IRIS Explorer User’s Guide
Release 3

Document Number NP2872
ISBN 1−85206−110−3

Appendix B

The Skm Language

This appendix describes the syntax and use of the Skm language to create scripts for the IRIS Explorer command interface.

B.1. Skm Syntax

The Skm scripting language uses prefix notation. All expressions are enclosed in parentheses and all commands precede arguments and operands. If you want to say "do something," the Skm expression is:

```
skm>(do something)
```

To add two numbers, the addition operator precedes the arguments:

```
skm>(+ 1 2)
3
```

To launch a module:

```
skm>(start "ReadImg")
```

White space separates operands and operators. White space is generally just the space character but may also be the tab or newline character. Non−numeric constants should be surrounded by double quotes. Numeric constants do not require quotes.

Footnote: Non−numeric constants without embedded spaces may actually be preceded by a single quote. Only strings containing embedded spaces need be surrounded by double quotes.

Assignment is performed with the *define* command. To define a variable called *var* and assign it the value 3, write:

```
skm> (define var 3)
```

To change its value, write:

```
skm> (define var 4)
```

To display the value of a variable, type the name of the variable at the prompt (without parentheses).

```
skm> var
4
```

B.1.1. The Skm Data Types

The basic Skm data types include:

- numbers
- symbols
- strings
- lists
- procedures

There is no distinction between real and integer numbers in Skm—all numbers have floating point precision. Symbols are identifiers that are preceded by a single quote. For example, *var* is a symbol below:

```
(define var 'foo)
```

Strings are character sequences surrounded by double quotes. String operators are discussed in a later section. Lists are sequences of identifiers surrounded by parentheses. Procedures are discussed in a subsequent section.

Any variable in Skm can be used to contain any data type. Thus, you can store both a number and a string into the same variable.

```
skm> (define var 23)
23
skm> (define var "hello there")
"hello there"
```

The Comment Character

The semicolon is the comment character in Skm. It comments out the remainder of the line on which it is located. As seen above, the define operator returns a value that is printed after the Skm statement is evaluated. In the discussion that follows, the printing of a return value (when present) is sometimes omitted for the sake of simplifying the presentation.

B.1.2. IRIS Explorer Operators

This section introduces the Skm commands used to perform functions in IRIS Explorer. Those commands that accept one or more module names as arguments can handle zero or more module names listed in sequence, or a Scheme list of module names. The semantics of the commands when given no arguments depends on the particular command, but in general it means to apply the command to all selected modules. **Table B−1** lists the available commands.

Command	Purpose
all−mods	List all modules
all−groups	List all groups
break−loop	Break a loop by halting a named module
connect	Establish a connection between two modules
connection?	Test a variable for connection information
connection−list	List all modules connections
copy	Copy selected modules and groups
cut	Cut selected modules and groups
destroy	Destroy selected modules and groups
disable	Disable selected modules and groups
disconnect	Remove a connection between two modules
dup	Duplicate selected modules and groups
enable	Enable selected modules and groups
exec−hilite−off	Turn off execution highlighting
exec−hilite−on	Turn on execution highlighting
fire	Fire selected modules or groups
get−param	Get the value of a module’s parameter
get−pfunc	Get the value of a module’s P−Func
help	Provide information about command syntax
help−mod	Show module’s help page
hide−widget	Hide a module’s widget
interrupt	Interrupt a running module
log	Show module’s log window
loop−ctlr?	Test for the active loop controller module
loop−ctlr−capable?	Test for a loop controller−capable module

loop–ctlr–off	Turn loop controller module off
loop–ctlr–on	Turn loop controller module on
main–log–off	Toggle the Log window off
main–log–on	Toggle the Log window on
maxi–at	Create a maximized module control panel
maxi	Maximize the control panels of selected module
micro	"Micro–ize" the control panel of a module
mini	Minimize the control panel of a module
module?	Test a variable for module start information
module–to–filename	Print the file path of a module's resource file (*.mres* file)
module–to–host	Print the name of the host a module is running on
param–list	Print a list of a module's parameter value(s)
paste	Paste the contents of the editing buffer
pause	Cause the script execution to pause
perf–report–off	Switch off performance reporting for selected modules
perf–report–on	Switch on performance reporting for selected modules
pfunc–list	Print a list of a module's P–Func value(s)
save	Save selected modules to a named map file
select	Select named modules
selected–mods	List the selected modules
set–app–main–win	Designate a maximized module control panel as the main window for an application. If this window is quit while IRIS Explorer is in application mode, the application itself quits.
set–param	Set a parameter value in a module
set–param–minmax	Set min and max values for a range parameter
set–pfunc	Set a module's P–Func value
show–widget	Expose a module's hidden widget (see hide–widget)
start–map	Start a map
start	Start a module
quit–explorer	Exit IRIS Explorer
unmaxi	Remove module's maximized control panel
unmicro	"Unmicro–ize" the control panel of a module
unselect	Unselect named modules
unselected–mods	List the modules not selected

