Chapter 1.
Scheme Support Component

Component: *Scheme Interface, *Scheme AIDE Application, *Part Management, *History and Roll

The Scheme Support Component (SCM), in the scm directory, contains the Scheme Interpreter, basic Scheme extensions, and interfaces which are used by the Scheme ACIS Interface Driver Extension demonstration application (Scheme AIDE).

SCM includes an example of how PART and PART_CONTEXT are used, an example of part management, an example of history management and rollback, and routines to implement commonly used functionality, such as environment settings, system settings, user interface, and journaling.

Scheme AIDE accepts Scheme commands that are entered in a command window and displays the visual results in a separate view window. The Scheme AIDE Main Program Component (TKMAIN) contains the Scheme AIDE application files.

In addition to demonstrating ACIS functionality, Scheme AIDE can be used as an example of a Scheme based ACIS application, or as a starting point for creating a new Scheme based ACIS application. Developers can also create their own Scheme extensions for use within Scheme AIDE (or any other Scheme based ACIS application).

Scheme Language in ACIS

Topic: *Scheme Interface, *Scheme AIDE Application

Scheme is a public domain language based on LISP and has several dialects. *Spatial* has selected the Elk version of Scheme as having the best blend of features for the needs of ACIS application developers. Scheme is:

Interpretive . . . . . Programs and commands are processed by a Scheme interpreter at run time.

Tail recursive . . . . . Complex algorithms can be coded using procedure recursion in a finite space.

Object–oriented . . . . . Both data and procedures are considered to be Scheme objects, which can be created, stored, and returned as results of procedures.

Block structured . . . . . Standard block structuring concepts are used along with recursion.

Value passing . . . . . Arguments are always passed by value, not by reference, which forces argument expressions to be evaluated before being passed.

The following sections summarize the native Scheme language. This information was extracted from the *Revised[4] Report on the Algorithmic Language Scheme* (R4RS), and describes the syntax, notation and terminology, lexical conventions, basic concepts, expressions, and program structure of Scheme.

Notation and Terminology

Topic: *Scheme Interface, *Scheme AIDE Application

Notation and terminology in the Scheme language means the use of parentheses, naming conventions, lexical conventions, identifiers, white space, and comments.

Parentheses

Topic: *Scheme Interface, *Scheme AIDE Application

Scheme uses ( ) (parentheses) notation for programs and other data. Use as many parenthetic statements as needed, as long as there are an equal number of open and close parentheses.

Naming Conventions

Topic: *Scheme Interface, *Scheme AIDE Application

Scheme uses the following naming conventions:

? . . . . . Ends procedures, called *predicates,* that always return a *Boolean* (logical true or false) value. For example, body? returns #t if the object is an ACIS BODY.

! . . . . . Ends procedures, called *mutation procedures,* that store values into previously–allocated locations. For example, set! GREEN 2 sets the GREEN variable to a new value of 2.

–> . . . . . Appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, list –> vector takes a list and returns a vector whose elements are the same as those of the list.

Lexical Conventions

Topic: *Scheme Interface, *Scheme AIDE Application

No distinction is made between uppercase and lowercase letters, except within character and string constants. For example, Foo is the same identifier as FOO, and #x1AB is the same number as #X1ab. The following paragraphs describe identifiers, white space, comments, and other notations used in Scheme.

Identifiers

Topic: *Scheme Interface, *Scheme AIDE Application

An identifier in Scheme is a sequence of letters, digits, and extended keyboard characters that do not begin with a number. The extended characters +, –, and . . . are also identifiers.

Identifiers have the following uses in Scheme programs:

- Any identifier can be used as a variable.
- When an identifier appears as a literal or within a literal, it denotes a symbol.
- Scheme reserves the following identifiers as syntactic keywords:

=>	do	or
and	else	quasiquote
begin	if	quote
case	lambda	set!
cond	let	unquote
define	let*	unquote–splicing
delay	letrec	

White Space and Comments

Topic: *Scheme Interface, *Scheme AIDE Application

White space characters are spaces and new lines. White space can occur between any two tokens, but not within a token. White space can also occur inside a string where it is necessary.

