Chapter 2.
Scheme Procedures

Topic: *Scheme Interface, *Scheme AIDE Application

This chapter describes how to write Scheme procedures to enhance the functionality provided in the Scheme ACIS Interface Driver Extension (Scheme AIDE). Scheme AIDE (formerly known as TKMAIN), is the part of ACIS that interprets and processes Scheme commands for a Scheme based application.

Scheme procedures can be used with the Scheme AIDE demonstration application or any other Scheme based application.

Scheme procedures are a way to create “mini –programs” based on primitive extensions already built into the ACIS Scheme Interpreter. Any Scheme primitive procedure compiled and linked into the Scheme Interpreter can be included in these procedures.

The Scheme Interpreter supports three types of procedures: *immediate*, *defined*, and *filed*. Scheme procedures do not need to be compiled or linked, because they are interpreted.

Note *This chapter assumes that the user is already familiar with the Scheme language.*

Immediate Procedures

Topic: *Scheme Interface, *Scheme AIDE Application

When creating a procedure from the command line, the procedure is automatically evaluated when the number of close parentheses matches the number of open parentheses and when a carriage return is entered. If close parentheses are omitted, the number of missing parentheses is returned.

When a procedure is to be used only once during a session, it is typed at the interpreter’s command line prompt. Because it is not *defined*, it is evaluated as soon as the interpreter finds a matching close parentheses for each open parentheses. Once it is evaluated, it is discarded. To run it again, it must be entered again (either by typing or recalling from a command buffer).

For example, to create a solid block 20 units on a side with one corner at the origin, enter at the acis> prompt in Scheme AIDE:

```
(solid:block (position 0 0 0) (position 20 20 20))
```

Defined Procedures

Topic: *Scheme Interface, *Scheme AIDE Application

When a procedure is used several times during a session, it can be *defined* and then run as many times as needed with appropriate parameters. However, because the procedure is not saved to a file, it is lost when the session ends. (If it is saved to a file, it is a *filed* procedure.)

The define statement identifies the name of a procedure. The lambda statement identifies the arguments to the procedure.

For example, to define a named procedure to create a cube, enter:

```
(define cube
    (lambda (h w d)
        (solid:block (position 0 0 0) (position h w d))))
```

To call the defined procedure, enter at the acis> prompt in Scheme AIDE:

```
(cube 20 20 20)
```

Filed Procedures

Topic: *Scheme Interface, *Scheme AIDE Application

Procedures can be created, named, stored in a file, loaded into the interpreter and then used as many times as needed. If a procedure is needed in more than one session of the interpreter, it should be saved in a file so that it is not lost when the session ends.

Creating a Procedure File

Topic: *Scheme Interface, *Scheme AIDE Application

Procedure files can be created using any ASCII text editor. Procedure files are usually named <*name*>.scm. This convention is used by *Spatial*, but it is not required. For example, a file might be named myfile.scm. A procedure file contains one or more procedures.

Example 2-1 shows the contents of the file myfile.scm in which the procedure my_view, which takes no arguments, is defined.

```
(define front #f)
(define my_view
    (lambda ()
        (begin
            (set! front (view:dl 320 350 300 300))
            (view:set–title ”front” front)
            (view:set (position 0 –100 0) (position 0 0 0)
                (gvector 0 0 1) front))))
```

Example 2-1. myfile.scm Procedure File

Loading a Procedure File

Topic: *Scheme Interface, *Scheme AIDE Application

The load command reads a procedure file into the interpreter. For example, to load the myfile.scm file (from example 2-1) into the interpreter, enter the following command at the acis> prompt in Scheme AIDE:

```
(load ”myfile.scm”)
```

Autoloading a Procedure File

Topic: *Scheme Interface, *Scheme AIDE Application

The Scheme primitive autoload automatically loads a file containing one or more procedures when a particular procedure call is made. If that file has already been loaded once, it is not loaded again.

For example, issue the following autoload command to autoload the myfile.scm file (from example 2-1) when the user first tries to use the my_view procedure:

```
(autoload my_view ”myfile.scm”)
```

autoload is particularly useful for loading portions of an application as they are needed. If a user does not call an autoload procedure, that procedure’s file is not loaded.

