(Yet Another) Tiny Scheme

A Partial Revised(5) Report Scheme Implementation

Michael K K Montague

Table of Contents

1 Introduction

2 Expressions

2.1 Literal Expressions

`quote` *datum* [Syntax]

`(quote` *datum*`)` evaluates to *datum*. *datum* may be any external representation of a Scheme object. This notation is used to include literal constants in Scheme code.

```
(quote a)
    ⇒ a
(quote #(a b c))
    ⇒ #(a b c)
(quote (+ 1 2))
    ⇒ (+ 1 2)
```

`(quote` *datum*`)` may be abbreviated as `'`*datum*. The two notations are equivalent in all respects.

```
'a
    ⇒ a
'#(a b c)
    ⇒ #(a b c)
'()
    ⇒ ()
'(+ 1 2)
    ⇒ (+ 1 2)
'(quote a)
    ⇒ (quote a)
''a
    ⇒ (quote a)
```

Numerical constants, string constants, character constants, and boolean constants evaluate "to themselves"; they need not be quoted.

```
'"abc"
    ⇒ "abc"
"abc"
    ⇒ "abc"
'145932
    ⇒ 145932
145932
    ⇒ 145932
'#t
    ⇒ #t
#t
    ⇒ #t
```

2.2 Procedure Calls

`(`*operator operand1* `...)`

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The *operator* and *operand* expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments.

```
(+ 3 4)
    ⇒ 7
((if #f + *) 3 4)
    ⇒ 12
```

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables `+` and `*`. New procedures are created by evaluating lambda expressions. Procedure calls return one value.

2.3 Procedures

`lambda` *formals body* [Syntax]

Syntax: *formals* should be a formal arguments list as described below, and *body* should be a sequence of one or more expressions.

Semantics: 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 some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The result of the last expression in the body will be returned as the result of the procedure call.

```
(lambda (x) (+ x x))
    ⇒ #<procedure: #x123456>
((lambda (x) (+ x x)) 4)
    ⇒ 8
(define reverse-subtract
    (lambda (x y) (- y x)))
(reverse-subtract 7 10)
    ⇒ 3
(define add4
    (let ((x 4))
        (lambda (y) (+ x y))))
(add4 6)
    ⇒ 10
```

formals should have one of the following forms:

- (*variable1* ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.
- *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*.

- `(variable1 ... variable_n . variable_n+1)`: If a space-delimited period precedes the last variable, then the procedure takes n or more arguments, where n is the number of formal arguments before the period (there must be at least one). The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

It is an error for a *variable* to appear more than once in *formals*.

```
((lambda x x) 3 4 5 6)
    ⇒ (3 4 5 6)
((lambda (x y . z) z) 3 4 5 6)
    ⇒ (5 6)
```

2.4 Assignments

`set!` *variable expression* [Syntax]

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

2.5 Conditionals

`if` *test consequent alternate* [Syntax]

`if` *test consequent* [Syntax]

Syntax: *test*, *consequent*, and *alternate* may be arbitrary expressions.

Semantics: An `if` expression is evaluated as follows: first, *test* is evaluated. If it yields a true value, then *consequent* is evaluated and its value is returned. Otherwise *alternate* is evaluated and its value is returned. If *test* yields a false value and no *alternate* is specified, then the result of the expression is unspecified.

```
(if (> 3 2) 'yes 'no)
    ⇒ yes
(if (> 2 3) 'yes 'no)
    ⇒ no
(if (> 3 2) (- 3 2) (+ 3 2))
    ⇒ 1
```

`cond` *clause1 clause2 ...* [Syntax]

Syntax: Each *clause* should be of the form

```
(test expression1 ...)
```

where *test* is any expression. Alternatively, a *clause* may be of the form

```
(test => expression)
```

The last *clause* may be an "else clause," which has the form

```
(else expression1 expression2 ...)
```

Semantics: A `cond` expression is evaluated by evaluating the *test* expressions of successive *clause*s in order until one of them evaluates to a true value. When a *test* evaluates to a true value, then the remaining *expression*s in its *clause* are evaluated

in order, and the result of the last *expression* in the *clause* is returned as the result of the entire `cond` expression. If the selected *clause* contains only the *test* and no *expression*s, then the value of the *test* is returned as the result. If the selected *clause* uses the `=>` alternate form, then the *expression* is evaluated. Its value must be a procedure that accepts one argument; this procedure is then called on the value of the *test* and the value returned by this procedure is returned by the `cond` expression. If all *test*s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its *expression*s are evaluated, and the value of the last one is returned.

```
(cond
    ((> 3 2) 'greater)
    ((< 3 2) 'less))
    ⇒ greater
(cond
    ((> 3 3) 'greater)
    ((< 3 3) 'less)
    (else 'equal))
    ⇒ equal
(cond
    ((assv 'b '((a 1) (b 2))) => cadr)
    (else #f))
    ⇒ 2
```

