Scheme Quick Reference

COSC 18—Fall 2003

This document is a quick reference guide to common features of the Scheme language. It is not intended to be a complete language reference, but it gives terse summaries of some of the most common syntax and built-in procedures of the language. Swindle extensions to standard Scheme are indicated by the notation [Swindle].

Notation

The following notational conventions are used in this document.

Text denoting literal Scheme code is written in `typewriter` font. Parameters are indicated by writing their names in *italics*, subject to the following type conventions:

f	A procedure (arbitrary)
g	A generic function
p	A predicate (function returning `#t` or `#f`)
n	A number
z	An exact integer
s	An identifier (symbol)
lst	A proper list
c	A pair ("cons box")
w, x, y	Arbitrary Scheme objects
v	A vector
expr ...	Zero or more occurrences of *expr*
$expr_1$ $expr_2$ ...	One or more expressions
[*item*]	Indicates an optional item or items.

Multiple variables of a given type will be distinguished by giving them subscripts, *e.g.,* w_1, w_2.

1 Syntax

1.1 Identifiers

An identifier is a sequence of one or more letters, digits, and *extended alphabetic characters*, beginning with a character that cannot begin a number (such as a digit, sign, or decimal point). In addition, `+`, `-`, and `...` by themselves are considered identifiers (even though their leading characters could otherwise begin numbers).

The extended alphabetic characters include the following:

```
! % & * + - . / : < = > ? @ $ ^ _
```

1.2 Types

Swindle provides objects that represent types. By convention, the names of identifiers bound to type objects are written in `<angle-brackets>`. The following table gives a subset of the class hierarchy provided by default in Swindle. Subclass relationships are indicated by indentation, multiple inheritance is shown with parenthetic comments at the end of the line.

```
<top>                                   (also called <obj>)
   <object>
      <class>
      <procedure-class>
      <primitive-class>
      <generic>                         (also under <function>)
      <method>                          (also under <function>)
   <function>
      <generic>
```

```
      <method>
      <procedure>                       (also under <builtin>)
   <builtin>
      <sequence>
         <improper-list>
            <pair>
            <list>
               <nonempty-list>          (also under <pair>)
               <null>
         <vector>
         <string>
      <symbol>
      <boolean>
      <char>
      <number>
         <exact>
         <inexact>
         <complex>
            <real>
               <rational>
                  <integer>
      <end-of-file>
      <port>
         <input-port>
         <output-port>
      <macro>
      <void>
      <box>
      <promise>
      <procedure>                       (also under <function>)
```

1.3 Booleans

The expression `#t` denotes *true*, and `#f` denotes *false*. In addition, any Scheme object other than `#f` is considered "true" with respect to conditional forms such as `if`, `and`, `or`, and `cond`.

1.4 Lists

The expression `'()` denotes the empty list, sometimes called *null*. A *proper list* is either the empty list, or a `<pair>` whose second element (cdr) is a proper list.

2 Special Forms

A special form is a Scheme expression with a special evaluation rule. Some of the most common special forms are given here, along with a brief synopsis of the evaluation rule for each.

`(define` *s expr*`)`
: Defines the identifier *s* to have the value returned by the evaluation of *expr*.

`(begin` $expr_1$ $expr_2$ ...`)`
: Evaluates expressions from left to right and returns the value of the last.

`(quote` *expr*`)`
: Returns *expr* unevaluated. Equivalently, this can be written `'`*expr*.

`(lambda (`s_1 ...`)` $expr_1$ $expr_2$ ...`)`
: Creates a new procedure whose formal parameters are $s_1, s_2, \ldots$ and whose body is the given list of expressions.

`(if` *test* $expr_1$ $expr_2$`)`
: Evaluates *test*, which is an arbitrary Scheme expression. If the resulting value is `#f`, $expr_2$ is evaluated; otherwise $expr_1$ is evaluated.

`(cond (`$test_1$ $elst_1$`) (`$test_2$ $elst_2$`) ... [(else` $elst_e$`)])`
: Evaluates each *test* expression in left-to-right order. If $test_i$ evaluates to a true value, then the corresponding sequence of expressions $elst_i$ is evaluated, and the value of the last expression is returned. The optional `else` clause is evaluated if none of the preceding tests yields a true value.

`(and` *expr* ...`)`
: Evaluates expressions from left to right, and returns the value of the first expression that returns `#f`. If no expression returns `#f`, the value of the last expression is returned, or `#t` if there are no expressions.

`(or` *expr* ...`)`
: Evaluates expressions from left to right, and returns the value of the first expression that returns a true value (i.e., not `#f`). If there are no expressions, or all expressions return `#f`, then the `or` returns `#f`.

`(let ((`s_1 $texpr_1$`) (`s_2 $texpr_2$`) ...)` $expr_1$ $expr_2$ ...`)`
: Evaluate $expr_1, expr_2, \ldots$ in an environment with s_1 bound to the value of $texpr_1$, s_2 bound to the value of $texpr_2$, etc.