Table B–1 Skm Commands

B.1.3. Command Syntax

Table B–2 summarizes the command syntax. Square brackets enclose optional arguments.

This information is available from within Skm by typing **`(help)`**. You can also type **`(help–command "prefix")`** and you will get a list of all commands beginning with the given prefix.

Command

(all–mods)

(all–groups)

(break–loop "modulename")

(connect "modulename" "portname" "modulename" "portname")

(connection? variable)

```
(connection−list "modulename")
(copy "modulename" ...)
(cut "modulename" ...)
(destroy "modulename" ...)
(disable "modulename"...)
(disconnect "modulename" "portname" "modulename" "portname")
(dup "modulename" ...)
(enable "modulename"...)
(exec−hilite−off "modulename"...)
(exec−hilite−on "modulename"...)
(fire "modulename" ...)
(get−param "modulename" "portname")
(get−pfunc "modulename" "portname")
(help [prefix])
(help−mod"modulename" ...)
(hide−widget "modulename" "widgetname")
(interrupt "modulename"...)
(log "modulename"...)
(loop−ctlr? variable)
(loop−ctlr−capable? variable)
(loop−ctlr−off "modulename"...)
(loop−ctlr−on "modulename"...)
(main−log−off)
(main−log−on)
(maxi−at "modulename" [  [ ‘at X Y]  [ ‘size W H] ] )
(maxi "modulename" ...)
(micro "modulename"...)
(mini "modulename"...)
(module? variable)
(module−to−filename "modulename")
(module−to−host "modulename")
(param−list "modulename")
(paste)
(pause seconds)
(perf−report−off  "modulename" ...)
(perf−report−on  "modulename" ...)
(pfunc−list "modulename")
(save  "filename" "modulename" ...)
(select "modulename" ...)
(selected−mods)
(set−app−main−win "modulename")
(set−param "modulename" "portname" value)
(set−param−minmax "modulename" "portname" minval maxval)
(set−pfunc "modulename" "portname" value)
(show−widget "modulename" "widgetname")
(start−map "map−name" [ ‘at X Y] )
```

(start "modulename" [‘at X Y])
(quit−explorer)
(unmaxi "modulename")
(unmicro "modulename")
(unselect "modulename" ...)
(unselected−mods)

Table B−2 Skm Command Syntax

Some commands have an alternate syntax. Any command that takes a double quoted string specifying a module name can also take a variable of type *<Explorer Module>*. Such variables are returned by the *start* command when it executes successfully. Similarly, a shortcut way of specifying a connection is to use a variable of type *<Explorer Connection>*. These variables are returned by the *connect* command. Use of these special variables is explained in greater detail later.

B.1.4. IRIS Explorer−related Command Descriptions

This section discusses the most commonly used IRIS Explorer−related Skm commands, along with those whose operation is not obvious.

start

The *start* command is used to start modules. Here are a few examples:

```
(start "ReadImg")
(start "SharpenImg" "at" 200 300)
```

The *start* command returns a value of type *<ExplorerModule>* which contains the name of the started module; therefore, you can assign the return value in the following manner:

```
(define mod (start "ReadImg"))
```

The variable *mod* can now stand in for the string "ReadImg" whenever needed.

destroy

Modules can be destroyed using the *destroy* command. So to destroy the module *ReadImg*, you can type:

```
(destroy "ReadImg")
```

or if the value returned by *start* was saved in a variable named *mod*, you can use:

```
(destroy mod)
```

Note that assigning the value returned by *start* is more than a convenience; it is required under some circumstances. There, you requested that the module *ReadImg* be launched. It is possible, though, that a *ReadImg* module already exists. If so, then the module started as a result of this code will be named *ReadImg<2>*, not *ReadImg*. Subsequent connect commands that explicitly refer to the string *ReadImg* will reference the pre−existing *ReadImg* module and will fail to have the desired effect.

