?dontBlock t* when you create a form, then the *hiDisplayForm* function returns immediately.

The *?unmapAfterCB* Argument

The *?unmapAfterCB* argument governs when the X Window System removes the form from the screen.

You have two opportunities for the X Window System to remove the form:

- While SKILL evaluates the callback.
- After SKILL evaluates the callback.



The *?unmapAfterCB* Argument

Controlling the Cursor Icon

You can control the cursor icon that appears to let the user know the state of the process that is running. For example, if you change the cursor to an hourglass icon this will let the user know to wait for the process to complete.

Get the cursor icon value

hiGetCursor() => integer ID of the cursor type in the current window

Set the cursor icon value

hiSetCursor(wid x_cursor) => *t* if the cursor was set, *nil* if it was not set

Example:

```
hiSetCursor( hiGetCurrentWindow() `hicHourglass)
```

Controlling the Cursor Icon

x_cursor Constant	X Windows Equivalent
hicArrow	XC_left_ptr
hicCross	XC_cross
hicHand	XC_hand2
hicHelp	XC_question_arrow
hicIbeam	XC_xterm
hicNo	XC_pirate
hicSizeAll	XC_fleur
hicSizeNESW	—
hicSizeNS	XC_sb_v_double_arrow
hicSizeNWSE	—
hicSizeWE	XC_sb_h_double_arrow
hicUpArrow	XC_sb_up_arrow
hicWait	XC_watch
hicArrowHourglass	—
hicHourglass	—
hicNoCursor	—

Frequently Asked Questions

What role do the *f1*, *f2*, *f3*, *fields*, *form*, and *ExampleForm* variables play?

Which of these variables must be global?

Which can be local?

```
let( ( f1 f2 f3 fields form )  
  f1 = hiCreateStringField(  
    ?name 'field1  
    ...  
  )  
  f2 = hiCreateStringField(  
    ?name 'field2  
    ...  
  )  
  ...  
  fields = list( f1 f2 f3 )  
  form = hiCreateAppForm(  
    ?name      'ExampleForm  
    ?fields    fields  
    ...  
  )  
  hiDisplayForm( form )  
) ; let
```

Field data structure
Symbolic field name

Field data structure
Symbolic field name

Form data structure
MUST be global

Form data structure

Frequently Asked Questions

This table explains the role and the scope of each variable in the overhead example.

Variables	Role	Scope (local or global)
<i>f1, f2, f3</i>	Each value is a field data structure.	Make these variables local.
<i>field1, field2, field3</i>	Each is the symbolic name of a field. Each value is irrelevant.	The scope is irrelevant, because the variable name, instead of the value, is used.
<i>fields</i>	The value is a list of the field data structures.	Make this variable local.
<i>form</i>	The value is a form data structure.	Make this variable local.
<i>ExampleForm</i>	The symbolic name of the form. The value is a form data structure.	Make this variable global.

Form Fields

All field creation routines share similar arguments and return the field structure.

Field Category	Field Type	Creation Function
Type-in	single-line string	hiCreateStringField
	multi-line string	hiCreateMLTextField
	integer numeric	hiCreateIntField
	floating point numeric	hiCreateFloatField
	list	hiCreateListField
	2-D point	hiCreatePointField
	bounding box	hiCreateBBoxField
Enumerated Choice	2-D point list	hiCreatePointListField
	toggle	hiCreateToggleField
	boolean	hiCreateBooleanButton
	radio	hiCreateRadioField
	list box	hiCreateListBoxField
	cyclic	hiCreateCyclicField
	layer cyclic	hiCreateLayerCyclicField
Command	scale	hiCreateScaleField
	button	hiCreateButton
	button box	hiCreateButtonBoxField

Form Fields

Use the Cadence Finder to list all the functions whose name starts with *hiCreate* and end with *Field*.

Lab Overview

Lab 14-4 Exploring Forms

Lab 14-5 Creating a Form

- You create a form capable of changing the name and size of an associated window.
- Your form title displays the window ID. There are three fields whose default values reflect the current state of the window.
 - Name field
 - Height field
 - Width field
- Click on **Defaults** or **Apply** to restore the window to its original condition.
- This lab has several optional sections for advanced students.

Lab Overview

Module Summary

In this module, we covered

- Context sensitive pop-up menus
- Fixed menus
- Dialog boxes
- List boxes
- Forms

Module Summary

Category	Functions
Fixed menus	<i>hiCreateVerticalFixedMenu</i> <i>hiCreateHorizontalFixedMenu</i> <i>hiDisplayFixedMenu</i> <i>hiGetWindowFixedMenu</i> <i>hiRemoveFixedMenu</i> <i>hiAddFixedMenu</i>
Dialog boxes	<i>hiDisplayAppDBox</i>
List boxes	<i>hiShowListBox</i>
Forms	<i>hiCreateStringField</i> <i>hiCreateAppForm</i> <i>hiDisplayForm</i> <i>hiSetCallbackStatus</i> <i>hiHighlightField</i>

Module 15: Advanced Customization

Objectives

- Customize the Composer software and the Virtuoso Layout Editor.
 - Add a menu item to a pull-down menu.
 - Insert a menu item in a pull-down menu.
 - Delete a menu item from a pull-down menu.
 - Add a pull-down menu to the menu banner.
 - Reorganize the menus in the menu banner.
- Customize form field default values.
- Manage the Layer Selection Window

Terms and Definitions

Module Scope

This module covers the following topics:

- Customizing Composer, Virtuoso Layout Editor, and other Design Editor applications.
- Customizing the default value and initial value of a statically defined field in a SKILL form.

You need different techniques to customize the CIW and the LSW.

Module Scope

This module focuses on one of several approaches to customizing certain Design Framework II applications, such as the Composer and the Virtuoso Layout Editor applications.

Different techniques are required for customizing the CIW and the LSW.

Design Editor Applications

The Design Editor is a software interface that associates application software with view types. The Design Editor is not itself a tool.

The Composer and Layout Editor software are Design Editor applications.

All Design Editor applications register certain of their SKILL functions with the Design Editor. These application SKILL functions are known as Design Editor trigger functions.

The Design Editor calls these registered SKILL functions to configure the design window when the user does any of the following tasks:

- Opens a cellview.
- Descends the design hierarchy into a cellview.
- Ascends the design hierarchy into a cellview.

The view type of the cellview determines which trigger functions the Design Editor invokes.

Cadence supplies the basic set of Design Editor trigger functions for each supported view type.

You can define several user trigger functions for each supported view type.

Design Editor Applications

The *deGetAllViewTypes* Function

The *deGetAllViewTypes* function returns a list of all registered Design Editor view types.

```
deGetAllViewTypes() =>  
  ("graphic" "maskLayout" "schematic" "schematicSymbol" ... )
```

The *deGetAppInfo* Function

The *deGetAppInfo* function returns the information associated with the application registered for a *viewType*. You can extract specific trigger information with the *->* operator.

```
deGetAppInfo( "schematic" )->menuTrigger  
=> schMenuTrigger
```

Design Editor User Trigger Functions

This course presents a user trigger function known as the user postinstall trigger function.

The phrase *user postinstall trigger* function implies the following:

- You define the user postinstall trigger function.
Cadence does not define it.
- The Design Editor invokes your function after it configures a design window.

To define a user postinstall trigger function in your *.cdsinit* file, do the following:

1. Declare the user trigger function.
2. Register your trigger function with the Design Editor.

Design Editor User Trigger Functions

The Design Editor User Postinstall Trigger

Declare a user postinstall trigger function to accept ONE argument as in this example.

```
procedure( TrUserPostInstallTrigger( args )  
    ...  
    ) ; procedure
```

The single argument to any Design Editor trigger function is a disembodied property list that includes the following fields:

```
args->window  
args->libId  
args->libName  
args->cellName  
args->viewName  
args->accessMode  
args->action
```

You must use the -> operator to access the fields. Do not use the ~> operator.

Declaring a User Postinstall Trigger Function

Registering a User Postinstall Trigger Function

Use the *deRegUserTriggers* function to register your user postinstall trigger as in this example.

```
deRegUserTriggers( "schematic"  
    nil          ;;; no user application trigger  
    nil          ;;; no user menu trigger  
    'TrUserPostInstallTrigger  
)
```

- The first argument is the view type associated with the application.
Composer = *"schematic"*
Layout Editor = *"maskLayout"*
- Pass *nil* for the second and third arguments.
- The fourth argument is the function that you want for the user postinstall trigger. Use the single quote syntax.

Registering a User Postinstall Trigger Function

The *deRegUserTriggers* Function

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

Developing a User Postinstall Trigger Function

Develop a user postinstall trigger function the same way you develop any SKILL function.

Be aware that the Design Editor traps any SKILL errors that a user postinstall trigger function causes.

If you are using the SKILL Debugger, configure it to display a stack trace automatically.

Developing a User Postinstall Trigger Function

Debugging a User Postinstall Trigger Function

After you have declared and registered a user postinstall trigger function, you can test and debug it.

To test your user postinstall trigger function, follow these steps:

1. Open an application window.
2. Determine if there are any errors or warnings in the CIW.
3. Determine if the windows menu banner or pull-down menu has been appropriately customized.

Debugging a User Postinstall Trigger Function

Redefining a User Postinstall Trigger Function

When you determine the cause of an error, follow these steps:

1. Edit the source of your user postinstall trigger function.
2. Redefine the user postinstall trigger function. You do not have to register it again when you redefine it.
3. Open another application window. Determine if it has been appropriately customized.

Redefining a User Postinstall Trigger Function

The *deUnRegUserTriggers* Function

The Design Editor allows only one user postinstall trigger per view type. When developing and testing your customizations, be sure to unregister your postinstall trigger before you invoke the *deRegUserTriggers* function a second time.

Be careful when you reload a file that both declares your customization function and registers it. Unregister your postinstall trigger before you reload the file.

What You Put in the *.cdsinit* File

After you have tested your user postinstall trigger function, declare it and register it in your *.cdsinit* file.

Example

```
procedure( TrUserPostInstallTrigger( args )
  ...
) ; procedure

deRegUserTriggers( "schematic"
  nil
  nil
  'TrUserPostInstallTrigger
)
```

What You Put in the *.cdsinit* File

Three User Postinstall Trigger Examples

On the next several pages, there are three examples that implement the following customizations:

- Adding a menu item to the Composer Edit menu
- Adding a pull-down menu to a Composer window
- Reordering the Virtuoso Layout Editor pull-down menus

Three User Postinstall Trigger Examples

Adding a Menu Item to the Composer Edit Menu

In your *.cdsinit* file, declare your user postinstall trigger function and register it with the Design Editor.

```
procedure( TrUserPostInstallTrigger( args )
  ...
) ; procedure

deRegUserTriggers( "schematic"
  nil
  nil
  'TrUserPostInstallTrigger
)
```

Make the *TrUserPostInstallTrigger* function do these tasks:

1. Build your menu item if it is not already built.
2. Add your menu item to the Edit menu if it is not already in the Edit menu.

```
hiAddMenuItem( schEditMenu TrMenuItem )
```

The *schEditMenu* global variable contains the data structure of the Composer Edit pull-down menu.

Adding a Menu Item to the Composer Edit Menu

The *hiAddMenuItem* Function

Use the *hiAddMenuItem* function to add a menu item to a menu. Pass data structures for both the menu and the menu item.

The *hiInsertMenuItem* Function

Use the *hiInsertMenuItem* function to insert a menu item in a menu. Pass data structures for both the menu and the menu item. The third parameter is the position of the menu item in the menu.

```
hiInsertMenuItem( schEditMenu TrMenuItem 0 )
```

Finding the Edit Menu Data Structure

The *schEditMenu* global variable contains the data structure of the Composer Edit pull-down menu.

Follow these steps to verify this fact by inspection.