`case` *key clause1 clause2 . . .* [Syntax]

Syntax: key may be any expression. Each *clause* should have the form

```
((datum1 ...) expression1 expression2 ...)
```

where each *datum* is an external representation of some object. All the *datum*s must be distinct. The last *clause* may be an "else clause," which has the form

```
(else expression1 expression2 ...)
```

Semantics: A `case` expression is evaluated as follows. *key* is evaluated and its result is compared against each *datum*. If the result of evaluating *key* is equivalent (in the sense of `eqv?`) to a *datum*, then the expressions in the corresponding *clause* are evaluated from left to right and the result of the last expression in the *clause* is returned as the result of the `case` expression. If the result of evaluating *key* is different from every *datum*, then if there is an else clause its expressions are evaluated and the result of the last is the result of the `case` expression; otherwise the result of the `case` expression is unspecified.

```
(case
    (* 2 3)
    ((2 3 5 7) 'prime)
    ((1 4 6 8 9) 'composite))
    ⇒ composite
(case
    (car '(c d))
    ((a e i o u) 'vowel)
    ((w y) 'semivowel)
```

```
        (else 'consonant))
        ⇒ consonant
```

`and` *test1 ...* [Syntax]

The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then `#t` is returned.

```
(and (= 2 2) (> 2 1))
    ⇒ #t
(and (= 2 2) (< 2 1))
    ⇒ #f
(and 1 2 'c '(f g))
    ⇒ (f g)
(and)
    ⇒ #t
```

`or` *test1 ...* [Syntax]

The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then `#f` is returned.

```
(or (= 2 2) (> 2 1))
    ⇒ #t
(or (= 2 2) (< 2 1))
    ⇒ #t
(or #f #f #f)
    ⇒ #f
(or (memq 'b '(a b c)) (/ 3 0))
    ⇒ (b c)
```

2.6 Binding Constructs

The two binding constructs `let` and `letrec` give Scheme a block structure, like Algol 60. The syntax of the two 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 of the variables become bound; while in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

`let` *bindings body* [Syntax]

Syntax: *bindings* should have the form

```
((variable1 init1) ...)
```

where each *init* is an expression, and *body* should be 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.

Semantics: The *init*s are evaluated in the current environment (in some unspecified order), 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* is returned. Each binding of a *variable* has *body* as its region.

```
(let
    ((x 2) (y 3))
    (* x y))
    ⇒ 6
(let
    ((x 2) (y 3))
    (let
        ((x 7)
        (z (+ x y)))
        (* z x)))
    ⇒ 35
```

`letrec` *bindings body* [Syntax]

Syntax: *bindings* should have the form

```
((variable1 init1) ...)
```

and *body* should be 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.

Semantics: The *variable*s are bound to fresh locations holding undefined values, the *init*s are evaluated in the resulting environment (in some unspecified order), each *variable* is assigned to the result of the corresponding *init*, the *body* is evaluated in the resulting environment, and the value of the last expression in *body* is returned. Each binding of a *variable* has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

```
(letrec
    ((even?
        (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
    (odd?
        (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
    (even? 88))
    ⇒ #t
```

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*. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the *init*s are lambda expressions and the restriction is satisfied automatically.

2.7 Sequencing

`begin` *expression1 expression2 . . .* [Syntax]

The *expression*s are evaluated sequentially from left to right, and the value of the last *expression* is returned. This expression type is used to sequence side effects such as input and output.

```
(define x 0)
(begin
    (set! x 5)
    (+ x 1))
    ⇒ 6
```

3 Program Structure

3.1 Programs

A Scheme program consists of a sequence of expressions and definitions. Expressions are described in chapter Chapter 2 [Expressions], page 2; definitions are the subject of the rest of the present chapter.

Definitions occurring at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top level environment or modify the value of existing top-level bindings. Expressions occurring at the top level of a program are interpreted imperatively; they are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

At the top level of a program `(begin` *`form1 ...`*`)` is equivalent to the sequence of expressions, definitions, and syntax definitions that form the body of the `begin`.

3.2 Definitions

Definitions are valid in some, but not all, contexts where expressions are allowed. They are valid only at the top level of a *program* and at the beginning of a *body*.

