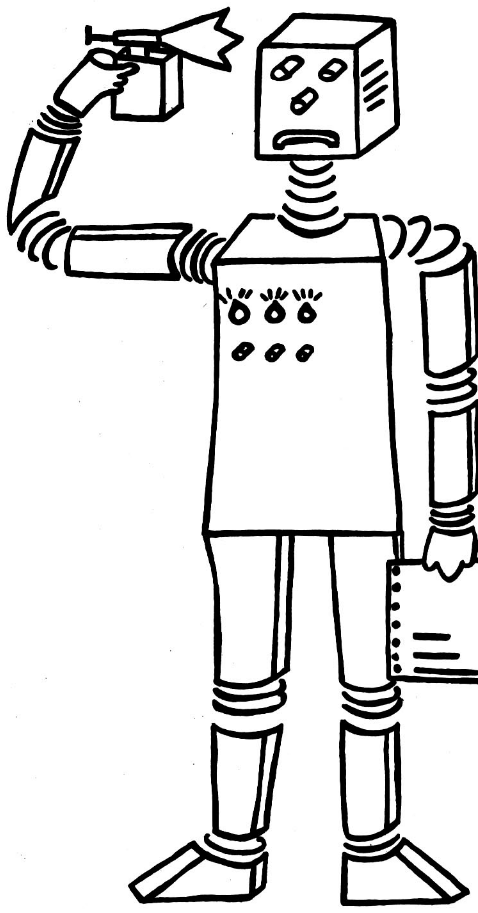
SEKI-Working Paper

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Manual of FranzScheme

Norbert Eisinger

SEKI Working Paper SWP-87-11

Manual of Franz Scheme

Norbert Eisinger
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, D-6750 Kaiserslautern, W.-Germany

Manual of FranzScheme

Abstract: Scheme is a Lisp dialect designed to have an exceptionally clear and simple semantics. FranzScheme is a Scheme interpreter written for educational purposes in Franz Lisp. The FranzScheme language is exactly the one used in the Abelson, Sussman & Sussman textbook. FranzScheme is deliberately kept small and easy to grasp; its intention is not to provide a full environment for professional programming, but a simple tool for teaching Scheme.

Brief History of Scheme

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and very few different methods of expression formation.

The first description of Scheme was written in 1975. A Revised Report appeared in 1978, which described the evolution of the language as its MIT implementation was upgraded to support an innovative compiler. Three distinct projects began in 1981 and 1982 to use variants of Scheme for courses at MIT, Yale, and Indiana University. An introductory computer science textbook using Scheme was published in 1984 [Harold Abelson and Gerald Jay Sussman with Julie Sussman: *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge MA, USA, 1985].

As might be expected of a language used primarily for education and research, Scheme has always evolved rapidly. This was no problem when Scheme was used only within MIT, but as Scheme became more widespread, local subdialects began to diverge until students and researchers occasionally found it difficult to understand code written at other sites. Fifteen representatives of the major implementations of Scheme therefore met in October 1984 to work toward a better and more widely accepted standard for Scheme. The outcome was a Revised Revised Report on Scheme and later a Revised^3 Report on Scheme, reporting their unanimous recommendations.

FranzScheme is a Scheme interpreter written in Franz Lisp for educational purposes by Dan Friedman at Indiana University, later modified by Manfred Meyer at the University of Kaiserslautern. The language it implements is the one used in the Abelson, Sussman & Sussman textbook, hence it does not comply with later reports. For instance, the object `nil` does have ambiguous meanings, as a symbol, as the truth value `false`, and as the empty list, although meanwhile this has been sorted out in the Scheme reports. Also, the language is deliberately kept small and easy to grasp, its intention is not to provide a full environment for professional programming.

This Manual of FranzScheme was written by Norbert Eisinger, borrowing most of the details of the presentation from The Revised Revised Report on Scheme.

Semantics

Scheme is a statically scoped programming language. Each use of an identifier is associated with a lexically apparent binding of that identifier. In this respect Scheme is like Algol 60, Pascal, and C, but unlike dynamically scoped languages such as APL and traditional Lisp.

Scheme has latent as opposed to manifest types. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are APL, Snobol, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, Pascal, and C.

All objects created in the course of a Scheme computation, including all procedures and variables, have unlimited extent. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include APL and other Lisp dialects.

Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative process in constant space, even if the iterative process is described by a syntactically recursive procedure. Thus with a tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar.

Scheme procedures are objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp and ML.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. ML, C, and APL are three other languages that always pass arguments by value. Lazy ML passes arguments by name, so that an argument expression is evaluated only if its value is needed by the procedure.

Top Level Interpreter Commands

(exit)

Terminates the FranzScheme session. The parantheses may be omitted for this command.

(vi ***filename*****)**
(vi)

The argument must be a string representing the name of a file. Calls the editor vi for the given file. After terminating vi, the control returns to FranzScheme, and the modified file can now be loaded. From the second call on the argument may be omitted, then the filename from the previous call is used. In this case the parantheses may be omitted for the command.

```
        (vi "mydirectory/myfile")
```

(vil ***filename*****)**
(vil)

Like vi, except that the edited file is automatically loaded upon return to FranzScheme.

```
        (vil "mydirectory/myfile")
```

(trace ***name1 name2*** **...)**

The arguments must be procedure names. Causes the procedures to print information about their arguments and results whenever they are called.

(untrace ***name1 name2*** **...)**
(untrace)

The arguments must be procedure names. Turns off the trace modus for the given procedures. If no argument is given, the trace modus is turned off for all procedures. In this case the parantheses may be omitted for the command.

(break ***name1 name2*** **...)**

The arguments must be procedure names. Causes the interpreter to interrupt immediately after any of the procedures is called and to enter a break level. On the break level the user can investigate the current environment. There are some special commands available on the break level, among them:

(help)	prints information about the break commands
(scheme)	aborts the evaluation and returns to the interpreter top level
(continue)	continues the interrupted procedure.

(unbreak ***name1 name2*** **...)**
(unbreak)

The arguments must be procedure names. Turns off the break modus for the given procedures. If no argument is given, the break modus is turned off for all procedures. In this case the parantheses may be omitted for the command.

Special Forms

variable

An expression consisting of a symbol that is not the keyword of a special form indicates a variable reference. Evaluates to the value stored in the location to which *variable* is bound in the current environment. It is an error to reference an unbound *variable.*

(operator operand1 ...)

A list whose first element is not the keyword of a special form indicates a procedure call. The *operator* and the *operand* expressions are evaluated and the resulting procedure is called with the resulting arguments. Evaluates to the result of this procedure call.

```
(+ 3 4)                     -->     7
((if t   + *) 3 (+ 2 2))    -->     7
((if nil + *) 3 (+ 2 2))    -->     12
```

(**quote** *expression*)
'*expression*

Evaluates to *expression.* This notation is used to include literal constants in Scheme code. The notation '*expression* is an abbreviation of (quote *expression*). The two notations are equivalent in all respects.

```
(quote (+ 1 2))             -->     (+ 1 2)
'(+ 1 2)                    -->     (+ 1 2)
(quote a)                   -->     a
'a                          -->     a
(quote (quote a))           -->     'a
''a                         -->     'a
```

(**if** *condition consequent alternative*)

First evaluates *condition.* If it yields the value **t** (or another value different from **nil**), then *consequent* is evaluated and its value is returned. If it yields the value **nil**, *alternative* is evaluated and its value is returned.

```
(if (< 1 2) 'yes 'no)       -->     yes
(if (< 2 1) 'yes 'no)       -->     no
(if (< 1 2) (+ 3 4) 'no)    -->     7
```

(**cond** *clause1 clause2 ...*)

Each *clause* must be a list of one or more expressions. The first expression in each clause is a boolean expression called the "guard" for the *clause.* The guards are evaluated in order until one of them evaluates to the value **t** (or another value different from **nil**). If this happens for a guard, then the remaining expressions in its *clause* are evaluated in order, and the result of the last expression in the selected *clause* is returned as the result of the entire conditional expression. If the selected *clause* contains only the guard, then the value of the guard is returned as the result.

```
(cond ((> 1 2) 'greater)
      ((< 1 2) 'less))      -->     less
```