1. Open a Composer design window.
2. Make it the current window.
3. Call the *hiGetBannerMenus* function.

```
hiGetBannerMenus( hiGetCurrentWindow() ) =>  
  ( schematicTools schFileMenu4 schWindowMenu  
    schEditMenu schAddMenu schCheckMenu schSheetMenu )
```

4. Each variable in the return result holds the data structure of the corresponding pull-down menu.
Specifically, the *schEditMenu* variable contains the data structure for the Composer Edit menu.

Finding the Edit Menu Data Structure

The *hiGetBannerMenus* Function

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

The *TrGetMenuWithTitle* Function

You can use the *TrGetMenuWithTitle* function to determine the data structure of the menu in the window banner with the given title.

```
procedure( TrGetMenuWithTitle( wid title )
  let( ( menus menuVariable )
    menus = hiGetBannerMenus( wid )
    menuVariable =
      car(
        exists( var menus
          symeval(var)->_menuTitle == title
        )
      )
    when( menuVariable symeval( menuVariable ))
  ) ; let
```

The *exists* Function

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

The *symeval* Function

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

Avoiding Errors and Warnings

Make sure that your user postinstall function verifies the following conditions:

- Before building your menu item, verify that it does not already exist. Consider this example.

```
unless( boundp( 'TrMenuItem )  
    TrMenuItem = hiCreateMenuItem(  
        ...  
    )  
    ) ; unless
```

- Before attempting to modify the menu, verify that the menu data structure is in memory and that the menu item is not present in the menu, as in this example.

```
boundp( 'schEditMenu ) && !schEditMenu->TrMenuItem
```

Avoiding Errors and Warnings

Analyzing the Menu Data Structure

Let M be a variable whose value is a menu data structure.

Let $TrMenuItem$ be the symbolic name of a menu item.

Then $M \rightarrow TrMenuItem$ is non- nil when the item is in the menu, and nil otherwise.

The *boundp* Function

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

Use the *boundp* function to determine whether a variable is bound.

- The *boundp* function returns t , if the variable is bound to a value.
- It returns nil , if it is not bound to a value.

What You Put into the *.cdsinit* File

To make the Edit menu in Composer windows have the *Example* menu item, include the following code in your *.cdsinit* file:

```
procedure( TrUserPostInstallTrigger( args )
  unless( boundp( 'TrMenuItem )                                check #1
    TrMenuItem = hiCreateMenuItem(
      ?name      'TrMenuItem
      ?itemText  "Example"
      ?callback  "println( 'Example )"
    )
  ) ; unless
when(
  boundp( 'schEditMenu ) &&
  !schEditMenu->TrMenuItem                                check #2
  hiAddMenuItem( schEditMenu TrMenuItem )
) ; when
) ; procedure

deRegUserTriggers( "schematic"
  nil
  nil
  'TrUserPostInstallTrigger
)
```

What You Put into the *.cdsinit* File

Check #1

Check #1 determines whether the menu item exists.

Check #2

Check #2 determines whether the menu exists without the menu item present.

Adding a Pull-Down Menu to Composer Windows

Use the same approach as the first example.

In this example, make the *TrUserPostInstallTrigger* function do these tasks:

1. Build your pull-down menu, if it is not already in memory.
2. Use the `->` operator to access the *args* argument.
3. Add your menu to the window banner if it is not already there.

```
hiInsertBannerMenu( args->window, TrSchematicPulldownMenu(), 0)
```

Note: *args* is a datastructure provided by the trigger function which contains information about the window presently being opened. *args->window* contains the window id for the window.

Adding a Pull-Down Menu to Composer Windows

Building Your Pull-Down Menu

The Design Editor calls your trigger function each time the user opens a Composer window. To avoid building your menu repeatedly, the following code checks whether the menu already exists.

```
procedure( TrCreateSchematicPulldownMenu( )  
  if( boundp( 'TrSchematicPulldownMenu )  
    then TrSchematicPulldownMenu  
    else  
      hiCreatePulldownMenu( 'TrSchematicPulldownMenu ... )  
    ) ; if  
  ) ; procedure
```

What You Put into the *.cdsinit* File

To make every Composer window have your pull-down menu on the far right, include the following code in your *.cdsinit* file:

```
procedure( TrCreateSchematicPulldownMenu( )
  ...
) ; procedure

procedure( TrUserPostInstallTrigger( args )
  hiInsertBannerMenu(
    args->window
    TrCreateSchematicPulldownMenu( )
    length( hiGetBannerMenus( args->window ) )
  )
) ; procedure

deRegUserTriggers( "schematic"
  nil
  nil
  'TrUserPostInstallTrigger
)
```

See below

Window ID
Your pull-down

View type
No user app trigger
No user menu trigger
User postinstall trigger

What You Put into the *.cdsinit* File

The *TrCreateSchematicPulldownMenu* Function

```
procedure( TrCreateSchematicPulldownMenu( )
  if( boundp( 'TrSchematicPulldownMenu ) then
    TrSchematicPulldownMenu
  else
    hiCreatePulldownMenu(
      'TrSchematicPulldownMenu
      "Example Menu"
      list(
        hiCreateMenuItem( ;list of menu items
          ?name      'One
          ?itemText  "One"
          ?callback  "println( 'One )"
        )
        hiCreateMenuItem(
          ?name      'Two
          ?itemText  "Two"
          ?callback  "println( 'Two )"
        )
      ) ; list
    ) ; hiCreatePulldownMenu
  ) ; if
) ; procedure
```


Reordering Layout Editor Pull-Down Menus

In your *.cdsinit* file, include source code to declare the *TrUserPostInstallTrigger* function and register it as Design Editor user postinstall trigger function.

```
procedure( TrUserPostInstallTrigger( args )  
    ...  
    ) ; procedure  
  
deRegUserTriggers( "maskLayout"  
    nil  
    nil  
    'TrUserPostInstallTrigger  
    )
```

In this example, make the *TrUserPostInstallTrigger* function do these tasks:

1. Retrieve the list of pull-down menus already in the banner.
2. Remove them all.
3. Insert them one by one at position 0.

Reordering Layout Editor Pull-Down Menus

Use the same approach as the first and second examples.

What You Put into the *.cdsinit* File

To make every layout editor window appear with the pull-down menus in reverse order, include the following code in your *.cdsinit* file:

```
procedure( TrReverseBannerMenus( wid )
  let( ( theBannerMenuList )
    theBannerMenuList =
      hiGetBannerMenus( wid )
      hiDeleteBannerMenus( wid )
      foreach( menuSymbol theBannerMenuList
        hiInsertBannerMenu( wid menuSymbol 0 )
      ) ; foreach
  ) ; let
) ; procedure
```

Save menus

Remove all menus

Reinsert menus

```
procedure( TrUserPostInstallTrigger( args )
  TrReverseBannerMenus( args->window )
)
```

```
deRegUserTriggers( "maskLayout"
  nil
  nil
  'TrUserPostInstallTrigger
)
```

View type

No user app trigger

No user menu trigger

User postinstall trigger

What You Put into the *.cdsinit* File

The *hiGetBannerMenus* Function

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

The *hiDeleteBannerMenus* Function

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

The *TrReverseBannerMenus* Function

Customizing Forms

Assume your goal is to customize the default value or initial value of a form field.

To use SKILL to customize a form, follow these steps:

1. Declare a function that customizes a form field's default value.
2. Use the *defUserInitProc* function to register your function as a *user init proc* with SKILL.

When SKILL brings this form into memory, SKILL calls your function.

The next several pages explain these steps in greater detail.

Customizing Forms

Certain applications use an ASCII text file to control the default values for their forms.

The *defUserInitProc* function causes your function to be loaded immediately after the context you have chosen.

Customizing the Default Value of a Form Field

For each field in a form built with SKILL, there is a corresponding SKILL expression that sets the field value.

Whenever the user sets a field value in a form, the corresponding SKILL expression executes. You can observe this SKILL expression in the `~/CDS.log` file.

A similar SKILL expression sets the default value.

Example

Consider the **File Form** discussed in the User Interface module. Follow these steps:

1. Set the File Name field to the `/tmp/report.txt` value.
2. Observe the resulting SKILL expression in the CIW.

```
TrFileForm->TrFileNameField->value = "/tmp/report.txt"
```

A similar expression sets the default value. Use *defValue* instead of *value*.

```
TrFileForm->TrFileNameField->defValue = "/tmp/report.txt"
```

Customizing the Default Value of a Form Field

Design Framework II Initialization Sequence

A SKILL context is a binary file containing compiled SKILL code. Cadence supplies an initialization SKILL function for each context.

Cadence partitions applications into one or more SKILL contexts that load either at startup or later on demand.

At startup, the Design Framework II environment performs these steps:

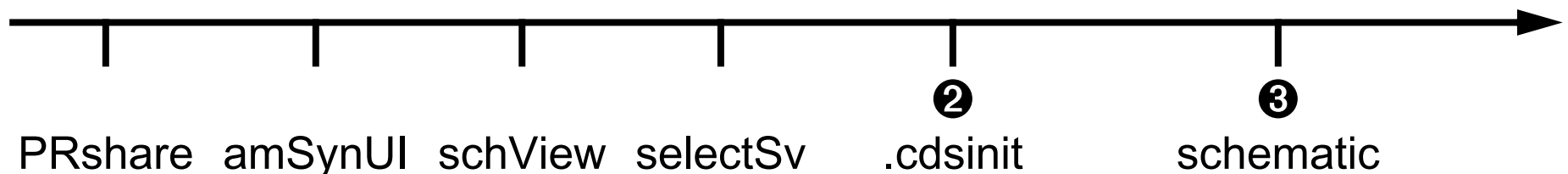
1. Loads one or more SKILL contexts.

2. Loads the *.cdsinit* customization file.

The search order for the *.cdsinit* file is <install_dir>/tools/dfII/local, the current directory, and the home directory.

Once the environment finds the *.cdsinit* file, the search stops. However, a *.cdsinit* can load other *.cdsinit* files.

3. Responds to user activity by loading other SKILL contexts.



Building your own context is beyond the scope of this course. See the SKILL User Guide for details.

Design Framework II Initialization Sequence

Each executable can load different contexts. For example, the *icfb* executable loads the following contexts: *PRshare*, *amSynUI*, *schView* and *selectSv*. Study the log file to determine the contexts that your executable loads prior to the *.cdsinit* file.

The context files are in the *<install_dir>/etc/context* directory.

A priority for the Design Framework II environment is to minimize startup time for application. For large applications, loading a SKILL context is faster than loading the original SKILL source code.

When Cadence builds a context, Cadence specifies a context initialization function to create data structures that are not appropriate to save in the context, such as ports, database objects, and window IDs.

After the Design Framework II environment loads a context upon demand, it calls the Cadence context initialization function. However, if you load a context with the *loadContext* function, you must use the *callInitProc* function to invoke the initialization function.

Some applications build forms dynamically. Customizing these forms is beyond the scope of this course.

The *defUserInitProc* Function

You can supply a user initialization SKILL function for any Cadence context.

Immediately after Design Framework II loads a context into memory, Design Framework II calls these functions:

- The initialization SKILL function that Cadence supplies.
- The user initialization SKILL function that you supply for that context.

Your user initialization SKILL function can customize the menus and forms that are defined in a context.

Example

```
procedure( TrCustomizeContext()  
  ...  
) ; procedure  
  
defUserInitProc( "schView" 'TrCustomizeContext )
```

↑
Context

↑
**User
initialization
function**

The *defUserInitProc* Function

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

The first argument to the *defUserInitProc* function is the name of a context. The second argument denotes the SKILL function that you want the Design Framework II environment to call when it loads the context.