The solution is to avoid referring to modules by the name used in the *start* command. Instead, assign the value returned by *start* to a variable and use the variable for subsequent references.

start−map

Maps are started using the *start−map* command:

```
(start-map "cfd")
```

connect

The *connect* command takes an output module and port and an input module and port:

```
(connect "ReadImage" "Output" "SharpenImg" "Img In")
```

As with *start*, a value is returned by *connect* that may be saved in a variable:

```
(define c1 (connect "ReadImg" "Output" "SharpenImg" "Img In"))
```

You can delete the connection with:

```
(disconnect "ReadImg" "Output" "SharpenImg" "Img In")
```

or

```
(disconnect c1)
```

Variables containing module and connection information persist even after the IRIS Explorer objects to which they refer are gone. Thus, you can reconnect the modules just disconnected with:

```
(connect c1)
```

Testing Variable Types

Skm provides the predicates *module?* and *connection?* to test the type of a variable.

```
(module? c1)
()
(connection? c1)
t
```

A predicate returns either `()` (false) or `t` (true). The following:

```
(module? m1)
t
```

means that the variable *m1* contains information about a module started some time in the past. The `true` return value does not necessarily mean the module still exists.

B.2. Skm Language Features

The sections that follow describe the programming language features of Skm.

B.2.1. Comparison Operators

Skm provides two basic comparison operators: *eq?* and *eqv?*. The first, *eq?*, determines if two symbols evaluate the same. The latter, *eqv?*, is more general. It determines the equality of both symbols and numbers.

eq?

Below are examples using *eq?* to test the equality of symbols.

```
(eq? 'test 'test)
t
(eq? 'test 'hat)
()
```

But, you can't use *eq?* for numbers:

```
(eq? 3 3)
()
```

eqv?

Here, *eqv?* is shown for numbers and symbols:

```
(eqv? 3 3)
t
(eqv? 3 4)
()
(eqv? 'test 'test)
t
```

Numbers can also be compared with the following operators:

```
=, !=, <, <=, >, >=
```

You can use these operators only to compare numbers.

B.2.2. Conditional Operators

A conditional test can be performed using the *if* operator, which takes the form:

```
(if test-expr then-expr else-expr)
```

For example:

```
(if (eq? expr-1 expr-2)
  (do-something)
  (do-something-else))
```

If more than one statement belongs in either the *then* or the *else* portion of the *if* construct, use the *begin* operator. It is analogous to curly braces in C and to *begin−end* in Pascal−like languages.

```
(if (eq? expr-1 expr-2)
  (begin
     (do-this)
     (do-that))
     (do-something-else))
```

B.2.3. String Operators

Skm provides the set of string operators listed below:

string?	Returns true if its argument is a string.
string−length	Returns length of string argument.
string−ref	Returns a character at indicated position.
string−set!	Sets a character at indicated position.
string=?	Compares two strings. Returns true if equal.
substring	Returns substring of given string between start and stop index.

string?

string? is a predicate that tests an argument to see if it is a string. It takes a single argument. *string?* returns `t` or `()`, depending on the type of its argument.

```
(string? "some-string")
t
(string? 23)
()
```

string−length

*string−length*returns the length of a string argument.

```
(string-length "some-string")
11
```

string−ref

*string−ref*returns a specific character from a given string. The first argument is the string; the second is the index of the character to be extracted. Indexing is origin zero (that is, the index of the first character in the string is zero). The returned character is still a string and may be used in all string commands.

```
(string-ref "hello" 1)
 "e"
```

string−set!

*string−set!*sets a character in a string. The first argument is the target string; the second argument is the index into the target string; the third argument is the character that replaces the indexed character in the target string. The target string is changed as a side effect. *string−set!* returns `t` if it succeeded and `()` otherwise. The index is origin 0.

```
(string-set! "hello" 0 "J")
t
;; string is changed to "Jello"
```

string=?