`(let* ((`s_1 $texpr_1$`) (`s_2 $texpr_2$`) ...)` $expr_1$ $expr_2$ ...`)`
: As `let`, except that each *texpr* is evaluated in an environment where all previous bindings (those further to the left) are visible.

`(letrec ((`s_1 $texpr_1$`) (`s_2 $texpr_2$`) ...)` $expr_1$ $expr_2$ ...`)`
: As `let`, except that each *texpr* is evaluated in an environment where all the variables $s_1, s_2, \ldots$ are visible. This permits definition of recursive procedures.

`(set!` *s expr*`)`
: Change the value of the lexically closest binding of *s* to the value resulting from evaluation of *expr*. *Note:* You may not use this form in your code unless the instructions explicitly say so.

`(define-class` *s* `(`*supers*`) (`*slot type* [*init*]`) ...)`
: Define a new class with the specified superclasses (zero or more) and slots (zero or more), and bind it to the name *s* in the global environment. The *init* expressions are optional, but if present, will provide default values for their slot. [Swindle]

`(make` *class* :key_1 $expr_1$ ...`)`
: Create a new instance of *class* with the specified initial values for its slots. If *class* has a slot named *s*, the corresponding keyword is `:s`. Keyword arguments may be specified in any order. [Swindle]

`(define-generic` *s* `(`*params*`))`
: Creates a new generic function named *s* which takes as many parameters as are specified in *params*. [Swindle]

`(define-method` *g* `(`*params*`)` $expr_1$ $expr_2$ ...`)`
: Define a new method on the specified generic *g*. The types of the given *params* form the specializer for this method. The number of parameters must match the number given when the generic was defined. [Swindle]

3 Standard Procedures

All the forms given here are standard procedures; that is, they are evaluated according to the standard Scheme rule, and all their arguments are evaluated before the procedure is applied.

3.1 Equality Testing

`(eq? x y)`
: Returns `#t` if x and y should be regarded as the same object, `#f` otherwise. Two symbols are `eq?` if they are spelled the same (case does not matter), the empty list `'()` is `eq?` to itself, and any pair is `eq?` to itself.

`(equal? x y)`
: Returns `#t` if x and y "print the same", `#f` otherwise. In Swindle, `equal?` is a generic function, and you can add new methods to change the behaviour of the `equal?` relationship.

`(=` n_1 n_2 `...)`
: Returns `#t` if $n_1 = n_2 = \ldots$. Note that this works only for numbers.

`(zero? n)`
: Returns `#t` if $n = 0$. Note that this works only for numbers.

3.2 List Structure Operations

`(null? x)` Returns `#t` if x is the empty list `'()`, `#f` for any other Scheme object.

`(cons x y)` Creates a new pair (of type `<pair>`) whose car is x and whose cdr is y.

`(car c)` Returns the first element of c.

`(cdr c)` Returns the second element of c.

`(cadr c) (caddr c) (cddr c) (cdar c)`
: These are defined as compositions of `car` and `cdr`. For instance, `(cadr c)` is equivalent to `(car (cdr c))`, and `(cddr c)` is equivalent to `(cdr (cdr c))`. All combinations of up to 4 a's and d's are defined for you by Scheme.

`(set-car! c expr)` Set the car of c to the value of *expr*. Do not use unless explicitly permitted to do so.

`(set-cdr! c expr)` Set the cdr of c to the value of *expr*. Do not use unless explicitly permitted to do so.

`(list` $expr_1$ $expr_2$ `...)` Create a new list consisting of the values of the given expressions.

`(list-ref lst z)`
: Return the element at index z of the given list, where 0 is the index of the first element in a list. It is an error if $z < 0$ or z is past the end of the list.

`(length lst)` Return the number of elements in the given list.

`(append` lst_1 lst_2`)` Append the two lists together, and return the resulting list.

`(reverse lst)` Return a new list which has same elements as *lst*, but in the opposite order.

`(member x lst)` Return the first sublist of *lst* whose car is x, or `#f` if x does not occur in the list. Uses `equal?` for comparison.

`(memq x lst)` As `member`, but uses `eq?` for comparision.

3.3 Arithmetic and Numeric Operators

`(=` n_1 n_2 `...) (<` n_1 n_2 `...) (>` n_1 n_2 `...) (<=` n_1 n_2 `...) (>=` n_1 n_2 `...)`
: Tests whether a sequence of numerical values are, respectively, equal, strictly increasing, strictly decreasing, non-decreasing, or non-increasing.

`(+` n_1 n_2 `...)` Addition.

`(add1 n)` Adds 1 to any number [Swindle].

`(-` n_1 n_2 `...)` Subtraction (negation, with a single argument).

`(sub1` n`)` Subtracts 1 from any number [Swindle].