Manage the Layer Selection Window (LSW)

Remove the LSW from the screen

```
leUnmapLSW( )
```

Restore the LSW to the screen

```
leRemapLSW( )
```

The LSW is only present with the layout tools. By default once the LSW is displayed for a layout window it is not removed from the screen and there is no close option on the window itself. The SKILL commands listed here are the only way remove and restore the window.

Manage the Layer Selection Window (LSW)

Historically this window could not be removed because it was always a reference to the layers in the current window. If you remove the LSW with *leUnmapLSW* you will need to restore it if you require it for another layout window. The system will not restore the LSW when a new layout window is opened. You will need to use *leRemapLSW*.

Lab Overview

Lab 15-1 Adding a Menu Item to the Composer Edit Menu

You study and run the lecture example.

Lab 15-2 Adding a Pull-Down Menu to Composer Window

You study and run the lecture example.

Lab 15-3 Reversing the Layout Editor Pull-Down Menus

You study and run the lecture example.

Lab 15-4 Customizing Initial Window Placement

You customize the Layout Editor to initially place all of its design windows in a given location.

Lab Overview

Module Summary

In this module, we discussed

- Customizing the Composer software and the Layout Editor.
- Customizing form field default values.

Module Summary

Module 16: Interprocess Communication

Objectives

- Discuss the scope of SKILL IPC.
- Discuss an example C program.
- Spawn the C program as a separate UNIX child process.
- Communicate synchronously with the child process.
- Communicate asynchronously with the child process.

Terms and Definitions

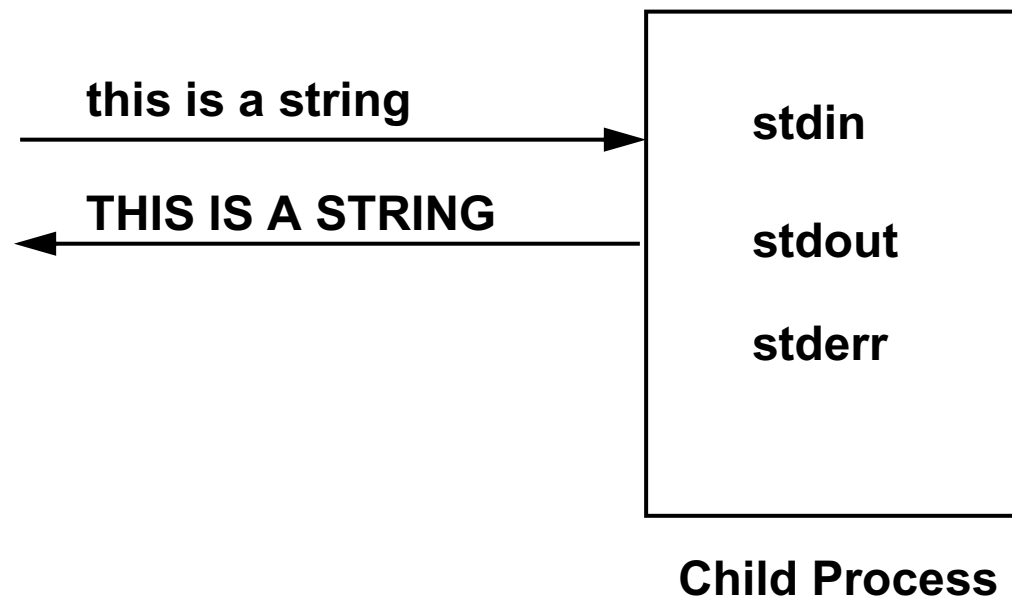
Blocking read	The parent process, the Design Framework II environment, is suspended until the data is available from a child process. During a <i>blocking read</i> , the parent's user interface and graphics freeze.
Deadly embrace	Two processes enter a <i>dead lock</i> if both wait for resources held by the other before releasing resources. A <i>blocking read</i> with a timeout limit avoids the possibility of a deadly embrace.
Kill command	The UNIX <i>kill</i> command sends a signal to a UNIX process. The <i>kill</i> command requires a UNIX process ID. The SKILL process ID is different from the UNIX process ID.

The Scope of SKILL IPC

Using SKILL IPC, an application can:

- Launch a C executable or shell command as a separate UNIX process.
- Write text strings to the child's *stdin* channel.
- Read text strings from the child's *stdout* and *stderr* channels.
- Suspend, resume, interrupt, and terminate the child process.

The example child program reads a text string from its *stdin* and writes the text string containing upper case characters to its *stdout*.



The Scope of SKILL IPC

The *toUpperCase.c* Source Code

```
#include <stdio.h>
#include <ctype.h>
#define TRUE 1
main()
{
    int c;
    char s[100],*p;
    while( TRUE ) {
        bzero(s,99);
        p = &(amp;s[0]);

        gets(p);

        while ( *p )
        {
            if (*p != ' ' && islower(*p))
                *p = toupper(*p);
            p++;
        }

        printf("%s \n" , s);
        fflush(stdout);
    }
}
```

Forever

Read stdin

Force case

Write stdout

The *toUpperCase.c* Source Code

The *toUpperCase.c* program:

- Reads a line from *stdin* using the *gets* function.
- Converts lowercase characters to uppercase characters.
- Uses the *printf* statement to write to *stdout*.

Comparing the Two Communication Styles

Synchronous communication features a single flow of control.

- Your user is not free to run other Design Framework II commands until synchronous communication with the child is complete.
- When you call the *TrUpperCase* function, the user cannot run another Design Framework II command until the *TrUpperCase* function returns.

Asynchronous communication does not grab your application's flow of control.

- Your user is free to run other Design Framework II commands.
- Asynchronous communication with the child occurs between Design Framework II commands.

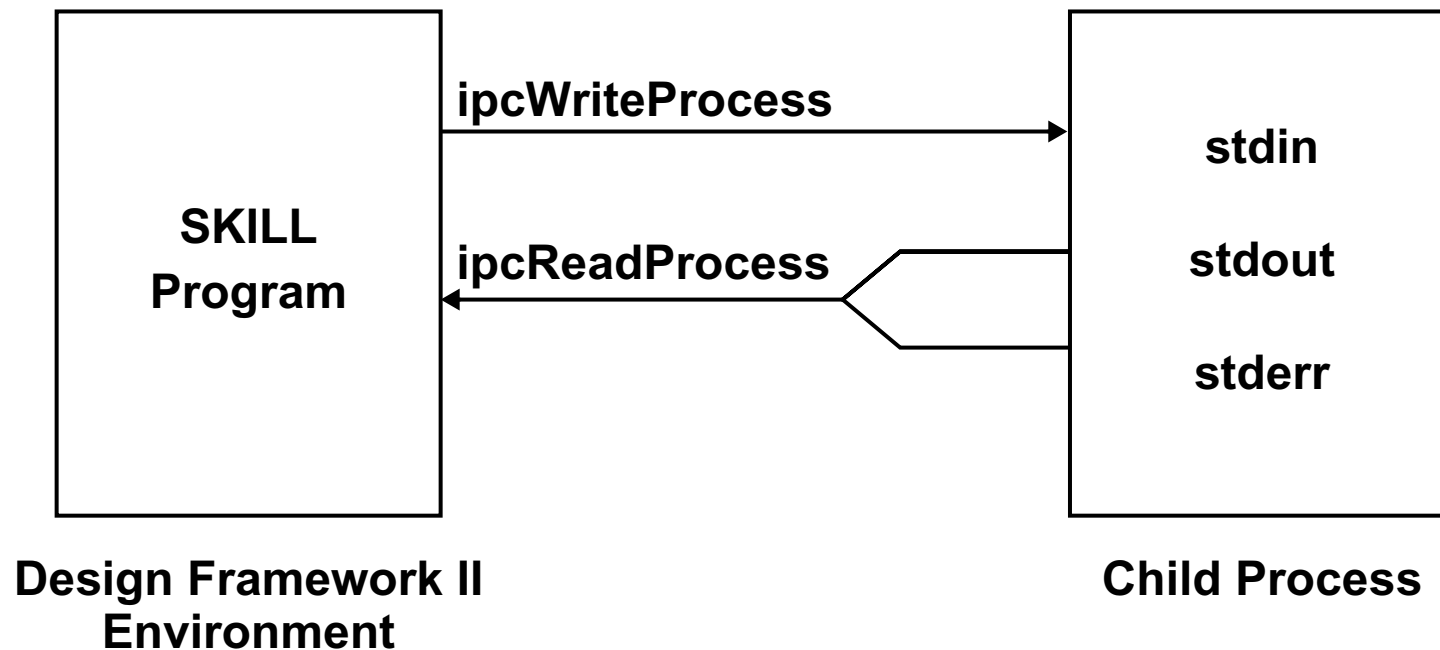
Comparing the Two Communication Styles

Spawning a Child Process

Use the *ipcBeginProcess* function to launch a child process.

The *ipcBeginProcess* function returns an integer that identifies the child process. This ID is not the same ID used by the UNIX environment.

The newly initiated child process communicates with its parent process using the standard descriptors: *stdin*, *stdout*, and *stderr*.



Spawning a Child Process

The *ipcBeginProcess* Function

Use the *ipcBeginProcess* function to spawn a UNIX process to execute a command or sequence of commands. The *ipcBeginProcess* function returns an integer process ID.

Argument	Required?	Type	Meaning
command	required	text	Spawns a UNIX process to execute <i>command</i> on <i>hostname</i> .
hostname	optional	text	The child process executes locally unless a network node is specified for <i>hostname</i> .
ioHandler	optional	text or function symbol	System invokes <i>ioHandler</i> when child data on <i>stdout</i> is available.
errHandler	optional	text or quoted function symbol	System invokes <i>errHandler</i> when child error on <i>stderr</i> is available.
postFunc	optional	text or quoted function symbol	System invokes <i>postFunc</i> when child terminates.

Synchronizing with the Child Process

You can synchronize a SKILL program with either of these two events in the child process:

- The child has been launched and can now accept data through its *stdin*.
- The child has terminated.

Whenever a SKILL program synchronizes with either of the above events, the SKILL program suspends itself and the Design Framework II user interface freezes until the event happens.

Use the *ipcWaitForProcess* function to synchronize with the child launch and the *ipcWait* function to synchronize with the child termination.

Synchronizing with the Child Process

The *ipcWaitForProcess* Function

Call the *ipcWaitForProcess* function before you send any data to the child process.