A semicolon (;) indicates the start of a comment. The comment continues to the end of the line on which the semicolon appears. Comments are invisible to Scheme, but the end of the line is visible as white space, which prevents a comment from appearing in the middle of an identifier or number.

Other Notations

Topic: *Scheme Interface, *Scheme AIDE Application

The following is a description of notations used in Scheme.

. + –	Used in numbers and can occur anywhere in an identifier except as the first character. A delimited plus (+) or minus (–) sign by itself is also an identifier. A delimited period (.) is used in the notation for pairs, and to indicate a rest–parameter in a formal parameter list. A delimited sequence of three successive periods (. . .) is also an identifier.
()	Used for grouping and for annotating lists.
’	Indicates *literal* data. Literal data is object information whose evaluation is inhibited so that information is used as data.
‘	Indicates almost–constant data.
,@	Used with back–quotes.
”	Delimits strings.
\	Used in the syntax for character constants and as an escape character within string constants.
[] {}	Reserved for future language extensions.
#	Introduces constants and number notations.
#\	Introduces a character constant.
#(	An open parenthesis introduces a vector constant. A closed parenthesis terminates a vector constant.
#b #o #d #x	Used in number notation, these mean binary, octal, decimal, and hexadecimal, respectively.

Basic Concepts

Topic: Scheme Interface, Scheme AIDE Application

Basic concepts in the Scheme language include variables and regions, true and false, and predicates.

Variables and Regions

Topic: *Scheme Interface, *Scheme AIDE Application

Any identifier that is not a syntactic keyword may be used as a variable. A *variable* names a location that stores a value and is *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable’s *value*.

Certain expression types create new locations and bind variables to those locations. lambda expressions explain the most fundamental of these *binding constructs*. The other binding constructs are let, let*, letrec, and do expressions.

True and False

Topic: *Scheme Interface, *Scheme AIDE Application

A conditional test returns a Boolean value indicating true or false. True is represented by the Boolean value #t and false is represented by the Boolean value #f. A conditional test can use any Scheme value as a Boolean value. Unless an argument to a conditional test explicitly has the value #f (false), it is interpreted as #t (true).

Predicates

Topic: *Scheme Interface, *Scheme AIDE Application

A predicate is any procedure that returns a Boolean. No object satisfies more than one of the following predicates, which define the following types:

boolean? . . . . . Identifies whether an object is a Boolean.

pair? . . . . . Identifies whether an object is a pair.

symbol? . . . . . Identifies whether an object is a symbol.

number? . . . . . Identifies whether an object is a number.

char? . . . . . Identifies whether an object is a character string.

string? . . . . . Identifies whether an object is a string.

vector? . . . . . Identifies whether an object is a vector.

procedure? . . . . . Identifies whether an object is a procedure.

Expressions

Topic: *Scheme Interface, *Scheme AIDE Application

A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.

Expression types are categorized as *primitive* or *derived* (also known as *compound)*. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive but can instead be explained in terms of the primitive constructs. They capture common patterns of usage and are convenient abbreviations.

Primitive Expression Types

Topic: Scheme Interface, Scheme AIDE Application

Primitive expression types include variable references, literal expressions, procedure calls, lambda expressions, conditionals, and assignments.

Variable References

Topic: *Scheme Interface, *Scheme AIDE Application

An expression consisting of a variable is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

Literal Expressions

Topic: *Scheme Interface, *Scheme AIDE Application

The following are examples of literal expression forms:

- (quote <datum>)

  Evaluates to <datum>, where <datum> is any external representation of a Scheme object. This notation is used to include literal constants in Scheme code. For example:

  ```
  (load "file.scm")
  ```

- '<datum>

  The abbreviation of (quote <datum>). The notations are equivalent. For example:

  ```
  (load 'file.scm)
  ```

- <constant>

  Numerical constants, string constants, character constants, and Boolean constants evaluate to themselves; they need not be quoted. For example:

  ```
  123
  ;; 123
  ```

Procedure Calls

Topic: *Scheme Interface, *Scheme AIDE Application

A procedure call has a form such as:

(<operator> <operand$_1$> . . .)

