FORMATION SKILL

Beginning to Use SKILL

Module 2

July 27, 2005

Module 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.

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.

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	<*cds*> &	Use an ampersand (&)
Nongraphic	<*cds*> -nograph	Do not use an ampersand (&)
Replay a session	<*cds*> -replay *~/OldCDS.log &*	

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

Graphic Sessions

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

Nongraphic Sessions

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

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

During a nongraphic session, the 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.

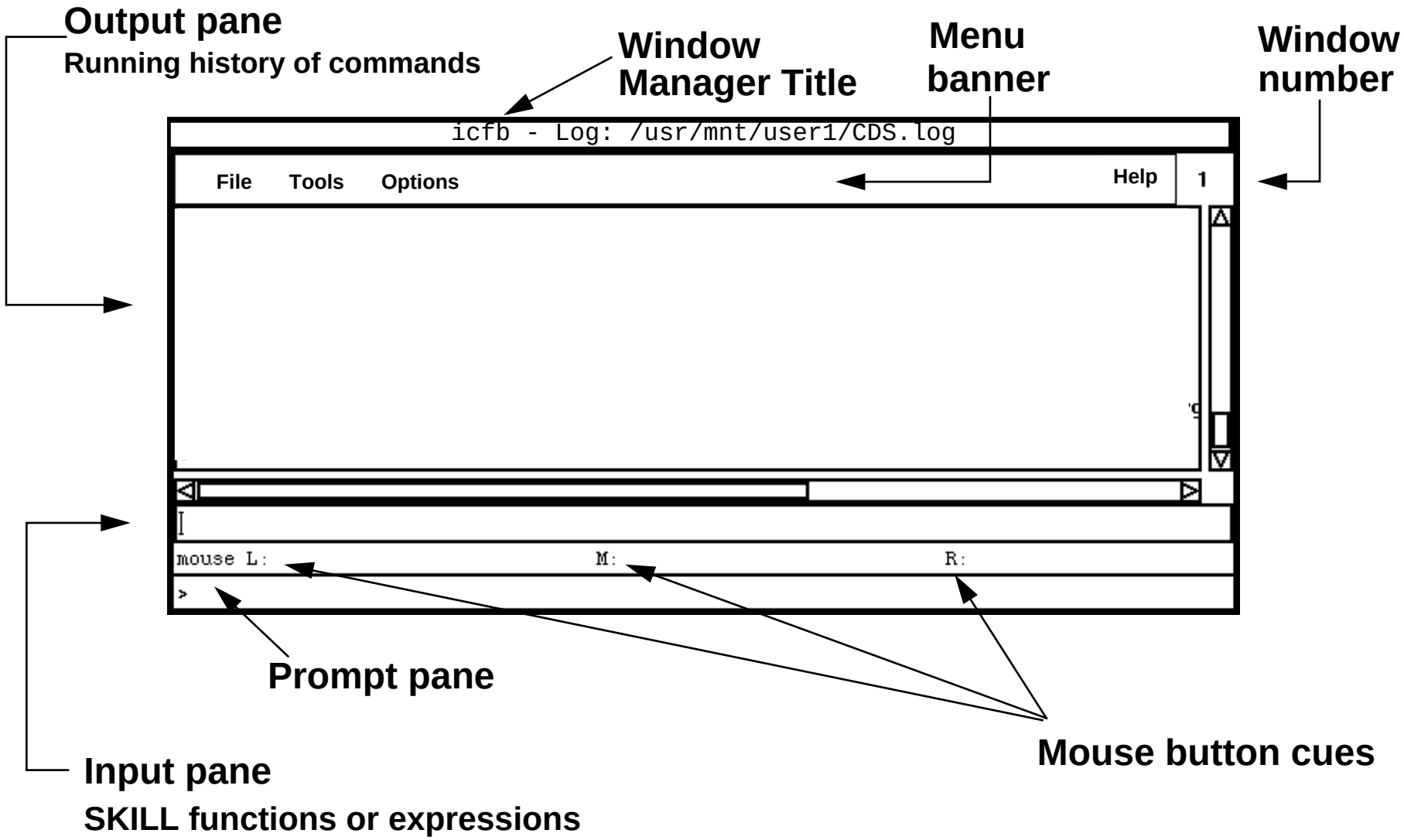
Typically, you use the *.cdsinit* file to define application bindkeys and load customer-specific SKILL utilities.

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

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

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

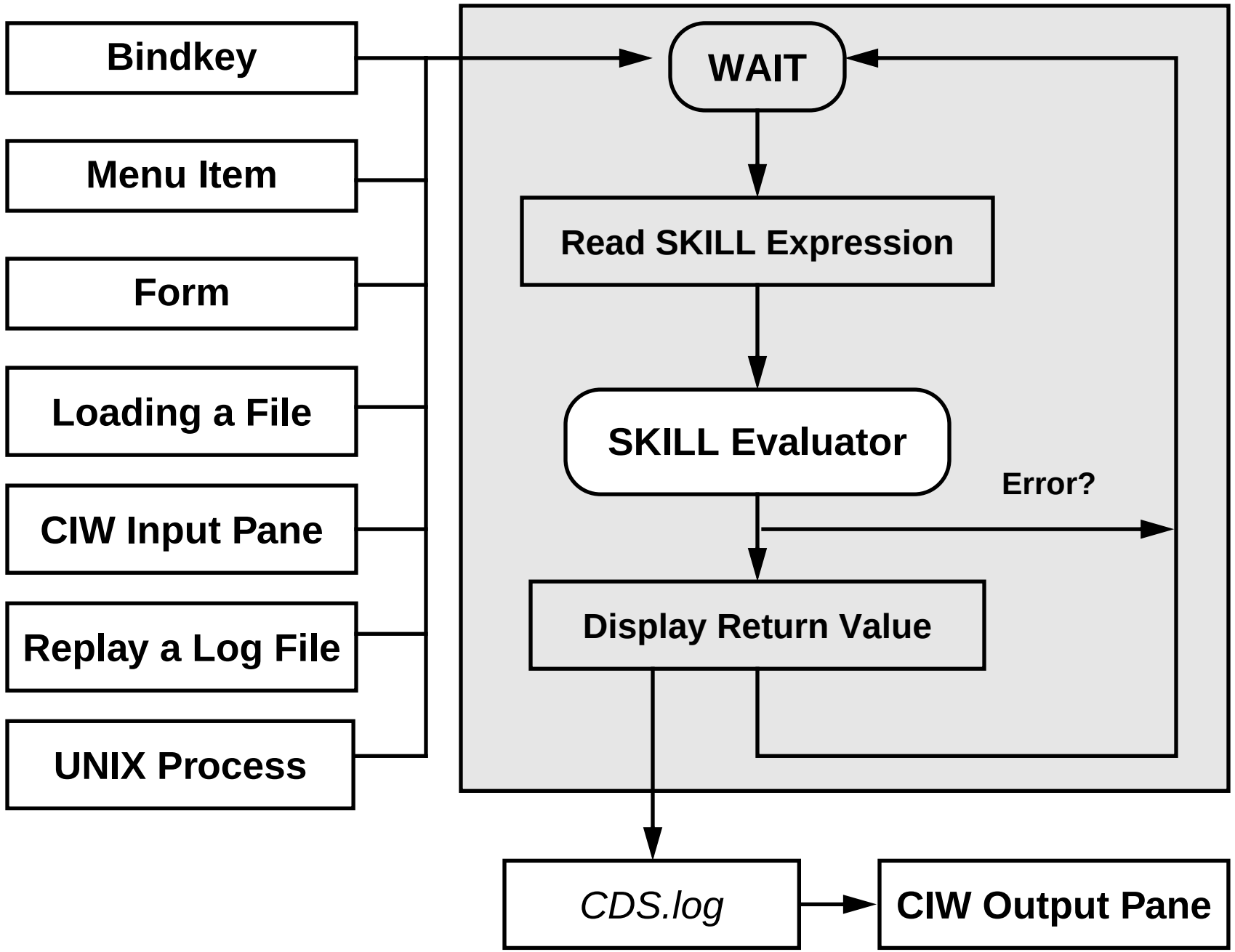
Command Interpreter Window



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



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 definition of the *TrBump* function is:

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

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

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

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

```
hiLogDragEvents( t )
```

The CDS.log File Code

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

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

Setting the Log Filter

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

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

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

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

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

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

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

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

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

SKILL Syntax Summary

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

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

Data

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

- You use the input syntax to specify data as an argument or to assign a value to a variable.
- 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	*5*	*5*	*x*	*x_count*
floating point	*5.3*	*5.3*	*f*	*f_width*
text string	*"this is text"*	*"this is text"*	*t*	*t_msg*
list	*'(1 "two" 3 4)*	*(1 "two" 3 4)*	*l*	*l_items*

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

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. Variable names are case-sensitive.

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

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

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

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.

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.

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")
```

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)
```

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

Lab Overview

Lab 2-1 Starting the Software

Lab 2-2 Using the Command Interpreter Window

Lab 2-3 Exploring SKILL Numeric Data Types

Lab 2-4 Exploring SKILL Variables

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.

The *print* and *println* Functions

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

This nongraphic session transcript illustrates *println*.

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

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

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

Displaying Data with Format Control

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

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

The first argument is a conversion control string containing directives.

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

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

The *printf* Function

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

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

The %L Directive

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

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

Solving Common Problems

These are common problems you might encounter:

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

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.

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
```

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
```

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 2-5 Displaying Data in the CIW

Lab 2-6 Solving Common Input Errors

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

cadence®

Developing a SKILL Function

Module 8

July 27, 2005

Module 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.

Curly Braces

The following statements refer to the example above:

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

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

Grouping Expressions with Local Variables

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

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

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

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

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

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

The *let* Function

You can freely nest *let* statements.

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

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

Whenever the flow of control enters a *let* function, 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
*Error* let: too few arguments (at least 2 expected, 1 given)
```

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

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

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

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.

Defining Required Function Parameters

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

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

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

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

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

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

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

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

Here's how 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:lly+dy
      urx+dx:ury+dy
      )
  ) ; let
) ; procedure
```

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

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

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

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

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

Defining Keyword Function Parameters

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

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

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

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

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

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

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

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

Collecting Function Parameters into a List

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

Example

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

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

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

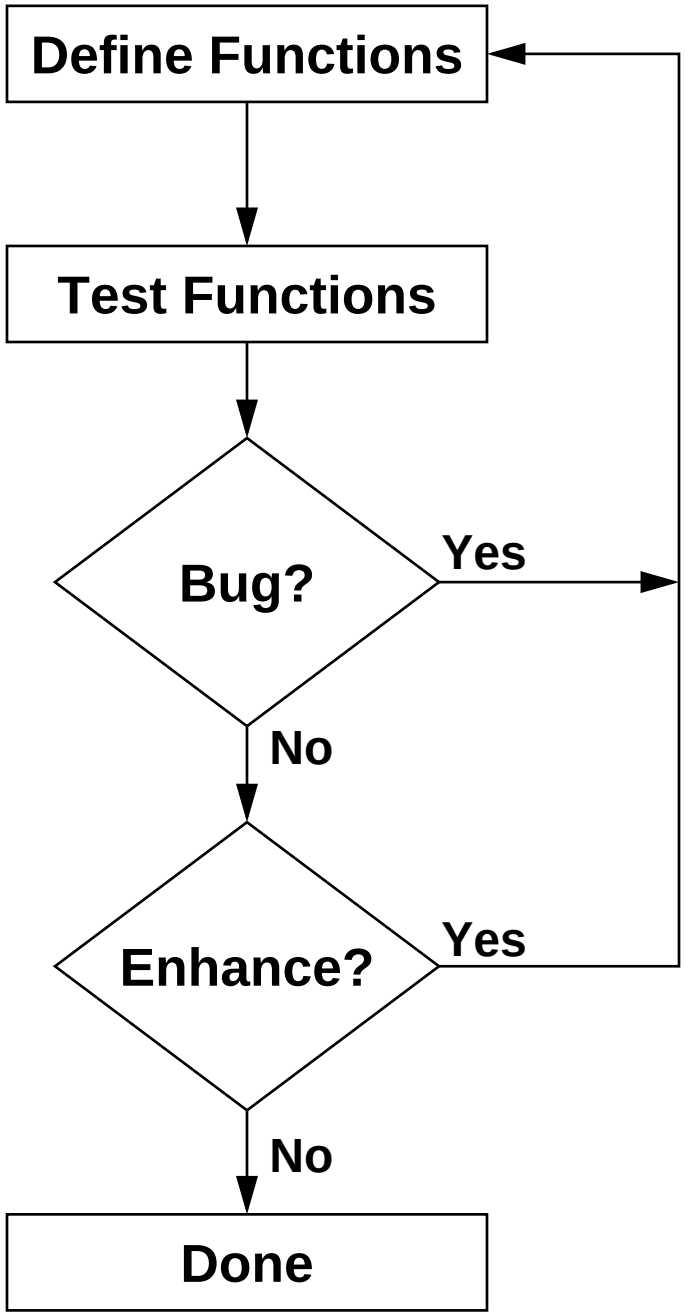
To call the *TrCreatePath* function, specify any number of points.

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

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

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

SKILL Development Cycle



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.

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.

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

```
:set nonumber
```

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

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

Why is that?

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

Redefining a SKILL Function

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

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

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

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

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

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

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

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

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

Lab Overview

Lab 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*.

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

cadence®

SKILL Lists

Module 3

July 27, 2005

Module Objectives

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

What Is a SKILL List?

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

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

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

SKILL functions commonly return lists you can display or process.

Design database list examples:

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

User interface list examples:

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

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

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

How 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 ))
```

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

Creating New Lists

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

- Specify all the elements literally. Apply the ’ operator to the list.

Expressions	Return Result
’(1 2 3)	*(1 2 3)*
’("one" 1)	*("one" 1)*
’(("one" 1) ("two" 2))	*(("one" 1) ("two" 2))*

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

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

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
result = '(2 3)	*(2 3)*
result = cons(1 result)	*(1 2 3)*

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

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

The *cons* Function

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

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

The *append* Function

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

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

Points of Confusion

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

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

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

Working with Existing Lists

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

The *nth* Function

Lists in 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 conceptually represented as an inverted tree. To traverse a left branch use **car** and to traverse a right branch use **cdr**.

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

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

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

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

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

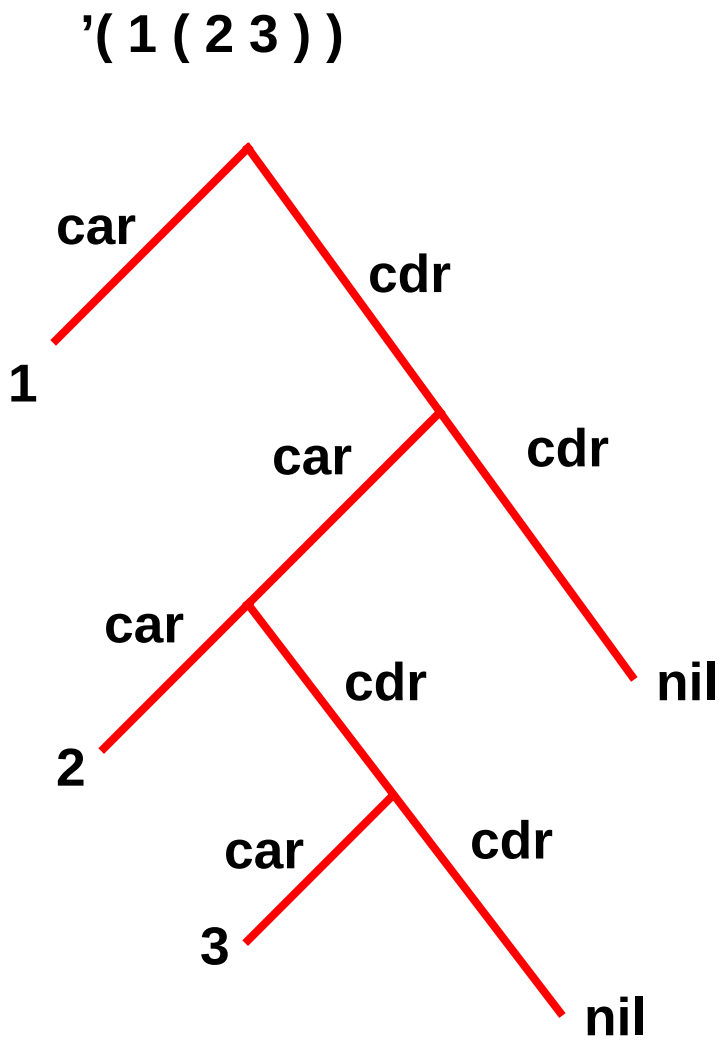
5: **car(car(cdr(alist))) => 2**

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

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

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

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



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?

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

Two-Dimensional Points

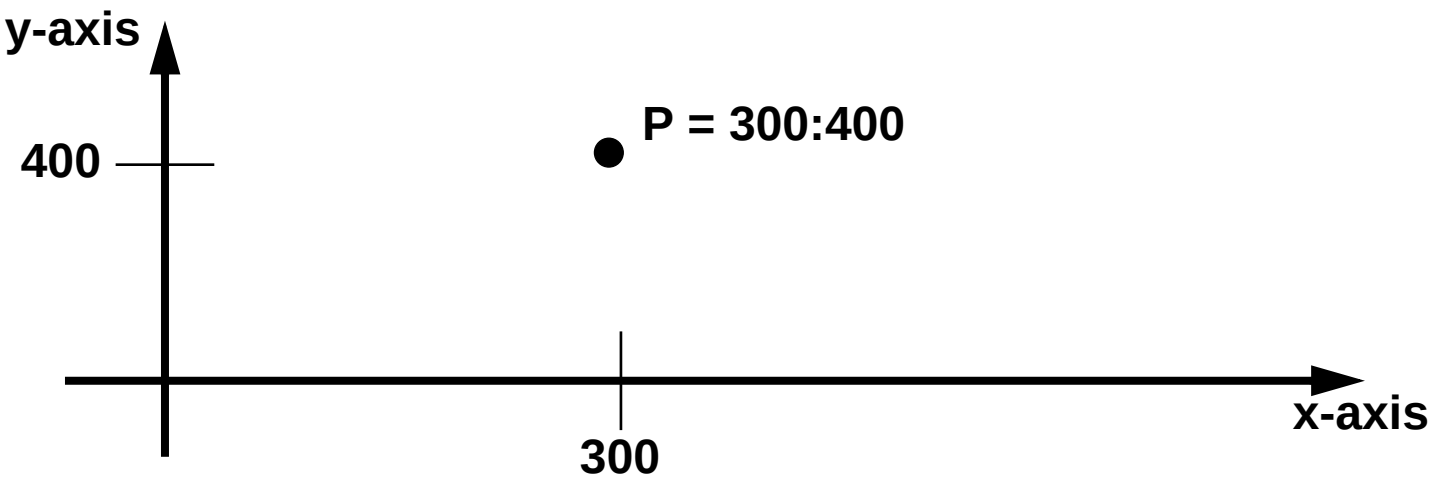
SKILL represents a two-dimensional point as a two-element list.

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
```