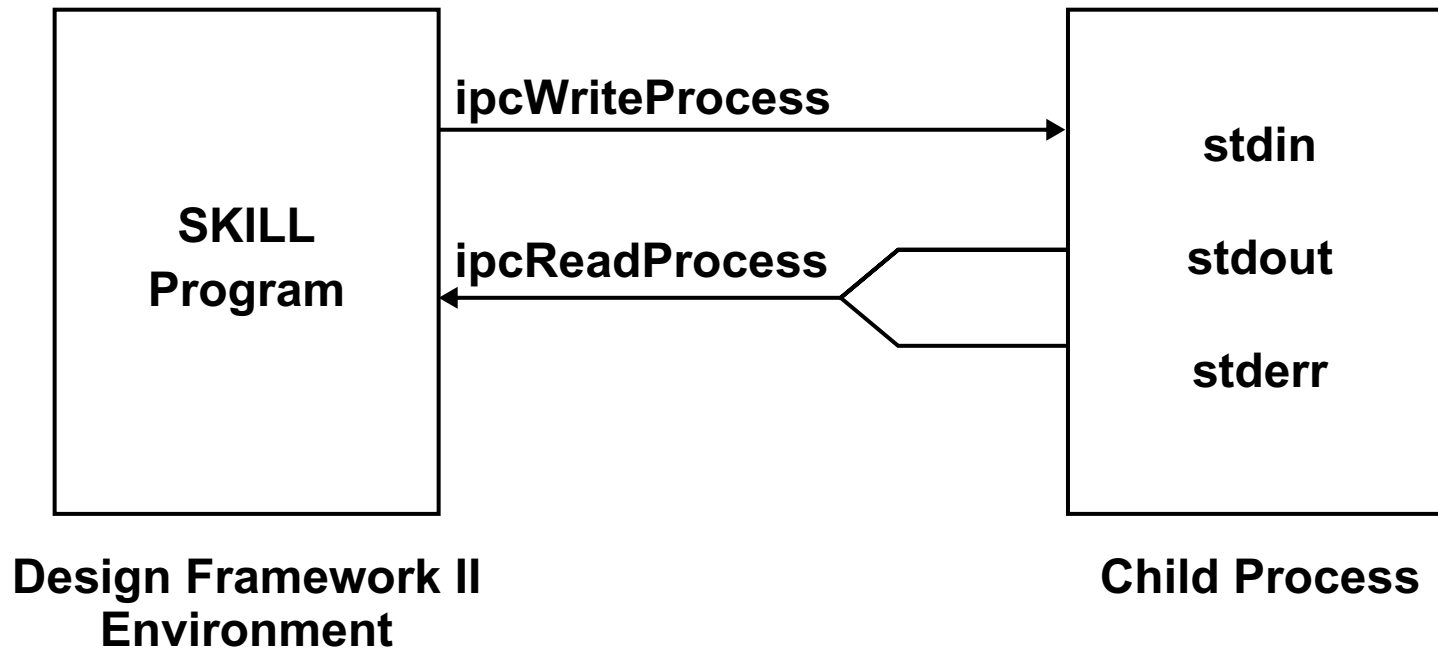
The *ipcWait* Function

Call the *ipcWait* function if you want to prevent the user from doing anything until the child process terminates. For example, the child process might create a file that is required for further steps the user might take.

Both the *ipcWaitForProcess* function and the *ipcWait* function freeze the Design Framework II user interface until they return. Although the Design Framework II user interface is frozen, the SKILL Evaluator is available to invoke the data handler or error handler of a child process.

Communicating with the Child Process

The child process communicates through its *stdin*, *stdout*, and *stderr* with the Design Framework II environment.



You can use either synchronous or asynchronous techniques to communicate with the child process.

Communicating with the Child Process

Synchronous Communication

Task	Synchronous Communication
Processing Child Data	<i>ipcReadProcess</i>
ProcessingChild Errors	<i>ipcReadProcess</i>
Determining Exit Status	<i>ipcGetExitStatus</i>
Detecting Child Termination	<i>ipclsAliveProcess</i>

Synchronous Communication

The *ipclIsAliveProcess* Function

If the child process is alive, the *ipclIsAliveProcess* function returns *t*. If the child process has terminated, the *ipclIsAliveProcess* returns *nil*.

The *ipcReadProcess* Function

The *ipcReadProcess* function reads data from the child function.

The second argument to *ipcReadProcess* specifies a timeout limit. This limit is the maximum time in seconds to wait before giving up the attempt to read.

- If the *ipcReadProcess* function times out before data is available from the child, the *ipcReadProcess* function returns an empty string.
- If 0 is given for the timeout value, the *ipcReadProcess* function only reads what is currently available.

Synchronous Communication Example

```
procedure( TrUpperCaseList( aList )
  let( ( childPID result )
    childPID = ipcBeginProcess(
      simplifyFilename(
        "../IPC/toUpperCase.out" )
      )
    ipcWaitForProcess( childPID )
    result = foreach( mapcar element aList
      ipcWriteProcess(
        childPID
        strcat( element "\n" )
      )
      ipcReadProcess( childPID 10 )
    ) ; foreach
    ipcKillProcess( childPID ))
    result
  ) ; let
) ; procedure

TrUpperCaseList( '( "a" "bc" "def" ) )
=> ( "A \n" "BC \n" "DEF \n" )
```

Spawn child
Full path

Synchronize

Write to child

Read from child

Terminate the child

Call function

Synchronous Communication Example

The *TrUpperCaseList* Function

The *TrUpperCaseList* function does the following:

- Accepts a list of text strings as its single argument.
- Returns the list of text strings that the child sends back.
- Uses the functions in this table.

Function	Purpose
<i>ipcBeginProcess</i>	Launch a child process to execute the compiled <i>UpperCase.c</i> program.
<i>simplifyFilename</i>	Determine the full path name to the executable.
<i>ipcWaitForProcess</i>	Synchronously wait for the child to launch.
<i>ipcWriteProcess</i>	Write each text string in a list to the child process.
<i>ipcReadProcess</i>	Synchronously wait for the child to send back data.
<i>ipcKillProcess</i>	Terminate the child process.

Asynchronous Communication

Task

Asynchronous Communication

Processing Child Data	Data handler Specify to <i>ipcBeginProcess</i>
Processing Child Errors	Error handler Specify to <i>ipcBeginProcess</i>
Determining Exit Status	Postfunction Specify to <i>ipcBeginProcess</i>
Detecting Child Termination	Postfunction Specify to <i>ipcBeginProcess</i>

Three SKILL functions govern asynchronous communication with the child process. The Design Framework II environment calls the following functions:

- The data handler when it receives data from the child's *stdout* channel.
- The error handler when it receives an error from the child's *stderr* channel.
- The postfunction when UNIX reports that the child has terminated.

Asynchronous Communication

The SKILL Evaluator executes calls to data handlers, calls to postfunctions, and user input in the order they are queued.

The *dataHandler* Function

You must declare the data handler to accept two arguments: *o_childId* and *t_data*. A single data handler can service multiple child processes because the process ID of the child is a parameter.

The child sends data to the Design Framework II process after each newline character or when the buffer fills or is flushed. If the data handler does not have time to read the data before the child process writes again, the data is not lost. The data handler is only called if data is available and the Design Framework II environment is idle.

The *errHandler* Function

You must declare your error handler function to accept two arguments: *o_childId* and *t_data*.

The *postFunc* Function

You must declare your postfunction to accept two parameters: *o_childId* and *x_exitStatus*.

Asynchronous Communication Example

The following pages show you how to do these tasks:

- Communicate asynchronously with a child process.
- Reimplement the previous synchronous communication example using asynchronous communication.

The new implementation requires a collection of functions to communicate through global variables.

Asynchronous Communication Example

Launching the Child Process

The *TrUpperCase_Launch* function launches the child and sends the first string in the list to the child process.

```
procedure( TrUpperCase_Launch( aList )
  let( ( childPID result )
    childPID = ipcBeginProcess(
      simplifyFilename(
        "./IPC/toUpperCase.out"
      )
      " "
      'TrUpperCase_DataHandler
      nil
      'TrUpperCase_PostFunction
    )
    ipcWaitForProcess( childPID )
    TrList = aList
    TrResult = nil
    TrUpperCase_Write( childPID )
    printf( "Launched child %d\n" childPID )
  ) ; let
) ; procedure

TrUpperCase_Launch( '( "a" "bc" "def" ) )
```

Spawn child
Determine full path

Synchronize
Initialize globals

First write to child

Launching the Child Process

The *TrUpperCase_Launch* Function

The *TrUpperCase_Launch* function accepts a list of text strings as its single argument. It returns *t*.

The *TrUpperCase_Launch* function uses the functions in this table.

Function	Purpose
<i>ipcBeginProcess</i>	Launch a child process to execute to the compiled <i>UpperCase.c</i> program.
<i>simplifyFilename</i>	Determine the full path name to the executable.
<i>ipcWaitForProcess</i>	Synchronously wait for the child to launch.
<i>TrUpperCase_Write</i>	Write a text string to the child process.

Writing Data to the Child

The several functions in the example communicate through two global variables.

- The *TrList* global variable contains the strings to be sent to the child process.
- The *TrResult* global variable contains the strings that the child process has returned.

The *TrUpperCase_Write* function does the following:

- Writes the first element in *TrList* to the child.
- Sets the *TrList* to be the remainder of the list.

```
procedure( TrUpperCase_Write( pid )  
  ipcWriteProcess(  
    pid  
    strcat( car( TrList ) "\n" )  
  )  
  TrList = cdr( TrList )  
) ; procedure
```

Access global

Update global

Writing Data to the Child

The *TrUpperCase_Write* Function

Receiving Data from the Child Process

Make your data handler accept these two arguments:

- *pid*
This argument identifies the child process.
- *data*
This argument is the text string that the child sent to the Design Framework II environment.

Example *TrUpperCase_DataHandler* function

```
procedure( TrUpperCase_DataHandler( pid data )  
  TrResult = append( TrResult list( data ))  
  if( TrList then  
    TrUpperCase_Write( pid )  
  else ipcKillProcess( pid )  
  ) ; if  
) ; procedure
```

Update global
Access global

This function appends the child data to the list in the global variable *TrResult* and checks the global variable *TrList* to decide whether to

- Send the next string to the child.
- Terminate the child.

Receiving Data from the Child Process

The *TrUpperCase_Write* Function

The *TrUpperCase_DataHandler* Function

The Design Framework II environment queues a call to the data handler when it receives data from the child's *stdout* channel.

The data handler executes when the SKILL Evaluator is idle.

The *TrUpperCase_PostFunction* Function

Make your postfunction accept two arguments:

- *pid*
This argument identifies the child process.
- *status*
This argument is the UNIX exit status of the terminated child process.

Use global variables to communicate with the rest of your application.

The return value of the postfunction is irrelevant.

Example of *TrUpperCase_PostFunction*

```
procedure( TrUpperCase_PostFunction( pid status )  
    printf( "Child %d result: %L\n" pid TrResult )  
    ) ; procedure
```

The *TrUpperCase_PostFunction* Function

When a child process terminates, the Design Framework II environment queues a call to the child's postfunction, if one exists.

The call to the postfunction executes when the SKILL Evaluator is idle.

Controlling the Child Process

The following functions control the child process.

Function	Argument	Action
<i>ipcStopProcess</i>	child ID	Suspends the child process.
<i>ipcContProcess</i>	child ID	Resumes the child process.
<i>ipcKillProcess</i>	child ID	Terminates the child process.
<i>ipcKillAllProcesses</i>	none	Terminates all child processes.
<i>ipcSoftInterrupt</i>	child ID	Sends a <i>kill</i> -2 to the child process.

Controlling the Child Process

Lab Overview

Lab 16-1 Compiling a C program

Lab 16-2 Exploring Synchronous Communication

Lab 16-2 Exploring Asynchronous Communication

Lab Overview

Module Summary

In this module, we covered

- Launching a child process with the *ipcBeginProcess* function.
- Synchronizing the child process with the *ipcWaitForProcess* function.
- Waiting for the child to terminate with the *ipcWait* function.
- Writing text strings to the child's *stdin* using the *ipcWriteProcess* function.
- Reading text strings from the child's *stdout* using the *ipcReadProcess* function.
- Launching a child process with a data handler and a postfunction.

Module Summary

Category	Functions
Launching a child process	<i>ipcBatchProcess</i> <i>ipcBeginProcess</i>
Synchronizing with a child process	<i>ipcWaitForProcess</i> <i>ipcWait</i>
Controlling a child process	<i>ipcStopProcess</i> <i>ipcContProcess</i> <i>ipcSoftInterrupt</i> <i>ipcKillProcess</i> <i>ipcKillAllProcesses</i>
Data communication	<i>ipcReadProcess</i> <i>ipcWriteProcess</i> <i>data handlers</i> <i>postfunctions</i>

Module 17: Data Structures

Objectives

- Implementing records in SKILL
 - Disembodied property lists
 - SKILL structures
- Arrays
- Associative data structures
 - Association tables
 - Association lists

Terms and Definitions

Disembodied Property Lists

A disembodied property list is logically equivalent to a record. Unlike C structures or PASCAL records, new fields can be dynamically added or removed.