The *string=?* function compares two strings for equality. If equal, `t` is returned.

```
(string=? "skm" "skm")
t
(string=? "skm" "scheme")
()
```

substring

The *substring* function returns a substring of a string using the supplied begin and end indices.

```
(substring "abcde" 0 3)
"abc"
```

string−append

The *string−append* function appends two strings.

```
(string-append "abc" "def")
"abcdef"
```

B.2.4. Loading Script Files

To read in a script file from within script code, use the *load* command:

```
(load "myfile.skm")
```

B.3. Procedures in Skm

Creating a procedure in Skm is done with the *define* and *procedure* constructs. The following defines a procedure that takes a single argument, declared formally as 'a,' and adds one to it. The value returned by the procedure, if any, is the last value of the procedure.

```
(procedure (a)
 (+ a 1)
)
```

This procedure doesn't have a name yet (procedures can be "anonymous" in Skm). The following gives the procedure the name plus1.

```
(define plus1 (procedure (a) (+ a 1)) )
```

You would invoke this procedure with:

```
(plus1 3)
4
```

Here is the definition and use of a procedure called "add−pair," which takes two numbers and adds them together:

```
(define add-pair
 (procedure ( arg1 arg2 )
 (+ arg1 arg2)
 )
)

(add-pair 2 2)
4
```

B.3.1. Using a Skm Procedure

Connecting image processing modules in IRIS Explorer is a common occurrence. Image processing modules that operate on a single input typically have an input port named "Img In" and an output port named "Img Out".

This procedure simplifies connecting such modules by supplying these port names automatically.

```
(define image-conn
 (procedure ( mod1 mod2 )
 (connect mod1 "Img Out" mod2 "Img In")))
```

You can use this procedure as follows:

```
(image-conn "SharpenImg" "DisplayImg")
```

B.3.2. Skm Output

Skm output is handled by the *print* command. It takes one or more arguments and generates output to the IRIS Explorer console (the shell from which IRIS Explorer was launched).

```
(define v1 "abc")
(define v2 123)
```

```
(define m1 (start "ReadImg"))
(print v1 v2)
abc123
(print "My module is " m1)
My module is #< IRIS Explorer Module Ref: ReadImg >
```

B.3.3. Looping

In Skm, the procedure *for−loop* is used to implement a style of iteration similar to that used by C. The format of the *for−loop* call is:

```
(for-loop start op stop step loop-body-proc)
```

Like the typical C `for loop` structure, the Skm *for−loop* takes a start count, a comparison operator, a stop count, and an increment. Instead of having a loop body as does the C construct, *for−loop* takes as its final argument a loop−body construct (explained below) or a procedure of one input argument. The input argument to this loop−body procedure is the loop count.

To create a loop that prints all numbers between zero and nine (inclusive), you write:

```
(for-loop 0 < 10 1
   (loop-body(i)
  (print i)
)
0
1
2
3
4
5
6
7
8
9
```

The loop−body construct is used as the final argument to *for−loop*. The loop−body construct takes no arguments or a single input argument. In the latter case, the argument is a variable that is scoped locally to the loop body and that serves as the loop counter.

If the loop counter is not needed, loop−body is used as follows:

```
(for-loop 0 < 3 1 ;; do something 3 times
  (loop-body()
  (do-something)
  )
)
```

Since the built−in *print* procedure can be called with a single input argument, the procedure can be used directly as the last argument to *for−loop* (without having to use the loop−body construct). Here is an alternate implementation of the print example above:

```
(for-loop 0 < 10 1 print)
```

The next example presents a doubly nested loop. The outer loop variable is `i` and the inner loop variable is `j`.

```
(for-loop 0 < 5 1
  (loop-body(i)
  "outer loop: i = " i )
  (for-loop 1 < 4 1
```

```
  (for-loop 1 < 4 1
     (loop-body(j)
       (print " inner loop: j = " j )
     )
  )
)
)

outer loop: i = 0
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 1
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 2
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 3
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 4
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
()
```

B.3.4. Lexical Scoping of Variables

Variables created with *define* are globally scoped regardless of the context in which *define* occurs. Thus, the following uses of *define* both introduce variables into the global environment.

```
(define var1 1)
1
(begin
(define var2 2))
2
(print var1 " " var2)
1 2
```

The `(begin ... )` block in the second example does not behave as does the creation of a local variable in C where the variable goes out of scope at the end of the block:

```
{
int var2 = 2;
}
```

Arguments passed to a procedure in Skm are scoped locally to that procedure. Thus, in the following example *define* introduces neither a nor b into the global environment:

```
(define test
   (procedure ( a b )
      (define a 1)
```

```
        (define b 2)))
```

It seems at first that the example above contradicts the earlier statement that *define* always introduces variables into the global environment. However, *define* here is not being used to create a variable (the variables a and b are created at the entry to the procedure). Rather, *define* is being used to change the value of these variables. You can also use the assigned operator `set!` here.