Running a Procedure

Topic: *Scheme Interface, *Scheme AIDE Application

To run (evaluate) a procedure after its file has been loaded, enter the name of the procedure followed by its arguments at the interpreter’s command line prompt:

```
(procedure–name arguments)
```

For example, to run the procedure called my_view (from example 2-1), enter the following command at the acis> prompt in Scheme AIDE:

```
(my_view)
```

Examples for Scheme AIDE

Topic: Scheme AIDE Application

This section provides some Scheme examples that demonstrate many of the capabilities of ACIS and Scheme AIDE.

Initialization

Topic: *Scheme AIDE Application

The acisinit.scm file is read in each time Scheme AIDE is loaded. Scheme AIDE searches the current working directory, the user's home directory, and each directory in the load path for acisinit.scm.

This file is not required, but it is a good place to put common initializations, to define commonly –used utility procedure, and to load needed files. Example 2-2 shows a sample acisinit.scm file that references the file created in example 2-1.

```
; acisinit.scm
; ------------------------------------------------------------
; Purpose---
; This is an example startup file. It is loaded when Scheme AIDE
; program is started. Scheme AIDE looks for a file called
; acisinit.scm in the working directory, your home
; directory (defined by the environment variable HOME or the
; variables HOMEDRIVE and HOMEPATH, and the directories specified
; by the environment variable LOADPATH. If it finds the file, it
; will load it and not search the remaining directories.
; ------------------------------------------------------------

;; load some useful procedures for Scheme AIDE
(load "acis.scm")
(autoload 'apropos "apropos.scm")

;; load the file with my_view defined
(load "myfile.scm")
```

Example 2-2. Sample acisinit.scm File

Finding Commands Using Apropos or Help

Topic: *Scheme AIDE Application

apropos, or help, is a standard Scheme procedure that displays primitive commands given some input string that is part or all of a command. For example, (apropos "journal") lists all the solid:*. Scheme command primitives to print. Execute the (apropos "journal") command to get a list similar to the one shown below.

```
; ------------------------------------------------------------
; Purpose---
; This is an example 'apropos' listing.
; ------------------------------------------------------------
acis> (apropos "journal")
;; journal:on
;; journal:off
;; journal:load
;; journal:step
;; journal:save
;; journal-step
;; journal:abort
;; journal:pause
;; journal-abort
;; journal:resume
;; journal:append
;; ds:journal-on
acis>
```

Example 2-3. Sample 'apropos'

Example 2-4 shows one implementation of help.

```
; ------------------------------------------------------------
; Find Scheme procedures whose name includes a given string.
; For example, to find all procedures that include the word
; "curve" issue the command (help "curve")
; ------------------------------------------------------------
(define help)
(let ((found))
    (define (got-one sym)
        (if (bound? sym)
            (begin
                (set! found #t)
                (print sym)))))
```

```
(set! help (lambda (what)
    (if (symbol? what)
        (set! what (symbol->string what))
        (if (not (string? what))
            (error 'apropos "string or symbol expected")))
    (set! found #f)
    (do ((tail (oblist) (cdr tail))) ((null? tail))
        (do ((l (car tail) (cdr l))) ((null? l))
            (if (substring? what (symbol->string (car l)))
                (got-one (car l)))))
    (if (not found)
        (format #t "~a: nothing appropriate~%" what))
    #v))
```

Example 2-4. Help in Scheme

In Example 2-4, the let statement is responsible for printing each matching command that is found.

The set statement chains through the list of Scheme objects known to the Scheme Interpreter, looking for a substring match with the input string. For example, if the user entered the (help "solid") expression, one of the primitives to match is solid:block. When a match is found, got–one is called, and the full command name string is printed. If no match is found, "nothing appropriate" is printed.

The apropos procedure usually sets up to be autoloaded upon its first use using the inherent Scheme primitive(autoload 'apropos "apropos.scm") command.