The : Operator

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

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

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

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

The *xCoord* and *yCoord* Functions

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

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

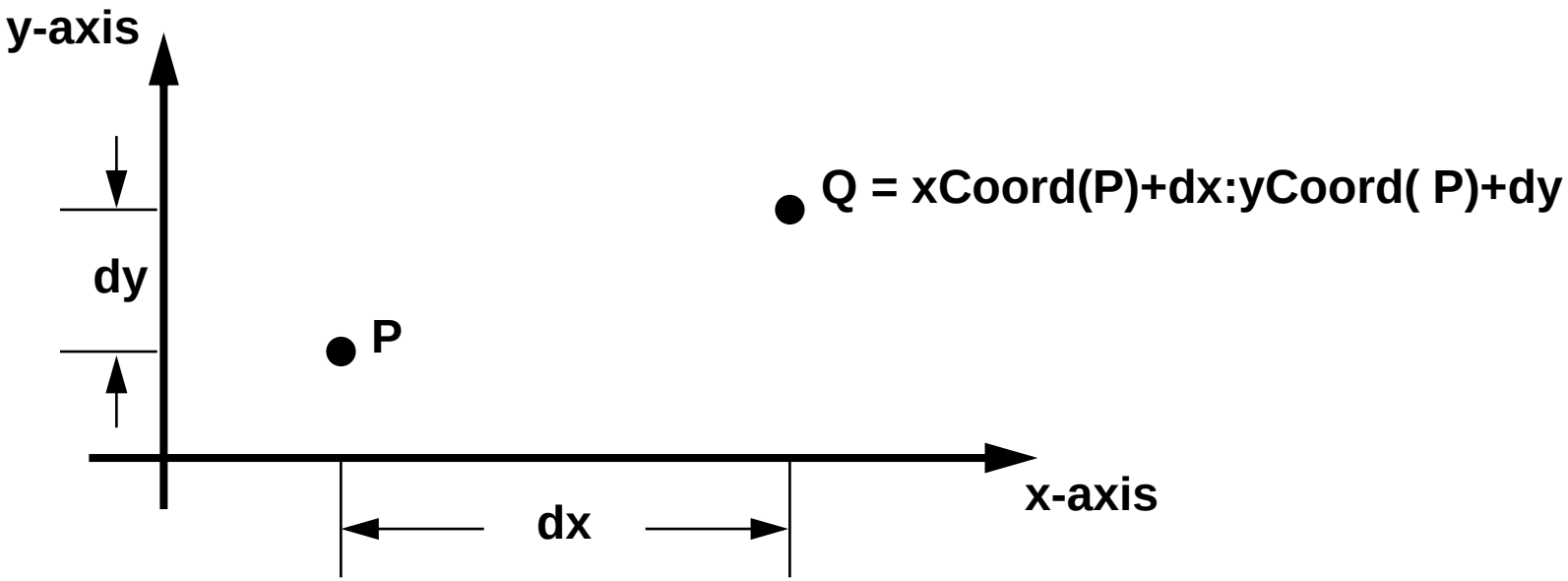
Computing Points

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

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

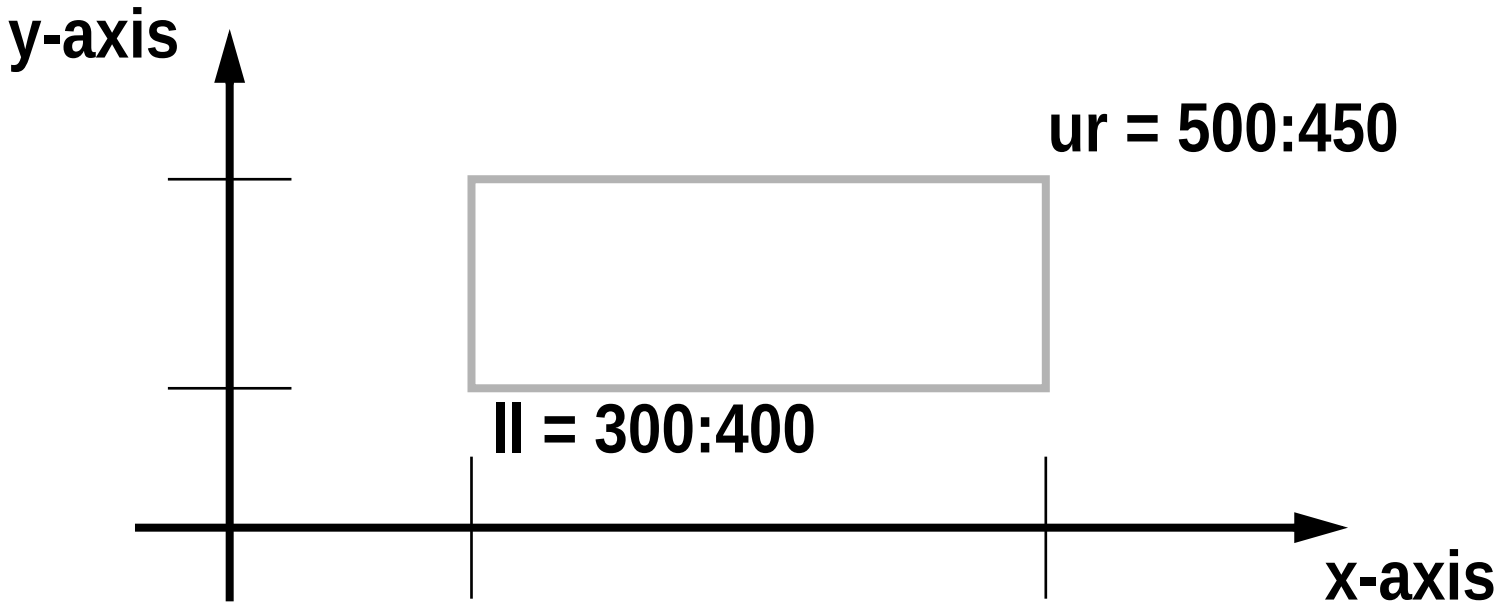
Computing a point from another point is easy.

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



Bounding Boxes

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))

Remember that : is a point operator.

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

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

Creating a Bounding Box

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

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

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

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

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

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

Retrieving Elements from Bounding Boxes

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

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

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

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

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

Lecture Exercises

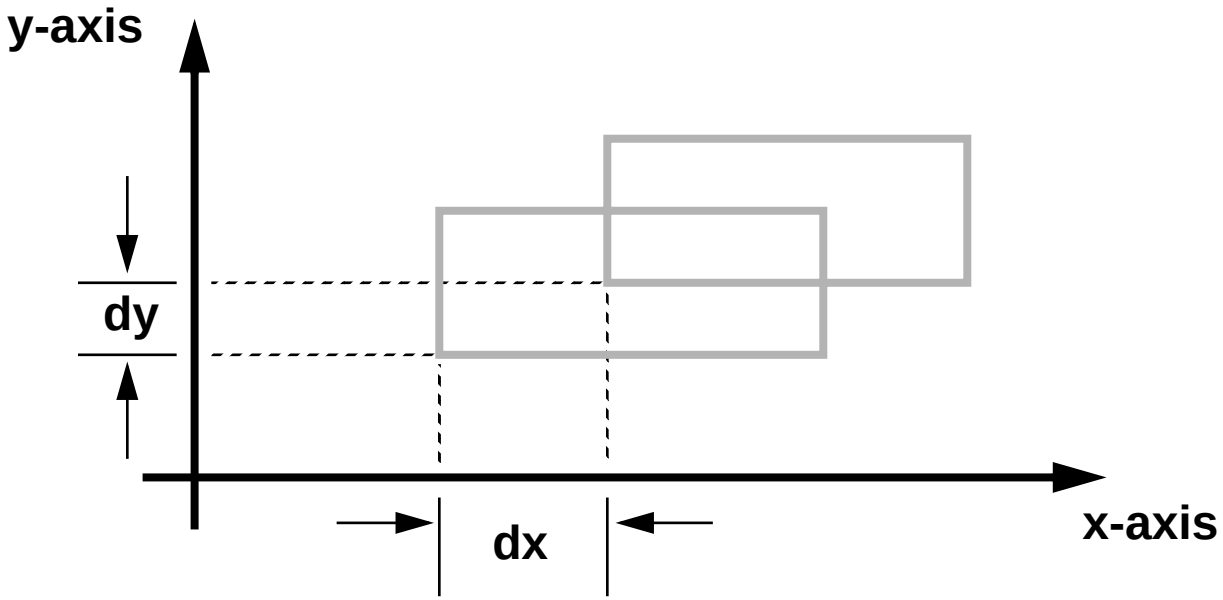
The exercises on the next few pages illustrate techniques for manipulating bounding boxes:

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

Offsetting a Box

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

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



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

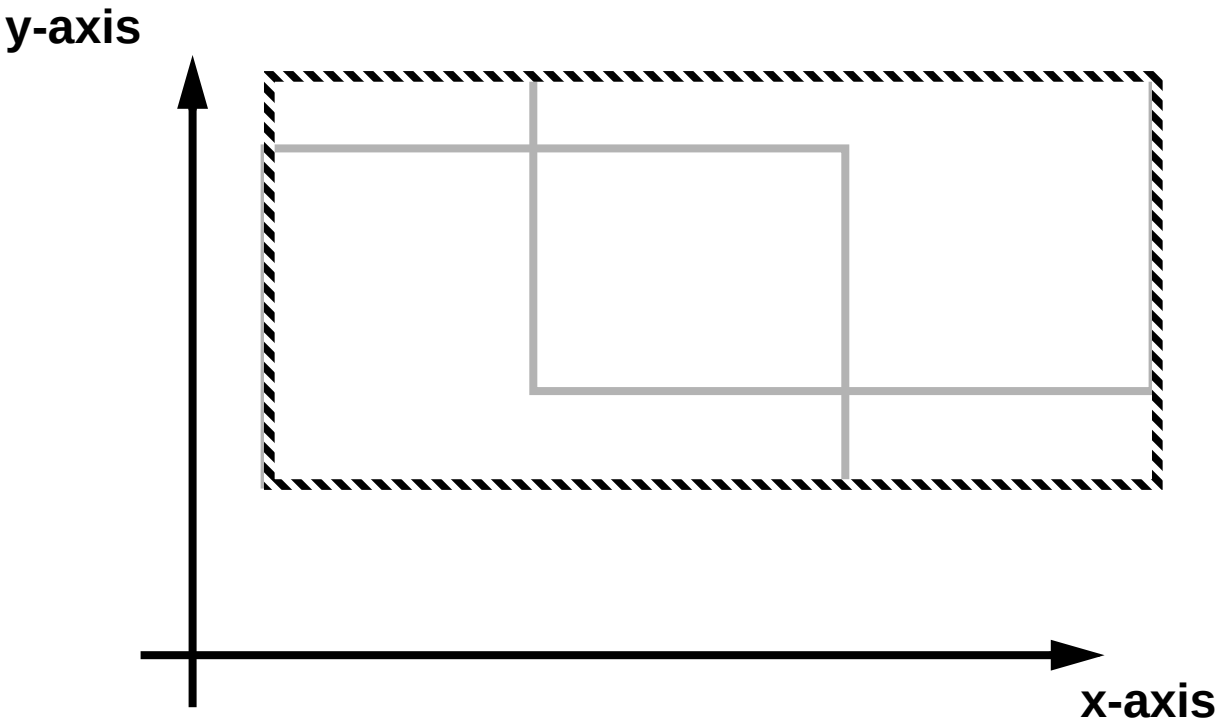
```
boxLL = ... : ...
boxUR = ... : ...
list( boxLL boxUR )
```

To view the solution, turn the page upside down.

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

Finding the Smallest Bounding Box

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



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

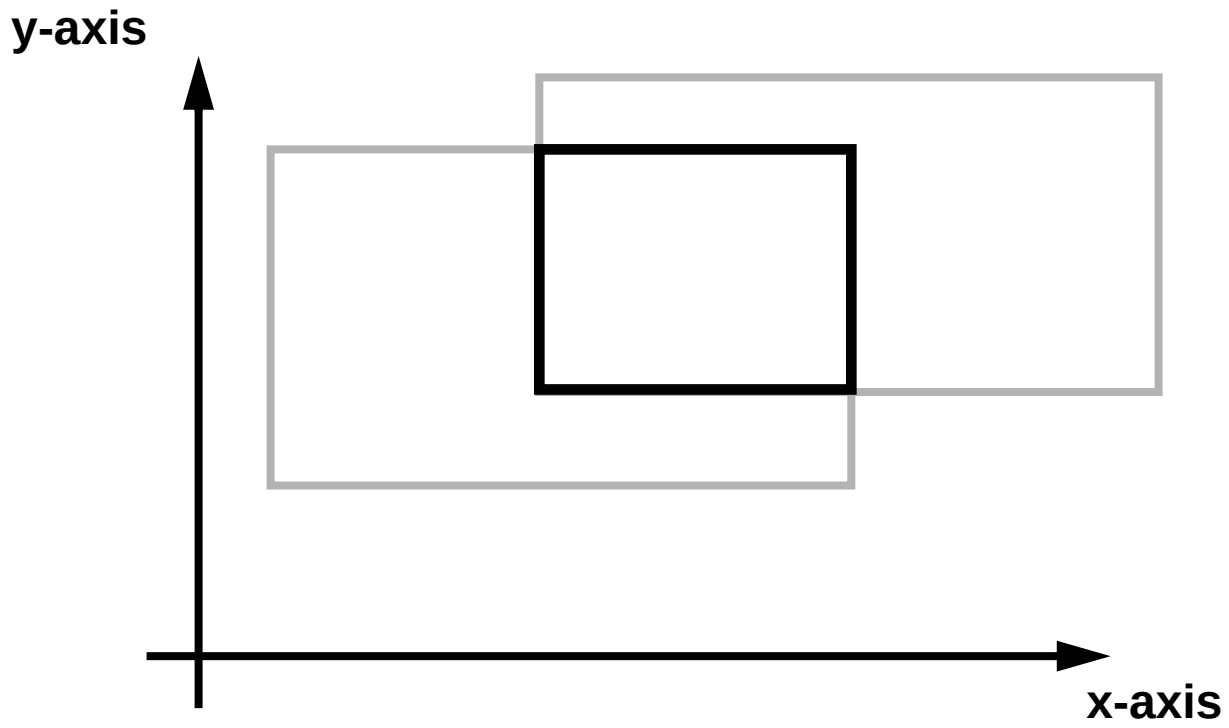
```
boxLL = min( ... ... ):min( ... ... )
boxUR = max( ... ... ):max( ... ... )
list( boxLL boxUR )
```

To view the solution, turn the page upside down.

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

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
car	car(...)	lower left corner	ll = car(bBox)
cadr	car(cdr(...))	upper right corner	ur = cadr(bBox)
caar	car(car(...))	x-coord of lower left corner	llx = caar(bBox)
cadar	car(cdr(car(...))	y-coord of lower left corner	lly = cadar(bBox)
caadr	car(car(cdr(...)))	x-coord of upper right corner	urx = caadr(bBox)
cadadr	car(cdr(car(cdr(...]	y-coord of upper right corner	ury = cadadr(bBox)

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

The *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 Creating New Lists

Lab 3-2 Extracting Items from Lists

Module Summary

In this module we covered:

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

List Construction

Module 12

July 27, 2005

Module Objectives

- Review the techniques for building a list.
  - The ’ operator
  - The *list* function
  - The *cons* function
  - The *append* function
- Use the *foreach mapcar* function to make a list that corresponds one to one with a given list.
- Use the *setof* function to make a filtered copy of a list.
- Use the *foreach mapcar* function and *setof* function together to make a list that corresponds one to one with a filtered list.

List 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 )
```

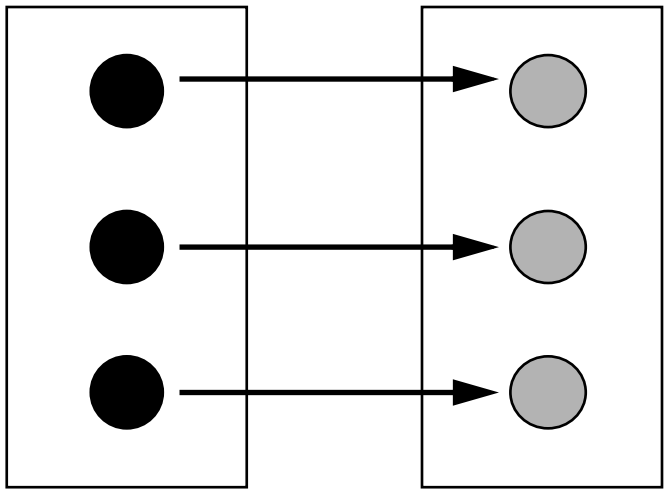
- Merging two lists together

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

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

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

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

Extended *foreach mapcar* Example

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

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

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

- However, you are responsible for collecting the return results. Use the **cons** function.
- In addition, the list will be in reverse order. Use the *reverse* function to make a copy of the list in the correct order.

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

winNames = reverse( winNames )
```

The *mapcar* Function

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

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

The *mapcar* function accepts two arguments:

- A function of one argument
- A list

The *mapcar* function

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

Examples

- A *foreach mapcar* expression

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

- An equivalent *mapcar* expression

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

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)
```

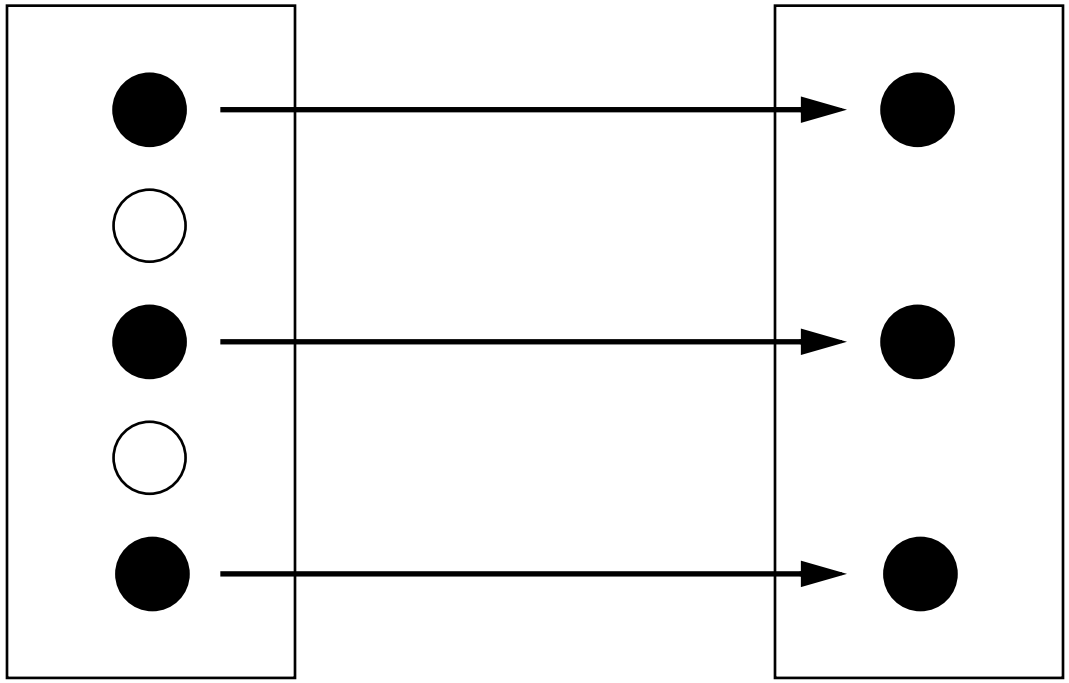
- A trace of a *foreach mapcar* expression
  funobj:0x21980a8 represents the function that SKILL dynamically generates to represent *x**2*.

```
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)*.



Filtered list

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

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

The *setof* Function

The *setof* function accepts several arguments:

- A local variable that holds a list element in the filter expressions.
- The list to filter.
- One or more expressions that compose the filter.

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

- Binds each element to the local variable.
- Evaluates the expressions in its body.
- Uses the result of the last expression to determine whether to include the element.

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

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

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

A *setof* Example

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

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

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

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

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:

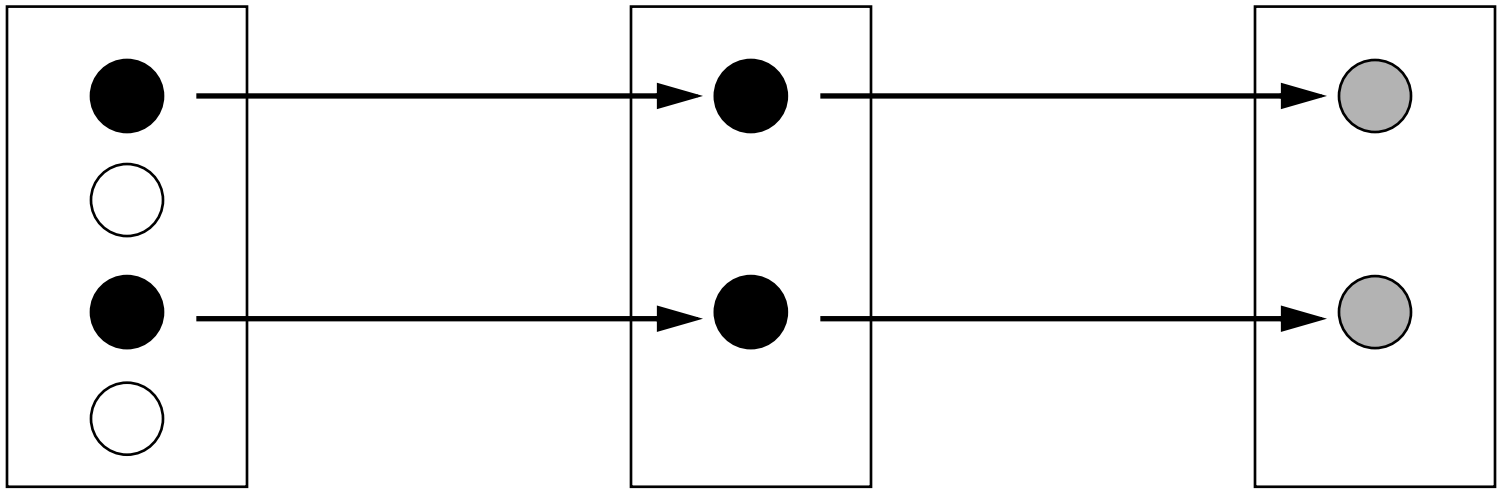
```
procedure( TrIntersect( list1 list2 )
  let( ( result )
    foreach( element1 list1
      when( member( element1 list2 )
        result = cons( element1 result )
      ) ; when
    ) ; foreach
    result
  ) ; let
) ; procedure
```

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

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

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.