A definition should have one of the following forms:

- `(define` *`variable expression`*`)`
- `(define (`*`variable formals`*`)` *`body`*`)`

  formals should 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 (`*`variable . formal`*`)` *`body`*`)`

  formal should be a single variable. This form is equivalent to

  ```
  (define variable
      (lambda formal body))
  ```

3.2.1 Top Level Definitions

At the top level of a program, a definition

```
(define variable expression)
```

has essentially the same effect as the assignment expression

```
(set! variable expression)
```

if *variable* is bound. If *variable* is not bound, however, then the definition will bind *variable* to a new location before performing the assignment, whereas it would be an error to perform a `set!` on an unbound variable.

```
(define add3
    (lambda (x) (+ x 3)))
(add3 3)
```

```
    ⇒ 6
(define first car)
(first '(1 2))
    ⇒ 1
```

3.2.2 Internal Definitions

Definitions may occur at the beginning of a *body* (that is, the body of a `lambda`, `let`, or `letrec` expression or that of a definition of an appropriate form). Such definitions are known as *internal definitions* as opposed to the top level definitions described above. 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*. For example,

```
(let
    ((x 5))
    (define foo (lambda (y) (bar x y)))
    (define bar (lambda (a b) (+ (* a b) a)))
    (foo (+ x 3)))
    ⇒ 45
```

A *body* containing internal definitions can always be converted into a completely equivalent `letrec` expression. For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
    (letrec
        ((foo (lambda (y) (bar x y)))
        (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

Just as for the equivalent '`letrec`' expression, it must be possible to evaluate each *expression* of every internal definition in a *body* without assigning or referring to the value of any *variable* being defined.

Wherever an internal definition may occur `(begin` *`definition1 ...`*`)` is equivalent to the sequence of definitions that form the body of the `begin`.

4 Builtin Procedures

4.1 Equivalence Predicates

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, and `equal?` is the coarsest. `Eqv?` is slightly less discriminating than `eq?`.

`eqv?` *obj1 obj2* [Procedure]

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if *obj1* and *obj2* should normally be regarded as the same object.

The `eqv?` procedure returns `#t` if:

- *obj1* and *obj2* are both `#t` or both `#f`.
- *obj1* and *obj2* are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
    ⇒ #t
```

- *obj1* and *obj2* are both numbers and are numerically equal according to the `=` procedure (see Section 4.2 [Numbers], page 13).
- *obj1* and *obj2* are both characters and are the same character according to the `char=?` procedure (see Section 4.6 [Characters], page 21).
- both *obj1* and *obj2* are the empty list.
- *obj1* and *obj2* are pairs, vectors, or strings that denote the same locations.
- *obj1* and *obj2* are the same procedure.

The `eqv?` procedure returns `#f` if:

- *obj1* and *obj2* are of different types.
- one of *obj1* and *obj2* is `#t` but the other is `#f`.
- *obj1* and *obj2* are symbols but

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
    ⇒ #f
```

- *obj1* and *obj2* are numbers for which the `=` procedure returns `#f`.
- *obj1* and *obj2* are characters for which the `char=?` procedure returns `#f`.
- one of *obj1* and *obj2* is the empty list but the other is not.
- *obj1* and *obj2* are pairs, vectors, or strings that denote distinct locations.
- *obj1* and *obj2* are different procedures.

```
(eqv? 'a 'a)
    ⇒ #t
(eqv? 'a 'b)
    ⇒ #f
```

```
(eqv? 2 2)
    ⇒ #t
(eqv? '() '())
    ⇒ #t
(eqv? 100000000 100000000)
    ⇒ #t
(eqv? (cons 1 2) (cons 1 2))
    ⇒ #f
(eqv? (lambda () 1)
      (lambda () 2))
    ⇒ #f
(eqv? #f 'nil)
    ⇒ #f
(let ((p (lambda (x) x)))
    (eqv? p p))
    ⇒ #t
```

`eq?` *obj1 obj2* [Procedure]

`eq?` is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`.