If all guards evaluate to the value **nil**, then the result of the entire conditional expression is the value **nil**. In order to prevent this, the keyword **else** may be used as the guard of the last *clause* to make sure that the last *clause* is always selected if none of the others is. The *clause* with guard **else** must contain at least one more expression.

```
        (cond  ((> 1 1) 'greater)
               ((< 1 1) 'less)
               (else    'equal))  -->        equal
```

(and *expression1 ...*)

Evaluates the *expressions* from left to right, returning **nil** as soon as one evaluates to **nil**. Any remaining *expressions* are not evaluated. If all the *expressions* evaluate to the value **t** (or another value different from **nil**), then the value **t** is returned.

```
        (and (= 2 2) (< 1 2))       -->        t
        (and (= 2 2) (< 2 1))       -->        nil
        (and (= 2 2) 'a (+ 3 4))    -->        t
```

(or *expression1 ...*)

Evaluates the *expressions* from left to right, returning **t** as soon as one evaluates to the value **t** (or another value different from **nil**). Any remaining *expressions* are not evaluated. If all the *expressions* evaluate to the value **nil**, then the value **nil** is returned.

```
        (or (= 2 2) (< 1 2))        -->        t
        (or (= 2 2) (< 2 1))        -->        t
        (or (= 2 3) nil (< 1 1))    -->        nil
        (or (= 2 3) (+ 3 4) nil)    -->        t
```

(sequence *expression1 expression2 ...*)

Evaluates the *expressions* sequentially from left to right and returns the value of the last *expression*. Used to sequence side effects such as input and output.

```
        (define x 1)
        (sequence (set! x 5)                   ; changes value of x
                  (1+ x))           -->        6
        (sequence
          (princ "4+1 equals ")
          (princ (1+ 4))                       ; prints 4+1 equals 5
          'done)                    -->        done
```

Some special forms, such as **lambda** and **let**, implicitly treat their bodies as **sequence** expressions.

(let ((*var1 form1*) ...) *expr1 expr2 ...*)

Evaluates the *forms* in the current environment (in some unspecified order), then creates an extended environment in which the *vars* are bound to new locations containing the corresponding results, and then evaluates the *exprs* in the extended environment from left to right, returning the value of the last one. Each binding of a *var* has *expr1 expr2 ...* as its scope.

```
        (let ((x 1) (y 2))
             (+ x y))               -->        3
```

```
(let ((x 1) (y 2))
   (let ((foo (lambda(z) (+ x y z)))
         (x 11))
        (foo 3)))            -->       6
```

The nesting of **let** and **letrec** gives Scheme a block structure. The difference between **let** and **letrec** is that in a **let** the *forms* are not within the scope of the *vars* being bound, whereas in a **letrec** they are.

(letrec ((*var1 form1*) ...) *expr1 expr2* ...)

Creates an extended environment in which the *vars* are bound to new locations containing unspecified values, then evaluates the *forms* in the extended environment (in some unspecified order), then stores the result of each *form* in the location to which the corresponding *var* is bound, and then evaluates the *exprs* in the extended environment from left to right, returning the value of the last one. Each binding of a *var* has the entire **letrec** expression as its scope.

```
(let ((x 1) (y 2))
   (letrec
        ((foo (lambda(z) (+ x y z)))
         (x 11))
        (foo 3)))            -->       16
```

One restriction of **letrec** is very important: it must be possible to evaluate each *form* without referring to the value of a *var*. In the normal use of **letrec** all *forms* are lambda expressions and the restriction is satisfied automatically.

(set! *var expression*)

Expects that *var* is a bound variable. Does not evaluate *var*, but evaluates *expression* and stores the result in the location to which *var* is bound. The result of the entire **set!** expression is not specified, it is called only for its side effect.

```
(let ((x 1))
      (set! x (+ x 7))
      x)                     -->       8
```

(lambda (*var1* ...) *expression*)

Each *var* must be a symbol. The lambda expression evaluates to a procedure with formal argument list (*var1* ...) and procedure body *expression*. 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 identifiers in the formal argument list to new locations, the corresponding actual argument values will be stored in those locations, and *expression* will then be evaluated in the extended environment. The result of this evaluation will be returned as the result of the procedure call.

```
(lambda(x) (+ x 2))          -->      <PROCEDURE>
((lambda(x) (+ x 2)) 4)      -->       6
(let ((x 1) (y 2))
  ((lambda(x) (+ x y)) 4))-->          6
```

(lambda (*var1* ...) *expression1 expression2* ...)

may be used as an abbreviation of **(lambda** (*var1* ...) **(sequence** *expression1 expression2* ...))