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 Layer Shape Report

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

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

Lab 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" )
```

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

cadence®

Data Structures

Module 17

July 27, 2005

Module Objectives

- Implementing records in SKILL
  - Disembodied property lists
  - SKILL structures
- Arrays
- Associative data structures
  - Association tables
  - Association lists

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 any value, typically *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
```

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" )
  => card@0xa8e8378
```

3. Use the *type* function to determine the structure name.

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

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 )
```

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)
dayNum = 0
foreach( day days
  week[dayNum++] = 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 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 => card@0xa8e8378
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
```

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.

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()
     makeTable( 'TrSparseArray )
) ; procedure

TrTimesTable = Tr2D_DynamicArray( )
for( i 0 9
  for( j 0 9
    TrTimesTable[ list( i j ) ] = i*j
  ) ; for
) ; for
```

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
```

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

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 ))
```

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.

Module Summary

In this module, we covered

- Implementing records in SKILL with
  - Disembodied property lists
  - SKILL structures
- SKILL Arrays
- Association Lists
- Association Tables

Database Queries

Module 5

July 27, 2005

Module 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 *viewType* such as maskLayout, schematic, or symbolic.
Pin	A physical implementation of a terminal.
Terminal	A logical connection point of a component.

Database Objects

You can use 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.

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.

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

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

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

The *geGetWindowCellView* Function

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

The *geGetEditCellView* Function

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

The ~> Operator

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

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

Summary of ~> Syntax

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

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

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

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

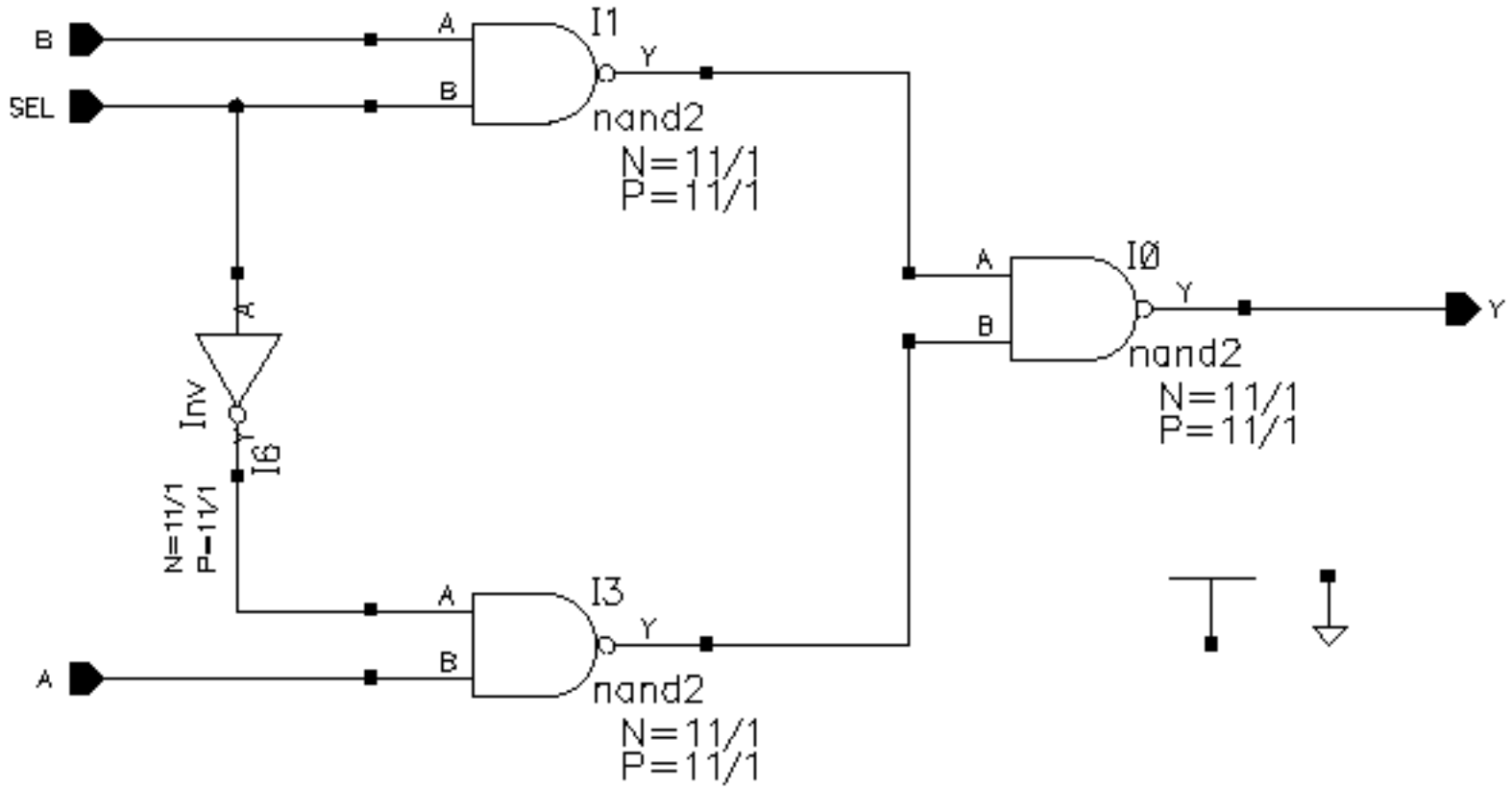
Querying Designs with the ~> Operator

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

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

The *cellView* Object Type

Example: *master mux2 schematic*

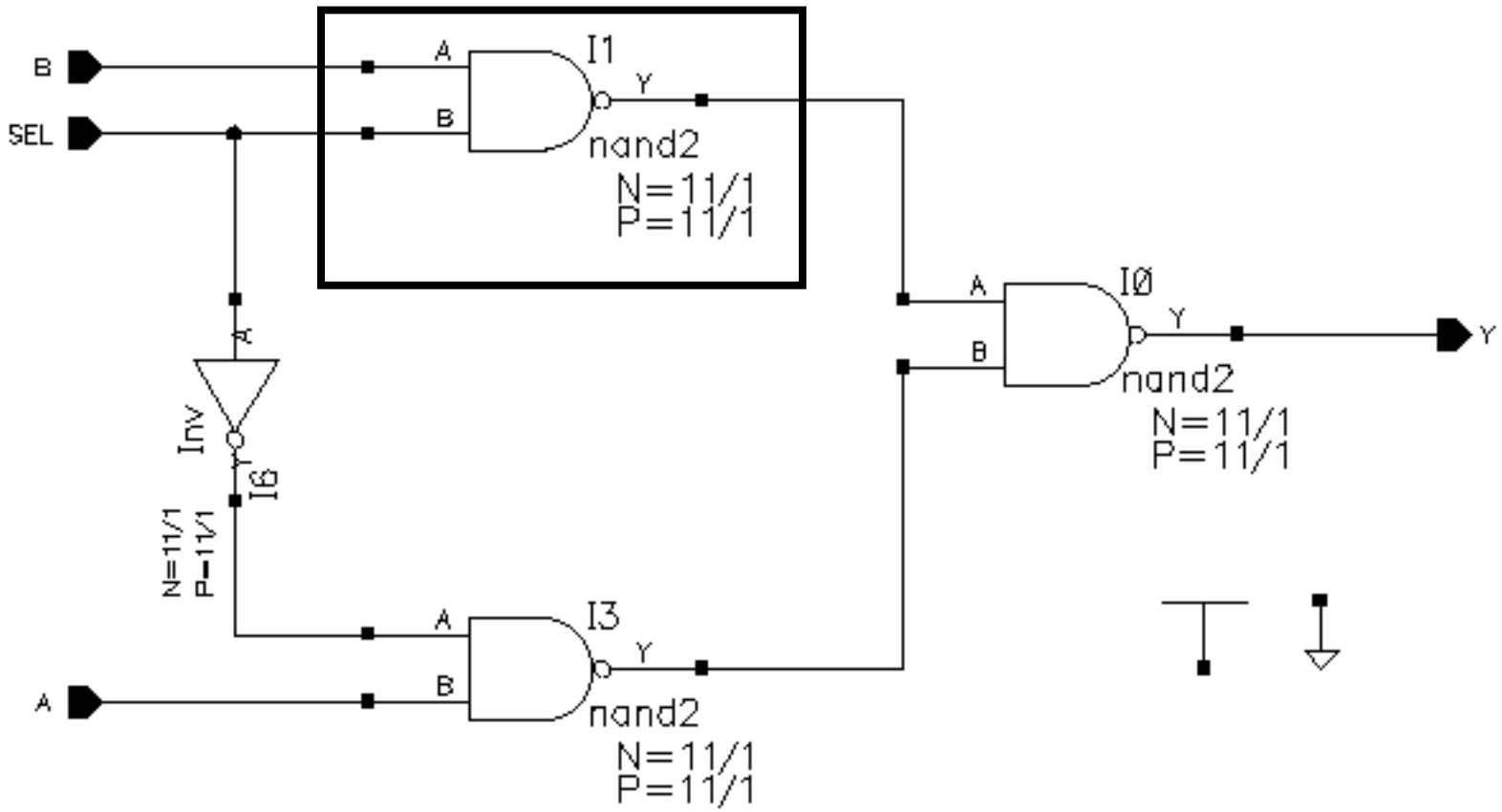


The *cellView* object type includes the following attributes among others:

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

Instances

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



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

The *inst* Object Type

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

- The list of the instances in the design

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

- The list of instance names

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

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

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

- The list of master cell names

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

The *inst* Object Type

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

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

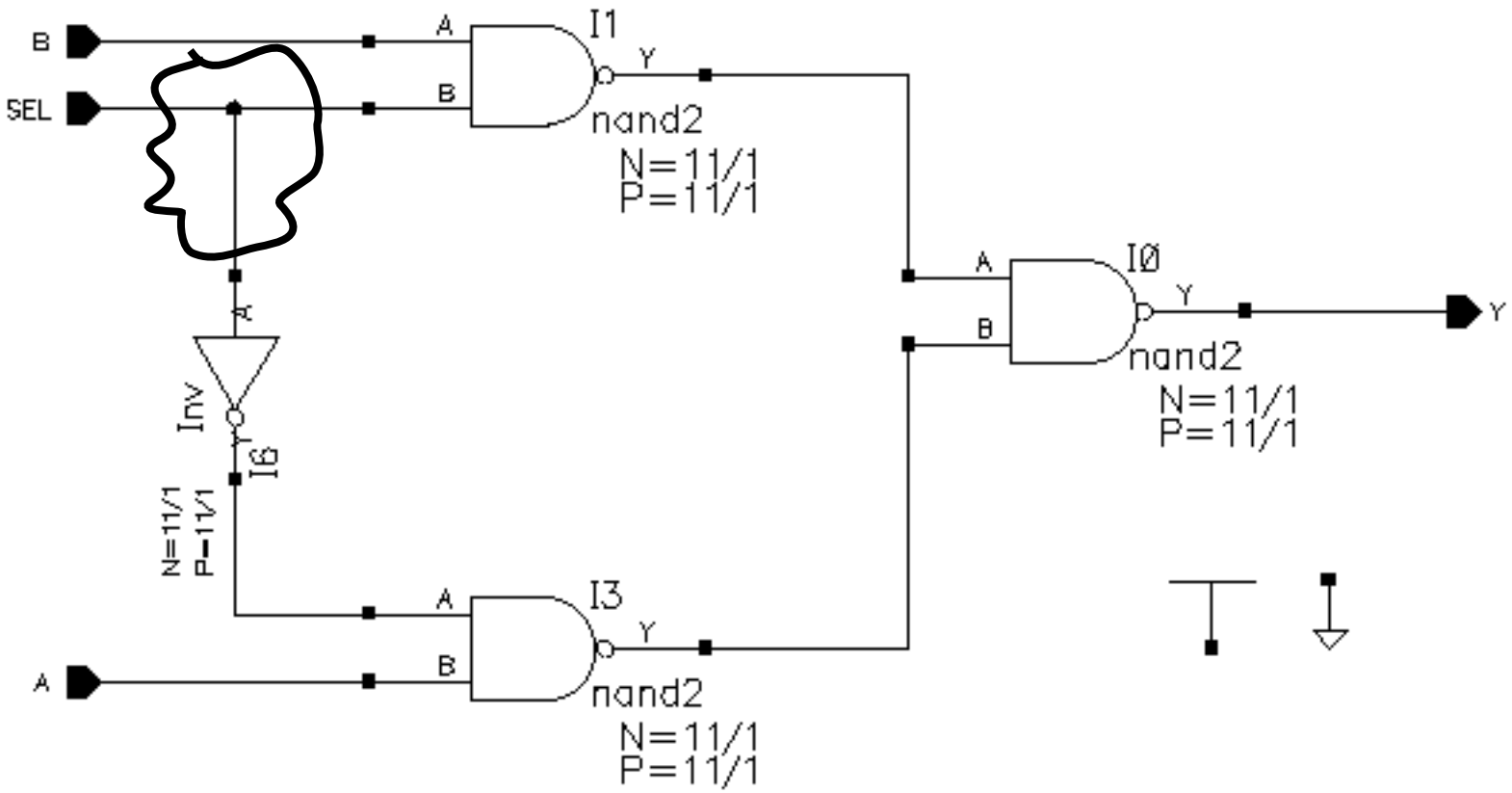
The *dbFindAnyInstByName* Function

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

See the Cadence online documentation to read about this function.

Nets

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



The *net* Object Type

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

- The nets in the design

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

- The number of nets

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

- The names of the nets

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

The *net* Object Type

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

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

The *dbFindNetByName* Function

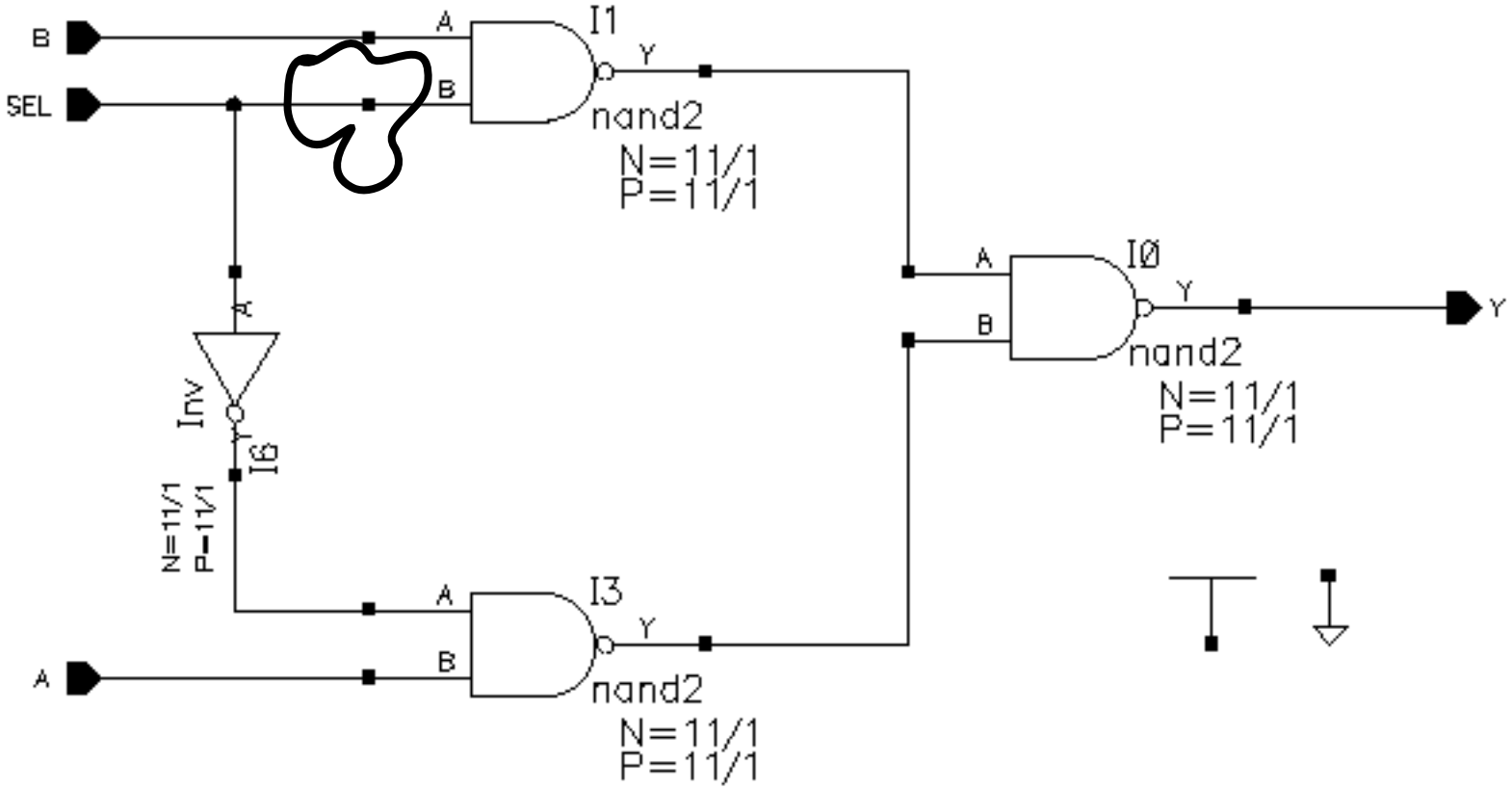
See the Cadence online documentation to read about this function.

Instance Terminals

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

- Each instance contains zero or more instance terminals.
- Each net connects zero or more instance terminals.