If example 2-4 were saved to a file called my_help.scm, it could be invoked by:

```
(load "my_help.scm")
(help "position")
;; cell:classify-position
;; ray:position
;; position:z
;; position:y
;; position:x
;; ds:pick-position
;; pick:position
;; position:project-to-plane
;; position:project-to-line
;; vertex:position
;; rbd:scheme-get-position
;; position:view
;; position:copy
;; position:set!
;; rb-position-hook
;; point:position
;; the_position_hook
;; solid:classify-position
;; position:offset
;; position:set-z!
;; position:set-y!
;; position:set-x!
;; position:distance
;; position:closest
;; entray:position
;; face:ray-at-position
;; position:transform
;; position:interpolate
;; position?
;; position
;; law:eval-position
```

Example 2-5. Invoking my_help.scm in Scheme

Describing a Scheme Object

Topic: *Scheme Interface, *Scheme AIDE Application

describe is a Scheme procedure that can be used to display the type of a Scheme object, and is useful for debugging Scheme procedures, and as a quick reference for the basic Scheme types. Example 2-6 shows one implementation of describe.

```
; ------------------------------------------------------------
; Describe a Scheme object
; ------------------------------------------------------------
```

```
(define (describe x)
    (fluid-let
        ((print-depth 2)
            (print-length 3))
        (format #t "~s is " (if (void? x) '\#v x)))

    (case (type x)
        (integer
            (format #t "an integer.~%"))

        (real
            (format #t "a real.~%"))

        (null
            (format #t "an empty list.~%"))

        (boolean
            (format #t "a boolean value (~s).~%"
                (if x 'true 'false)))

        (void
            (format #t "void (the non-printing object).~%"))

        (character
            (format #t "a character, ascii value is ~s~%"
                (char->integer x)))

        (symbol
            (format #t "a symbol.")
            (let ((l (symbol-plist x)))
                (if (null? l)
                    (format #t " It has no property list.~%")
                    (format #t "~%Its property list is: ~s.~%"
                        l))))

        (pair
            (if (pair? (cdr x))
                (let ((p (last-pair x)))
                    (if (null? (cdr p))
                        (format #t "a list of length ~s.~%"
                            (length x))
                        (format #t "an improper list.~%")))
                (format #t "a pair (cons cell).~%")))

        (environment
            (format #t "an environment.~%"))
```

```
(string
    (if (eqv? x "")
        (format #t "an empty string.~%")
        (format #t "a string of length ~s.~%"
            (string-length x))))

(vector
    (if (eqv? x '#())
        (format #t "an empty vector.~%")
        (if (and (feature? 'oops) (memq (vector-ref x 0)
                '(class instance)))
            (if (eq? (vector-ref x 0) 'class)
                (begin
                    (format #t "a class.~%~%")
                    (describe-class x))
                (format #t "an instance.~%~%")
                (describe-instance x))
            (format #t "a vector of length ~s.~%"
                (vector-length x)))))

(primitive
    (format #t "a primitive procedure.~%"))

(compound
    (format #t "a compound procedure (type ~s).~%"
        (car (procedure-lambda x))))

(control-point
    (format #t "a control point (continuation).~%"))

(promise
    (format #t "a promise.~%"))

(port
    (format #t "a port.~%"))

(end-of-file
    (format #t "the end-of-file object.~%"))

(macro
    (format #t "a macro.~%"))

(entity
    (format #t "an entity of type ~s~%" (entity:debug x)))

(position
    (format #t "a position.~%"))
```

```
(gvector
    (format #t "an ACIS vector.~%"))

(view
    (format #t "a view. ~%"))

(color
    (format #t "a color. ~%"))

(curve
    (format #t "a curve. ~%"))

(else
    (let ((descr-func (string->symbol
        (format #f "describe-~s"
            (type x)))))
        (if (bound? descr-func)
            ((eval descr-func) x)
            (format #t "an object of unknown type (~s)~%"
                (type x)))))))
```

Example 2-6. The Describe Declaration

In Example 2-6, the case statement supplies a format for printing based on the object type of the input.