A disembodied property list is a list that starts with *nil* followed by alternating name/value pairs.

```
aCard = '( nil rank "ace" suit "spades" )
```

Use the `->` operator to retrieve a value for a given name.

```
aCard->rank => "ace"  
aCard->suit => "spades"
```

Use the `->` operator together with the `=` operator to update or add a field.

```
aCard->faceUp = t
```

Implementing Records in SKILL

Disembodied Property Lists

The first element of the disembodied list does not have to be *nil*. It can be any SKILL data object. The access operation ignores the first element.

You can use a disembodied property list to consolidate several arguments into a single argument. Similarly, you can use a disembodied property list to consolidate several global variables into a single global data structure.

When you apply the `~>??` operator to a database object, this operator builds a disembodied property list. This list consists of the attributes and values of the database object.

The *get* and *getq* Functions

The `->` operator is implemented by the *getq* function. Its second argument is not evaluated. That is, it is implicitly quoted as in this example.

```
getq( aCard rank ) => "ace"
```

The *get* function is a generalized version of the *getq* function. Both of its arguments are evaluated.

SKILL Structures

A SKILL structure is also logically equivalent to a record.

The *defstruct* function declares a template. It **automatically** defines a SKILL creation function with the slot names as keyword parameters.

Example

1. Define a *card* structure and allocate an instance of *card*.

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

2. Allocate an instance of *card* and store a reference to it in *aCard*.

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

3. Use the *type* function to determine the structure name.

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

SKILL Structures

The *defstruct* Function

The *defstruct* function declares a creation function, which you use to make instances of the structure. SKILL bases the name of the creation function on the name of the structure.

The *printstruct* Function

The *printstruct* function dumps out a structure instance. It recursively dumps out any slot value that is also a structure instance.

Comparing SKILL Structures and Disembodied Property Lists

- A structure is more efficient than a disembodied property list, when you know the slots in advance, and you specify them when you declare the structure.
- A structure instance requires less space than a disembodied property list and slot access is faster.

Accessing and Updating Structures

The `->` operator provides slot access.

```
aCard->rank => "ace"  
aCard->rank = 8 => 8
```

Use `->?` to get a list of the slot names.

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

Use `->??` to get a list of slot names and values.

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

You can create slots dynamically for a specific instance by simply referencing them with the `->` operator.

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

Accessing and Updating Structures

Arrays

SKILL provides arrays.

- Elements of an array can be any datatype.
- SKILL provides run-time array bounds checking.

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

Example

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

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

Arrays

Arrays are one dimensional. You can implement higher dimensional arrays using single dimensional arrays.

SKILL checks the array bounds each time you access the array during runtime. Accessing an array outside the array bounds causes an error.

Arrays and Structures

Structure instances are arrays. The first element of the array holds the type, and the last element is reserved for a disembodied property list to hold dynamically added slots.

```
aCard => array[5]:21556040  
arrayp( aCard ) => t
```


2-Dimensional Arrays

You can implement a 2-dimensional array as an array of row arrays.

```
procedure( Tr2DArray( nRows mCols )
  let( ( rowArray row )
    declare( rowArray[ nRows ] )
    for( i 0 nRows-1
      declare( row[ mCols ] )
      rowArray[ i ] = row
    ) ; for
    rowArray
  ) ; let
) ; procedure
```

```
TrTimesTable = Tr2DArray( 10 10 )
```

```
for( i 0 9
  for( j 0 9
    TrTimesTable[i][j] = i*j
  ) ; for
) ; for
```

2-Dimensional Arrays

Association Tables

An association table is a collection of key/value pairs. You can use these SKILL data types as keys in a table:

- integer
- string
- list
- symbol
- database object

Use the syntax for array access to store and retrieve entries in a table.

The *makeTable* function defines and initializes the association table. The arguments include:

- The table name (required).
- The default entry (optional).
The default of the default is the *unbound* symbol.

Association Tables

An association table is implemented as a hash table.

Association table access uses the *equal* function to compare keys.

Several list-oriented functions also work on tables, including iteration. Use these functions to manipulate tables:

- The *foreach* function executes a collection of SKILL expressions for each key in a table.
- The *setof* function returns a list of keys in a given table that satisfies a given criterion.
- The *length* function returns the number of keys in a table.
- The *remove* function removes a key from a table.

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

Using Association Tables as Dynamic Arrays

You can easily use an association table for a 1-dimensional array. Use integers as the keys.

Use an association table in situations where it is obviously wasteful or impossible to allocate the entire array as in these data structures:

- A sparse array, most of whose entries are unused
- Multidimensional arrays

To implement a 2-dimensional array, use keys that are lists of index pairs.

```
procedure( Tr2D_DynamicArray()  
  let( ( )  
    makeTable( gensym( 'TrSparseArray ) )  
  ) ; let  
) ; procedure  
  
TrTimesTable = Tr2D_DynamicArray( )  
for( i 0 9  
  for( j 0 9  
    TrTimesTable[ list( i j ) ] = i*j  
  ) ; for  
) ; for
```

Using Association Tables as Dynamic Arrays

Associating Shapes with Layer Purpose Pairs

You can use an association table to associate a shape with a layer purpose pair as follows:

- Use a list composed of the layer name and the purpose as a key to access the table.
- The value for the key is the list of shapes on the layer purpose pair.

Making the Table

```
cv = geGetWindowCellView( )
tableName = sprintf( nil "%s %s %s"
  cv~>libName cv~>cellName cv~>viewName )
shapeTable = makeTable( tableName nil )
=> table:master mux2 layout
```

Populating the Table with Shapes

```
foreach( shape cv~>shapes
  layerPurpose = shape~>lpp
  shapeTable[ layerPurpose ] =
    cons(
      shape
      shapeTable[ layerPurpose ]
    )
) ; foreach
```

Associating Shapes with Layer Purpose Pairs

The *shapes* Attribute

Every *cellView* database object has a *shapes* attribute, whose value is a list of all shapes in the design.

The *layerPurposePairs* Attribute

In practice, if you need to process all the shapes in a design by layer purpose pair, retrieve the *layerPurposePairs* attribute, whose value is a list of *LP* database objects. Each *LP* database object has these attributes:

- a *shapes* attribute, which is a list of all Shapes that are on that layer purpose pair.
- *layerName* and *layerNum* attributes, which identify the layer purpose pair.

Traversing Association Tables

Use the *foreach* function to visit every key in an association table.

The following code displays each key/value pair in the CIW.

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

The following functions also work with tables:

- The *forall* function
- The *exists* function
- The *setof* function

Traversing Association Tables

The *readTable* and *writeTable* Functions

Use these functions to read a table from a file and to write a table to a file.

Association Lists

A list of key/value pairs is a natural means to record associations.

An association list is a list of lists. The first element of each list is the key.

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

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

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

The *rplaca* function updates the first element of a list.

Use the *rplaca* function to replace the *car* of the *cdr* of the association list entry as shown here.

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

Association Lists

The *tableToList* Function

You can also convert an association table to an association list. Use the *tableToList* function to build an association list from an association table.

The *append* Function

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

- For the first argument, you specify the association table.
- For the second argument, you specify the disembodied property list, association list, or other association table whose data is to be appended to the first association table.

Lab Overview

Lab 17-1 Exploring Associative Tables

Lab 17-2 Exploring Association Lists

Lab 17-3 Exploring Disembodied Property Lists

- Extend the Layer Shape Report to maintain a separate count for each type of shapes in a design. Eliminate the miscellaneous count.
- Three versions
 - Use an association table.
 - Use an association list.
 - Use a disembodied property list.

Lab Overview

Module Summary

In this module, we covered

- Implementing records in SKILL with
 - Disembodied property lists
 - SKILL structures
- SKILL Arrays
- Association Lists
- Association Tables

Module Summary

Module 18: OpenAccess

Objectives

- Understand OpenAccess
- Understand how OpenAccess is related to DFII
- Learn how to obtain OpenAccess
- Locate additional information on OpenAccess

Terms and Definitions

OpenAccess	New public API and reference database to allow tool inter operability
Si2	Silicon Integration Initiative, Inc. — Organization aimed at improving quality and reducing costs of integrated silicon systems. This is the parent organization of the OpenAccess coalition which will control future development of the API.

OpenAccess and SKILL

- OpenAccess provides a SKILL layer which allows SKILL functions to operate just as they presently do.
- In theory, SKILL should continue to function correctly. In practice, some functions will work differently and some new functions will be added to handle new concepts that did not exist in CDBA.
- You can still write your programs in SKILL for DFII on OpenAccess.
- If there are third-party tools or custom tools you create that use the OA API, you may want to interface to the OA database directly through the OA API rather than SKILL

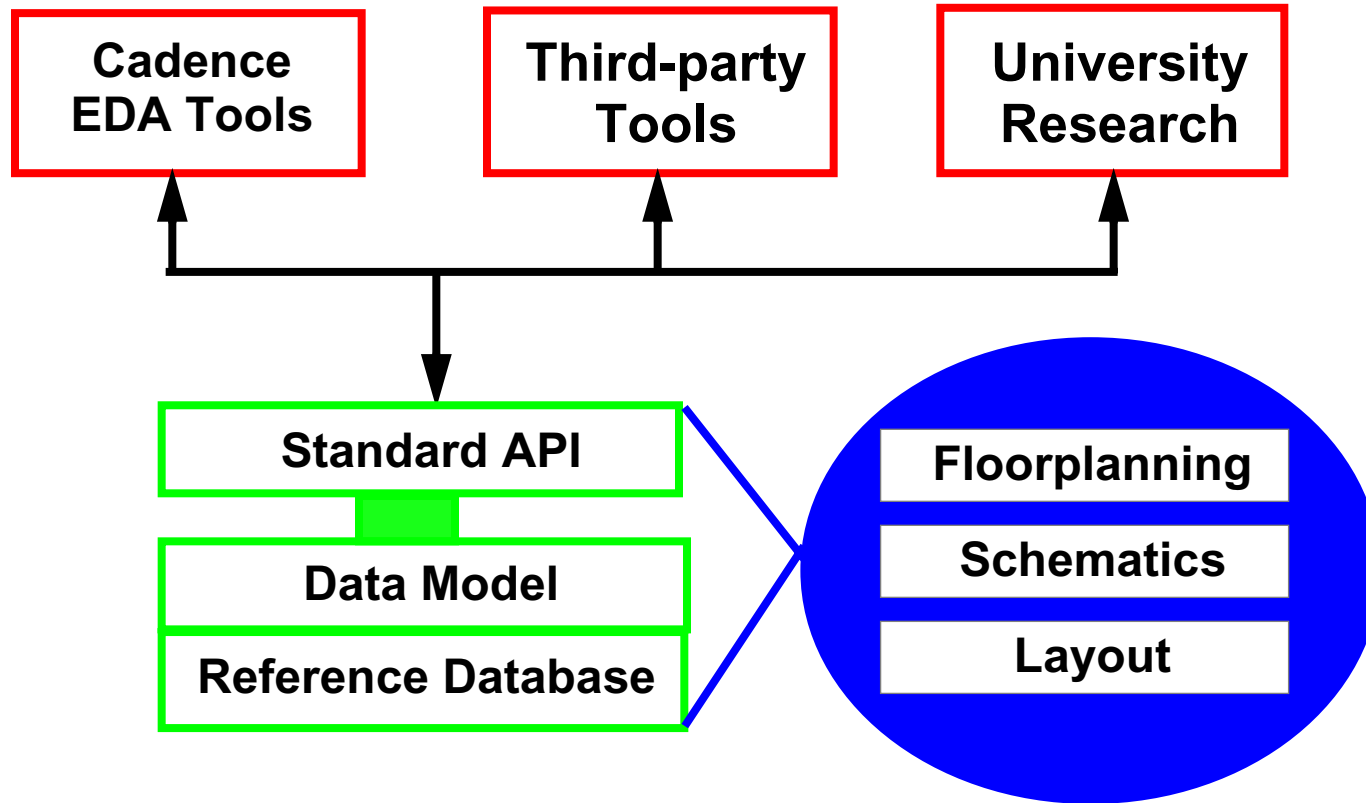
OpenAccess and SKILL

What Is OpenAccess?