`eq?` and `eqv?` are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, and non-empty strings and vectors.

```
(eq? 'a 'a)
    ⇒ #t
(eq? (list 'a) (list 'a))
    ⇒ #f
(eq? '() '())
    ⇒ #t
(eq? car car)
    ⇒ #t
(let ((x '(a)))
    (eq? x x))
    ⇒ #t
(let ((x '#()))
    (eq? x x))
    ⇒ #t
(let ((p (lambda (x) x)))
    (eq? p p))
    ⇒ #t
```

`equal?` *obj1 obj2* [Procedure]

`equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers and symbols. A rule of thumb is that objects are generally `equal?` if they print the same. `equal?` may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)
    ⇒ #t
```

```
(equal? '(a) '(a))
    ⇒ #t
(equal? '(a (b) c)
        '(a (b) c))
    ⇒ #t
(equal? "abc" "abc")
    ⇒ #t
(equal? 2 2)
    ⇒ #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))
    ⇒ #t
```

4.2 Numbers

The only numberical type supported by Tiny Scheme is exact integers.

`number?` *obj* [Procedure]
`integer?` *obj* [Procedure]

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is of the named type, and otherwise they return `#f`.

```
(number? 10)
    ⇒ #t
(number? 'a)
    ⇒ #f
(integer? 10)
    ⇒ #t
(integer? 'a)
    ⇒ #f
```

`=` *z1 z2 z3 . . .*, [Procedure]
`<` *x1 x2 x3 . . .*, [Procedure]
`>` *x1 x2 x3 . . .*, [Procedure]
`<=` *x1 x2 x3 . . .*, [Procedure]
`>=` *x1 x2 x3 . . .*, [Procedure]

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

These predicates are required to be transitive.

`zero?` *z* [Procedure]
`positive?` *x* [Procedure]
`negative?` *x* [Procedure]
`odd?` *n* [Procedure]
`even?` *n* [Procedure]

These numerical predicates test a number for a particular property, returning `#t` or `#f`.

`max` *x1 x2 ...*, [Procedure]
`min` *x1 x2 ...*, [Procedure]

These procedures return the maximum or minimum of their arguments.

`+` *z1 ...*, [Procedure]
`*` *z1 ...*, [Procedure]

These procedures return the sum or product of their arguments.

```
(+ 3 4)
    ⇒ 7
(+ 3)
    ⇒ 3
(+)
    ⇒ 0
(* 4)
    ⇒ 4
(*)
    ⇒ 1
```

`-` *z1 z2* [Procedure]
`-` *z* [Procedure]
`-` *z1 z2 ...*, [Procedure]
`/` *z1 z2* [Procedure]
`/` *z* [Procedure]
`/` *z1 z2 ...*, [Procedure]

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

```
(- 3 4)
    ⇒ -1
(- 3 4 5)
    ⇒ -6
(- 3)
    ⇒ -3
(/ 40 4 5)
    ⇒ 2
```

`abs` *x* [Procedure]

`abs` returns the absolute value of its argument.

```
(abs -7)
    ⇒ 7
```

`remainder` *n1 n2* [Procedure]

This procedure implements number-theoretic (integer) division. *n2* should be non-zero.

```
(remainder 13 4)
    ⇒ 1
(remainder -13 4)
```

```
    ⇒ -1
(remainder 13 -4)
    ⇒ 1
(remainder -13 -4)
    ⇒ -1
```

`number->string` *z* [Procedure]
`number->string` *z radix* [Procedure]

radix must be an exact integer, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. The procedure `number->string` takes a number and a radix and returns as a string an external representation of the given number in the given radix. The result returned by `number->string` never contains an explicit radix prefix.

`string->number` *string* [Procedure]

Returns a number of the maximally precise representation expressed by the given *string*. If *string* is not a syntactically valid notation for a number, then `string->number` returns `#f`.

4.3 Booleans

The standard boolean objects for true and false are written as `#t` and `#f`. What really matters, though, are the objects that the Scheme conditional expressions (`if`, `cond`, `and`, `or`) treat as true or false. The phrase "a true value" (or sometimes just "true") means any object treated as true by the conditional expressions, and the phrase "a false value" (or "false") means any object treated as false by the conditional expressions.

Of all the standard Scheme values, only `#f` counts as false in conditional expressions. Except for `#f`, all standard Scheme values, including `#t`, pairs, the empty list, symbols, numbers, strings, vectors, and procedures, count as true.

Programmers accustomed to other dialects of Lisp should be aware that Scheme distinguishes both `#f` and the empty list from the symbol `nil`.

Boolean constants evaluate to themselves, so they do not need to be quoted in programs.

```
#t
    ⇒ #t
#f
    ⇒ #f
'#f
    ⇒ #f
```

`not` *obj* [Procedure]

`not` returns `#t` if *obj* is false, and returns `#f` otherwise.

```
(not #t)
    ⇒ #f
(not 3)
    ⇒ #f
(not (list 3))
    ⇒ #f
(not #f)
```

```
    ⇒ #t
(not '())
    ⇒ #f
(not (list))
    ⇒ #f
(not 'nil)
    ⇒ #f
```

`boolean?` *obj* [Procedure]

`boolean?` returns `#t` if *obj* is either `#t` or `#f` and returns `#f` otherwise.