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



The *instTerm* Object Type

You can retrieve the following data:

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

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

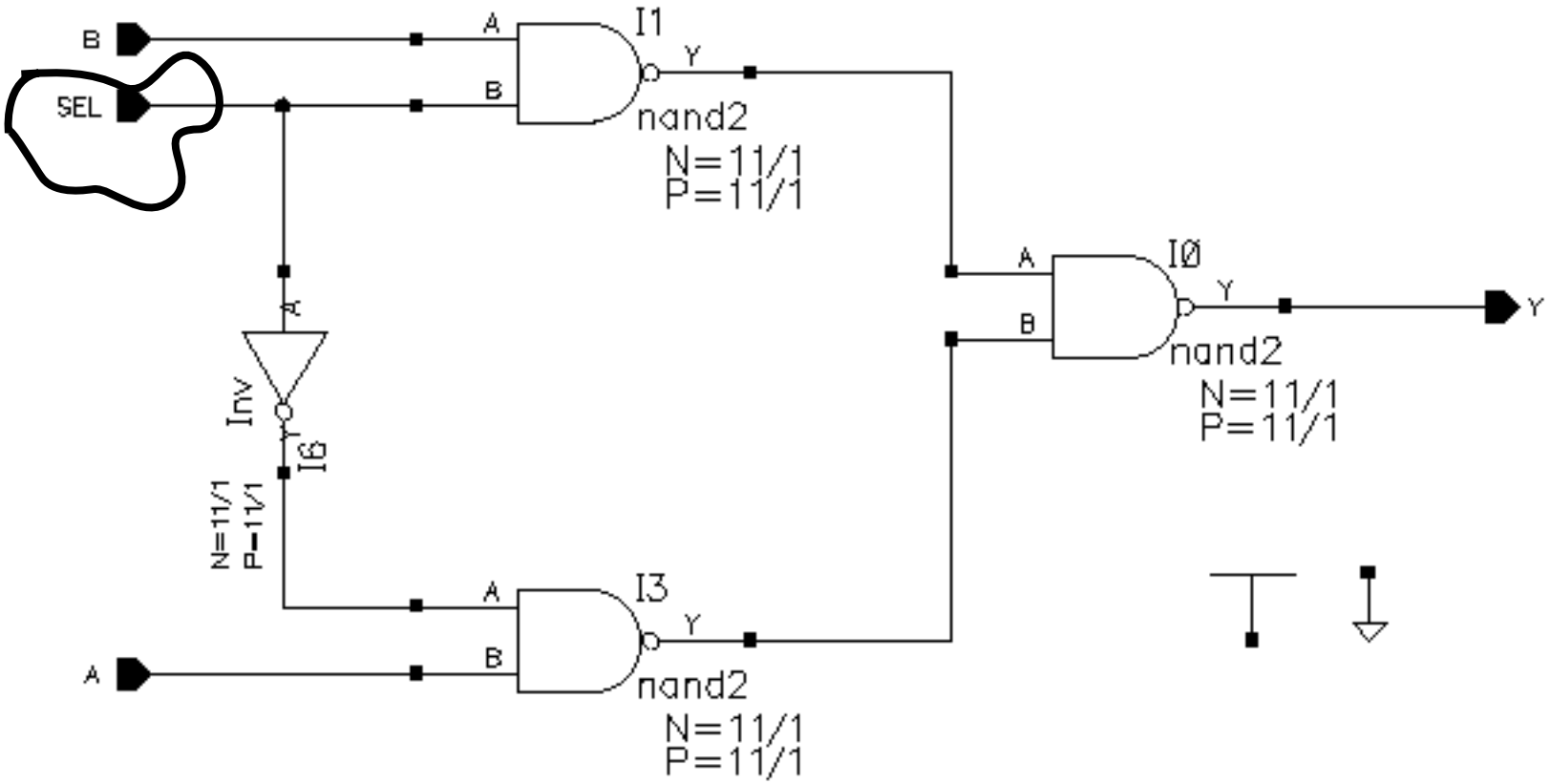
- The name of the net that attaches to the *B instTerm*

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

Terminals

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

Internal nets do not have terminals.



The *term* Object Type

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

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

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

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

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

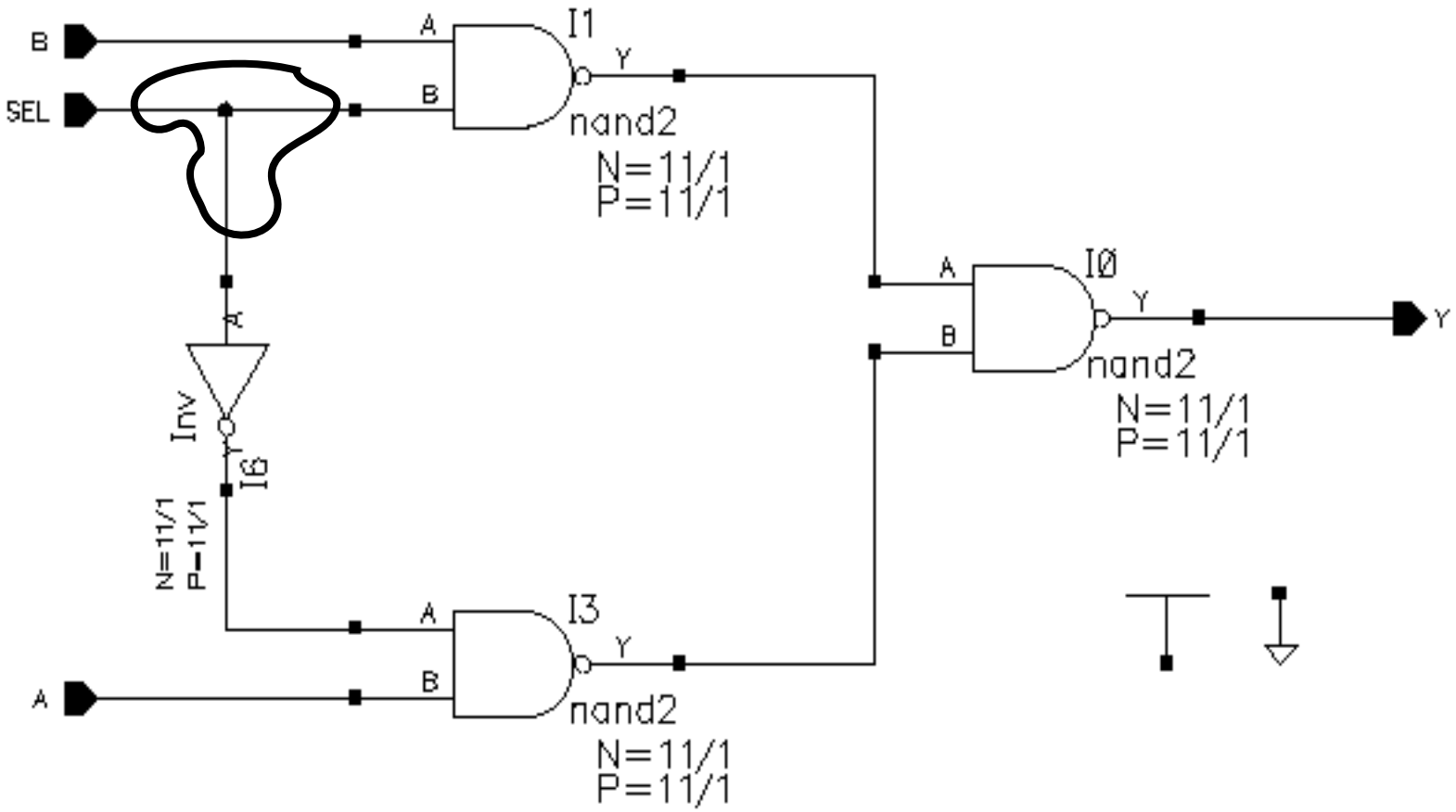
```
mux2~>nets~>name =>
   ("B" "net7" "net6" "Y" "A"
    "vdd!" "gnd!" "SEL" "net4" )
```

The *dbFindTermByName* Function

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

See the Cadence online documentation to read about this function.

Figures and Shapes



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

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

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

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

The *geGetSelSet* Function

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

The *geSelectFig* Function

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

Shape Attributes

All shapes have several common attributes.

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

Lab Overview

Lab 5-1 Querying Design Databases

In this lab, you open these designs:

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

You enter SKILL expressions to answer the following questions:

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

Module Summary

In this module, we covered

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

Flow of Control

Module 9

July 27, 2005

Module Objectives

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

Terms and Definitions

Iteration	To repeatedly execute a collection of SKILL expressions.

Relational Operators

Use the following operators to compare data values.

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

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

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

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

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.

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"
```

Branching

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

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
```

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)
```

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
```

The *when* and *unless* Functions

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

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

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

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

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

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

The *case* Function

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

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

```
case( shapeType
   ( "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

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                  expN1expN2 ....
```

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-*nil*	while

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

The *for* Function

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

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

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

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

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

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 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.

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 *prog* and *return* Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Lab Overview

Lab 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.

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

Windows

Module 4

July 27, 2005

Module 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.

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" )
```

The *geOpen* Function

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

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

Prompt Line

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

```
hiRemovePromptLine()
```

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

Using Bindkeys

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

There are several different uses for bindkeys:

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

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

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

This example illustrates the different uses of bindkeys.

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

   The Zoom In command starts.

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

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

   The Pan command starts.

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

   The Pan command finishes.

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

   The Zoom In command finishes.

Defining Bindkeys

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

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

The *hiGetAppType* Function

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

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

The *hiSetBindKey* Function

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

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

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

Describing Events

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

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

Examples

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

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

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

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?

The *hiGetBindKey* Function

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

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

The *hiSetBindKeys* Function

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

Standard Bindkey Definitions

The following files contain the standard bindkey definitions:

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

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

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

The *alias* Function

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

```
alias( bk hiSetBindKey )
```

Opening a Text Window

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

Example

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

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

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

Example

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

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

The *view* Function

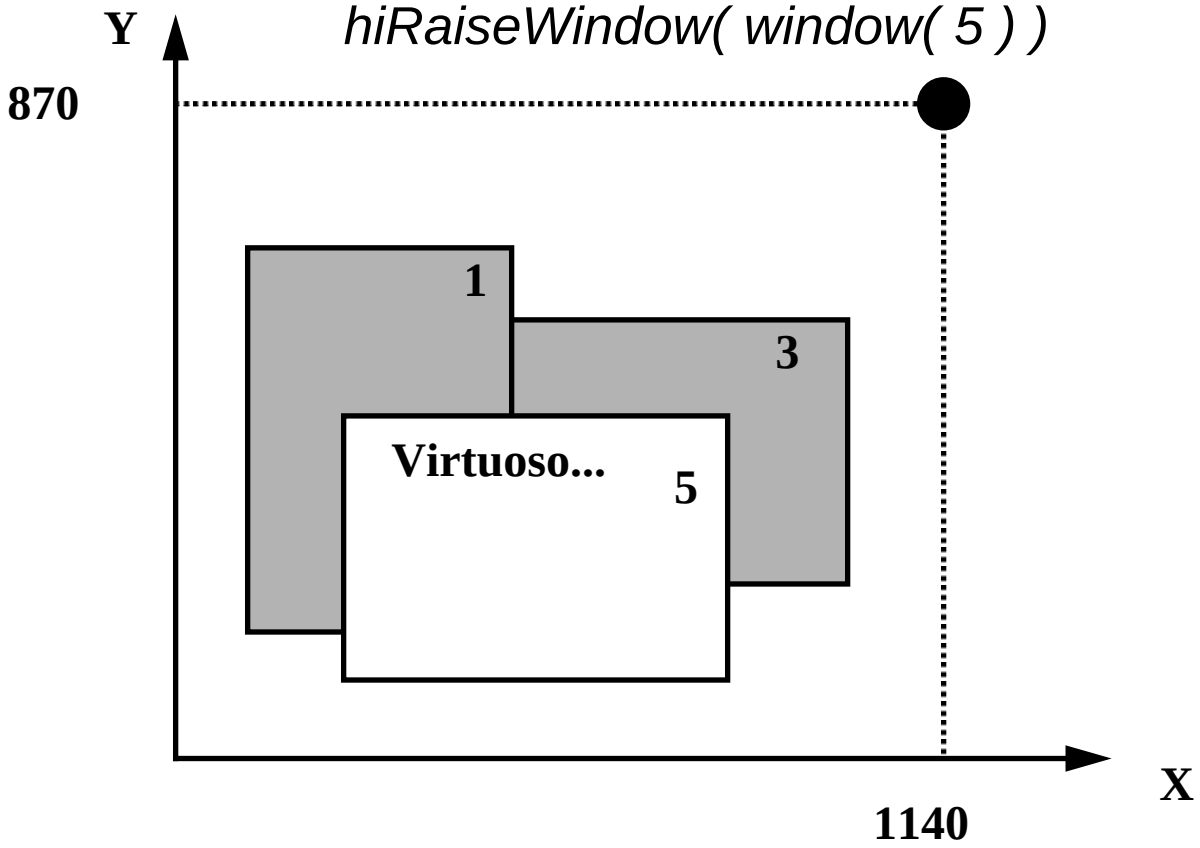
The *view* function takes several optional arguments.

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

Manipulating Windows

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

hiRaiseWindow(window(5))



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.

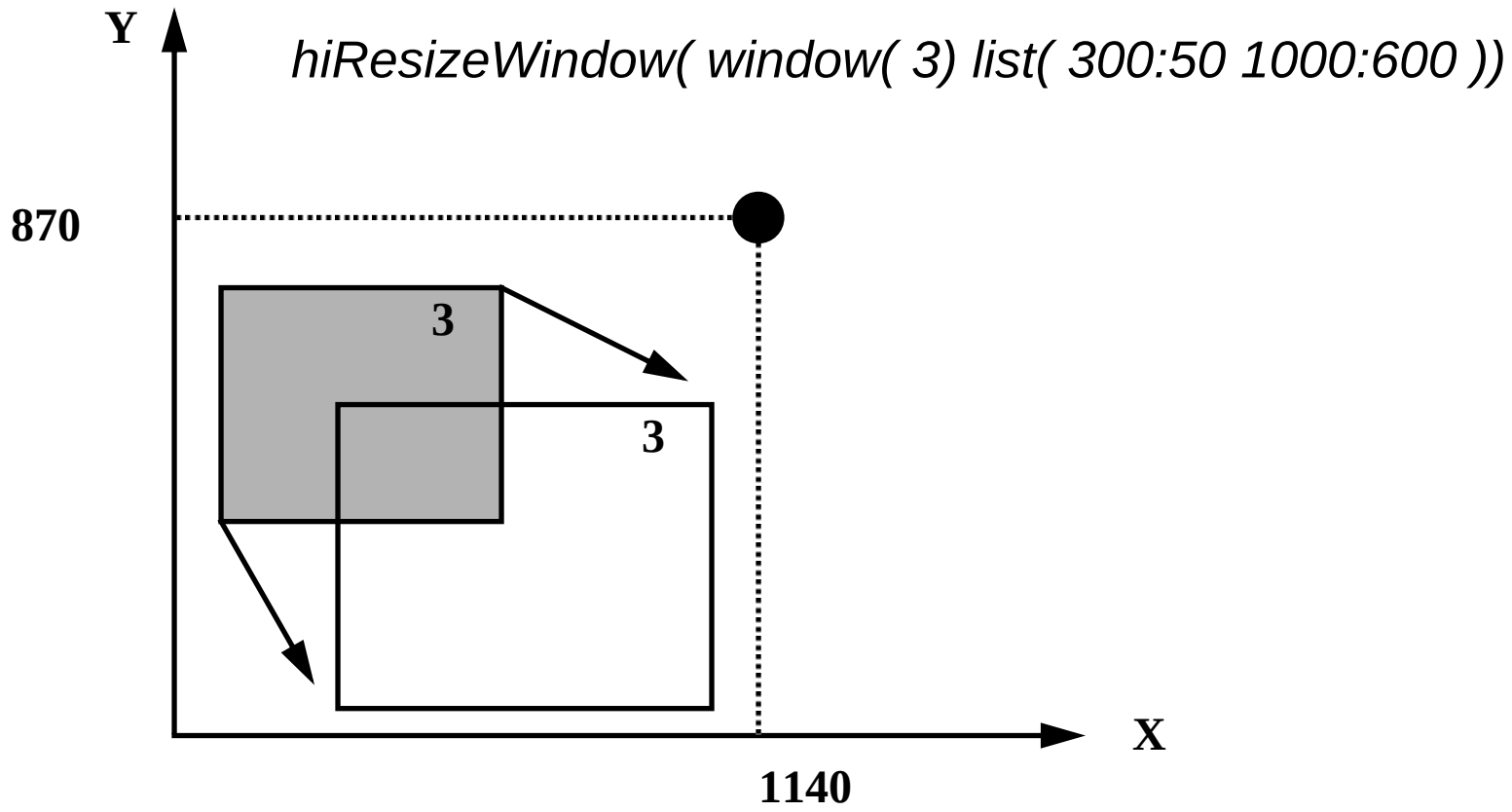
```
hiRaiseWindow( window(5 )) => t
```

- 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.



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.

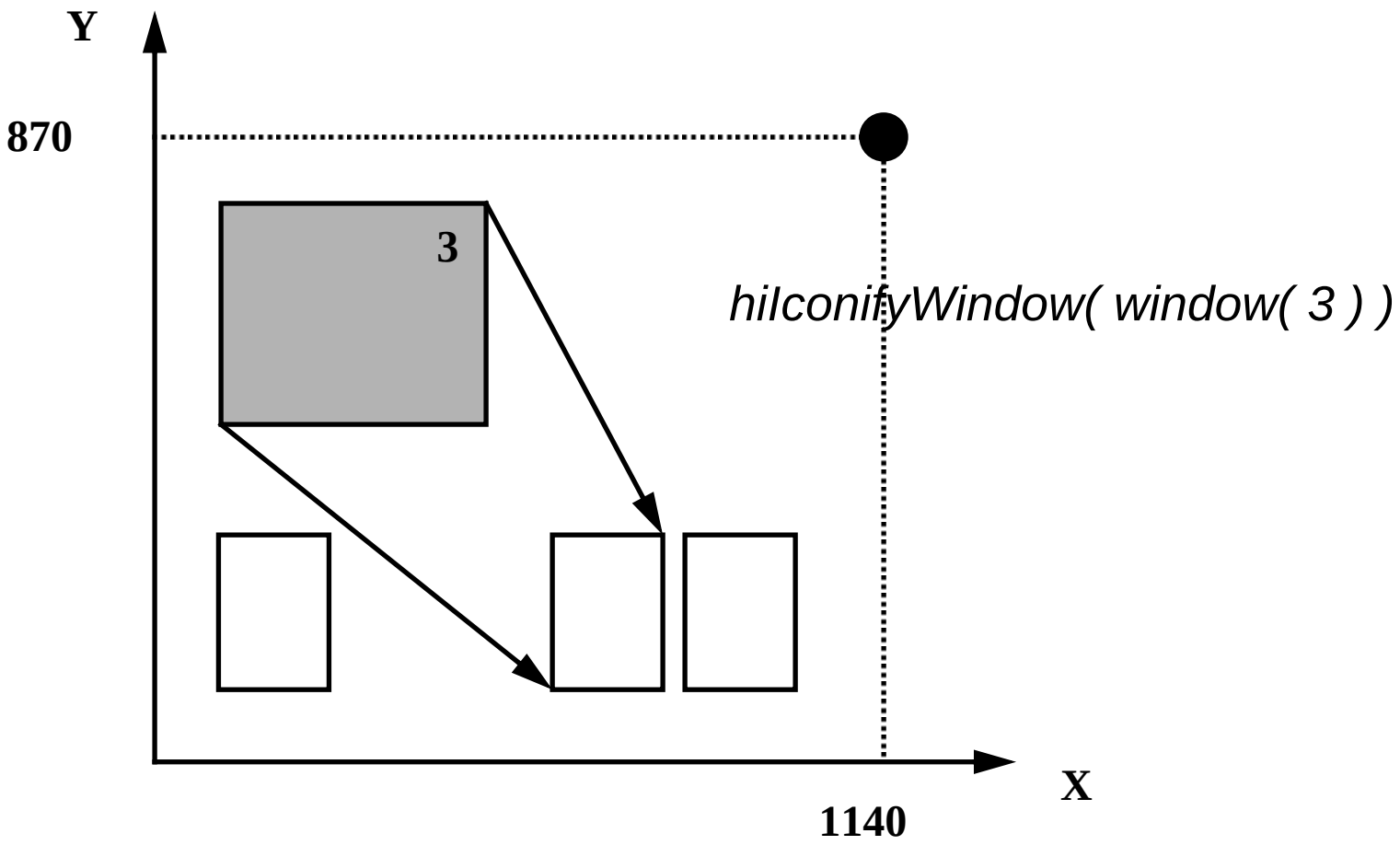
```
hiGetAbsWindowScreenBBox( window(1) t ) =>
  ((200 300) (650 700))
```

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



The *hiIconifyWindow* Function

The *hiIconifyWindow* function iconifies an open window.

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

The *hiGetWindowIconifyState* Function

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

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

The *hiDeiconifyWindow* Function

The *hiDeinconifyWindow* function opens an iconified window.

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

Lab Overview

Lab 4-1 Opening Windows

Lab 4-2 Resizing Windows

Lab 4-3 Storing and Retrieving Bindkeys

Lab 4-4 Defining a Show File Bindkey

Module Summary

In this module, we discussed

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

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

Menus

Module 6

July 27, 2005

Module 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

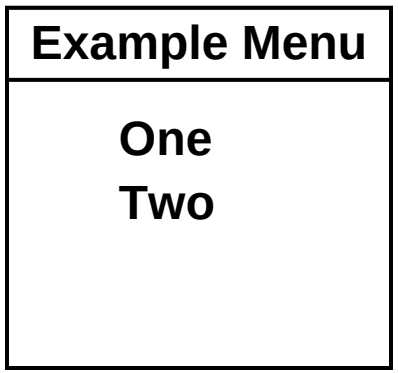
Use the *hiCreateSimpleMenu* function to build a pop-up menu with choices.

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

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

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

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



The *hiCreateSimpleMenu* Function

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

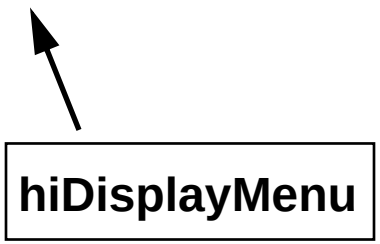
Displaying a Pop-Up Menu

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

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

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

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



The *hiDisplayMenu* Function

Example Argument	Meaning
TrExampleMenu	Use the variable you passed to *hiCreateSimpleMenu*. Do not apply the ’ operator to the *TrExampleMenu* variable because you want to refer to the value of the variable.

Creating a Pull-Down Menu

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

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

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

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

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

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

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

The *hiCreateMenuItem* Function

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

The *hiCreatePulldownMenu* Function

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

Inserting a Pull-Down Menu

Assume that a single window is already open.

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

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

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

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

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

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

The *hiInsertBannerMenu* Function

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

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

Deleting a Pull-Down Menu

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

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

This example removes the leftmost menu in the CIW.

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

Notice that the *hiDeleteBannerMenu* function requires the index of the pull-down menu. You can use the result of *hiGetBannerMenus()* to figure out the index by counting down the list to the position of the menu. The first menu in the list is index 0.

The *hiDeleteBannerMenu* Function

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

Lab Overview

Lab 6-1 Exploring Menus

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

Module Summary

In this module, we covered

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

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

cadence®

File I/O

Module 10

July 27, 2005

Module Objectives

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

Writing Data to a File

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

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

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

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

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

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

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

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
```

Reading Data from a File

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

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

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

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

The *infile* Function

The *gets* Function

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

- Variable that receives the next line.
- Input port.