Write a procedure call by enclosing the expressions for the procedure call and the arguments to be passed to it in parentheses. The operator and operand expressions are evaluated and the resulting procedure is passed to the resulting arguments. New procedures are created by evaluating lambda expressions.

```
(+ 1 2 3 4)
;; 10
```

Example 1-1. Procedure Call Example

Lambda Expressions

Topic: *Scheme Interface, *Scheme AIDE Application

A lambda expression evaluates to a procedure. The environment in effect when the lambda expression was evaluated is remembered as part of the procedure. When the procedure is later called with actual arguments, the environment in which the lambda expression was evaluated is extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values are stored in those locations, and the expressions in the body of the lambda expression is evaluated sequentially in the extended environment. The result of the last expression in the body returns as the result of the procedure call.

The form of the lambda expression is:

(lambda <formals> <body> . . .)

where <formals> is a formal argument list, and <body> is a sequence of one or more expressions.

The forms for <formals> are as follows:

- (<variable$_1$> . . .)

  The procedure takes a fixed number of arguments. When the procedure is called, the arguments are stored in the bindings of the corresponding variables.

```
(define test (lambda (num1 num2)(+ num1 num2)))
(test 2 3)
;; 5
```

Example 1-2. Lambda Expressions Example

The forms for <variable> are as follows:

- <variable>

  The procedure takes any number of arguments. When the procedure is called, the sequence of actual arguments is converted into a newly allocated list and the list is stored in the binding of the <variable>.

```
(define test (lambda (ls1 ls2) (car ls2)))
(test (list 1 2) (list 3 4 5 6))
;; 3
```

Example 1-3.

- (<variable$_1$> . . . <variable$_{n-1}$> . <variable$_n$>)

  If a space–delimited period precedes the last variable, the variable stored in the binding of the last variable is a newly allocated list of the actual arguments left over after all the other actual arguments are matched up against the other formal arguments.

```
(define test (lambda (. ls) (car ls)))
(test 1 2 3 4 5 6)
;; 1
```

Example 1-4.

A <variable> cannot appear more than once in <formals>.

Conditionals

Topic: *Scheme Interface, *Scheme AIDE Application

A primitive conditional expression has one of the following forms:

- (if <test> <consequent> <alternate>)
- (if <test> <consequent>)

where <test>, <consequent>, and <alternate> are arbitrary expressions.

An if expression is evaluated as follows: first, <test> is evaluated. If <test> yields TRUE, then <consequent> is evaluated and its value returns; otherwise, <alternate> is evaluated and its value returns. If <test> yields FALSE and no <alternate> is specified, the result of the expression is unspecified.

Assignments

Topic: *Scheme Interface, *Scheme AIDE Application

A primitive assignment expression has the following form:

(set! <variable> <expression>) – <expression>

This expression is evaluated, and the resulting value is stored in the location to which <variable> is bound. The location to which <variable> is bound must be either in some region enclosing the set! expression or at top level. The result of the set! expression is unspecified.

```
(define test 1)
(set! test 2)
;; 1
```

Example 1-5.

Derived Expression Types

Topic: Scheme Interface, Scheme AIDE Application

Derived expression types include conditionals, binding constructs, sequencing, and iteration.

Derived Conditionals

Topic: *Scheme Interface, *Scheme AIDE Application

The form of a derived conditional expression is:

(cond <clause$_1$> <clause$_2$> . . .)

where each <clause> is in the form (<test> <expression> . . .) and where <test> is any expression. The last <clause> is an "else clause," which has the form (else <expression$_1$><expression$_2$>. . .).

A cond expression is evaluated by evaluating the <test> expressions of successive <clause>s in order until one of them evaluates to TRUE, the remaining <expression>s in its <clause> are evaluated in order, and the result of the last <expression> in the <clause> returns as the result of the entire cond expression. If the selected <clause> contains only the <test> and no <expression>s, the value of the <test> returns as the result. If all <test>s evaluate FALSE values, and there is no else clause, the result of the conditional expression is #f; if there is an else clause, its expressions are evaluated and the value of the last one returns.