- Created by Cadence as common database and API for next-generation integration of digital and custom flow tools.
- Released to industry OpenAccess Coalition under Si2 as open database for tool integration.
- High-capacity connectivity-based database with structures to compactly handle large designs. Increased capacity and smaller memory footprint (for example, 800Mb CDBA vs. 250Mb OA) will also increase performance.
- Well documented C++ based public API for easy integration of third-party tools.

What Is OpenAccess?

OpenAccess Database

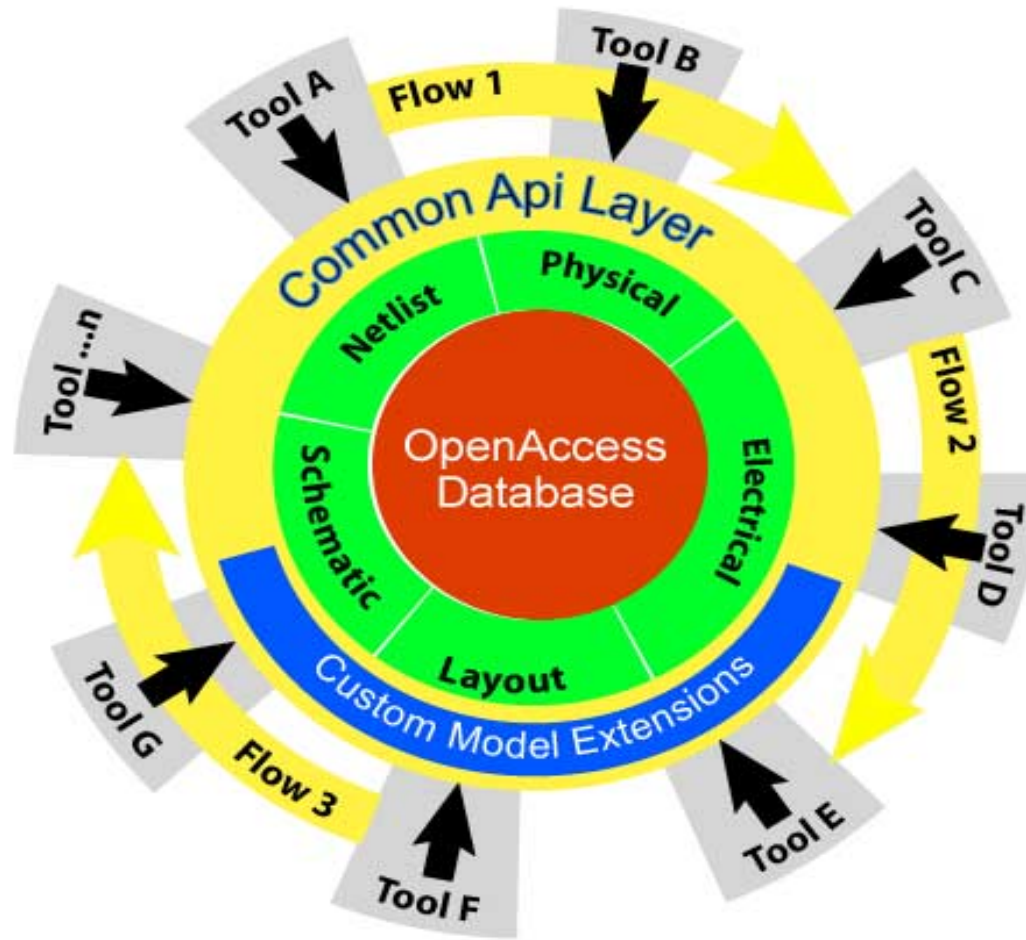


OpenAccess Database

Advantages of OpenAccess:

- Single database eliminates translation steps
- Single data model eliminates loss of data
- Correct representation is between each application and the database, not a function of downstream tools
- Name mapping in the API means each tool uses its own naming rules
- All existing design data is available to any tool that decides to use it
- Non-proprietary — uses OpenSource model for distribution and enhancements

Database-Centric Flows



Database-Centric Flows

- All tools communicate through the common API layer
- Custom model extensions are allowed and can be created by customer or third party
- Multiple tool flows share data without the need for translation

Code Examples

Tech Database LPP API

Open the tech DB for a given cellview

```
oaCellView::getTech( )
```

Look up a layer by name

```
oaLayer::find( oaTech *tech, const oaString &name )
```

Look up a purpose by number

```
gnPurpose::find( oaTech *tech, oaPurposeNum num )
```

Code Examples

What is Si2?

Overview

Si2 is an organization of industry-leading silicon systems and tool companies focused on improving productivity and reducing cost in creating and producing integrated silicon systems.

Mission

To provide value-added services and to transfer technologies that reduce the cost and complexity of silicon design and integration.

Key Strategies

- Facilitate collaborative customer-driven solutions definition, development, and implementation
- Engage and support EDA Industry Council thrusts
- Provide dedicated support services, which accelerate adoption of solutions
- Technology transfer
- Project management
- Testing services

What is Si2?

History

Si2 was founded in 1988 as CAD Framework Initiative, Inc. Today Si2 represents semiconductor manufacturers, electronic systems companies, and EDA tool suppliers in North America, Europe, and Asia.

What Is the OpenAccess Coalition?

- The OpenAccess coalition is an organized group of CAD tool vendors and users who collaborate and make decisions on the future changes to OA.
- Cadence does not have the final approval on changes; the coalition does.
- Modifications are requested by members and then voted on to determine if they will be included in the specification and API.

What Is the OpenAccess Coalition

What Does OpenAccess Mean for IC 5.0?

- IC 5.0 is scheduled to be released in two streams
 - IC 5.0 on CDBA (released in 7/02)
 - IC 5.02 on OpenAccess (release to limited customers end of Q4 2002)
- Enhancements and bug fixes will continue to be made in both streams. We will continue supporting our CDBA customers into the foreseeable future.
- Functionality will be identical initially (OA stream will have larger capacity)
- Enhancements that require features of OA will eventually cause some divergence between the streams

What Does OpenAccess Mean for IC 5.0?

How Does OpenAccess Affect DFII?

- CDBA library structure remains in place.
- API layer interfaces between standard SKILL and ITKdb database functions to underlying database.
- In theory, this should mean that all old SKILL should continue to function correctly. In practice, there are going to be some functions that will work differently and some new functions to handle new concepts that did not exist in CDBA.
- SKILL registration and audit will be key tools in a successful migration.
- Migration will only be supported from 4.4.X design databases.
- Initially DFII on OA will be targeted to customers who need extra capacity/performance.
- General rollout slated for 2004.

How Does OpenAccess Affect DFII?

How Will DFII on OA Be Supported?

- Tools and flows for migration will be provided
- Customers using DFII on OA and Cadence customers integrating their own tools will be supported by Cadence Customer Support and adoption team.
- Third-party integrators into OA will be directed to the OpenAccess Coalition for support.
- Dedicated OA adoption team will make the transition to OA efficient.

How Will DFII on OA Be Supported?

How to Obtain OpenAccess

As Part of the Cadence Release Stream

Each Cadence tool set will have a release on OpenAccess. The API and database will be shipped as part of the Cadence release stream.

Directly from the OpenAccess Web Site

You can go to the web site and download the API and database directly from the OpenAccess Consortium. You must be an OpenAccess coalition member to download the data. Details on access are posted on the web site.

How to Obtain OpenAccess

Where to Go for Information

Cadence Information

http://www.cadence.com/company/open_access.html

Si2 Information

<http://www.si2.org>

OpenAccess Coalition Information

<http://www.openEDA.org>

OpenAccess Training

- Students should:
 - Have experience with C++ software development
 - Have experience with some EDA database
 - Understand the basics of physical IC design data