```
(boolean? #f)
    ⇒ #t
(boolean? 0)
    ⇒ #f
(boolean? '())
    ⇒ #f
```

4.4 Pairs and Lists

A *pair* (sometimes called a *dotted pair*) is a record structure with two fields called the car and cdr fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. The car and cdr fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose cdr is a list. The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type (it is not a pair); it has no elements and its length is zero. The empty list is written `()` .

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list.

`pair?` *obj* [Procedure]

`pair?` returns `#t` if *obj* is a pair, and otherwise returns `#f`.

```
(pair? '(a . b))
    ⇒ #t
(pair? '(a b c))
    ⇒ #t
(pair? '())
    ⇒ #f
(pair? '#(a b))
    ⇒ #f
```

`cons` *obj1 obj2* [Procedure]

Returns a newly allocated pair whose car is *obj1* and whose cdr is *obj2*. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

```
(cons 'a '())
    ⇒ (a)
(cons '(a) '(b c d))
    ⇒ ((a) b c d)
(cons "a" '(b c))
    ⇒ ("a" b c)
(cons 'a 3)
    ⇒ (a . 3)
(cons '(a b) 'c)
    ⇒ ((a b) . c)
```

`car` *pair* [Procedure]

Returns the contents of the car field of *pair*. Note that it is an error to take the car of the empty list.

```
(car '(a b c))
    ⇒ a
(car '((a) b c d))
    ⇒ (a)
(car '(1 . 2))
    ⇒ 1
```

`cdr` *pair* [Procedure]

Returns the contents of the cdr field of *pair*. Note that it is an error to take the cdr of the empty list.

```
(cdr '((a) b c d))
    ⇒ (b c d)
(cdr '(1 . 2))
    ⇒ 2
```

`set-car!` *pair obj* [Procedure]

Stores *obj* in the car field of *pair*. The value returned by `set-car!` is unspecified.

`set-cdr!` *pair obj* [Procedure]

Stores *obj* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

`null?` *obj* [Procedure]

Returns `#t` if *obj* is the empty list, otherwise returns `#f`.

`list?` *obj* [Procedure]

Returns `#t` if *obj* is a list, otherwise returns `#f`. By definition, all lists have finite length and are terminated by the empty list.

```
(list? '(a b c))
    ⇒ #t
(list? '())
    ⇒ #t
(list? '(a . b))
    ⇒ #f
(let ((x (list 'a)))
```

```
(set-cdr! x x)
(list? x))
⇒ #f
```

`list` *obj* ..., [Procedure]

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)
    ⇒ (a 7 c)
(list)
    ⇒ ()
```

`length` *list* [Procedure]

Returns the length of *list*.

```
(length '(a b c))
    ⇒ 3
(length '(a (b) (c d e)))
    ⇒ 3
(length '())
    ⇒ 0
```

`append` *list* ..., [Procedure]

Returns a list consisting of the elements of the first *list* followed by the elements of the other *list*s.

```
(append '(x) '(y))
    ⇒ (x y)
(append '(a) '(b c d))
    ⇒ (a b c d)
(append '(a (b)) '((c)))
    ⇒ (a (b) (c))
```

The resulting list is always newly allocated, except that it shares structure with the last *list* argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))
    ⇒ (a b c . d)
(append '() 'a)
    ⇒ a
```

`reverse` *list* [Procedure]

Returns a newly allocated list consisting of the elements of *list* in reverse order.

```
(reverse '(a b c))
    ⇒ (c b a)
(reverse '(a (b c) d (e (f))))
    ⇒ ((e (f)) d (b c) a)
```

`list-tail` *list k* [Procedure]

Returns the sublist of *list* obtained by omitting the first *k* elements. It is an error if *list* has fewer than *k* elements. `list-tail` could be defined by

```
(define list-tail
    (lambda (x k)
        (if (zero? k)
            x
            (list-tail (cdr x) (- k 1)))))
```

`list-ref` *list k* [Procedure]

Returns the *k*th element of *list*. (This is the same as the car of `(list-tail` *list k*`)`.) It is an error if *list* has fewer than *k* elements.