(define *var expression*)

Is usually called at the top level, so that it is not nested within any other expression. Does not evaluate *var*, but evaluates *expression*. If *var* is a bound variable, the result of the evaluation is stored in the corresponding location, with the same effect as the assignment (**set!** *var expression*). If *var* is an unbound variable, then it is bound to a new location containing the result of the evaluation, which means that the current environment is extended. The value of the entire **define** expression is *var*, the name being defined.

```
        (define plus +)            -->        plus
        (define one 1)             -->        one
        (plus one one)             -->        2
```

(define (*var var1* ...) *expression1 expression2* ...)

may be used as an abbreviation of (**define** *var* (**lambda** (*var1* ...) *expression1 expression2* ...).

When (**define** *var expression*) is called at the beginning of the body of a **lambda**, **let**, **letrec**, or **define** expression, it is called a local definition as opposed to the global definition described above. The scope of a local definition is the body of the **lambda**, **let**, **letrec**, or **define** expression (including in the case of **letrec** the *forms*).

```
        (let ((x 1) (y 2))
          (define (foo z) (+ x y z))
          (define (foo+1 a) (1+ (foo a)))
          (foo+1 3))               -->        7
```

Local definitions can always be reexpressed by equivalent **letrec** expressions. For example, the expression above is equivalent to

```
        (let ((x 1) (y 2))
          (letrec
            ((foo (lambda(z) (+ x y z)))
             (foo+1 (lambda(a) (1+ (foo a)))))
            (foo+1 3)))            -->        7
```

Booleans and Equivalence Predicates

nil

The boolean value for falsity. The **nil** object is self-evaluating, it does not need to be quoted in programs.

```
        nil                         -->       nil
        'nil                        -->       nil
```

t

The boolean value for truth. The **t** object is self-evaluating, it does not need to be quoted in programs.

```
        t                           -->       t
        't                          -->       t
```

(not *obj***)**

Returns **t** if *obj* is **nil** and returns **nil** otherwise.

(eq? *obj1 obj2***)**

Returns **t** if *obj1* is identical in all respects to *obj2*, otherwise returns **nil**. If there is any way at all that a user can distinguish *obj1* and *obj2*, then eq? will return **nil**.

```
        (eq? 'a 'a)                 -->       t
        (eq? 'a 'A)                 -->       nil
        (eq? 'a 'b)                 -->       nil
        (eq? '(a) '(a))             -->       nil
        (eq? "a" "a")               -->       nil
        (eq? 1024 1024)             -->       unspecified
        (eq?    (cons 'a 'b)
                (cons 'a 'b))       -->       nil
```

(= *obj1 obj2***)**

Returns **t** if *obj1* and *obj2* are identical objects or if they are equivalent numbers or lists or strings. Two objects are generally considered equivalent, if they print the same. If its arguments are circular data structures, = may fail to terminate.

```
        (= 'a 'a)                   -->       t
        (= 'a 'A)                   -->       nil
        (= 'a 'b)                   -->       nil
        (= '(a) '(a))               -->       t
        (= "a" "a")                 -->       t
        (= 1024 1024)               -->       t
        (=      (cons 'a 'b)
                (cons 'a 'b))       -->       t
```

Pairs and Lists

nil

The empty list, which has no elements and length zero. The **nil** object is self-evaluating, it does not need to be quoted in programs. The empty list is not a pair.

Larger lists are built out of pairs (sometimes called dotted pairs). A pair is a structure with two fields called car and cdr. The notation for a pair is (*c1* . *c2*) where *c1* is the value of the car field and *c2* is the value of the cdr field. If the value of the cdr field is nil, the pair is written as (*c1*). If the value of the cdr field is itself a pair (*c21* . *c22*), the pair is written as (*c1 c21* . *c22*) and so on.

```
'(a . b)                    -->      (a . b)
'(a . nil)                  -->      (a)
'(a . (b . nil))            -->      (a b)
'(a . (b . (c . nil)))      -->      (a b c)
'(a . (b . c))              -->      (a b . c)
```