If example 2-6 were saved to a file called my_desc.scm, it could be invoked by:

```
(load "my_desc.scm")
(describe position)
;;#[primitive position] is a primitive procedure.
```

Example 2-7. Invoking my_desc.scm in Scheme

Limiting List Output

Topic: *Scheme Interface, *Scheme AIDE Application

Scheme lists, such as entity lists, can often appear quite long when printed. Example 2-8 terminates output of any list after the first 20 elements have been printed.

```
; ------------------------------------------------------------
; Limit list printing to the first 20 elements.
; ------------------------------------------------------------
(set! print-length 20)
```

Example 2-8. Limiting List Output

If example 2-8 were saved to a file called my_lim.scm, it could be invoked by:

```
(load "my_lim.scm")
(option:list)
;; (("abl_cap_fudge" . 3) ("abl_c#aps" . #t)
;; ("abl_off_x#curves" . #f) ("abl_rem#ote_ints" . #t)
;; ("abl_require_on#_support" . #t) ("add_bl_atts" . #f)
;; ("addr#ess_debug" . #t) ("all_free_edges" . #f)
;; ("angular_control" . #t) ("api_checking" . #t)
;; ("auto_disp#lay" . #t) ("backup_boxes" . #t)
;; ("bin#ary_format" . #f) ("bl_rem#ote_ints" . #t)
;; ("blend_make_s#imple" . #t) ("brief_comp#_debug" . #t)
;; ("brief_c#urve_debug" . #t) ("brief_m#esh_debug" . #t)
;; ("brief_pc#urve_debug" . #t) ("brief_s#urface_debug" . #t) ...)
```

Example 2-9. Invoking my_lim.scm in Scheme

Displaying an Entity's Statistics

Topic: *Scheme AIDE Application

Example 2-10 displays an entity's vital statistics, including its composition (type, wires, lumps, shells, etc.), status, and bounding box. The entity–stats procedure is passed a single entity as its parameter.

```
; ------------------------------------------------------------
; Display an entity's statistics.
; ------------------------------------------------------------
(define entity-stats
    (lambda ()
        (begin
            (format #t "Please pick an entity...\n")
        (let ((ent (pick:entity (read-event))))
                (if (entity? ent)
            (begin
                (format #t "wires: ~a \n" (length
                    (entity:wires ent)))
                (format #t "lumps: ~a \n" (length
                    (entity:lumps ent)))
                (format #t "shells: ~a \n" (length (entity:shells
                    ent)))
                (format #t "faces: ~a \n" (length
                    (entity:faces ent)))
                (format #t "loops: ~a \n" (length
                    (entity:loops ent)))
                (format #t "edges: ~a \n" (length
                    (entity:edges ent)))
                (format #t "vertices: ~a \n" (length
                    (entity:vertices ent)))
                (format #t "type: ~a \n" (entity:debug ent))
                (format #t "displayed: ~a \n"
                    (entity:displayed? ent))
                (format #t "faceted: ~a \n" (entity:faceted? ent))
                (format #t "highlighted: ~a \n"
                    (entity:highlighted? ent))
                (format #t "material: ~a \n"
                    (entity:material ent))
                (format #t "owner: ~a \n" (entity:owner ent))
                (format #t "bounding box: ~a \n" (entity:box ent))
            )
            (format #t "No entity selected\n")
        )
    ;; Return the selected entity
        ent))))
```

Example 2-10. Entity Statics

If example 2-10 were saved to a file called my_stat.scm, it could be invoked by:

```
(load "my_stat.scm")
(entity-stats)
;; Please pick an entity...
;; wires: 0
;; lumps: 1
;; shells: 1
;; faces: 6
;; loops: 6
;; edges: 12
;; vertices: 8
;; type: solid body
;; displayed: #t
;; faceted: #f
;; highlighted: #f
;; material: #f
;; owner: #[entity 38 1]
;; bounding box: (#[position -15 -20 -30] . #[position 15 25 35])
;; #[entity 6 1]
```

Example 2-11. Invoking my_stat.scm in Scheme

Suppressing Garbage Collection Messages

Topic: *Scheme Interface, *Scheme AIDE Application

Garbage collection is the Scheme Interpreter's process of recovering and reusing memory. Example 2-12 suppresses the notification messages that occur during garbage collection (but does not suppress garbage collection itself). It is useful when procedures create many objects and use lots of memory, especially for recursive operations and intermediate results. No Scheme object is ever destroyed unless Scheme can prove that that object is never used again.

```
; ------------------------------------------------------------
; Suppress (#f) or enable (#t) garbage collection messages
; ------------------------------------------------------------
(set! garbage-collect-notify? #f)
```