```
(list-ref '(a b c d) 2)
    ⇒ c
```

`memq` *obj list* [Procedure]
`memv` *obj list* [Procedure]
`member` *obj list* [Procedure]

These procedures return the first sublist of *list* whose car is *obj*, where the sublists of *list* are the non-empty lists returned by `(list-tail` *list k*`)` for *k* less than the length of *list*. If *obj* does not occur in *list*, then `#f` (not the empty list) is returned. `memq` uses `eq?` to compare *obj* with the elements of *list*, while `memv` uses `eqv?` and `member` uses `equal?`.

```
(memq 'a '(a b c))
    ⇒ (a b c)
(memq 'b '(a b c))
    ⇒ (b c)
(memq 'a '(b c d))
    ⇒ #f
(memq (list 'a) '(b (a) c))
    ⇒ #f
(member (list 'a)
        '(b (a) c))
    ⇒ ((a) c)
(memv 101 '(100 101 102))
    ⇒ (101 102)
```

`assq` *obj alist* [Procedure]
`assv` *obj alist* [Procedure]
`assoc` *obj alist* [Procedure]

alist (for "association list") must be a list of pairs. These procedures find the first pair in *alist* whose car field is *obj*, and returns that pair. If no pair in *alist* has *obj* as its car, then `#f` (not the empty list) is returned. `assq` uses `eq?` to compare *obj* with the car fields of the pairs in *alist*, while `assv` uses `eqv?` and `assoc` uses `equal?`.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)
    ⇒ (a 1)
(assq 'b e)
    ⇒ (b 2)
(assq 'd e)
```

```
    ⇒ #f
(assq (list 'a) '(((a)) ((b)) ((c))))
    ⇒ #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
    ⇒ ((a))
(assv 5 '((2 3) (5 7) (11 13)))
    ⇒ (5 7)
```

4.5 Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eqv?`) if and only if their names are spelled the same way.

It is guaranteed that any symbol that has been returned as part of a literal expression, or read using the `read` procedure, and subsequently written out using the `write` procedure, will read back in as the identical symbol (in the sense of `eqv?`). The `string->symbol` procedure, however, can create symbols for which this write/read invariance may not hold because their names contain special characters or letters in the non-standard case.

`symbol?` *obj* [Procedure]

Returns `#t` if *obj* is a symbol, otherwise returns `#f`.

```
(symbol? 'foo)
    ⇒ #t
(symbol? (car '(a b)))
    ⇒ #t
(symbol? "bar")
    ⇒ #f
(symbol? 'nil)
    ⇒ #t
(symbol? '())
    ⇒ #f
(symbol? #f)
    ⇒ #f
```

`symbol->string` *symbol* [Procedure]

Returns the name of *symbol* as a string. If the symbol was part of an object returned as the value of a literal expression or by a call to the `read` procedure, and its name contains alphabetic characters, then the string returned will contain characters in lower case. If the symbol was returned by `string->symbol`, the case of characters in the string returned will be the same as the case in the string that was passed to `string->symbol`. It is an error to apply mutation procedures like `string-set!` to strings returned by this procedure.

```
(symbol->string 'flying-fish)
    ⇒ "flying-fish"
(symbol->string 'Martin)
    ⇒ "martin"
(symbol->string (string->symbol "Malvina"))
    ⇒ "Malvina"
```

`string->symbol` *string* [Procedure]

Returns the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves. See `symbol->string`.

```
(eq? 'mISSISSIppi 'mississippi)
    ⇒ #t
(string->symbol "mISSISSIppi")
    ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt"))
    ⇒ #f
(eq? 'JollyWog (string->symbol (symbol->string 'JollyWog)))
    ⇒ #t
(string=? "K. Harper, M.D." (symbol->string
    (string->symbol "K. Harper, M.D.")))
    ⇒ #t
```

4.6 Characters

Characters are objects that represent printed characters such as letters and digits. Characters are written using the notation #*character*. Case is significant. If *character* in #*character* is alphabetic, then the character following *character* must be a delimiter character such as a space or parenthesis. Characters written in the #\ notation are self-evaluating. That is, they do not have to be quoted in programs.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have `-ci` (for "case insensitive") embedded in their names.

`char?` *obj* [Procedure]

Returns `#t` if *obj* is a character, otherwise returns `#f`.

`char=?` *char1 char2* [Procedure]
`char<?` *char1 char2* [Procedure]
`char>?` *char1 char2* [Procedure]
`char<=?` *char1 char2* [Procedure]
`char>=?` *char1 char2* [Procedure]

These procedures impose a total ordering on the set of characters.