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

The *when* Function

The first expression within a *when* expression is a condition. 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 format directives commonly found include the ones in this table.

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

The following is an example of output from the *~/CDS.log*.

```
\o "#.cshrc"
\o "for"
\o "Solaris"
\o "#Cadence"
\o "Training"
\o "Database"
\o "setup"
\o "for"
\o "the"
\o "97A"
\o "release"
\o "#"
```

Opening a Text Window

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

```
view( "~/SKILL/.cdsinit" ) => window:5
```

The *view* function is very useful for displaying a report file to the user.

The *view* function resolves a relative pathname in terms of the SKILL path. This is a list of directories you can establish in your *.cdsinit*.

See Module 7, *Customization*, for specifics on the SKILL path.

To select text from your *CDS.log* file, try the following:

```
view(
   "~/CDS.log" ;;; pathname to CDS.log
   nil           ;;; default location
   "Log File"    ;;; window title
   t             ;;; auto update
) => window:6
```

The *view* Function

The *view* function takes several optional arguments.

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

Lab Overview

Lab 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.

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

ANNEXE

Cellview Data Model

Module 13

July 27, 2005

Module 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 *viewType*, such as *maskLayout*, *schematic*, or *symbolic*
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 *geGetSelSet* function
- Using the ~> operator to retrieve attributes and user-defined properties.

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

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.

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.

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.

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 *nil*.
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:

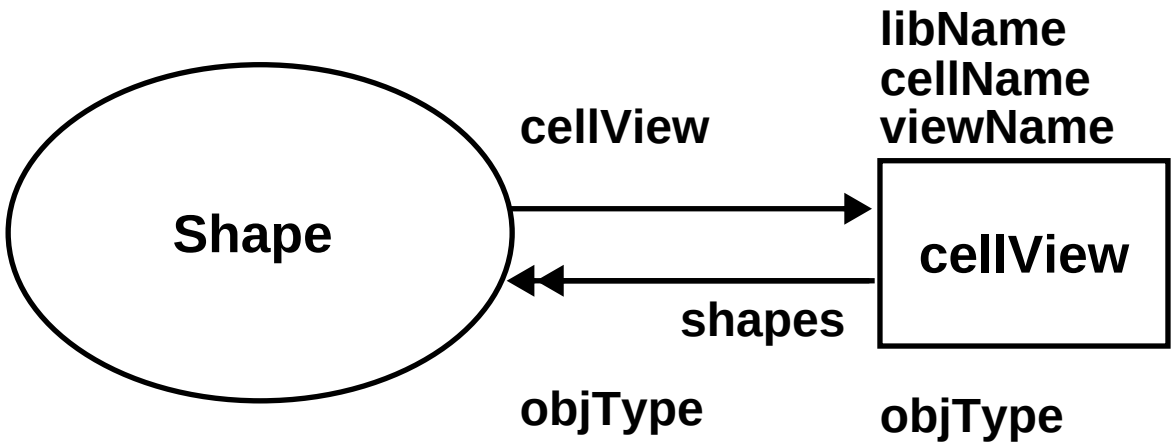
- All object types have an *objType* attribute.
- The *inst* and *mosaicInst* object types both have a *master* attribute.
- All shape object types and the *inst* object type have a *bBox* attribute.
- All shape object types have a *layerName* attribute.

An object *class* is a collection of object types with similar attributes.

Class	**Scope**
Generic Database Object	All database objects
Figure	Any selectable database object
Instance	The *inst* and *mosaicInst* database objects
Shape	All shape database objects

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-to-1(0) relationship.
- A double-headed, thin arrow represents a 1-to-many(0) relationship.

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
Shape	*objType*	
Shape	*cellView*	1 *cellView* database object.
cellView	*objType*	
cellView	*libName*	
cellView	*cellName*	
cellView	*viewName*	
cellView	*shapes*	0 or more *Shape* database objects.

The *~/SKILL/CaseStudy/RoadMap.il* defines a function that builds the complete road map as a layout cellview. See Lab 13-10, *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
```

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
```

Cellviewtype examples

If cell "cellA" with view "layout" exists in library "test," open the cellview for read.

```
cellview = dbOpenCellViewByType("test" "cellA" "layout")
```

Open cell "cellA" with view "layout" in library "test" for "append" mode. Create the cellview if it does not exist.

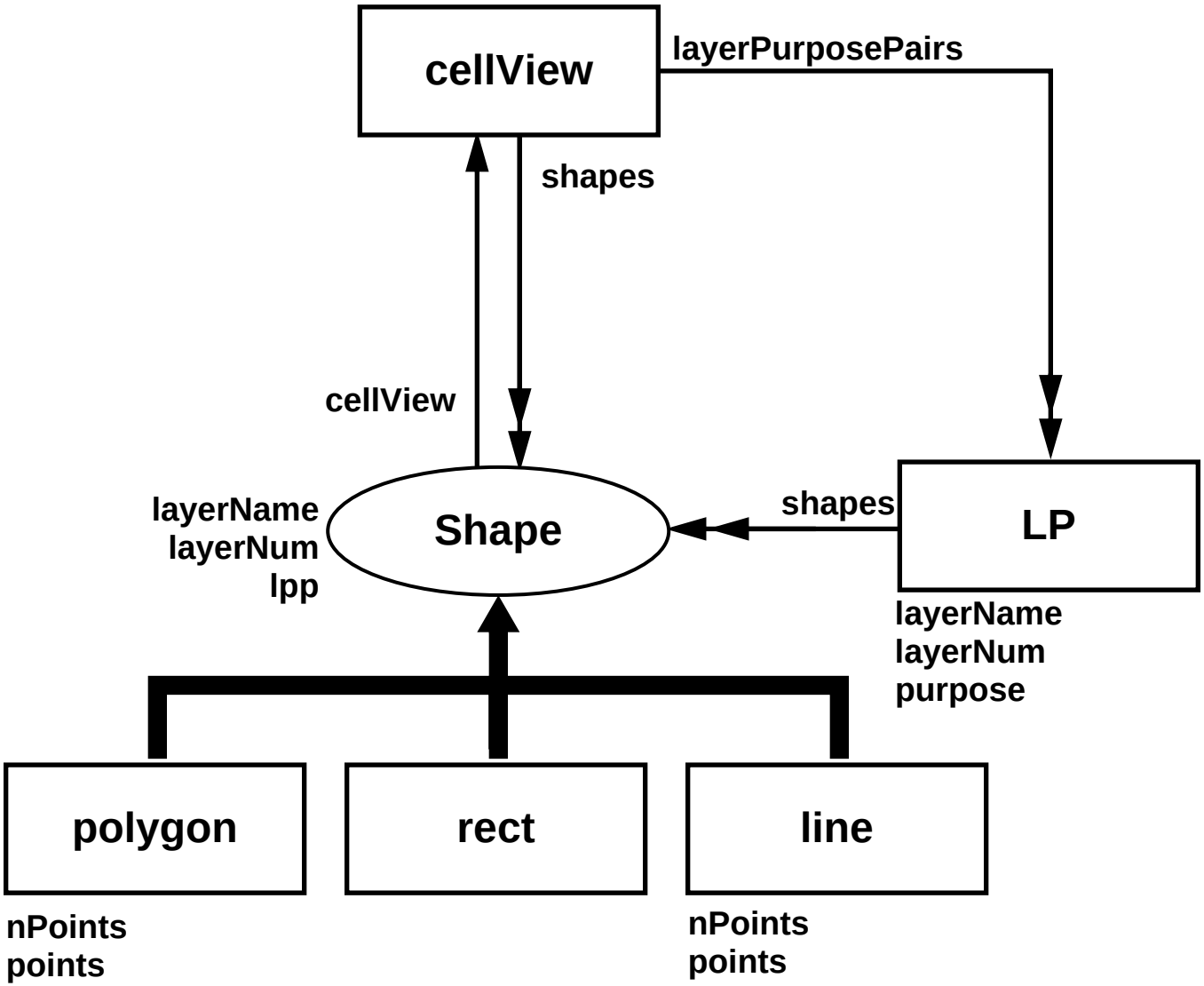
```
cellview = dbOpenCellViewByType("test" "cellA" "layout" "maskLayout" "a")
```

Open cell "cellA" with view "layout" in library "test" for "append" mode only if the cellview already exists.

```
cellview = dbOpenCellViewByType("test" "cellA" "layout" "" "a")
```

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

Shapes



Note: Not all shape object types are shown.

is a kind of

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
```

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 *drawing*.
`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
~>bBox =	Move a rectangle	*aRect~>bBox = newBBox*
~>layerName =	Change a shape's layer	*aRect~>layerName = "thinox"*
~>points =	Change a path's points	*aPath~>points = newPoints*
~>width =	Change a path's width	*aPath~>width = 3.0*

There are attributes you are not allowed to set.

```
aPath~>nPoints = 6
Message: *Error* setSGq: dbSetq:
   Can not set attribute - nPoints
```

The *geTransformUserBBox* function transforms a bounding box according to a displacement vector, a rotation, and a magnification.

```
newBBox = geTransformUserBBox( aRect~>bBox
   list( 0:deltaY "R0" )
   )
```

The *geTransformUserPoint* Function

The *geTransformUserPoint* function transforms a point 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.

Refer to the reference documentation for further information on these functions.

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.

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*.

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 CBDA memory pool to UNIX when you exit the Design Framework II environment.

Lab Overview

Lab 13-1 Enhancing the Layer Shape Report SKILL Program

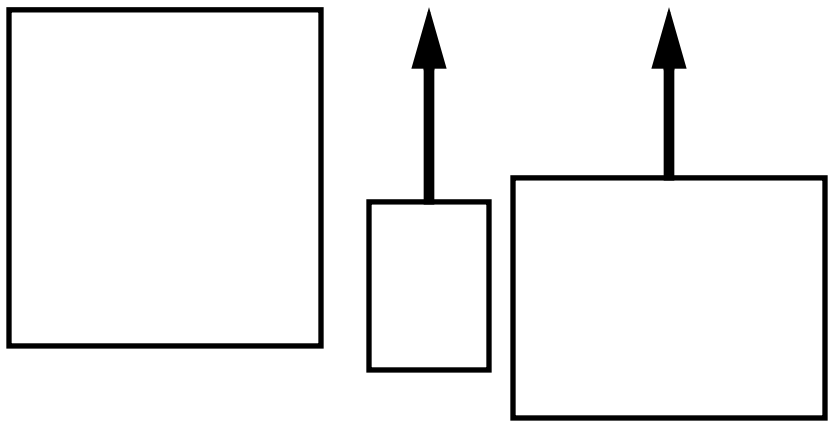
You develop a SKILL function that produces a report to count the number of each kind of shape on each layer in a design.

Lab 13-2 Creating a Cellview

You create a cellview containing various shapes.

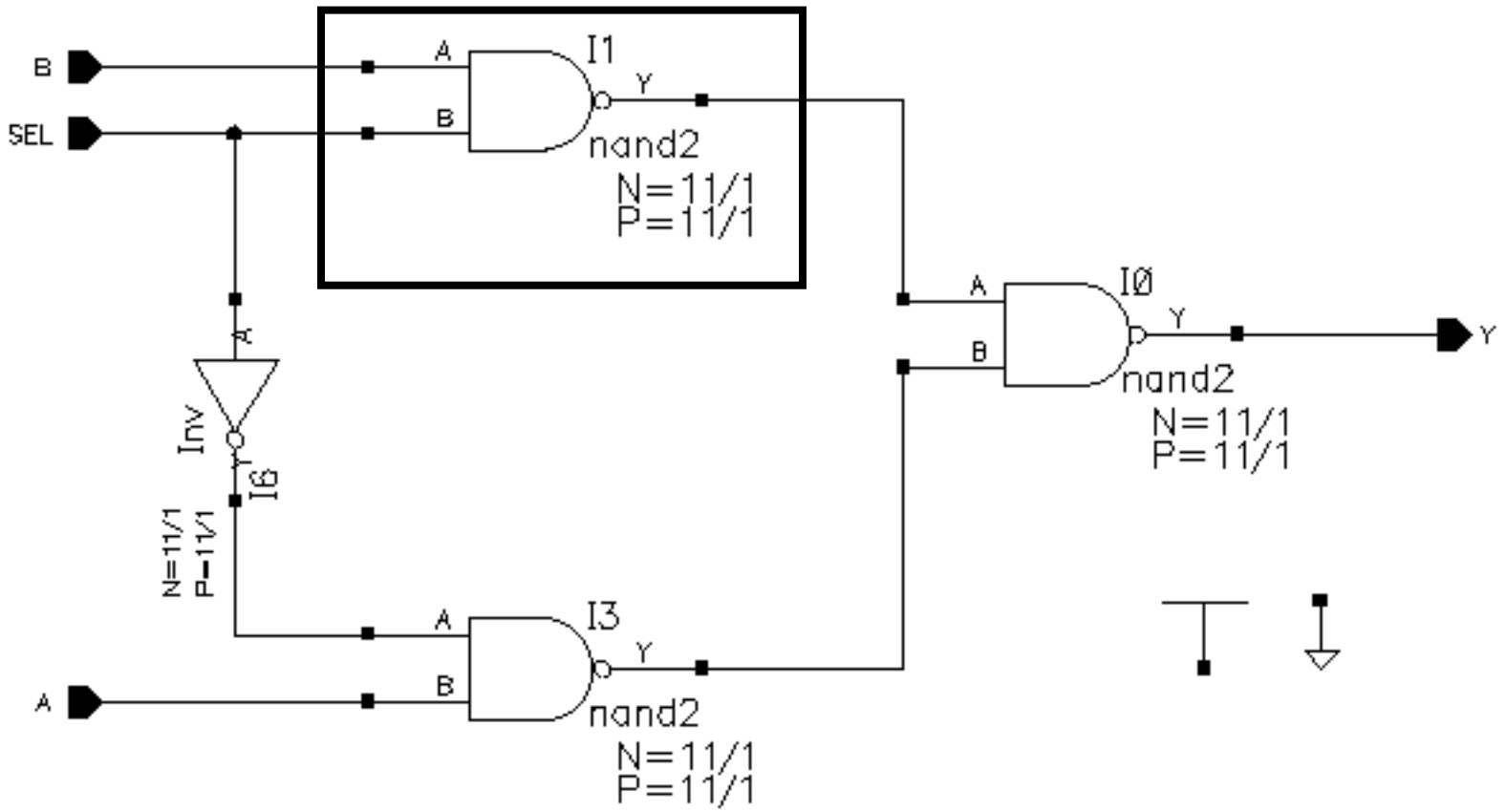
Lab 13-3 Aligning Rectangles

You develop a SKILL function to align the top sides of a collection of rectangles.



Hierarchy

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



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
```

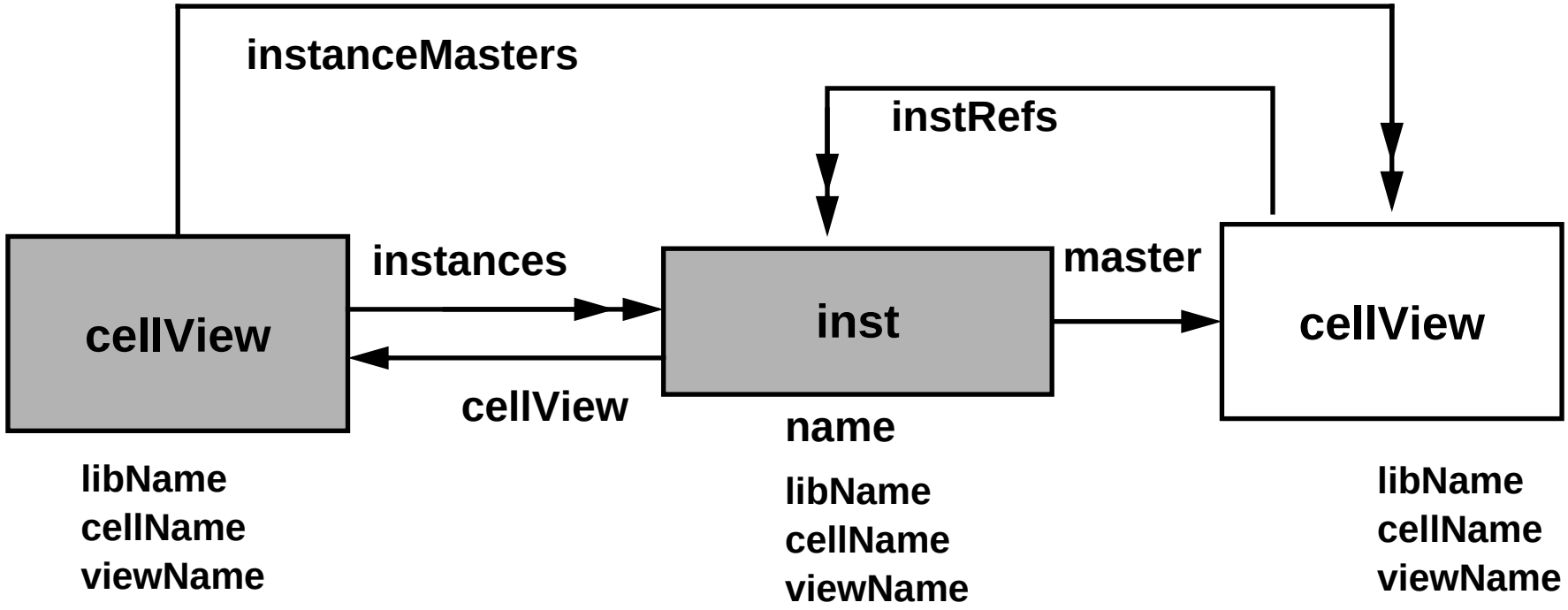
- The list of master cell names

```
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* 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"*.

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
```

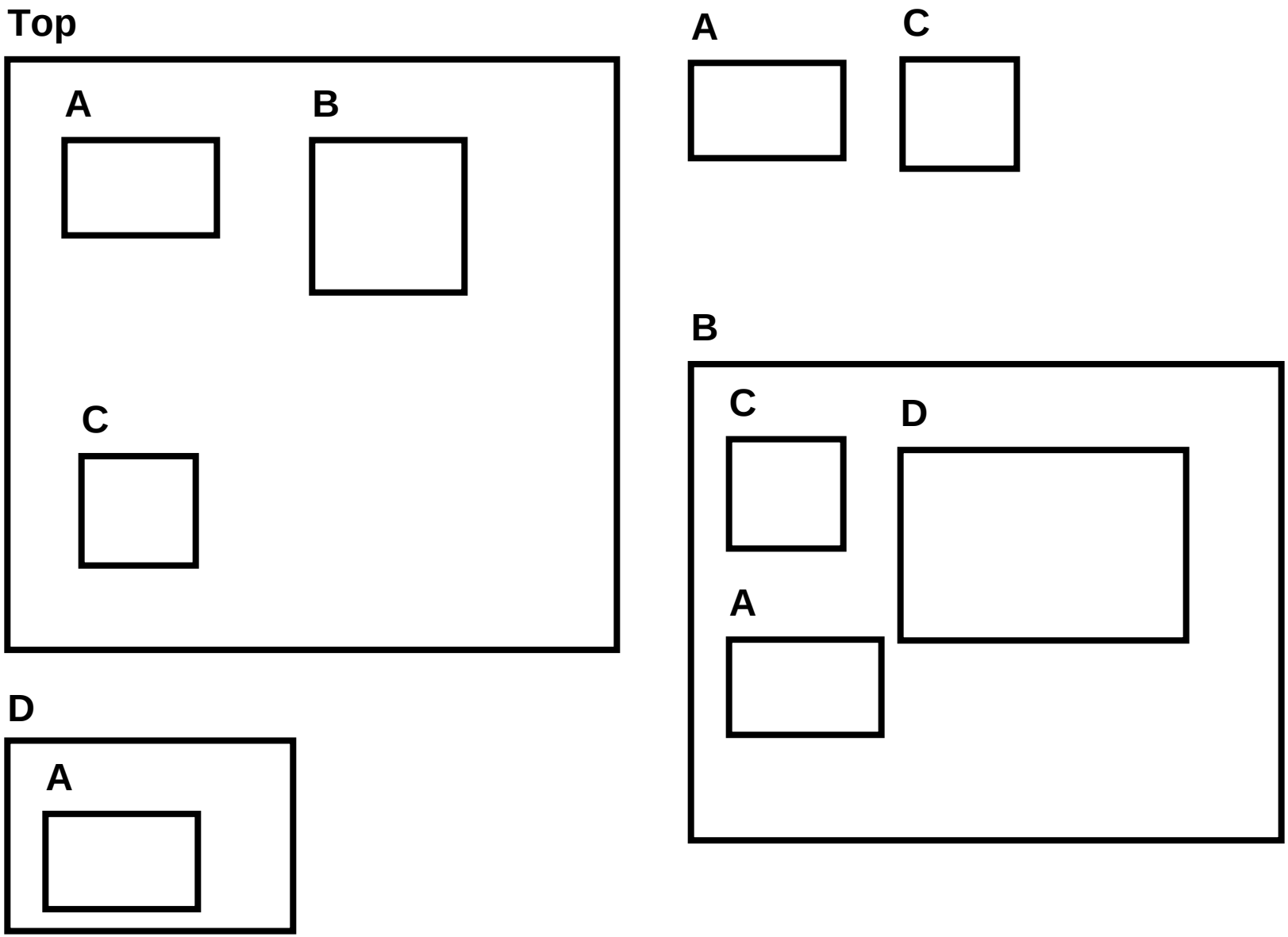
*Can you account for the return value **t** by studying the source code?*

This more realistic format string prints the information in columns.