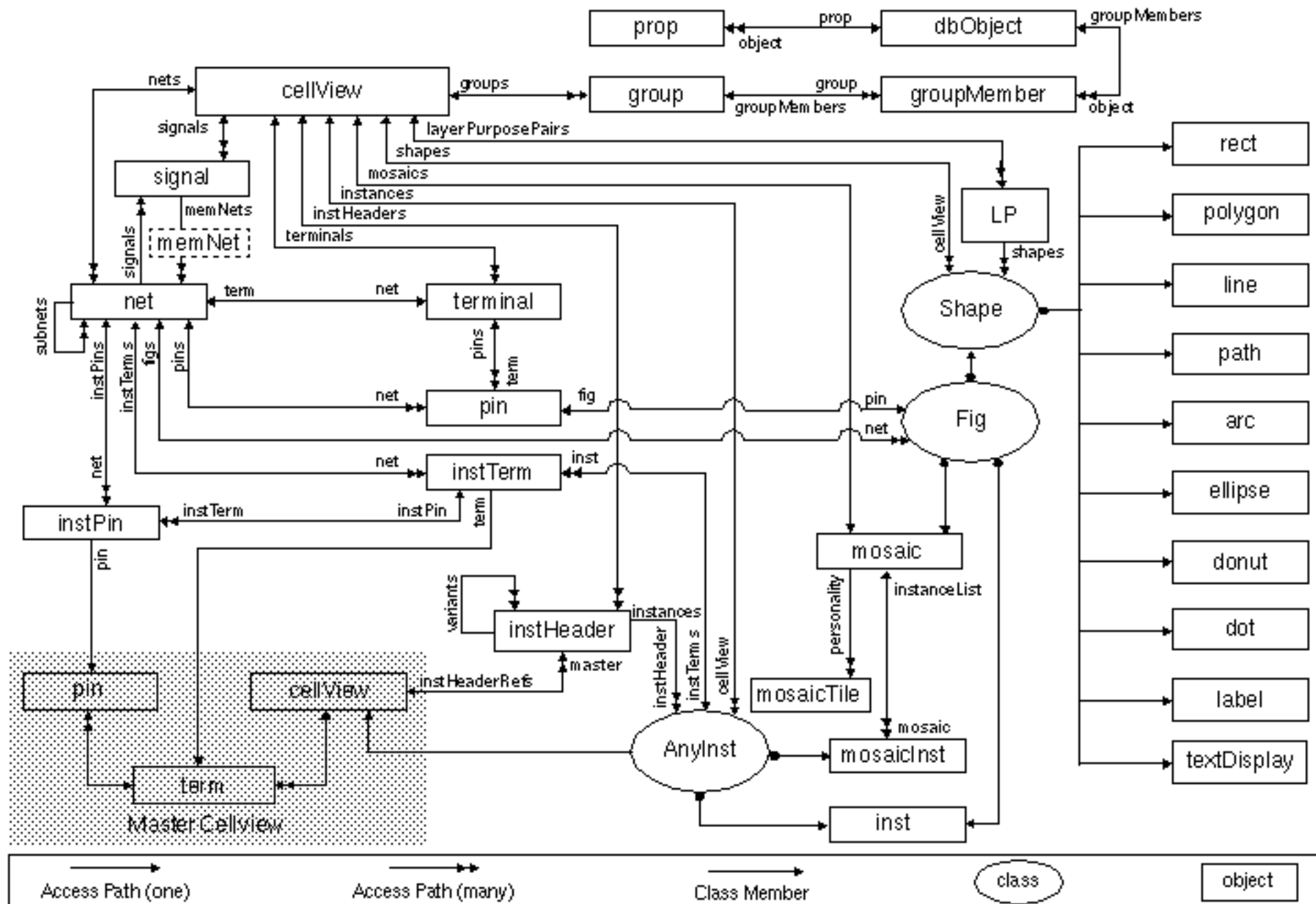
Where to Go for Information

Appendix A

Cellview Data Model Map

Appendix A: Cellview Data Model Map

SKILL CellView Information Model



Appendix B

Case Study Source Code

~/SKILL/CaseStudy/ShowFileBrowser.il

```
;;; the database objects are stored in an association table

trDatabaseObjectTable = nil

defstruct( trDbObjectStruct
  dbObject;;; the database object
  dbNeighborhood;;; the neighborhood of the object
  browserId;;; the window id of the browser
)

;;; dbObjectReference is something like "inst123456"
;;; dbObject is something like db:123456
;;; dbReference is something like "inst: I0 master: inv"

;;; trGetDbObjectReference
;;; insures that the dbObject is entered in the table.
;;; returns the database object reference

procedure( trGetDbObjectReference( dbObject )

  let( ( dbstr dbObjectReference dbObjectStruct )

    dbstr = sprintf( nil "%L" dbObject )

    dbObjectReference = strcat(
      dbObject~>objType ;;; use the objType in symbol name
      substring( dbstr 4 );;; use same digits as in db:123456789
    )
    dbObjectStruct = trDatabaseObjectTable[ dbObjectReference ]
    unless( dbObjectStruct
      trDatabaseObjectTable[ dbObjectReference ] =
        dbObjectStruct = make_trDbObjectStruct( ?dbObject dbObject )
    ) ; unless

    dbObjectReference
  ) ; let
) ;procedure

;;; trDbObjectFromString
;;; returns the database object that corresponds to the textString

procedure( trDbObjectFromString( textString "t" )
  "return the database object that corresponds to the textString"
```

```

trDatabaseObjectTable[ textString ]->dbObject

) ; procedure
;;; trDbObjectReferences
;;; returns an expression with all database objects replaced
;;; by database object references.

procedure( trDbObjectReferences( expression )
  cond(
    ( !expression nil )
    ( dbobjectp( expression ) trGetDbObjectReference( expression ) )
    ( ddIsId( expression ) trGetDbObjectReference( expression ) )
    ( listp( expression )
      cons(
        trDbObjectReferences( car(expression))
        trDbObjectReferences( cdr(expression))
      ) ; cons
    )
    ( t expression )
  ) ; cond
) ; procedure

;;; trNeighborhood
;;; returns the neighborhood of the database object.
;;; enters the database object in the table.

procedure( trNeighborhood( dbObject )
  let( ( attributes attributeValue dbObjectReference )
    attributes= dbObject~>?
    attributeValue= nil
    dbObjectReference= trGetDbObjectReference( dbObject )

    unless( memq( 'objType attributes) ) ;;;; dm bug
      attributes = cons( 'objType attributes)
    ) ; when

    trDatabaseObjectTable[ dbObjectReference ]->dbNeighborhood =
      cons( dbObjectReference
        foreach( mapcar attribute attributes
          attributeValue = dbGet( dbObject attribute)
          list( attribute trDbObjectReferences( attributeValue ) )
        ) ; foreach
      ) ; cons
  ) ;let
) ; procedure

```

```

;;; dbAppReferenceForDBObject
;;; returns an application specific text string
;;; describing the database object
;;; This is used for the browser title

procedure( dbAppReferenceForDBObject( dbObject )
  case( dbObject~>objType
    ( "cellView"
      sprintf( nil "%s: %s %s %s"
        dbObject~>objType
        dbObject~>cellView~>libName
        dbObject~>cellView~>cellName
        dbObject~>cellView~>viewName
        ;;; dbObject~>cellView~>versionName
      )
    )
    ( "net"
      sprintf( nil
        "net: %s"
        dbObject~>name
      )
    )
    ( "inst"
      sprintf( nil
        "inst: %s master: %s"
        dbObject~>name
        dbAppReferenceForDBObject( dbObject~>master )
      )
    )
    ( "instTerm"
      sprintf( nil
        "instTerm: %s %s %s"
        dbObject~>name
        dbAppReferenceForDBObject( dbObject~>net )
        dbAppReferenceForDBObject( dbObject~>inst )
      )
    )
    ( "term"
      sprintf( nil
        "term: %s direction: %s %s"
        dbObject~>name
        dbObject~>direction
        dbAppReferenceForDBObject( dbObject~>net )
      )
    )
  )
)

```

```

    ( t;;; the most generic way to represent a database object symbol
      trGetDBObjectReference( dbObject )
    )
  ) ; case
) ; procedure

;;; trGetTextSelection
;;; returns the reverse of the usual text selection in the view file window

procedure( trGetTextSelection( wid )
  reverse( hiGetTextSelection( wid ) )
) ; procedure

;;; trGetCurrentDBObject
;;; returns the the current database object
;;; which is either the one whose reference is selected
;;; or the browser window.
;;; Note that trGetCurrentDBObject() assumes that
;;; the current window is a browser window.

procedure( trGetCurrentDBObject()
  "return the current database object"
  let( ( browserId dbObject selection dbObjectReference )
    browserId= hiGetCurrentWindow()
    selection= car( trGetTextSelection( browserId ) )
    if( selection
      then
        trDBObjectFromString( selection )
      else
        trDatabaseObjectTable[ browserId->dbObjectReference ]->dbObject
    ) ;if
  ) ; let
) ; procedure

;;; trBrowserPopupMenuCB
;;; the callback routine for the Browser pop-up menu

procedure( trBrowserPopupMenuCB( actionSymbol)
  let( ( browserId )
    browserId = hiGetCurrentWindow()
    caseq( actionSymbol
      ( browse
        trShowFileBrowser(
          trDBObjectFromString(
            car( trGetTextSelection( browserId ) )
          )
        )
      )
    )
  )
) ; procedure

```

```

    )
    ( raiseParentBrowser
      when( windowp(browserId->parentBrowser)
        hiRaiseWindow( browserId->parentBrowser )
      ) ; when
    )
    ( raiseChildBrowser
      when( windowp(browserId->childBrowser)
        hiRaiseWindow( browserId->childBrowser )
      ) ; when
    )
    ( raiseCellViewWindow
      hiRaiseWindow(
        trParentCellViewWindow( trGetCurrentDbObject()~>cellView ) )
    )
    ( raiseCIW
      hiRaiseWindow( hiGetCIWindow() )
    )
    ( clearSelection
      hiUnselectTextAll( browserId )
    )
    (t println( `(Bad call to trBrowserPopupMenuCB: ,actionSymbol)))
  ) ;cond
) ; let
) ; procedure

```

```

;;; trObjectMenus
;;; is an association table that stores a pop-up for each object type

```

```

trObjectMenus = nil

```

```

hiCreateSimpleMenu(
  'trFigurePopUp
  "Figure Menu"
  '(
    "Select in Graphics Editor"
    "Probe Net in Graphics Editor"
  )
  '(
    "trObjTypeMenuCB( 'selectFig )"
    "trObjTypeMenuCB( 'probeFigureNet )"
  )
  "" ; empty help string
) ; hiCreateSimpleMenu

```

```

hiCreateSimpleMenu(
  'trNetPopUp
  "Net Menu"
  '(

```

```

    "Probe in Graphics Editor"
  )
  '(
    "trObjTypeMenuCB( 'probeNet )"
  )
  "" ; empty help string
) ; hiCreateSimpleMenu

trFigureObjTypes = '(
  "arc"
  "dot"
  "donut"
  "ellipse"
  "label"
  "line"
  "path"
  "polygon"
  "rect"
  "textDisplay"
  "inst" ;;; conflicts with the inst menu
  ;;; "mosaic"
  ;;; "mosaicInst"
)

;;; trInitObjectMenus
;;; stores the various object menus in the trObjectMenu table

procedure( trInitObjectMenus( )
  trObjectMenus = makeTable( "Object Menus" nil )

  ;;; store the trFigurePopUp for each figure object type
  foreach( objType trFigureObjTypes
    trObjectMenus[ objType ] = trFigurePopUp
  ) ; foreach

  trObjectMenus[ "net" ] = trNetPopUp
) ; procedure

;;; trObjectTypePopUp
;;; pops up the object menu for the selected object

procedure( trObjectTypePopUp()
  "pop up the menu based on the text selection"
  let( ( menuToDisplay dbObject )
    dbObject = trGetCurrentDbObject()
    menuToDisplay = trObjectMenus[ dbObject~>objType ]
    if( menuToDisplay

```



```

        then
            hiDisplayMenu( menuToDisplay )
        else
            printf( "\nNo menu for %L" dbObject )
        ) ; if
    ) ; let
) ; procedure

;;; trObjTypeMenuCB
;;; is the common callback for all the object type menus

procedure( trObjTypeMenuCB( actionSymbol )
    let( ( browserId dbObject )
        browserId= hiGetCurrentWindow()
        dbObject= trGetCurrentDbObject()
        caseq( actionSymbol
            ( openCellView
                trParentCellViewWindow( dbObject ) )
            ( selectFig
                trSelectFig( dbObject ) )
            ( probeFigureNet
                trProbeFigureNet( dbObject ) )
            ( probeNet
                trProbeNet( dbObject ) )
            ( t printf( "\nBad call to trObjTypeMenuCB: %L" actionSymbol ) )
        ) ; caseq
    ) ; let
) ; procedure

;;; trOpenCellView
;;; opens a window on cellView

procedure( trOpenCellView( cv )
    geOpen(
        ?window nil
        ?lib      cv~>libName
        ?cell      cv~>cellName
        ?view      cv~>viewName
        ;;; ?versioncv~>versionName
        ?mode      "r"
    )
) ; procedure

;;; trParentCellViewWindow
;;; returns a window viewing the dbObject's cellView ...
;;; Either take the first window viewing the cellview
;;; or open a new window

```

```

procedure( trParentCellViewWindow( dbObject )
  let( ( theWindow theCellView )
    theCellView= dbObject~>cellView
    theWindow=
      car(
        exists( wid hiGetWindowList()
          hiGetWidgetType( wid ) == "graphics" &&
          theCellView == geGetWindowCellView( wid )
        ) ; exists
      ) ; car
    theWindow || trOpenCellView( theCellView )
  ) ; let
) ; procedure

;;; trSelectFig
;;; selects the figure in the parent cell view window

procedure( trSelectFig( dbObject )
  let( ( cellViewWindowId )
    cellViewWindowId = trParentCellViewWindow( dbObject )
    when( cellViewWindowId
      geDeselectAll( cellViewWindowId )
      geDeleteAllProbe( cellViewWindowId ) ;;; a probe can obscure the selection
      if( geSelectFig( dbObject )
        then
          hiRaiseWindow( cellViewWindowId )
          t
        else
          printf( "\nUnable to select figure %L on lpp %L"
            dbObject dbObject~>lpp )
          nil
      ) ; if
    ) ; when
  ) ; let
) ; procedure

;;; trProbeNet
;;; puts a net probe on the net
;;; using trProbeColor

trProbeColor = '( "y1" "drawing" )

procedure( trProbeNet( dbObject )
  let( ( cellViewWindowId )
    cellViewWindowId = trParentCellViewWindow( dbObject )
    when( cellViewWindowId
      geDeleteAllProbe( cellViewWindowId )
      geMakeProbeWithColor(

```

```

        ?probeType "net"
        ?object dbObject
        ?color trProbeColor
        ?window cellViewWindowId
    )
    hiRaiseWindow( cellViewWindowId )
) ; when
) ; let
) ; procedure

;;; trProbeFigureNet
;;; probes the figure's net

procedure( trProbeFigureNet( dbObject )
let( ( cellViewWindowId net )
    cellViewWindowId = trParentCellViewWindow( dbObject )
    cond(
        ( !cellViewWindowId
            printf( "\nUnable to Probe: can't locate Cell View Window" )
        )
        ( !( net = dbObject~>net )
            printf( "\nFigure %L has no associated net" dbObject )
        )
        ( t
            geDeleteAllProbe( cellViewWindowId )
            geMakeProbeWithColor(
                ?probeType "net"
                ?object net
                ?color trProbeColor
                ?window cellViewWindowId
            )
            hiRaiseWindow( cellViewWindowId )
        )
    ) ; cond
) ; let
) ; procedure

procedure( trBrowserPopUpMenu( browserId )
let( ( menuSymbol )
    menuSymbol = hiCreateSimpleMenu(
        'browserPopUp
        "Browser Menu" ;menu title
        '(
            "Browse Selection"
            "Raise Parent Browser"
            "Raise Most Recent Child Browser"
            "Raise CellView Window";;; should be on other menu(s) ??
            "Raise CIW"