Example 2-12. Suppressing Garbage Collection

Finding All Entities

Topic: *Scheme AIDE Application

Example 2-13 prints out a list of Scheme objects that are entities, and the type of each entity. If the procedure is called using (find-ents ""), a list of all entities prints. If the procedure is called with (find-ents "<*string*>"), a list of all entities containing <*string*> in their name prints.

```
; ------------------------------------------------------------
; Find and display a list of entities
; The parameter "what" restricts the list to entities with a
; specified string in their name.
; ------------------------------------------------------------

; Bind find-ents in the toplevel environment, so that it can
; find all of the entities.
(define find-ents)

; Define the procedure got-one that outputs the the symbol name
; and the entity type of symbol.
(let ((found))
       (define (got-one sym)
           (if (bound? sym)
               (begin
                   (set! found #t)
                   (format #t "~s ~s ~%" (symbol->string sym)
                       (entity:debug (eval sym))
           )
       )
   )
))

; Define the procedure find-ents that searches the current
; environment object list and find those symbol names that are
; of type entity.

; It first checks to see if the object passed is a symbol. If it
; is not a valid symbol, an error is reported.

; Next, it loops over oblist (a list of object lists). Thus,
; it takes each sublist and loops over all of the component object
; and then goes on to the next sub-list.

; If the object name contains the string passed, it is then tested
; to see if it is bound to the current environment. If this is the
; case, the object is checked to see if it is of type entity. If
; this is the case, the procedure got-one is called. If any of
; these fail, the next object is processed.

; If no object are found that contain the string passed, a message
; is returned as such.

; Once all of the objects have been processed, the procedure
; returns #v, or null.
```

```
(set! find-ents
    (lambda (what)
        (if (symbol? what)
            (set! what (symbol->string what))
            (if (not (string? what))
                (error 'find-ents "string or symbol expected")
            )
        )


        (set! found #f)
        (do ((tail (oblist) (cdr tail))) ((null? tail))
            (do ((l (car tail) (cdr l))) ((null? l))
                (if (substring? what (symbol->string (car l)))
                (if (bound? (car l))
                    (if (entity? (eval (car l)))
                        (got-one (car l))
                    )
                )
                )
            )
        )
        (if (not found)
            (format #t "~a: nothing appropriate~%" what)
        )
    #v)
    )
```

Example 2-13. Finding an Entity

If example 2-13 were saved to a file called my_ent.scm, it could be invoked by:

```
(load "my_ent.scm")
; Create something to look for.
(define my_block (solid:block
    (position -10 -20 -30) (position 15 25 35)))
;; my_block
(find-ents "block")
;; "my_block" "solid body"
```

Example 2-14. Invoking my_ent.scm in Scheme

Displaying Top Level Environment

Topic: *Scheme AIDE Application

Example 2-15 prints out the current Scheme top –level environment and is useful for debugging.

```
; ------------------------------------------------------------
; Display environment stats
; ------------------------------------------------------------

(define env-stats
    (lambda ()
        (display "\nactive wcs color: ")
        (display (env:active-wcs-color))
        (display "\ndefault color: ")
        (display (env:default-color))
        (display "\nhighlight color: ")
        (display (env:highlight-color))
        (display "\nauto display: ")
        (display (env:auto-display))
        (display "\npoint size: ")
        (display (env:point-size))
        (display "\npoint style: ")
        (display (env:point-style))
        (display "\nactive view: ")
        (display (env:active-view))
        (display "\nviews: ")
        (display (env:views))
        (display "\n")
    )
)
```

Example 2-15. Environment Statics

If example 2-15 were saved to a file called my_envstat.scm, it could be invoked by:

```
(load ”my_envstat.scm”)
(env–stats)
;; active wcs color: #[color 0 1 0]
;; default color: #[color 0.8 0.5 0.2]
;; highlight color: #[color 0 1 1]
;; auto display: #t
;; point size: 10
;; point style: X
;; active view: #[view 1076013904]
;; views: (#[view 1076013904])
```