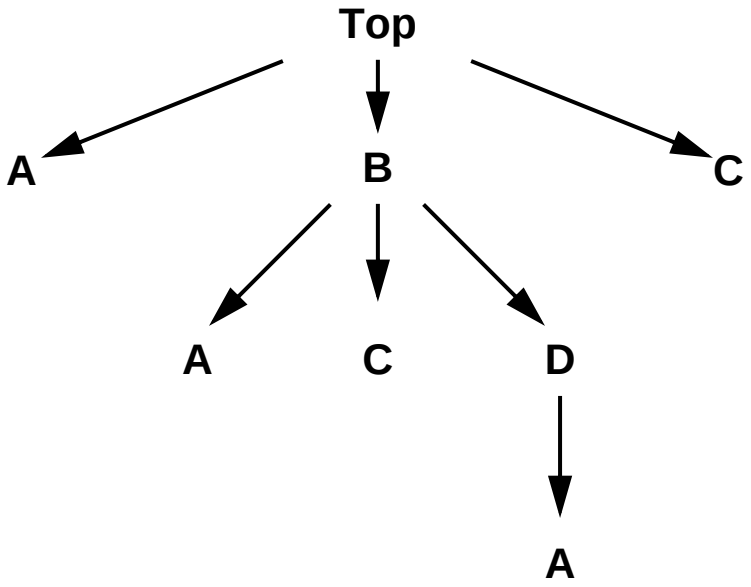
```
format =
"Instance: %-10s Master: %-10s within CellView: %-10s %-10s %-10s \n"
```

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)*.



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.

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*.

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
```

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.

Lab Overview

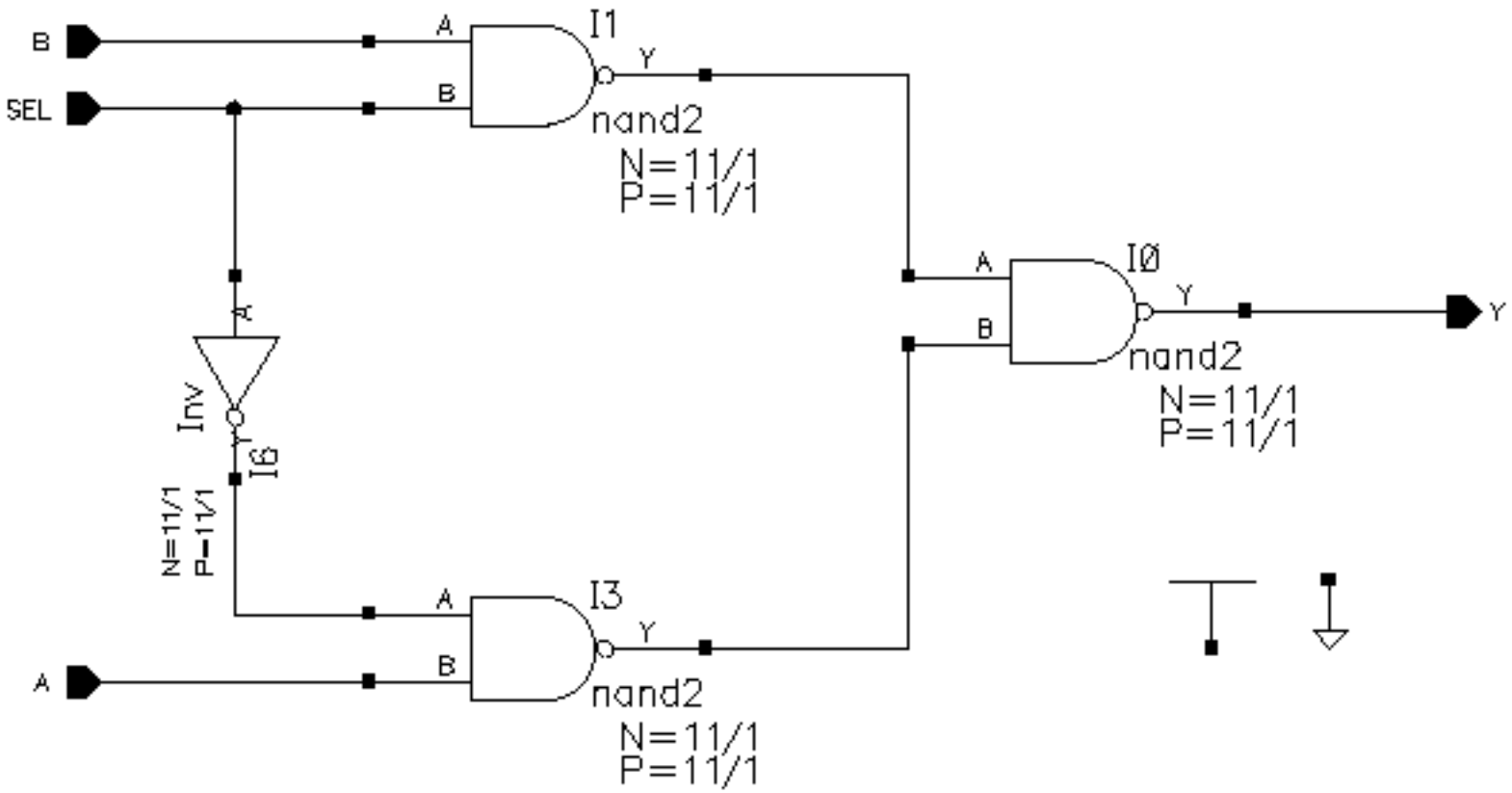
Lab 13-4 Exploring Hierarchy

Lab 13-5 Traversing a Hierarchical Design

You run the *TrHierarchyTraversal* function on *master mux2 layout* and turn on tracing to observe the recursion.

Connectivity

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

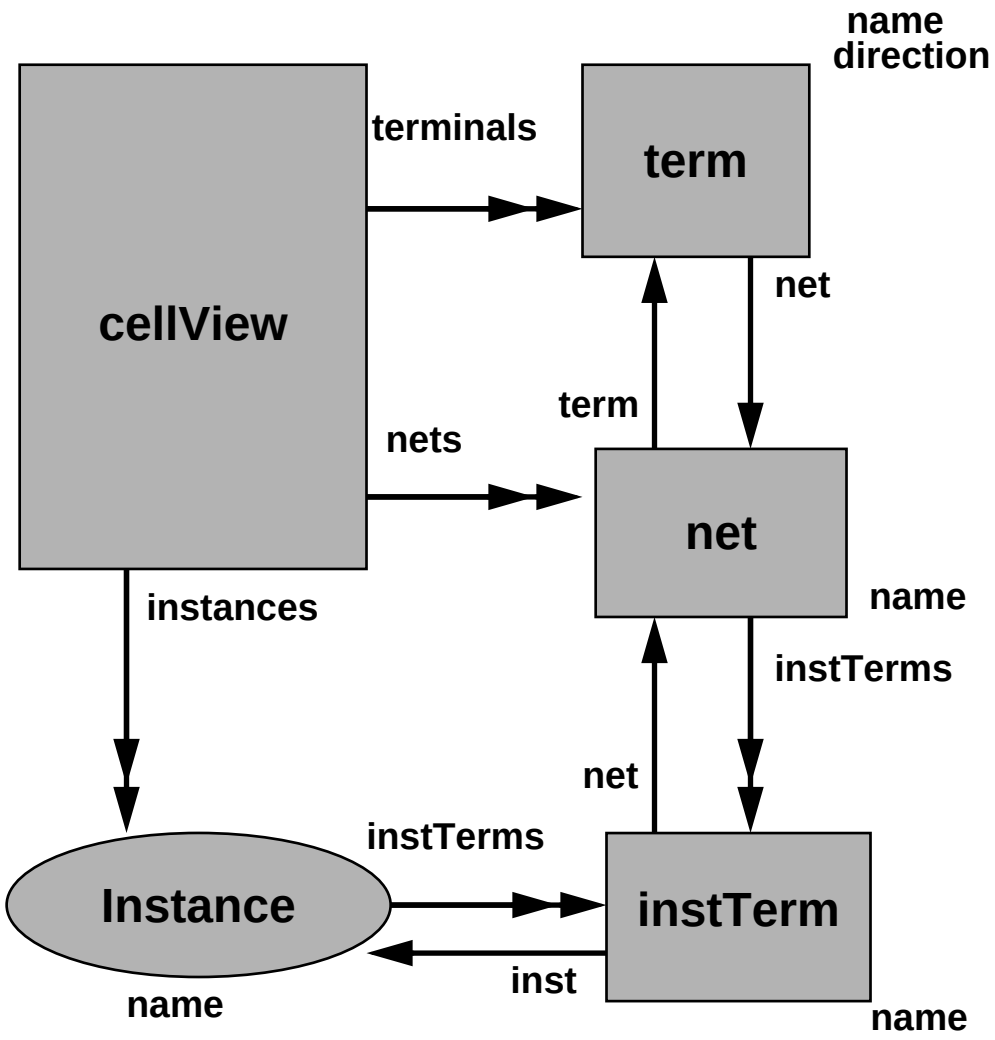


External nets connect a terminal with zero or more instance terminals.

Internal nets connect zero or more instance terminals.

Each instance terminal is on one instance.

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.

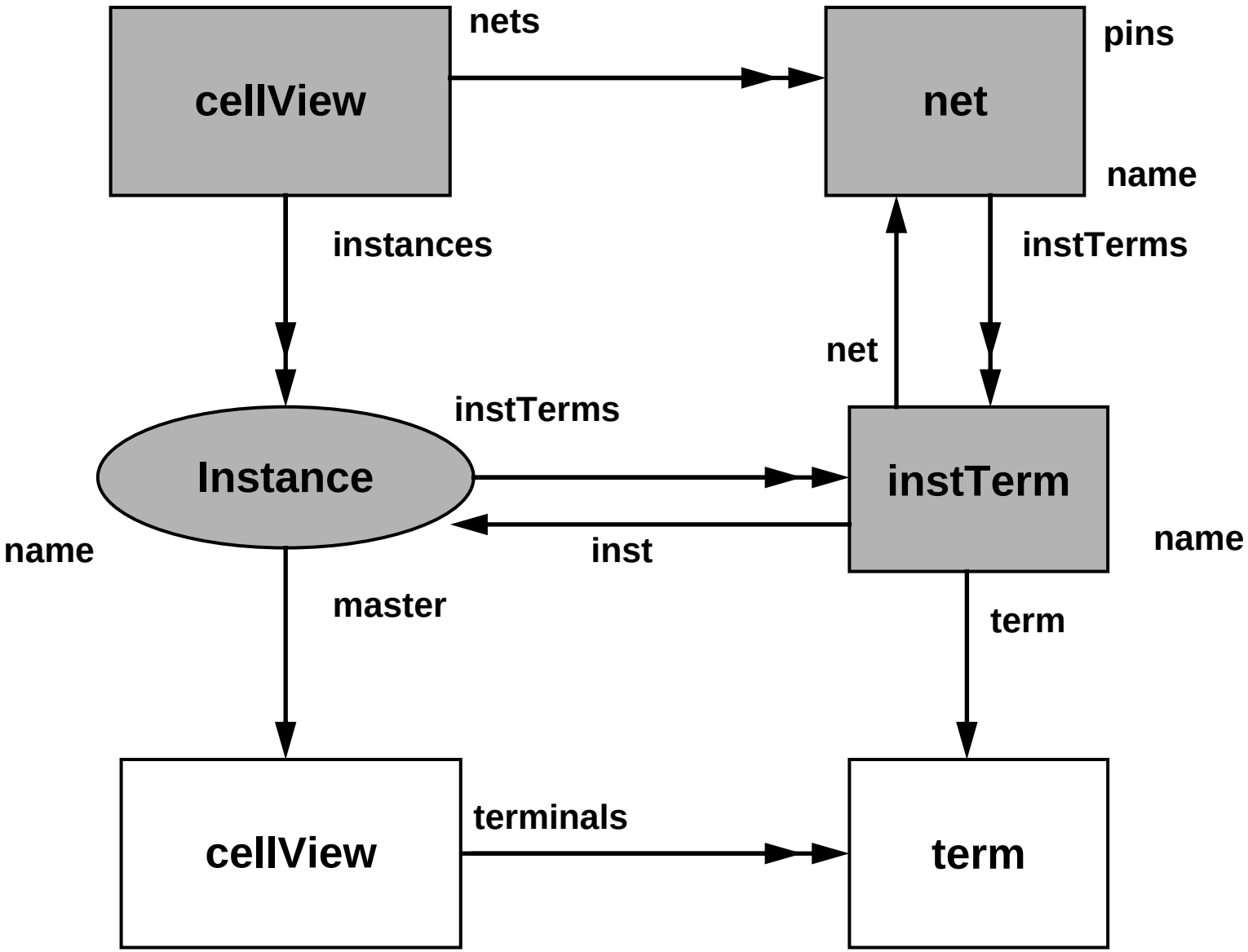
A net runs between one or more internal instance terminals. Each instance terminal is on a unique instance. In addition, a net can connect to the outside world by means of a terminal.

Each *net* database object has

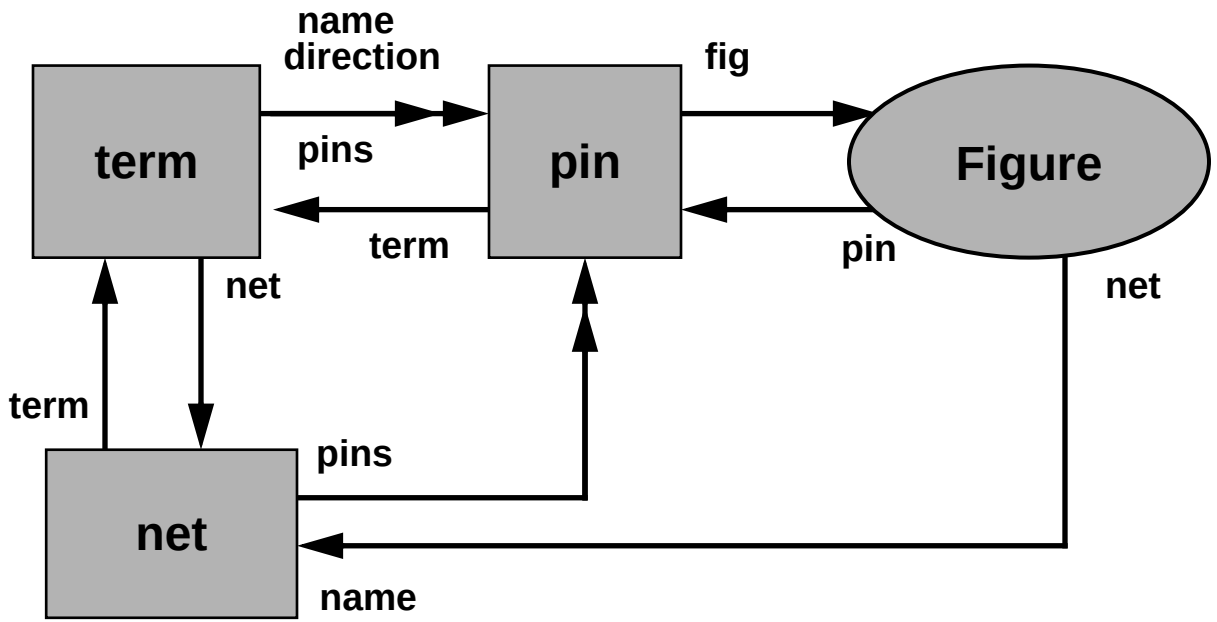
- A *term* attribute, whose value is the unique *term* database object for this net.
- An *instTerms* attribute, whose value is a list of *instTerm* database objects.

Each *instTerm* object has an *inst* attribute, whose value indicates the instance.

Connectivity Between Instances and Masters



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* or *Instances*.

This example determines what type of figure is associated with the terminal pin(s).