`(*` n_1 n_2 `...)` Multiplication.

`(/` n_1 n_2 `...)` Real or rational division (associates to the right).

`(quotient` n_1 n_2 `...)` Quotient from integer division

`(remainder` n_1 n_2 `...)` Remainder from integer division

`(modulo` n_1 n_2 `...)` Least non-negative residue of remainder.

`(abs` n`)` Absolute value.

`(min` n_1 n_2 `...)` `(max` n_1 n_2 `...)` Returns the smallest (largest) value among the given numeric values.

`(sqrt` n`)` Square root.

`(expt` n_1 n_2`)` Computes $n_1^{n_2}$. $0^x = 1$ if $x = 0$, 0 otherwise.

`(exp` n`)` Computes e^n.

`(log` n`)` Computes $\ln(n)$.

`(gcd` z `...)` Returns the greatest common divisor of its arguments (0 if none).

`(floor` n`)` Returns the largest integer not greater than n.

`(ceiling` n`)` Returns the smallest integer not less than n.

`(sin` n`)` `(cos` n`)` `(tan` n`)` Standard trigonometric functions. Angles in radians.

`(atan` n`)` Arc tangent (radians).

3.4 Higher-Order Procedures

`(apply` *f* *args*`)`
: Call the function *f* with the specified arguments, as if you had written `(`*f* arg_1 arg_2 `...)`

`(map` *f* lst_1 lst_2 `...)`
: Apply *f* to each element of all the input lists, in some order, and collect the return values into a list, which is returned from `map`.

`(filter` *p* *lst*`)`
: Returns a list consisting of the elements from *lst* for which *p* returns a true value. [Swindle]

`(foldr` *f* *x* *lst*`)`
: Uses *f* to combine all the elements of *lst* in right-to-left order, using *x* as the starting value. Returns the resulting value. The `foldl` function does the same, except it processes the list in left-to-right order. [Swindle]

3.5 Vector Operations

`(vector` $expr_1$ $expr_2$ `...)`
: Creates a new vector containing the values of the given expressions.

`(make-vector` *z* `[`*expr*`])`
: Creates a new vector of *z* elements, each initialized to the value of the given *expr*, if provided (otherwise, values are initially undefined).

`(vector-ref` *v* *z*`)`
: Returns the element at position *z* of vector *v*, where 0 is the index of the first element of the vector. It is an error if $z < 0$ or *z* is past the last element in the vector.

`(vector-set!` *v* *z* *expr*`)`
: Set the value at position *z* of vector *v* to the value of the *expr*. Vectors have fixed length, so it is an error to set an element outside the created range of the vector.

3.6 Miscellaneous

`(not` *expr*`)` Returns `#t` if *expr* is `#f`, otherwise returns `#f`.

`(pair?` *expr*`)` Returns `#t` if *expr* is a pair, otherwise returns `#f`.

`(procedure?` *expr*`)` Returns `#t` if *expr* is a procedure, otherwise returns `#f`.

`(number?` *expr*`)` `(complex?` *expr*`)` `(real?` *expr*`)` `(rational?` *expr*`)` `(integer?` *expr*`)`
Returns `#t` if *expr* is, respectively, a number, a complex number, a real number, a rational number, or an integer; `#f` otherwise. Each numeric type is a subset of the previous numeric types in the list.

`(display` *expr*`)` Writes a printable representation of *expr* to the default output port.

`(newline)` Writes a newline character to the default output port.

3.7 Stream Operations

These operations are not part of standard Scheme, but we will be working with them during the term, so they are provided here for your reference.

`(null-stream)` Returns an object representing the empty stream.

`(null-stream?` x`)` Returns `#t` if x is the empty stream, otherwise `#f`.

`(cons-stream` x *stream*`)`
Creates a new stream whose first element is x and the rest of which is a promise to evaluate *stream*.

`(stream-car` *stream*`)` Returns the first element of *stream*.

`(stream-cdr` *stream*`)` Returns the "rest" of *stream*, forcing evaluation of the promise.

`(map-stream` f *stream*`)`
Returns a new stream which is the result of applying function f to each element of *stream*.

`(map-streams` f $stream_1$ $stream_2$ ... $stream_n$`)`
Returns a new stream which is the result of applying f, a function which takes n arguments, to corresponding elements of $stream_1$, $stream_2$, *etc.*.

`(filter-stream` p *stream*`)`
Returns a new stream which contains all the elements of *stream* for which p returns a true value.

`(fold-stream` f x *stream*`)`
Uses f to accumulate all the elements of *stream*, using x as the starting element. Does not delay evaluation of the given *stream*.

`(make-stream` x_1 x_2 ... x_n`)`
Returns a new finite stream consisting of all the specified elements.

`(stream->list` *stream* z`)` Returns a list consisting of (up to) the first z elements of *stream*.

`(integers-from` z`)`
Returns a stream of the integers in ascending order, starting with z.