The following examples illustrate some conditional expressions.

```
(define test
    (lambda (c)
        (if (char=? c #\a) #t #f)
    )
)
;; test
(test #\a)
;; #t
(test #\b)
;; #f
```

Example 1-6.

```
; Returns absolute value of input number.
(define test
    (lambda (num)
        (if (> num 0) num (– 0 num))
    )
)
;; test
(test 1)
;; 1
(test –1)
;; 1
```

Example 1-7.

```
(define test
    (lambda (num)
        (cond
            ((> num 0) num)
            ((< num 0) (– 0 num))
            (else 0)
        )
    )
))
;; test
(test 1)
;; 1
(test –1)
;; returns 1
(test 0)
;; 0
```

Example 1-8.

Binding Constructs

Topic: *Scheme Interface, *Scheme AIDE Application

The three binding constructs let, let*, and letrec give Scheme a block structure. The syntax of the three constructs is identical, but they differ in the regions they establish for their variable bindings. In a let expression, the initial values are computed before any variables become bound. In a let* expression, the bindings and evaluations are performed sequentially. In a letrec expression, all the bindings are in effect while their initial values are being computed, allowing mutually recursive definitions.

- (let <bindings> <body>)
  Where <bindings> has the form ((<variable$_1$> <init$_1$>) . . .). Where each <init> is an expression, and <body> is a sequence of one or more expressions. A <variable> cannot appear more than once in the list of variables being bound.

```
(let ((x 3)
    (* x 3))
;; 9
```

Example 1-9.

The <init>s are evaluated in the current environment, the <variable>s are bound to fresh locations holding the results, the <body> is evaluated in the extended environment, and the value of the last expression of <body> returns. Each binding of a <variable> has <body> as its region.

- (let* <bindings> <body>)

  Where <bindings> has the form ((<variable$_1$> <init$_1$>) . . .) and <body> is a sequence of one or more expressions.

```
(let* ((x 3))
    (define y (* x 3))
    (define z (+ y 3))
    (+ y z)
)
;; 21
```

Example 1-10.

let* is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (<variable$_1$> <init$_1$>) is that part of the expression to the right of the binding. Thus, the second binding is done in an environment in which the first binding is visible, and so on.

- (letrec <bindings> <body>)

  Where <bindings> has the form ((<variable$_1$> <init$_1$>) . . .) and <body> is a sequence of one or more expressions. It is an error for a <variable> to appear more than once in the list of variables being bound.

```
(letrec ((x 12) (y 3)) (* x y))
;; 36
```

Example 1-11.

The <variable>s are bound to fresh locations holding undefined values, the <init>s are evaluated in the resulting environment, each <variable> is assigned to the result of the corresponding <init>, the <body> is evaluated in the resulting environment, and value of the last expression in <body> returns. Each binding of a <variable> has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

One restriction on letrec is very important—it must be possible to evaluate each <init> without assigning or referring to the value of any <variable>.

Sequencing

Topic: *Scheme Interface, *Scheme AIDE Application

Sequencing has the following form:

(begin <expression1> <expression2> . . .)

The <expression>s are evaluated sequentially from left to right, and the value of the last <expression> returns. This expression type sequences side effects such as input and output.

```
(begin
    (display (+ 3 4)) (newline)
    (display (* 1 2)) (newline)
    (display (+ 1 3)) (newline))
;; 7
;; 2
;; 4
```

Example 1-12. Sequencing Example

Iteration

Topic: *Scheme Interface, *Scheme AIDE Application

The iteration construct is the do expression.. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits with a specified result value. The do expression has the following form:

(do ((<variable1> <init1> <step1>) . . .) (<test> <expression> . . .) <command> . . .)

A do expression is evaluated as follows:

1. <init> expressions are evaluated
2. <variable>s are bound to fresh locations
3. The results of <init> expressions are stored in the bindings of the <variable>s.