(cons *obj1 obj2***)**

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 **eq?** from every existing object.

```
(cons 'a 'b)                -->      (a . b)
(cons 'a nil)               -->      (a)
(cons 'a '(b))              -->      (a b)
(cons 'a '(b c))            -->      (a b c)
(cons 'a '(b . c))          -->      (a b . c)
```

(car *pair***)**

Returns the car field of *pair*, which must be a pair.

```
(car '(a . b))              -->      a
(car '(a))                  -->      a
(car '(a b))                -->      a
```

(cdr *pair***)**

Returns the car field of *pair*, which must be a pair.

```
(cdr '(a . b))              -->      b
(cdr '(a))                  -->      nil
(cdr '(a b))                -->      (b)
```

(set-car! *pair obj***)**

Stores *obj* in the car field of *pair*, which must be a pair. The result is *pair* after the manipulation. This procedure can be very confusing if used indiscriminately.

(set-cdr! *pair obj***)**

Stores *obj* in the cdr field of *pair*, which must be a pair. The result is *pair* after the manipulation. This procedure can be very confusing if used indiscriminately.

(null? ***obj*****)**

Returns **t** if *obj* is the empty list, otherwise returns **nil**.

(atom? ***obj*****)**

Returns **t** if *obj* is not a pair, otherwise returns **nil**.

```
(atom? nil)                -->      t
(atom? 'a)                 -->      t
(atom? "a")                -->      t
(atom? 2)                  -->      t
(atom? (cons 'a 'b))       -->      nil
```

(list ***obj1 ...*****)**

Returns a list of its arguments, that is a pair whose car is *obj1* and whose cdr is a list of the remaining arguments.

```
(list)                     -->      nil
(list 'a)                  -->      (a)
(list 'a 'b)               -->      (a b)
(list 'a 'b 'c)            -->      (a b c)
(list 'a (+ 2 3) 'c)       -->      (a 5 c)
```

Symbols

Symbols are objects whose usefulness rests entirely on the fact that two symbols are identical in the sense of **eq?** if and only if their names are spelled the same way. The name of a symbol must be different from a number or a string and may start with one of the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
               ! § % & * / : < = > ?
```

Subsequent characters may be drawn from the same set or from

```
0 1 2 3 4 5 6 7 8 9
+ - . _
```

In addition, there are some symbols that violate the rule for the first letter, in particular:

```
+ - 1+ -1+
```

The following characters must not be used in a symbol name:

```
) ( ] [ } { " ; blank
```

However, any sequence of characters enclosed between two vertical bars, e.g. | (.) |, is a symbol.

(symbol? *obj***)**

Returns **t** if *obj* is a symbol, otherwise returns **nil**.

```
(symbol? nil)              -->       t
(symbol? 'abc)             -->       t
(symbol? '*a+b-c%$?<>)     -->       t
(symbol? "abc")            -->       nil
(symbol? 2)                -->       nil
(symbol? +2)               -->       nil
(symbol? +2.2)             -->       nil
(symbol? '|+2.2|)          -->       t
(symbol? (cons 'a 'b))     -->       nil
(symbol? (car '(a b)))     -->       t
```

(explode *symbol***)**

Returns a list of symbols, each consisting of one of the characters in the name of *symbol*, which must be a symbol.

```
(explode nil)              -->       (n i l)
(explode 'abc)             -->       (a b c)
(explode '*a.b-c%$?<>)     -->       (* a |.| b - c % $ ? < >)
```

(implode *list***)**

Returns a symbol whose name is the concatenation of the names of the members of *list*, which must be a list of symbols with one-character-names.

```
(implode '(n i l))         -->       nil
(implode '(a b c))         -->       abc
(implode '(* a |.| b))     -->       *a.b
```

Numbers

A number is an integer or a real. An integer is a sequence of digits or a sequence of digits preceded by + or -. Reals can be written in several ways, which are indicated by the examples below. Numbers are self-evaluating, they do not need to be quoted in programs.

(number? ***obj*****)**

Returns **t** if *obj* is a number, otherwise **nil**.