`char-ci=?` *char1 char2* [Procedure]
`char-ci<?` *char1 char2* [Procedure]
`char-ci>?` *char1 char2* [Procedure]
`char-ci<=?` *char1 char2* [Procedure]
`char-ci>=?` *char1 char2* [Procedure]

These procedures are similar to `char=?` et cetera, but they treat upper case and lower case letters as the same. For example, `(char-ci=? #\A #\a)` returns `#t`.

`char-alphabetic?` *char* [Procedure]
`char-numeric?` *char* [Procedure]

`char-whitespace?` *char* [Procedure]
: These procedures return `#t` if their arguments are alphabetic, numeric, or whitespace characters, respectively, otherwise they return `#f`.

`char->integer` *char* [Procedure]
`integer->char` *n* [Procedure]
: Given a character, `char->integer` returns an exact integer representation of the character. Given an exact integer that is the image of a character under `char->integer`, `integer->char` returns that character. These procedures implement order-preserving isomorphisms between the set of characters under the `char<=?` ordering and some subset of the integers under the `<=` ordering.

`char-upcase` *char* [Procedure]
`char-downcase` *char* [Procedure]
: These procedures return a character *char2* such that `(char-ci=? char char2)`. In addition, if *char* is alphabetic, then the result of `char-upcase` is upper case and the result of `char-downcase` is lower case.

4.7 Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within doublequotes (`"`). A doublequote can be written inside a string only by escaping it with a backslash (\). A backslash can be written inside a string only by escaping it with another backslash. A string constant may continue from one line to the next.

The *length* of a string is the number of characters that it contains. This number is an exact, non-negative integer that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

In phrases such as "the characters of *string* beginning with index *start* and ending with index *end*," it is understood that the index *start* is inclusive and the index *end* is exclusive. Thus if *start* and *end* are the same index, a null substring is referred to, and if *start* is zero and *end* is the length of *string*, then the entire string is referred to.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The versions that ignore case have `-ci` (for "case insensitive") embedded in their names.

`string?` *obj* [Procedure]
: Returns `#t` if *obj* is a string, otherwise returns `#f`.

`make-string` *k* [Procedure]
`make-string` *k char* [Procedure]
: `make-string` returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

`string` *char ...*, [Procedure]
: Returns a newly allocated string composed of the arguments.

`string-length` *string* [Procedure]
Returns the number of characters in the given *string*.

`string-ref` *string k* [Procedure]
k must be a valid index of *string*. `string-ref` returns character *k* of *string* using zero-origin indexing.

`string-set!` *string k char* [Procedure]
k must be a valid index of *string*. `string-set!` stores *char* in element *k* of *string* and returns an unspecified value.

`string=?` *string1 string2* [Procedure]
`string-ci=?` *string1 string2* [Procedure]
Returns `#t` if the two strings are the same length and contain the same characters in the same positions, otherwise returns `#f`. `string-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

`string<?` *string1 string2* [Procedure]
`string>?` *string1 string2* [Procedure]
`string<=?` *string1 string2* [Procedure]
`string>=?` *string1 string2* [Procedure]
`string-ci<?` *string1 string2* [Procedure]
`string-ci>?` *string1 string2* [Procedure]
`string-ci<=?` *string1 string2* [Procedure]
`string-ci>=?` *string1 string2* [Procedure]
These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

`substring` *string start end* [Procedure]
string must be a string, and *start* and *end* must be exact integers satisfying
0 <= *start* <= *end* <= (`string-length` *string*)
`substring` returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

`string-append` *string* ..., [Procedure]
Returns a newly allocated string whose characters form the concatenation of the given strings.

`string->list` *string* [Procedure]
`list->string` *list* [Procedure]
`string->list` returns a newly allocated list of the characters that make up the given string. `list->string` returns a newly allocated string formed from the characters in the list *list*, which must be a list of characters. `string->list` and `list->string` are inverses so far as `equal?` is concerned.

`string-copy` *string* [Procedure]
Returns a newly allocated copy of the given *string*.

`string-fill!` *string char* [Procedure]
Stores *char* in every element of the given *string* and returns an unspecified value.

4.8 Vectors

Vectors are heterogenous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indexes* of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation `#(`*obj* `...,)`. For example, a vector of length 3 containing the number zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be written as following:

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector, not an expression evaluating to a vector. Like list constants, vector constants must be quoted:

```
'#(0 (2 2 2 2) "Anna")
    ⇒ #(0 (2 2 2 2) "Anna")
```

`vector?` *obj* [Procedure]
Returns `#t` if *obj* is a vector, otherwise returns `#f`.