```
cv~>terminals~>pins~>fig~>objType
```

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

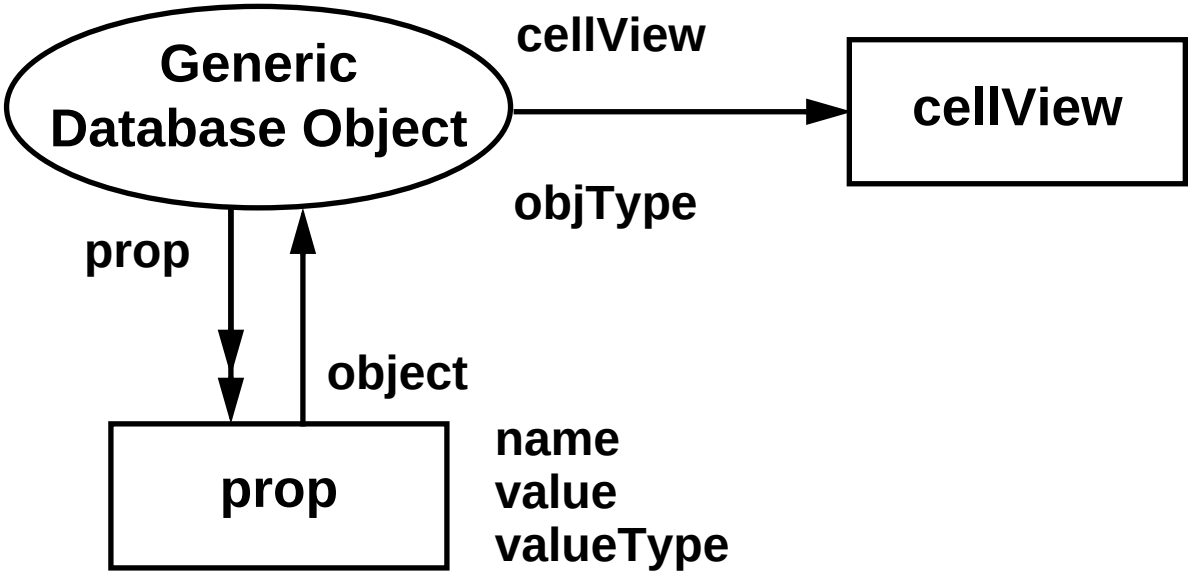
You develop a simple net-based netlister.

Lab 13-7 Developing an Instance-Based Netlister

You develop a simple instance-based netlister.

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: *"net"*, *"line"*, etc.
cellView	The cellView database object containing the database object.
prop	0, 1 or more prop database objects.

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 zero 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" ... )
```

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"
```

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.

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
```

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

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))
    ... )
```

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.

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

Module Summary

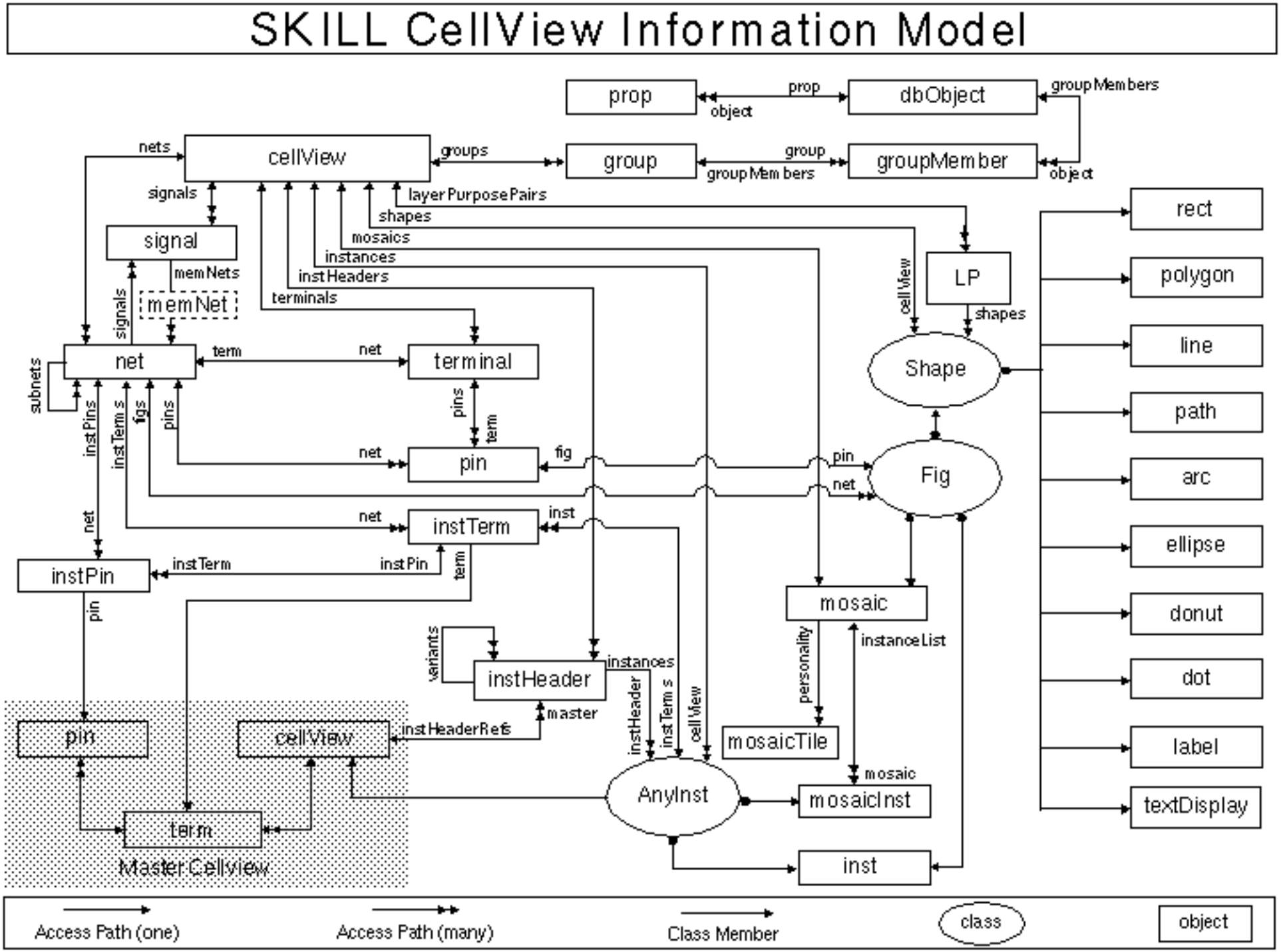
In this module, we

- 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

Cellview Data Model Map

Appendix A

July 27, 2005


SKILL CellView Information Model
prop
dbObject
object
groupMembers
cellView
nets
groups
group
groupMember
signals
layerPurposePairs
shapes
mosaics
instances
instHeaders
terminals
signal
memNets
memNet
LP
cellView
shapes
rect
polygon
line
path
arc
ellipse
donut
dot
label
textDisplay
net
term
terminal
subnets
instPins
instTerms
figs
pins
Shape
Fig
pin
fig
inst
instTerm
instPin
mosaic
personality
instanceList
variants
instHeader
master
instHeaderRefs
mosaicTile
mosaicInst
AnyInst
inst
Master Cellview
Access Path (one)
Access Path (many)
Class Member
class
object

Advanced Customization

Module 15

July 27, 2005

Module Objectives

- Customize the Virtuoso Schematic Editor and the Virtuoso® Layout Editor software.
  - 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

Module Scope

This module covers the following topics:

- Customizing Virtuoso Schematic Editor, 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.

This module focuses on one of several approaches to customizing certain Design Framework II applications, such as the Virtuoso Schematic Editor 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 Virtuoso Schematic Editor and Virtuoso 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.

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 two user trigger functions:

- *user postinstall trigger* function — modifies existing menus or sets variables
- *user menu trigger* function — adds new menus

The phrase *user trigger* function implies the following:

- You define the user trigger function.
  Cadence does not define it.
- The Design Editor invokes your function after it configures a design window.

To define a user trigger function in your *.cdsinit* file, do the following:

1. Declare the user trigger function.
2. Register your trigger function with the Design Editor.

The Design Editor User Triggers

Declare a *user postinstall trigger* function to accept ONE argument as in this example.

```
procedure( TrUserPostInstallTrigger( args )
   ...
) ; procedure
```

Declare a *user menu trigger* function to accept ONE argument as in this example.

```
procedure( TrUserMenuTrigger( 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.

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
   'TrUserMenuTrigger
   'TrUserPostInstallTrigger
)
```

- The first argument is the view type associated with the application.
  Virtuoso Schematic Editor = *"schematic"*
  Virtuoso Layout Editor = *"maskLayout"*
- Pass *nil* for any unused trigger, in this case the second argument.
- The third argument is the function that you want for the user menu trigger. Use the single quote syntax.
- The fourth argument is the function that you want for the user postinstall trigger. Use the single quote syntax.

The *deRegUserTriggers* Function

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

Virtuoso XL viewtypes

schematic: schSynthesisXL

layout: maskLayoutXL

Developing a User Trigger Function

Develop a user 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.

Debugging a User Trigger Function

After you have declared and registered a user trigger function, you can test and debug it.

To test your user 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.

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.

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
)
```

Three User Postinstall Trigger Examples

On the next several slides, there are three examples that implement the following customizations:

Customization	Trigger
Adding a menu item to the Edit menu in the schematic editor window	user postinstall trigger
Adding a pull-down menu to a schematic editor window	user menu trigger
Reordering the Virtuoso Layout Editor pull-down menus	user postinstall trigger

Adding a Menu Item to the Schematic 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 Edit pull-down menu in Virtuoso Schematic Editor.

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 )
```

Locating the Menu Items with the Menu Data Structure

Use the *hiGetMenuItems* function to get the menu items for a specific menu data structure.

This example will list all the menus and their menu items for the current window:

```
foreach( item hiGetBannerMenus( hiGetCurrentWindow()
  printf( "%L : %L\n" item hiGetMenuItems( eval( item ) ) )
)
```

Finding the Edit Menu Data Structure

The *schEditMenu* global variable contains the data structure of the Edit pull-down menu in Virtuoso Schematic Editor.

Follow these steps to verify this fact by inspection.

1. Open a Virtuoso Schematic Editor 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 Edit menu in Virtuoso Schematic Editor.

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
```

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->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 in the *.cdsinit* File

To make the Edit menu in Virtuoso Schematic Editor 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
)
```

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 Schematic Windows

Use the same approach as in the first example.

In this example, make the *TrUserMenuTrigger* function do these tasks:

1. Build your pull-down menu, if it is not already in memory.
2. Return the list of menus to add.

Test your menu function to add the pull-down menu to the window banner:

```
hiInsertBannerMenu( windowID, TrUserMenuTrigger(), 0)
```

Building Your Pull-Down Menu

The Design Editor calls your trigger function each time the user opens a Virtuoso Schematic Editor window. To avoid building your menu repeatedly, the following code checks whether the menu already exists.

```
procedure( TrUserMenuTrigger()
  if( boundp( 'TrSchematicPulldownMenu )
    then list( TrSchematicPulldownMenu )
    else
      list(hiCreatePulldownMenu( 'TrSchematicPulldownMenu ... ))
    ) ; if
  ) ; procedure
```

What You Put into the *.cdsinit* File

To make every Virtuoso Schematic Editor window have your pull-down menu on the far right, include the following code in your *.cdsinit* file:

```
procedure( TrUserMenuTrigger( args )
     ...
     )
) ; procedure

deRegUserTriggers( "schematic"
   nil
   'TrUserMenuTrigger
)
```

See below

Returns the menu list

View type

No user app trigger

User menu trigger

The *TrUserMenuTrigger* Function

```
procedure( TrUserMenuTrigger()
  if( boundp( 'TrSchematicPulldownMenu ) then
    list( TrSchematicPulldownMenu )
  else
    list( 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
    ) ; list
    ) ; 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.

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 )                          Save menus
    hiDeleteBannerMenus( wid )                         Remove all menus
    foreach( menuSymbol theBannerMenuList
      hiInsertBannerMenu( wid menuSymbol 0 )           Reinsert menus
    ) ; foreach
  ) ; let
) ; procedure

procedure( TrUserPostInstallTrigger( args )
  TrReverseBannerMenus( args->window )
)

deRegUserTriggers( "maskLayout"                        View type
  nil                                                  No user app trigger
  nil                                                  No user menu trigger
  'TrUserPostInstallTrigger                            User postinstall trigger
)
```

See the Cadence online documentation to read more about these functions:

- The *hiGetBannerMenus* Function
- The *hiDeleteBannerMenus* 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.

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.

You can also set form defaults from the *.cdsenv* file. When forms appear, the default values are already set. Some buttons are already turned on, some fields already contain text, and some other choices are already set.

Cadence supplies an initial set of defaults in a sample file located at

```
<your_install_dir>/tools/dfII/samples/.cdsenv
```

Your system administrator customizes this file for your site and puts it in the local environment file at

```
<your_install_dir>/tools/dfII/local/.cdsenv
```

You can override these site-specific settings by creating another .cdsenv file in your home or workarea directory. When the Cadence software loads, it reads your .cdsenv file after the site-specific file, so the settings in your file override the settings in files loaded earlier. These overrides apply to your system only.

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"
```

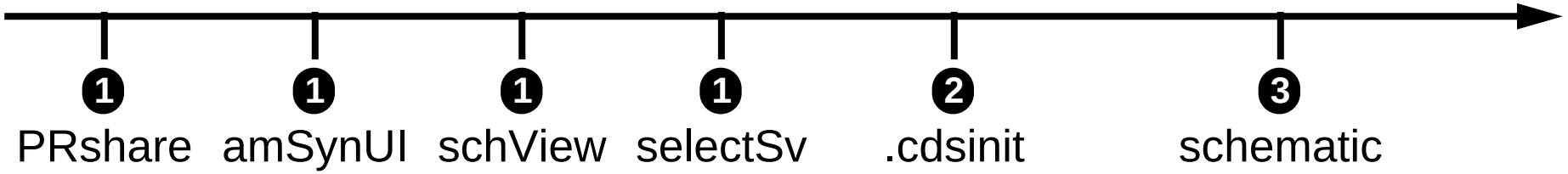
Design Framework II Initialization Sequence

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.

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 )
```

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.

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 Virtuoso Schematic Editor Edit Menu

You study and run the lecture example.

Lab 15-2 Adding a Pull-Down Menu to Virtuoso Schematic Editor 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 the Initial Window Placement

You customize the Layout Editor to initially place all of its design windows in a given location.

Module Summary

In this module, we discussed

- Customizing the Virtuoso Schematic Editor software and the Virtuoso Layout Editor
- Customizing form field default values

User Interface

Module 14

July 27, 2005

Module Objectives

- Context-sensitive pop-up menus
- Fixed menus
- Dialog boxes
- List boxes
- Forms

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()" )
```

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.

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
)
```

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 )
```

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.

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. You can also use the SKILL command.

```
hiGetCIWindow()->showFixedMenuLabels => t
```

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.

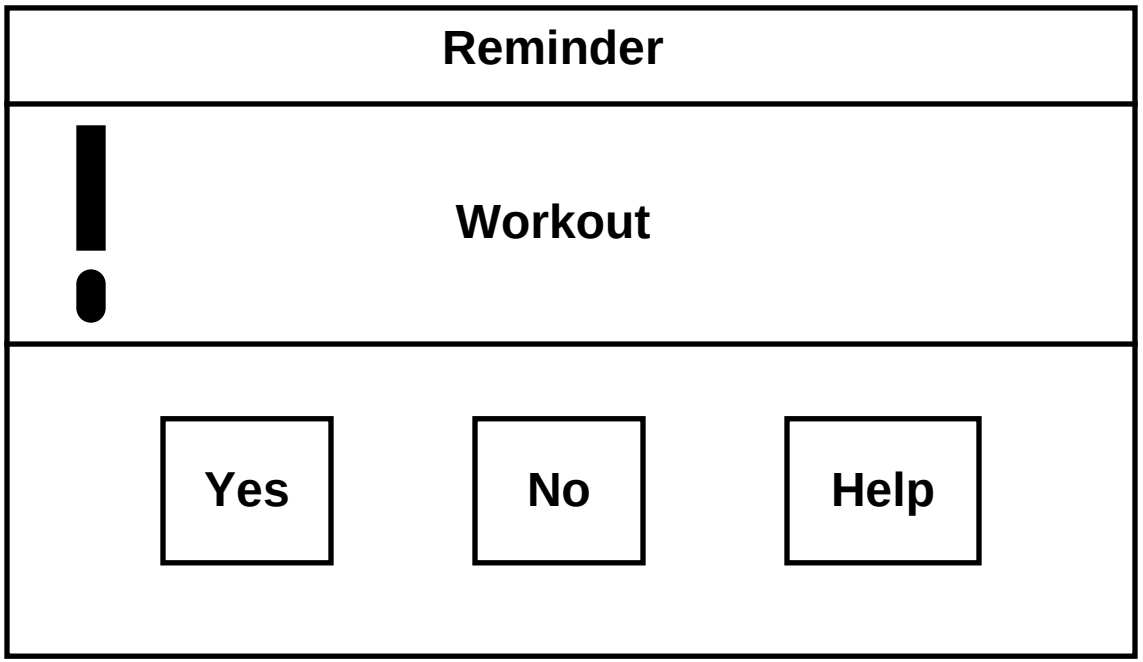
```
hiRemoveFixedMenu( wid ) => t
```

Lab Overview

Lab 14-1 Exploring Fixed Menus

- Create a vertical fixed menu.
- Attach the menu to an application window.

Dialog Boxes



Clicking the **Yes** button

- Displays *Workout completed* in the CIW.
- Removes the dialog box.

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   "Workout"
   ?callback   "TrReminderCB( \"Workout\" )"
   ?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 creates and displays a dialog box.
The dialog box is destroyed when it is removed from the screen.

Keyword Argument	Meaning
?name	The *dboxHandle* symbol is bound to the data structure of the dialog box and then reset to *nil* when box is dismissed.
?dboxBanner	Text that appears within the window manager banner text of the dialog box.
?dboxText	Message that appears in the dialog box.
?callback	Callback that executes whenever the user selects *OK* or *Yes*.
?dialogType	*hicWarningDialog*, *hicErrorDialog* etc. See *User Interface SKILL Functions Reference Manual*.
?dialogStyle	Blocking behavior of this dialog box. Acceptable choices are *'systemModal*, *'modal*, or *'modeless*.
?buttonLayout	*'OKCancel*, *'YesNo*, *'YesNoCancel*, etc. See *User Interface SKILL Functions Reference Manual*.
?location	See *User Interface SKILL Functions Reference Manual*.

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**	***hiDisplayAppDBox* returns**
modeless	Does not block any application.	Immediately.
modal	Blocks Design Framework II applications.	After box is dismissed.
systemModal	Blocks all applications in the system until the user responds to the dialog box.	After box is dismissed.

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
OKCancel	OK
YesNo	Yes
YesNoCancel	Yes/No
CloseHelp	
Close	

The following program displays modeless dialog boxes to verify the circumstances under which the system triggers the callback.