```
(number? 1)               -->     t
(number? -1)              -->     t
(number? +1234567890123)  -->     t
(number? -1.0)            -->     t
(number? +12345.67890)    -->     t
(number? +1.234567890e4)  -->     t   ; same value as above
(number? +1.2e-20)        -->     t   ; 1.2·10^-20, almost zero
(number? '-1)             -->     t
(number? '|-1|)           -->     nil
(number? '-1+)            -->     nil
(number? "-1")            -->     nil
(number? nil)             -->     nil
(number? 'abc)            -->     nil
```

In the following definitions assume that *n*, *n1*, *n2*, ... are numbers, that *i*, *i1*, *i2*, ... are integers. Warning: most arithmetical operations may be unreliable for reals.

(zero? ***n*****)**

Returns **t** if *n* is zero, otherwise **nil**.

(= ***n1 n2*****)**

Returns t if *n1* and *n2* are the same numbers, otherwise **nil.**

(< ***n1 n2*****)**

Returns t if *n1* is a smaller number than *n2*, otherwise **nil.**

(> ***n1 n2*****)**

Returns t if *n1* is a larger number than *n2*, otherwise **nil.**

(min ***n1 n2*** **...)**

Returns the smallest of the numbers given as arguments.

(max ***n1 n2*** **...)**

Returns the largest of the numbers given as arguments.

(1+ ***n*****)**

Returns the number one larger than *n*.

(-1+ ***n*****)**

Returns the number one smaller than *n*.

(+ *n1 n2* ...)

Returns the sum of the numbers given as arguments.

(* *n1 n2* ...)

Returns the product of the numbers given as arguments.

(- *n1*)

Returns the additive inverse of *n1*.

(- *n1 n2* ...)

Returns the number obtained by subtracting from *n1* the sum of all the other arguments.

(/ *n1*)

Returns the multiplicative inverse of *n1*.

(/ *n1 n2* ...)

Returns the number obtained by dividing *n1* by the product of all the other arguments.

(quotient *i1 i2*)

Returns the result of the number-theoretic integer division of *i1* by *i2*.

(remainder *i1 i2*)

Returns the remainder of the number-theoretic integer division of *i1* by *i2*.

```
(/ 4)                   -->     0.25
(/ 12 4)                -->     3
(/ 12 2 3)              -->     2
(/ 13 4)                -->     3.25
(quotient 13 4)         -->     3
(remainder 13 4)        -->     1
```

(sin *n*)

Returns the sine of *n*.

(cos *n*)

Returns the cosine of *n*.

(atan *n1 n2*)

Returns the arctangent of *n1* / *n2* in the range between -π and π.

(round *n*)

Returns the closest integer to *n*.

(floor *n*)

Returns the closest integer below *n*.

(ceiling *n*)

Returns the closest integer above *n*.

```
(round 2.5)             -->     3
(round 2.49999)         -->     2
(floor 2.49999)         -->     2
(ceiling 2.49999)       -->     3
```

(random *i*)

Returns a random non-negative integer smaller than *i* if *i* is given. If *i* is omitted, any integer, positive or negative, might be returned.

Strings

A string is a sequence of characters enclosed between two doublequote characters. Strings are self-evaluating, they do not need to be quoted in programs. In FranzScheme strings may be read and printed and passed around as arguments, but special operations on strings are not provided.

```
"abc"                          -->      "abc"
'"abc"                         -->      "abc"
"this contains | and ' "       -->      "this contains | and ' "
"this contains even
       (some)
  return characters and
  (other stuff)
)))"                           -->      "this contains even
                                               (some)
                                          return characters and
                                          (other stuff)
                                        )))"
```

The last example illustrates a common error of novices: a doublequote causes the interpreter to accumulate any input as part of a string until it encounters the next doublequote. To a user who forgot to type a closing doublequote and thus fails to notice that any input is still being accumulated, this often appears as if the system entirely refused to react.

Vectors

A vector is a structure of an arbitrary but fixed number of elements indexed by integers. 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 created by the procedure **make-vector**.

(vector? *obj***)**

Returns t if *obj* is a vector, otherwise returns **nil**.

(make-vector *length***)**
(make-vector *length fill***)**

Returns a newly allocated vector of *length* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

(vector-length *vec***)**

Returns the number of elements in the vector *vec*.

(vector-ref *vec i***)**

Returns the contents of element *i* of the vector *vec*. The index *i* must be a nonnegative integer less than **(vector-length** *vec***)**.

(vector-set! *vec i obj***)**

Stores *obj* as the contents of element *i* of the vector *vec*. The index *i* must be a nonnegative integer less than **(vector-length** *vec***)**. The value returned by **vector-set!** is unspecified.