Then, the iteration phase begins. Each iteration begins by evaluating <test>. If the result is FALSE, the <command> expressions are evaluated in order for effect, <step> expressions are evaluated in some unspecified order, <variable>s are bound to fresh locations, the results of <step>s are stored in the binding of the <variable>s, and the next iteration begins.

If <test> evaluates to TRUE, the <expression>s are evaluated from left to right and the value of the last <expression> returns as the value of the do expression. If no <expression>s are present, the value of the do expression is unspecified.

The region of the binding of a <variable> consists of the entire do expression except for the <init>s. A <variable> cannot appear more than once in the list of do variables. If a <step> is omitted, the effect is the same as if (<variable> <init> <variable>) was written instead of (<variable> <init>).

The following is an iteration example:

```
(define result 1)
;; result
(do ((i 1 (+ i 1))) ((> i 5) result)
    (set! result (* result i))
)
;; 120
```

Example 1-13. Iteration Example

Program Structure

Topic: *Scheme Interface, *Scheme AIDE Application

A Scheme program consists of a sequence of expressions and definitions. Programs are stored in files or entered interactively to a running Scheme system. *Expressions* on page 1–5 describes Scheme expressions. This section describes definitions.

Definitions that occur at the top level of a program are interpreted as declarations. These definitions cause bindings to be created in the top–level environment. Expressions occurring at the top level of a program are interpreted imperatively. These expressions are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

Definitions are valid in some, but not all, contexts where expressions are allowed. Definitions are valid only at the top level of a <program> and, in some implementations, at the beginning of a <body>. A definition must have one of the following forms:

- (define <variable> <expression>)

  This syntax is essential.
- (define (<variable> <formals> <body>))

  This syntax is not essential. <formals> must be either a sequence of zero or more variables, or a sequence of one or more variables followed by a space–delimited period and another variable (as in a<lambda> expression). This form is equivalent to: (define <variable> (lambda (<formals>) <body>))

```
(define factorial (lambda (num)
    (if (= num 0) 1 (* num (factorial (– num 1))))
    )
)
;; factorial
(factorial 5)
;; 120
```

Example 1-14.

- (define (<variable> . <formal>) <body>)

  This syntax is not essential. <formal> should be a single variable. This form is equivalent to: (define <variable> . (lambda <formal> <body>))

```
(define (factorial num)
    (if (= num 0) 1 (* num (factorial (– num 1)))))
;; factorial
(factorial 5)
;; 120
```

Example 1-15.

- (begin (<definition1> . . .)

  This syntax is essential. This form is equivalent to the set of definitions that form the body of the <variable>.

Top Level Definitions

Topic: Scheme Interface, Scheme AIDE Application

At the top level of a program, the following definition:

(define <variable> <expression>)

has essentially the same effect as the following assignment expression:

(set! <variable> <expression>)

if <variable> is bound. However, if <variable> is not bound, the definition binds <variable> to a new location before performing the assignment, whereas a <set!> cannot be performed on an unbound variable.

All Scheme implementations must support top level definitions.

Internal Definitions

Topic: Scheme Interface, Scheme AIDE Application

Some Scheme implementations permit definitions to occur at the beginning of a <body> (that is, the body of a <lambda>, <let>, <let*>, <letrec>, or <define> expression). Such definitions are known as *internal definitions*. The variable defined by an internal definition is local to the <body>. That is, <variable> is bound rather than assigned, and the region of the binding is the entire<body>.

Elk Incompatibilities with R4RS

Topic: *Scheme Interface, *Scheme AIDE Application

The following is a list of the areas where the Elk Extension Language does not conform to the R4RS. These are language features that could cause a Scheme program to run improperly under Elk, although it does run properly under an R4RS–conforming implementation.

- Quasi–quotation cannot be used to construct vectors.
- Rational and complex numbers are not implemented.
- All numbers are inexact.
- Prefixes for exact and inexact constants (#e and #i) are not implemented.
- exact–>inexact and inexact–>exact are not implemented.
- char–ready? is not implemented.
- transcript–on and transcript–off are not implemented.
- Elk is case sensitive.