`make-vector` *k* [Procedure]
`make-vector` *k fill* [Procedure]
Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

`vector` *obj ...*, [Procedure]
Returns a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)
    ⇒ #(a b c)
```

`vector-length` *vector* [Procedure]
Returns the number of elements in *vector* as an exact integer.

`vector-ref` *vector k* [Procedure]
k must be a valid index of *vector*. `vector-ref` returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)
    ⇒ 8
```

`vector-set!` *vector k obj* [Procedure]
k must be a valid index of *vector*. `vector-set!` stores *obj* in element *k* of *vector*. The value returned by `vector-set!` is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
    (vector-set! vec 1 '("Sue" "Sue"))
    vec)
    ⇒ #(0 ("Sue" "Sue") "Anna")
```

`vector->list` *vector* [Procedure]
`list->vector` *list* [Procedure]
`vector->list` returns a newly allocated list of the objects contained in the elements of *vector*. `list->vector` returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '#(dah dah didah))
    ⇒ (dah dah didah)
(list->vector '(dididit dah))
    ⇒ #(dididit dah)
```

`vector-fill!` *vector fill* [Procedure]
Stores *fill* in every element of *vector*. The value returned by `vector-fill!` is unspecified.

4.9 Control Features

`procedure?` *obj* [Procedure]
Returns `#t` if *obj* is a procedure, otherwise returns `#f`.

```
(procedure? car)
    ⇒ #t
(procedure? 'car)
    ⇒ #f
(procedure? (lambda (x) (* x x)))
    ⇒ #t
(procedure? '(lambda (x) (* x x)))
    ⇒ #f
```

`apply` *proc arg1 ... args* [Procedure]
proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list `(append (list` *arg1 ...*`,) args)` as the actual arguments.

```
(apply + (list 3 4))
    ⇒ 7
(define compose
    (lambda (f g)
        (lambda args
            (f (apply g args)))))
((compose sqrt *) 12 75)
    ⇒ 30
```

`map` *proc list1 list2 . . .*, [Procedure]

The *list*s must be lists, and *proc* must be a procedure taking as many arguments as there are *list*s and returning a single value. If more than one *list* is given, then they must all be the same length. `map` applies *proc* element-wise to the elements of the *list*s and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *list*s is unspecified.

```
(map cadr '((a b) (d e) (g h)))
    ⇒ (b e h)
(map (lambda (n) (* n n)) '(1 2 3 4 5))
    ⇒ (1 4 9 16 25)
(map + '(1 2 3) '(4 5 6))
    ⇒ (5 7 9)
(let ((count 0))
    (map (lambda (ignored)
        (set! count (+ count 1))
        count)
        '(a b)))
    ⇒ (1 2)
```

`for-each` *proc list1 list2 . . .*, [Procedure]

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls *proc* for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call *proc* on the elements of the *list*s in order from the first element(s) to the last, and the value returned by `for-each` is unspecified.

```
(let ((v (make-vector 5)))
    (for-each (lambda (i)
        (vector-set! v i (* i i)))
        '(0 1 2 3 4))
        v)
    ⇒ #(0 1 4 9 16)
```

4.10 Input and Output

`write` *obj* [Procedure]

Writes a written representation of *obj* to the current output. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the `#\` notation. `write` returns an unspecified value.

`display` *obj* [Procedure]

Writes a representation of *obj* to the current output. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by `write-char` instead of by `write`. `display` returns an unspecified value.

`write` is intended for producing machine-readable output and `display` is for producing human-readable output.

`newline` [Procedure]
Writes an end of line to the current output. Returns an unspecified value.

`write-char` *char* [Procedure]
Writes the character *char* (not an external representation of the character) to the current output and returns an unspecified value.

5 Implementation

5.1 Limitations

Tiny Scheme is a partial implementation of *Revised(5) Report on the Algorithmic Language Scheme*. In particular the following features are not supported.

- Continuations can not be captured. The procedures `call-with-current-continuation`, `call-with-values`, and `dynamic-wind` are not available.
- Only a single value can be returned. The procedure `values` is not available.
- Macros are not supported. The syntax `let-syntax`, `letrec-syntax`, `syntax-rules`, and `define-syntax` are not available.
- The procedure `eval` is not available.
- Only fixnums are supported as a numerical type. Only a subset of the numerical procedures are available.
- Ports are not supported. Only a subset of the input and output procedures are available.

5.2 Notes

Notes about the implementation of Tiny Scheme.

- *All* objects except fixnums and characters are heap allocated.
- The garbage collector uses mark-and-sweep to determine what objects to reclaim.
- There is no compiler: syntax errors will not be detected until the code is executed.

Index