```

```

        "Clear Selection"
    )

    '(
        "trBrowserPopupMenuCB( 'browse )"
        "trBrowserPopupMenuCB( 'raiseParentBrowser )"
        "trBrowserPopupMenuCB( 'raiseChildBrowser )"
        "trBrowserPopupMenuCB( 'raiseCellViewWindow )"
        "trBrowserPopupMenuCB( 'raiseCIW )"
        "trBrowserPopupMenuCB( 'clearSelection )"
    )
    "" ;empty help string
)
hiSetMenu( menuSymbol browserId )
menuSymbol;;; return value
) ; let
) ; procedure

procedure( trDumpDatabaseObject( dbObject pathname )
    "Use pp to dump the database object's neighborhood to the file"

    let( ( tmpPort dbNeighborhood )

        dbNeighborhood = trNeighborhood( dbObject )
        tmpPort = outfile( pathname "w" )
        pp( dbNeighborhood tmpPort )
        close( tmpPort )
    ) ; let

) ; procedure

;;; trOpenBrowser
;;; Opens a browser on the dbObject.
;;; Return the length of the trDatabaseObjectTable.

procedure( trOpenBrowser( dbObjectReference pathname )
    "Open a browser window on dbObject"

    let( ( browserId parentId dbObjectStruct )

        parentId = hiGetCurrentWindow();;; might be nil
        dbObjectStruct = trDatabaseObjectTable[ dbObjectReference ]
        browserId = view( ;;; Open up a view file
            pathname;;; the file contains the dumped database object
            trBrowserBBox;;; bbox
            dbAppReferenceForDbObject(;;; title
                dbObjectStruct->dbObject )
        )
    )

```

```

    ;; use window properties to cross-correlate the windows

    browserId->parentBrowser = parentId
    when( parentId
        parentId->childBrowser = browserId
        ) ; when

    dbObjectStruct->browserId = browserId

    browserId->dbObjectReference= dbObjectReference

    ;; install the pop-up menu

    trBrowserPopUpMenu( browserId )
    length( trDatabaseObjectTable )
    ) ; let
) ; procedure

;;; trShowFileBrowser
;;; Opens a browser on the dbObject.
;;; If one already exists, raise it.

procedure( trShowFileBrowser( dbObject )
    "Open a browser window on dbObject. If one already exists, raise it."
    let( ( dbObjectReference pathname dbObjectStruct )
        dbObjectReference = trGetDbObjectReference( dbObject )
        dbObjectStruct= trDatabaseObjectTable[ dbObjectReference ]
        if( windowp( dbObjectStruct->browserId )
            then
                hiRaiseWindow( dbObjectStruct->browserId )
            else
                pathname = strcat( "/tmp/" dbObjectReference )
                trDumpDatabaseObject( dbObject pathname )
                trOpenBrowser(
                    dbObjectReference
                    pathname
                )
            ) ; if
        ) ; let
    length( trDatabaseObjectTable )
    ) ; procedure

;;; trBrowseSelectedSet
;;; opens a browser on the first object in the selected set

procedure( trBrowseSelectedSet( @optional ( wid hiGetCurrentWindow() ) )
    let( ( object )
        object = car( geGetSelSet( wid ) )
    )

```

```

    when( object
        trShowFileBrowser( object )
    ) ; when
) ; let
) ; procedure

;;; trBrowseCellView
;;; opens a browser on the cellview in the window

procedure( trBrowseCellView( @optional ( wid hiGetCurrentWindow() ) )
    let( ( cv )
        cv = geGetWindowCellView( wid )
        when( cv
            trShowFileBrowser( cv )
        ) ; when
    ) ; let
) ; procedure

;;; trInitShowFileBrowser
;;; set bind keys used by the Show File Browser

procedure( trInitShowFileBrowser()
    trCloseBrowsers()
    trDatabaseObjectTable = makeTable( "Database Object Table" nil )
    trInitObjectMenus( )
    hiSetBindKey( "Show File" "None<Btn2Down>" "mousePopUp()" )
    hiSetBindKey( "Show File" "None<Btn3Down>" "trObjectTypePopUp()" )

    printf( "\nThe Show File Browser is initialized" )
    t
) ; procedure

;;; trCloseBrowsers
;;; closes all the browser windows

procedure( trCloseBrowsers()
    let( ( wid )
        foreach( key trDatabaseObjectTable
            when( wid = trDatabaseObjectTable[ key ]->browserId
                errset( hiCloseWindow( wid ) )
            ) ; when
        ) ; foreach
    ) ; foreach
) ; procedure

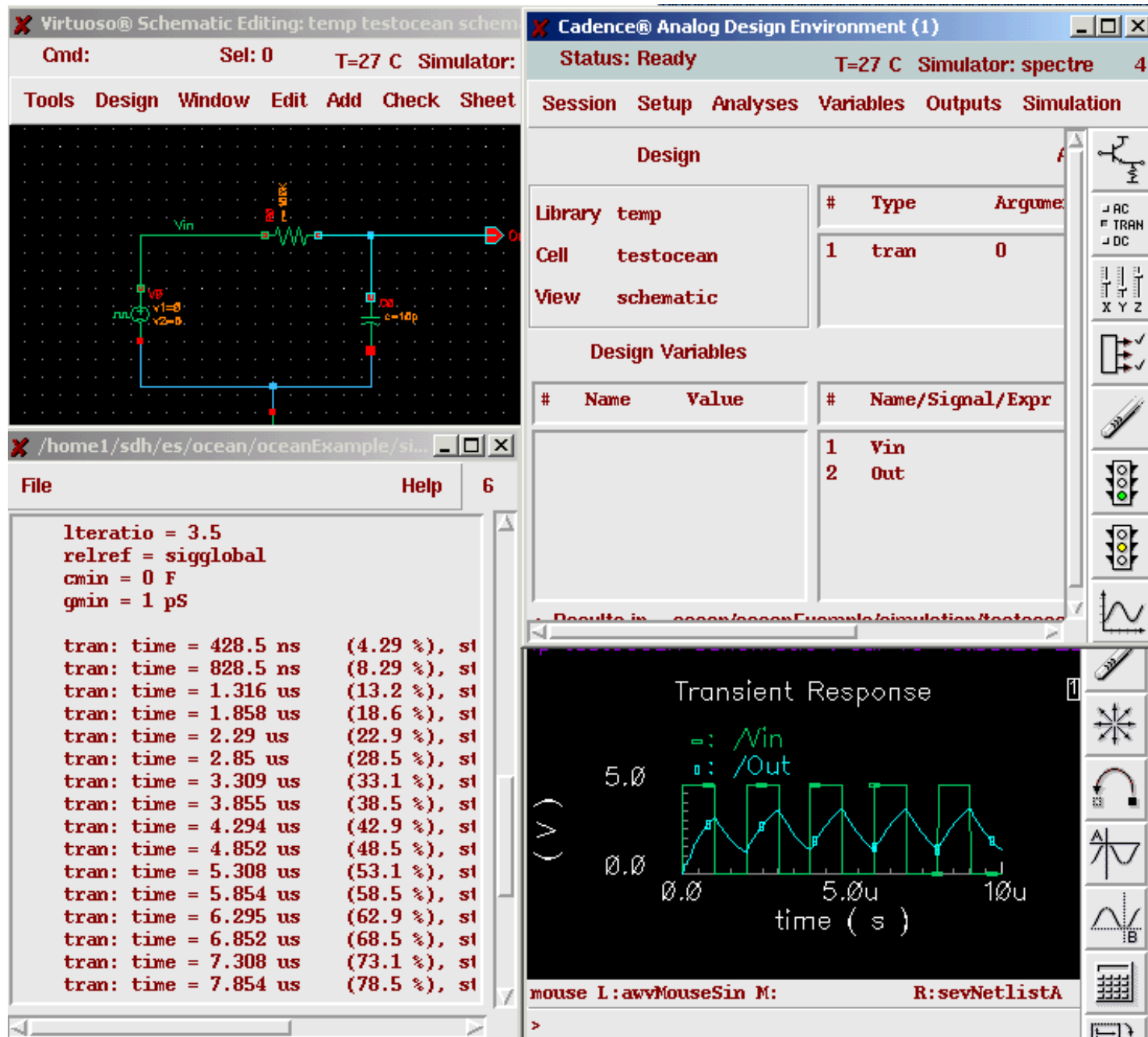
trInitShowFileBrowser()

```

Appendix C

OCEAN Example

Appendix C: OCEAN Example



OCEAN Example

OCEAN SKILL Script to Control Simulation

- The Analog Design Environment (ADE) provides the option of controlling the simulation flow using a scripted batch mode. This feature and the supporting API functions are called OCEAN (Open Command Environment for Analysis).
- You will run a simulation interactively and save the steps you execute into a script file.
- Then you can edit the script and re-execute the steps within the OCEAN environment by re-loading the script file.
- This is an example of how your knowledge of SKILL can be applied to daily engineering tasks.

OCEAN SKILL Script to Control Simulation

SKILL Script Output from Session->Save Script...

```
simulator( 'spectre ' )
design(
"/home1/userA/oceanExample/simulation/testocean/spectre/schematic/
netlist/netlist")
resultsDir(
"/home1/userA/oceanExample/simulation/testocean/spectre/schematic" )
analysis('tran ?stop "10u"  )
temp( 27 )
run()
selectResult( 'tran ' )
plot(getData("/Vin") getData("/Out") )
```

SKILL Script Output from Session->Save Script...

The content of this file reflects the simulation setup and execution just completed.

You can further edit this file and use SKILL constructs such as loops and variables to customize the script.

For example, you could place the expressions in a for loop and use a variable for the temperature to plot the results over a temperature range.

Running the SKILL Script in Batch Mode

Once you have saved and edited the SKILL script you can invoke the OCEAN batch mode environment:

```
unix> ocean
```

Then load the script to execute the simulation steps:

```
ocean> load "oceanScript.ocn"
```

The OCEAN script runs and the waveforms are plotted.

Running the SKILL Script in Batch Mode

Modifying the SKILL Script

You can modify the saved script and then re-execute it by loading it in the OCEAN environment.

This example shows a loop added to run the simulation at defined temperatures.

A unique result directory is generated for each run.

```
simulator( 'spectre )
design( "../oceanEx/simulation/testocean/spectre/schematic/netlist/netlist" )
analysis( 'tran ?stop "10u" )
templist = '(0 27 100)
foreach( tempval templist
    temp( tempval )
    rdir = resultsDir( sprintf( nil "Run_Temp_%d" tempval) )
    printf( "%L" rdir )
    run()
    selectResult( 'tran )
    plot(getData("/Vin") getData("/Out") )
)
```

Modifying the SKILL Script

The standard SKILL constructs include the:

- *templist* assignment
- *foreach* loop
- *sprintf* command
- *printf* command

The other functions are specific to OCEAN and include:

- *simulator*
- *design*
- *analysis*
- *temp*
- *resultsDir*
- *run*
- *selectResult*
- *plot*
- *getData*

OCEAN Demonstration

To see a demonstration of OCEAN, open the file `./animations/oceanscript.gif` in a Web browser.

OCEAN Demonstration