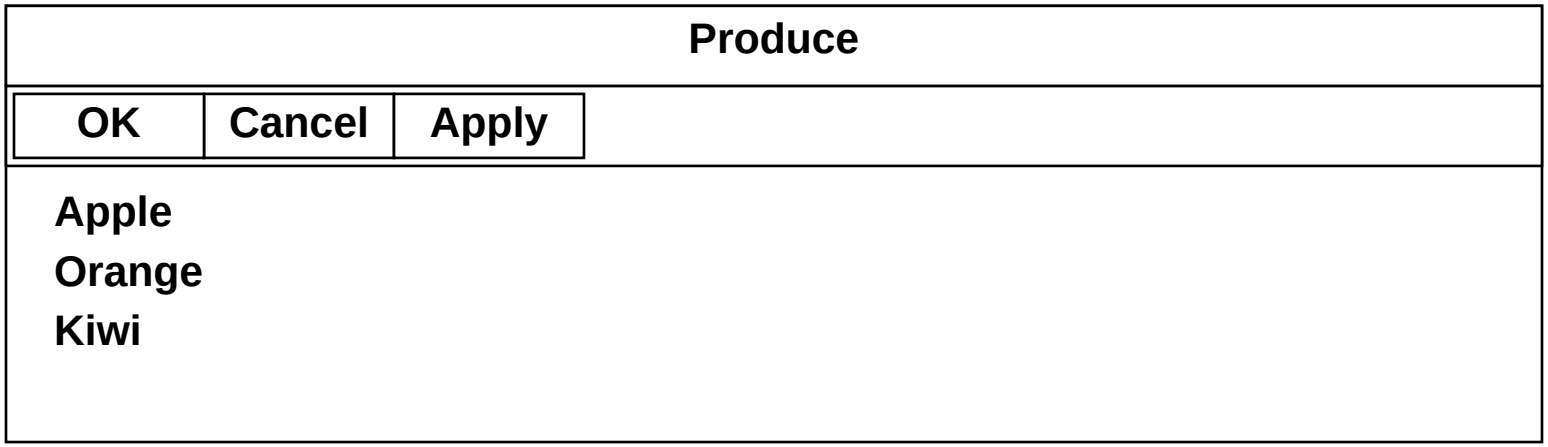
The source code is in the *~/SKILL/DialogBoxes/ButtonLayout.il* file.

```
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

List Boxes



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.

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 creates and displays a list box containing a group of text strings.

Keyword Argument	Meaning
?name	A global variable in which SKILL stores data structure.
?title	The title.
?choices	List of text string choices.
?callback	Function called when the user clicks **OK**, **Cancel**, or **Apply**.
?multipleSelect	Controls single item selection or multiple item selection.
*?applyButto*n	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	*t* the list box data structure	*t*	Disappears.
Apply	*t* the list box data structure		Remains on screen.
Cancel	*nil* *nil*	*nil*	Disappears.

The callback can use the SKILL expressions in this table.

Expression	Meaning
theListBox->value	Retrieves the list of selected items.
theListBox->choices	Retrieves the list of items that the user can select.
theListBox->choices = '(...)	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

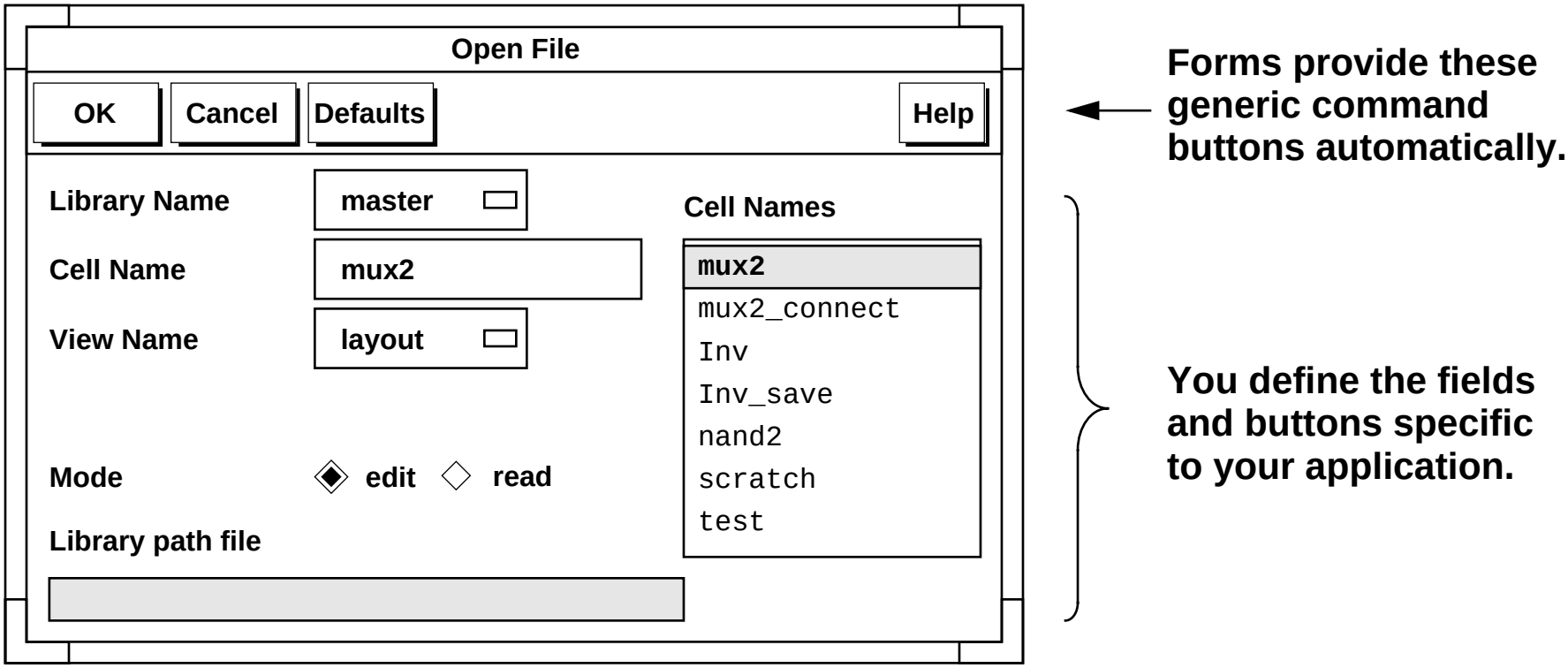
Standard Forms

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.



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:

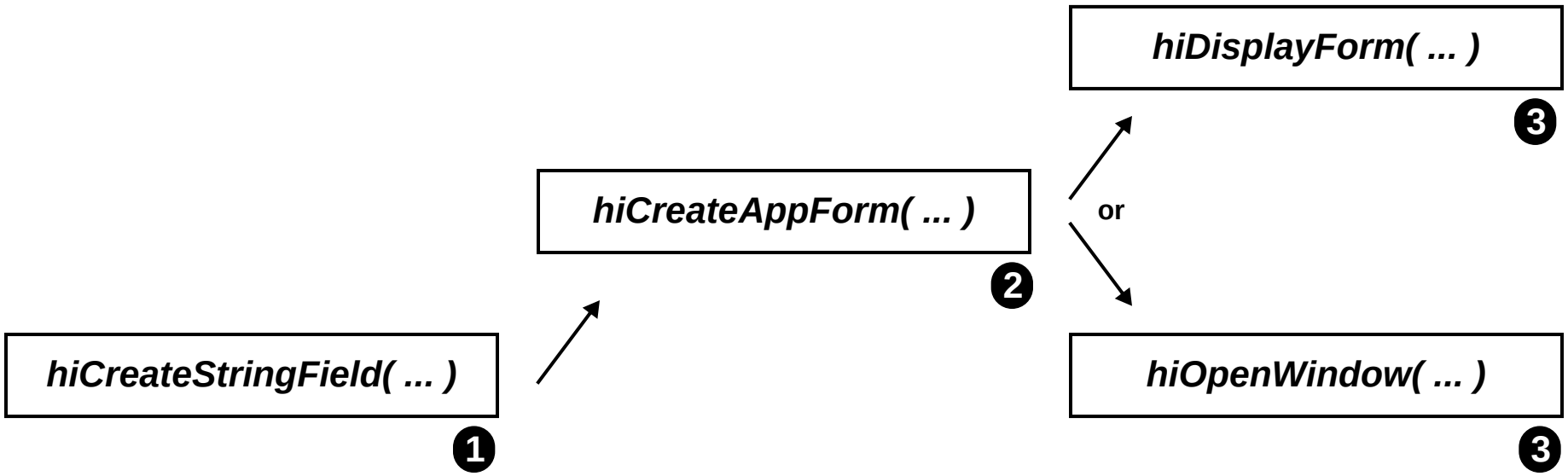
Button	Function
OK	Completes the command and removes the form from the screen.
Cancel	Cancels any changes or selections made to the form and removes the form from the screen.
Defaults	Sets all values in the form to their default values.
Apply	Completes the command and leaves the form on the screen.
Help	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.

Creating and Displaying a Form



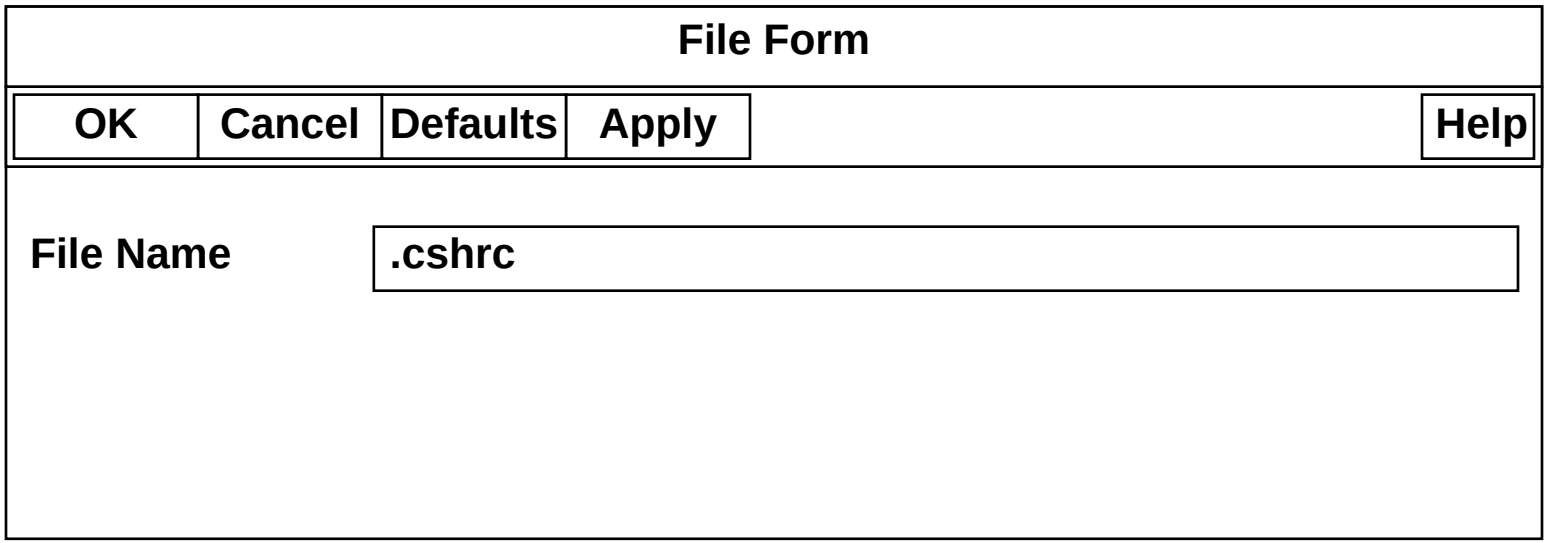
Create Forms from the Bottom Up

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.

Example File Form

This example shows a form with a single string field.



- 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.

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
```

The *hiCreateStringField* Function

Creates a string field entry for a form.

Keyword Argument	Meaning
?name	Symbolic name of this field
?prompt	Field prompt
?value	Initial value
?help	See *User Interface SKILL Functions Reference Manual*.
?defValue	Default value
?font	See *User Interface SKILL Functions Reference Manual*.
?callback	The function call(s) to execute when the user clicks into another field.
?format	See *User Interface SKILL Functions Reference Manual*.
?editable	*nil* => read-only *t* => 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
```

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.

The *hiCreateAppForm* Function

Returns the SKILL data structure of a form with the specified field entries.

Keyword Argument	Meaning
?name	A global variable. SKILL stores the form's data structure in this variable.
?fields	A list of the field data structures.
?formTitle	The form name, which appears in the form's window banner.
?callback	The function or functions to be called when the user selects **OK**, **Apply**, or **Cancel** in a form. Can also be a list containing *'(g_okAction g_cancelAction)*. Can be strings or symbols.
?unmapAfterCB	If *t*, SKILL removes the form after the callback returns. Otherwise, the form is removed when the user clicks **OK**. See *User Interface SKILL Functions Reference Manual*.
?formType	See *User Interface SKILL Functions Reference Manual*.
?dialogStyle	See *User Interface SKILL Functions Reference Manual*.
?buttonLayout	See *User Interface SKILL Functions Reference Manual*.
?initialSize	See *User Interface SKILL Functions Reference Manual*.

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                    Validate field
    hiSetCallbackStatus( theForm t )                      Clear unmap veto
    hiHighlightField(
      theForm                                             Form data structure
      'TrFileNameField                                    Field symbol
      'background )                                       Highlight type
    view(
      theForm->TrFileNameField->value )
  else
    hiSetCallbackStatus( theForm nil )                    Veto form unmap
    hiHighlightField(
      theForm 'TrFileNameField 'error )
  ) ; if
) ; procedure
```

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*.

Displaying the File Form

Button	Callback Evaluated	*hiDisplayForm* Return Value	Form Removed from Screen
OK	Yes	*t*	Yes
Apply	Yes	*does not return*	No
Cancel	No	*nil*	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.

The *hiDisplayForm* Function

By default, the *hiDisplayForm* function does not return until the user clicks **OK** or clicks **Cancel**.

The *?dontBlock* Argument

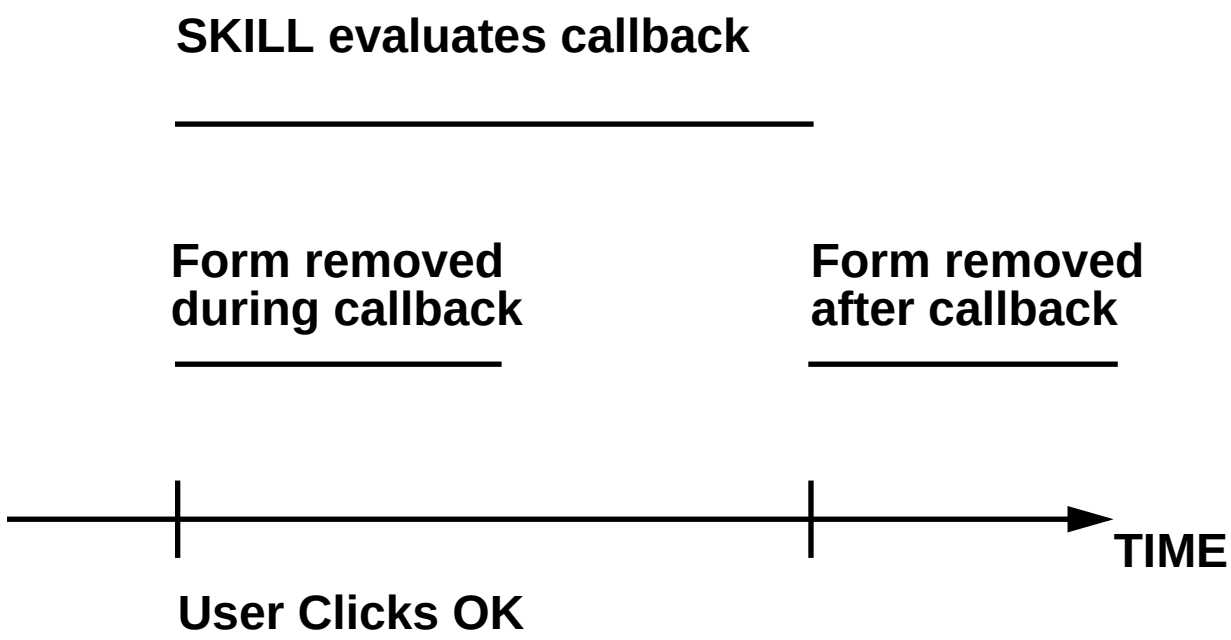
If you pass *?dontBlock t* when you create a form, then the *hiDisplayForm* function returns immediately.

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.



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)
```

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(                    Field data structure
    ?name 'field1                              Symbolic field name
    ...
    )
  f2 = hiCreateStringField(                    Field data structure
    ?name 'field2                              Symbolic field name
    ...
    )
  ...
  fields = list( f1 f2 f3 )
  form = hiCreateAppForm(                      Form data structure
    ?name    'ExampleForm                      MUST be global
    ?fields  fields
    ...
    )
  hiDisplayForm( form )                        Form data structure
) ; let
```

This table explains the role and the scope of each variable in the overhead example.

Variables	Role	Scope (local or global)
f1, *f2*, *f3*	Each value is a field data structure.	Make these variables local.
field1, *field2*, *field3*	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.
fields	The value is a list of the field data structures.	Make this variable local.
form	The value is a form data structure.	Make this variable local.
ExampleForm	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
	2-D point list	hiCreatePointListField
Enumerated Choice	toggle	hiCreateToggleField
	boolean	hiCreateBooleanButton
	radio	hiCreateRadioField
	list box	hiCreateListBoxField
	cyclic	hiCreateCyclicField
	layer cyclic	hiCreateLayerCyclicField
	scale	hiCreateScaleField
Command	button	hiCreateButton
	button box	hiCreateButtonBoxField

Use the Cadence Finder to list all the functions whose name starts with *hiCreate* and end with *Field*.

Some New Types of Fields

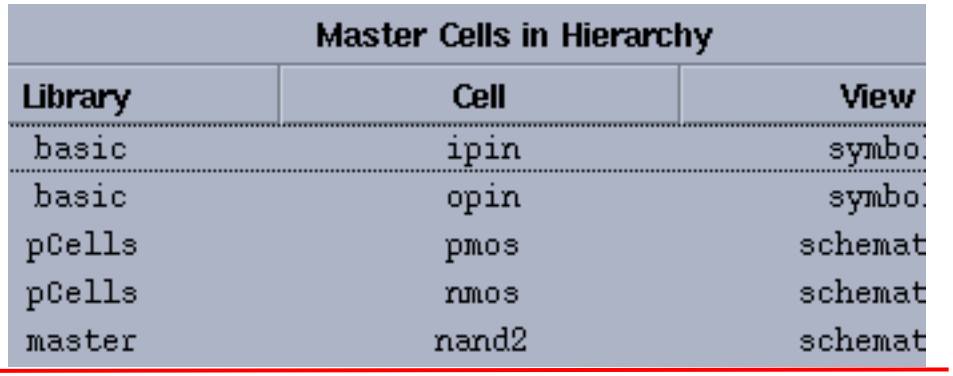
Tab Field: hiCreateTabField

Divide a form using horizontal tabs

Report Field: hiCreateReportField

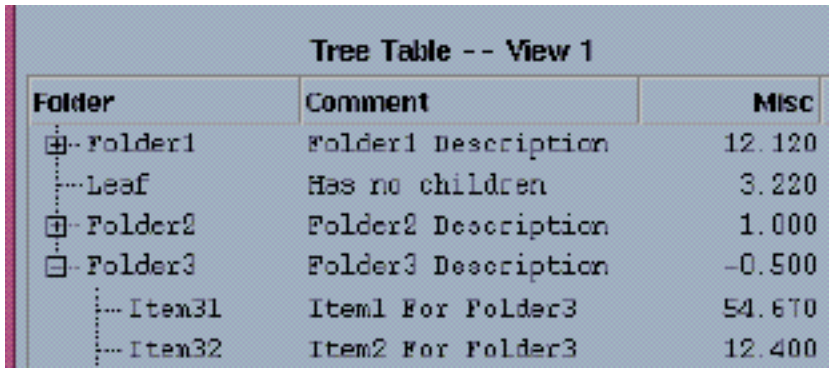
Table data with automatic re-sort

by clicking on a column header



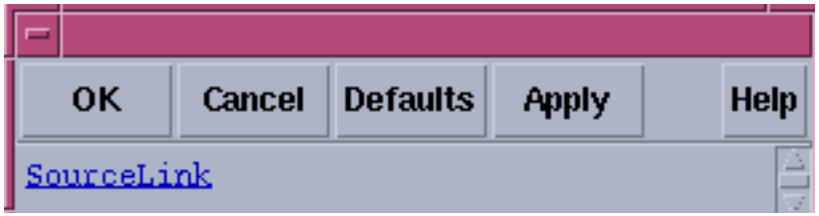
Tree Field: HiCreateTreeTable

An expandable tree to organize data



Hypertext Field: hiCreateHyperTextField

Hypertext links to web information



Tab Field Example:

```
tabField = hiCreateTabField(
?name 'pcellTabField
?fields fieldList
?tabs tabNames
?tabPlacement 'top
)
```

Report Field Example:

```
f3 = hiCreateReportField(
      ?name 'TrHierCellList
      ?title "Master Cells in Hierarchy"
      ?titleAlignment 'center
      ?choices list('("NONE" "NONE" "NONE"))
      ?headers list(
        '( "Library" 150 'center 'string t )
        '( "Cell"    150 'center 'string t )
        '( "View"    150 'center 'string t )
      )
      ?callback "TrGetViews(hiGetCurrentForm())"
     ) ; f3
```

Tree Field Example:

```
treeF=hiCreateTreeTable(
?name 'treeField
?title "Tree Table -- View 1"
?selectMode 'extended
?titleAlignment 'center
?callback "treeCB" ?expandCallback
"expandCB" ?collapseCallback "collapseCB"
?headers '(
("Folder" 120 'left)
("Comment" 150 'left)
("Misc" 80 'right 'float))
?choice rootTree )
```

Hypertext Field Examples:

```
f1 = hiCreateSimpleHypertextField(
   'TrHt1
   "Download Cadence Software"
   "http://www.cadence.com"
)
f2 = hiCreateHypertextField(
   ?name 'TrHt2
   ?value
   "<A name=slink
   href=http://www.cadence.com> SourceLink
   DFII FAQ</A>"
)
```

Lab Overview

Lab 14-4 Exploring Forms

Lab 14-5 Writing a Form Application

- You create a form capable of changing the name and size of an associated window.
- This lab has several optional sections for advanced students.

Module Summary

In this module, we covered

- Context-sensitive pop-up menus
- Fixed menus
- Dialog boxes
- List boxes
- Forms

Category	Functions
Fixed menus	*hiCreateVerticalFixedMenu* *hiCreateHorizontalFixedMenu* *hiDisplayFixedMenu* *hiGetWindowFixedMenu* *hiRemoveFixedMenu* *hiAddFixedMenu*
Dialog boxes	*hiDisplayAppDBox*
List boxes	*hiShowListBox*
Forms	*hiCreateStringField* *hiCreateAppForm* *hiDisplayForm* *hiSetCallbackStatus* *hiHighlightField*