Example 2-16. Invoking my_envstat.scm in Scheme

Listing Unbound and Bound Symbols

Topic: *Scheme Interface, *Scheme AIDE Application

Example 2-17 displays a list of all bound and unbound symbols known to the Scheme Interpreter. Symbols are bound to values at different points in time depending on the Scheme special form commands used, such as define, let, and set.

```
; ------------------------------------------------------------
; Display unbound and bound Scheme symbols
; ------------------------------------------------------------

(define symbols
    (lambda ()
        (begin
            (display ”UNBOUND SYMBOLS-------------------------\n”)
            (do ((tail (oblist) (cdr tail))) ((null? tail))
                (do ((l (car tail) (cdr l))) ((null? l))
                    (if (not(bound? (car l)))
                        (print (car l)))))
            (display ”BOUND SYMBOLS-----------------------------\n”)
            (do ((tail (oblist) (cdr tail))) ((null? tail))
                (do ((l (car tail) (cdr l))) ((null? l))
                    (if (bound? (car l))
                        (print (car l))))))))
```

Example 2-17. Displaying symbols

Handling Error Messages

Topic: *Scheme Interface, *Scheme AIDE Application

Example 2-18 formats error messages produced by Scheme AIDE.

```
; ------------------------------------------------------------
; Display error messages in a specific format
; ------------------------------------------------------------

(define (error-string error-msg)
    (string-append
        (format #f "~s: " (car error-msg))
        (apply format #f (cdr error-msg))
        ))
(set! error-handler
           (lambda error-msg
               (display "*** Error") (newline)
               (display (error-string error-msg)) (newline)
               ))
```

Example 2-18. Reformatting Error Messages

In Example 2-18, define specifies the format of the message. set! assigns to the error handler a procedure that uses this format.

```
(error 12 "My error 12 just occurred")
*** Error
12: My error 12 just occurred
```

Example 2-19. Error Handling

Scheme printf

Topic: *Scheme Interface, *Scheme AIDE Application

Example 2-20 implements a pseudo –printf command in Scheme.

```
; ------------------------------------------------------------
; Printf
; ------------------------------------------------------------
(define printf (lambda l (apply format (cons #t l)) (newline)))
```

Example 2-20. Scheme printf

Trace Scheme Procedures

Topic: *Scheme Interface, *Scheme AIDE Application

Example 2-21 implements tracing of compound Scheme procedures (the primitives are not traced).

```
; ------------------------------------------------------------
; Trace entry and exit from Scheme procedures.
; Procedures of type #[compound] are traced
; Procedures of type #[primitive] are not traced.
; ------------------------------------------------------------
(define trc:trace-list '(()))

;; reset-trace deletes all procedures from the list of procedures
;; to be traced.
(define (reset-trace) (set! trc:trace-list '(())))

;; trace adds a procedure to the list of of procedures to be
;; traced.
(define-macro (trace func)
    `(let ((the-func (eval ,func))
           (result #v))
       (if (assoc ',func trc:trace-list)
           (error 'trace "~s already trace on." ,func))
       (if (not (compound? ,func))
           (error 'trace "wrong argument type ~s
               (expected compound)" (type ,func)))
       (set! trc:trace-list
           (cons '()
               (cons (cons ',func the-func)
                   (cdr trc:trace-list))))
       (set! ,func
           (lambda param-list
               (format #t "# Entering ~s~%"
                   (cons ',func param-list))
               (set! result (apply the-func param-list))
               (format #t "# Exiting ~s ==> ~s~%"
                   (cons ',func param-list)
                   result)
               result))))

;; untrace deletes a procedure from the list of procedures
;; to be traced.
(define-macro (untrace func)
    `(let ((the-func (assoc ',func trc:trace-list)))
```

```
(define (remove! func)
    (let ((prev trc:trace-list)
            (here (cdr trc:trace-list)))
        (while (and here
                    (not (eq? func (car here))))
                (set! prev here)
                (set! here (cdr here)))
        (if (not here)
            (error 'remove "item ~s not found." func)
            (set-cdr! prev (cdr here)))))

(if the-func
        (begin (remove! ',func)
                (set! ,func (cdr the-func))))))
```

Example 2-21. Tracing

In Example 2-21, (trace <*procname*>) adds a procedure to the list of procedures to be traced, and binds a set of calls around the procedure that displays entering and exiting messages. (untrace <*procname*>) deletes a procedure from the list of procedures to be traced, and removes the entering and exiting message calls.

When a procedure on the list is called, trace displays "Entering <*procname*>", and when that procedure exits, trace displays "Exiting <*procname*>". reset–trace deletes the entire trace list.