The let Construct

Skm provides a special construct called *let*, which sets up a locally scoped environment. The format of *let* is:

```
(let      ((var1 init-val)
   (var2 init-val)
   ....
   (varN init-val))

let-body)
```

Any number of locally scoped variables can be created and initialized in the preamble of the *let* statement. The variables thus created and initialized are visible only within the code present in *let*, as seen in the following example:

```
(let ((a 1)        (b 2))
   (print "inside let, a = " a " b = " b))
inside let, a = 1 b = 2
```

B.3.5. List Processing

A list can be created using *list*.

```
(list 'a 'b 'c 'd)
(a b c d)
```

As indicated above, when Skm prints out a list, it surrounds the list with parentheses.

A Skm command that returns a list is *all−mods*:

```
(all-mods)
("SharpenImg" "ReadImg")
```

The first element of a list can be extracted using *first*. A list with the first element removed is obtained through the use of the *rest* command.

Footnote: *first* is also known as *car* and *rest* is also known as *cdr*.

```
(first '(a b c d))
a
(rest '(a b c d))
(b c d)
```

The list `(a b c d)` is quoted so that it will not be evaluated by Skm before being passed to *first* and *rest*.

B.4. Module Interface to Scripting

Modules in C or C++ may issue scripting commands through this API routine:

```
int cxScriptCommand( char * string );
```

This call sends a string containing the Skm code to the Map Editor for evaluation. A valid example is:

```
cxScriptCommand
( "(start \"ReadImg\" 'at 100 100 )" );
```

Embedded double quotes must be escaped in accordance with C syntax for string constants. The input string must contain a valid and complete Skm expression. You may not start an expression in one call and finish it in a subsequent call, as, for instance, in this example:

```
cxScriptCommand ( "(start \"ReadImg\" " );
cxScriptCommand ( " 'at 100 100 ) ");
```

However, multiple independent or nested expressions can be sent in a single call:

```
char * string = "(define a 1) (define b 2)"
cxScriptCommand ( string );
```

Multiple calls can be made as long as each one contains independent expressions:

```
cxScriptCommand ( "(define a 1)" );
cxScriptCommand ( "(define b 2)" );
```

Calling *cxScriptCommand* results in a message transmission and several context switches. Where it is possible to bundle expressions in a single string, performance will be improved.

Some validity checking is done before the script is sent to IRIS Explorer. If *cxScriptCommand()* returns a `-1`, then the validity test failed and no script code was sent. Currently, the validity test will fail if the string contains no parentheses or an uneven set of left and right parentheses.

Normal interpreter output for commands issued by modules is suppressed. Only those error messages generated as a result of module script commands appear on IRIS Explorer's `stderr`. Thus, input that is valid when the user types it at the > prompt, is not valid when generated by a module; in particular, querying the value of a variable if the query is not in parentheses.

Given this definition:

```
(define var 100)
100
```

the following is valid when typed by a user, but is not accepted from a module, since the printing of evaluated expressions is suppressed for module input:

```
var
100
```

CxScriptCommand() performs asynchronously with respect to module execution, and IRIS Explorer does not acknowledge Skm code execution.

Footnote: *cxScriptCommand* returns −1 if the Skm code was not sent to IRIS Explorer at all.

Consequently, a module cannot know for sure whether the script code had the desired effect. For debugging purposes, however, certain types of errors are noted on IRIS Explorer's console output (`stderr`).

As with input typed in directly by the user, script commands perform synchronously with respect to one another when required. Thus in the following sequence, *ReadImg* and *DisplayImg* will exist when the *connect* command is issued, assuming no failure during launch.

```
char * string1 = "(define m1 (start 'ReadImg))"
char * string2 = "(define m2 (start 'DisplayImg))";
char * string3 = "(connect m1 'Output m2 \"Img In\")";

char * string [LONG_LEN];

strcpy ( string, string1 );
```

```
strcpy ( string, string1 );
strcat ( string, string2 );
strcat ( string, string3 );

cxScriptCommand ( string );
```

Remember to not refer to modules by the name used in the *start* command (since the actual module launched may have an instance number appended to its name). Instead, use the name returned by the *start* command.