```
(define v
  (make-vector 3 'init)) -->      v
(vector-set! v 2 'two)   -->      unspecified
(vector-set! v 1 'one)   -->      unspecified
(vector-ref v 0)         -->      init
(vector-ref v 1)         -->      one
(vector-ref v 2)         -->      two
(vector-ref v 3)         -->      error
```

Input and Output

In order to transfer data between Scheme and external media, Scheme uses data objects called ports, which connect it to the external media. All reading and printing goes through these ports. A port may be open for input or output but not for both simultaneously. There is a standard input port usually connected to the keybord and a standard output port usually connected to the teletype. Ports are created by **infile** and **outfile** and must be explicitly destroyed by **close**.

(**port?** *obj*)

Returns **t** if *obj* is a port, otherwise returns **nil**.

(**infile** *filename*)

The argument must be a string representing the name of a file containing Scheme expressions. Returns a new port which is open for input and connected to the given file.

(**outfile** *filename*)

The argument must be a string representing the name of a file to be created. Returns a new port which is open for output and connected to the new file.

(**close** *port*)

Releases the given port.

(**read** *port end-of-file-indicator*)
(**read** *port*)
(**read**)

Expects that *port* is open for input. Returns the first Scheme expression not yet read from the medium to which *port* is connected (but does not evaluate this expression), updating *port* to point to the first character after this expression. If no Scheme expression can be obtained from the medium, **read** returns its second argument or **nil** if no second argument is given. If *port* is omitted, the standard input port is used.

(**princ** *obj port*)
(**princ** *obj*)

Expects that *port* is open for output. Prints the printed representation of *obj* to the medium to which *port* is connected. If the argument is omitted, the standard output port is used. If *obj* is a string, the enclosing doublequotes are not printed. The result of the **print** expression is not specified.

(**print** *obj port*)
(**print** *obj*)

Like **princ**, but starts on a new line and terminates its output with a space, which **princ** does not.

(**pretty-print** *obj port*)
(**pretty-print** *obj*)

Like **princ**, but indents to align the output according to the structure of *obj*.

(newline ***port*****)**
(newline)

Expects that *port* is open for output. Prints a "carriage return" character to the medium to which *port* is connected. If the argument is omitted, the standard output port is used. The result of the **newline** expression is not specified.

(load ***filename*****)**

The argument must be a string representing the name of a file containing Scheme expressions. All of these expressions are successively evaluated as long as no error occurs. When the first erroneous expression is encountered, **load** stops loading.

(load* ***filename*****)**

Like **load** except that after an erroneous expression loading continues with the rest of the file.

Environments, Procedures, and Miscellaneous

The set of all variable bindings in effect at some point in a program is known as the environment in effect at that point. Environments do not have a standard printed representation, but they can be stored in data structures and passed around like any other values. The most common thing to do with an environment is to provide it as an argument to **eval**. Procedures are created when **lambda** expressions are evaluated. Procedures do not have a standard printed representation, but they can be stored in data structures and passed around like any other values. The most common thing to do with a procedure is to call it with appropriate arguments.

(the-environment)

The current environment, that is the set of all bindings in effect when this expression is evaluated.

(make-environment *def1* ...)

Each of the arguments is a **define** expression. Returns a new environment, which is the current one extended by the given definitions.

(eval *expr env*)

Returns the value of evaluating *expr* relative to the environment *env*.

```
(let ((x 1) (y 2))
  (eval
    '(+ x y)
    (the-environment))) -->       3
(let ((x 1) (y 2))
  (define e (make-environment (define x 11)))
  (eval
    '(+ x y)
    e))                 -->       13
```

(apply *proc args*)

Requires that *proc* is a procedure and *args* is a list of proper arguments. Calls *proc* with the elements of *args* as the actual arguments and returns the value of this procedure call.

```
(apply + '(3 4))        -->       7
```

(error *expression1* ...)

When evaluated, this expression always signals an error and prints its arguments. It never returns any value.

(runtime)

Returns the cpu time used since the invocation of the current FranzScheme session. The time is measured in units of 1/60 seconds.