Global Namespace

There is a single global namespace in the script interpreter shared by the user typing at the IRIS Explorer user interface console and by all modules issuing scripting commands. The global namespace can be exploited by certain applications, but it also represents a hazard to the unwary.

For example, suppose you have written a module, *MyMod*, that under certain conditions will issue a scripting command to start another module. You have assigned the value returned by the *start* command to a variable, called *mod*, used for subsequent manipulations of the launched module.

A problem arises, though, if a user launches more than one instance of *MyMod*. All instances of *MyMod* will assign the result of the *start* command to the global interpreter variable *mod1*. The result is that *mod1* will contain a reference to the last started module and earlier references will be overwritten.

The solution to this problem is to name variables distinctively so that other modules avoid touching them. A good way to get distinctive names is to prepend the module name to the variable. Module names are unique within IRIS Explorer, so the variable namespaces should not overlap.

In some cases, this does not matter. For instance, in an IRIS Explorer application in which there is only one module that issues scripting commands, the module can name variables without restriction. There can even be more than one module issuing script commands in an application, as long as the module writer has coordinated them with respect to interpreter variable naming.

To be perfectly safe, an application map must be run as an application and not as a simple map. When run as an application, the application controls which modules are started. When run as a map, the application is not in total control since the user has access to the Module Librarian and the Map Editor and can thus always launch unrelated maps and modules that may themselves issue script commands.

B.4.2. A Complex Example

This is an example of advanced Skm programming. Given the built−in function *mods−to−list*, we will construct a procedure that determines if a particular module exists. The procedure is named *mod−exists?* and is specified as follows:

```
(define mod-exists?
 (procedure(mod)
 (mod-in-list? mod (all-mods))))
```

mod−exists? is fairly simple. It passes its single argument *mod* and the output of *all−mods* to a helper procedure, *mod−in−list,* which is defined below. As *mod−in−list?* is the last statement in the procedure, its return value will be returned by *mod−exists?*.

Here is the definition of *mod−in−list?*:

```
(define mod-in-list?
 (procedure ( mod m-list )
    (if (null? mod) ()
    (if (null? m-list) ()
     (let ((m (first m-list)))
     (if (null? m) ()
       (if (string=? mod m) t
```

```
        (mod-in-list? mod (rest m-list))))))))
```

mod−in−list? takes two arguments: a module name string and a list of module name strings. First, *mod−in−list?* validates its arguments by checking both the target module name and the list of modules to make sure they are not NULL. The procedure assigns the first element of the list to the temporary variable *m* using a *let* statement and compares it with the target module name. If equal, the procedure returns `t` (true).

If not equal, the procedure continues the search by recursively calling itself with the target string and the module list is reduced by having the first element removed.

Here is an improved version of *mod−exists?* that performs some error checking on its arguments.

```
(define mod-exists?
 (procedure(modname)
    (if (null? modname)
        ; then
        (print "mod-exists?: null arg")
        ; else
        (if (not (string? modname))
          ; then
          (print "mod-exists?: supplied with non-string arg")
      ; else
        (mod-in-list? modname (all-mods))))))
```

As defined, *mod−exists?* works only when given a module name as a string.

B.4.3. Known Bugs

Launching a module on a remote machine with the *on* argument to the start command will not work the first time if no Module Librarian has been created for that machine and if no prior module has been launched on that machine in the current IRIS Explorer session. When such a module fails to launch, it will (as a side effect) cause a Module Librarian to be created for that remote machine. After a few seconds have passed and the Librarian is present, subsequent attempts to launch modules on that machine should succeed.

Parentheses within double−quoted strings may confuse the Skm parser.

Note: When setting the value of a parameter using *set−param*, you may need to know something about the type of widget associated with the parameter.

B.4.4. References

Skm is similar to the Scheme programming language. A good Scheme language text may be of some value to those seeking advanced understanding of Skm.

A thorough, well−written text is *Scheme and the Art of Programming* by Springer and Friedman. A quick and amusing introduction is *The Little Lisper* by Friedman and Felleisen.

Additional information may be found on the network, for pointers see **`http://www.nag.co.uk/0/visual/IE/iecbb/Info.html`**.

B.4.5. Copyright Notice

Skm is based on the SIOD interpreter by George Carrette. Note that features of SIOD not explicitly documented herein are not supported. The following copyright notice accompanies SIOD as distributed on *altdorf.ai.mit.edu*.