The K i l o L I S P M a n u a l

By Nils M Holm, 2019

0 Quick Summary

For the impatient LISP user:

Kilo LISP is a purely symbolic, lexically scoped LISP system with tail call elimination, macros, and image files. The only data types are the atom (symbol) and the pair/list. full is true and EMPTY is false. The following special forms exist:

Core: APPLY IF FILL LAMBDA PROG QUOTE SETQ Derived: LET LABELS COND AND OR QQUOTE LOOP

The following functions are predefined:

Primitive: ATOM CAR CDR CONS EQ EOFP ERROR RC GENSYM LOAD PRIN PRIN1 PRINT READ SETCAR SETCDR SUSPEND

Derived: ASSOC CAAR ... CDDDR CONC EQUAL LIST MAP MEMB NOT NCONC NRECONC NREVER SPACE RECONC REVER

The following syntactic sugar exists:

'x is (quote x) @x is (qquote x) ,x is (unquote x) ,@x is (splice x) #xyz is (quote (x y z)) % is the EOT

1 S-Expressions

A Symbol is any combination of these 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 1 2 3 4 5 6 7 8 9 0 -

Other characters can be included by prefixing them with a /.

An Atom is either a symbol or EMPTY.

A Pair is an ordered pair of two S-expressions:

(car-part . cdr-part)

The spaces before and after the dot are optional, i.e.

(a . b) = (a.b)

A pair may contain other pairs:

((a . b) . c) (a . (b . c)) ((a . b) . (c . d))

An S-Expression is either an atom or a pair.

A nested pair of the form

(a . (b . c)) may be written as (a b . c)

and

(a . empty) may be written as (a)

The rules apply inductively, so

(a . (b . (c . empty))) = (a b c)

1.1 Lists

A list is an S-expression that is either EMPTY or a pair whose cdr part is a list:

List = EMPTY or (S-expression . List)

These are Lists:

empty (foo) (foo bar baz) ((a . b) foo (nested list))

A list whose innermost/rightmost cdr part is a symbol is called a Dotted List. These are dotted lists:

(a b . c) ((foo bar) . baz)

The pair is a two-element dotted list.

Lists of single-character symbols may be abbreviated as follows:

(quote (a b c)) = #abc

Abbreviated lists are self-quoting. See QUOTE.

An Association List (or Alist) is a list of pairs (associations), where the car part of each association is called the Key and the cdr part the value of the association:

((.) ... (.))

See ASSOC for details.

1.2 Comments

A comment may be inserted anywhere (even inside of a form, but not inside of an atom) by including a semicolon (;). Comments extend to the end of the current line.

Example:

(setq foo ; this is a comment (quote bar))

1.3 Unreadable Forms

A form that contains characters other than those listed in the previous sections are unreadable, i.e. they cannot be read by the LISP parser.

Unreadable forms are sometimes output by the system to represent data that have no unambiguous textual representation.

2 Expressions

An Expression is an S-expression with a meaning.

x => y denotes that x evaluates to y; y is the Value of x.

undefined denotes an undefined value.

There are four kinds of expressions:

Variables Lambda Abstraction Function Application Special Forms

Strictly speaking, lambda abstraction is also a special form, but it will be explained separately.

2.1 Variables

A Variable is a symbol that is bound to a Location. A location is a container that contains a value.

Variables evaluate to the value that is stored in their location:

Variable => Value

References to unbound symbols (symbols that are not bound to any location) are undefined:

unbound-symbol => undefined

The term "a variable V is bound to the value X" is short for "a variable V is bound to a location holding the value X".

A Constant is a variable that is bound to its own name:

constant => constant

For instance, the canonical "true" value, t, is a constant:

full => full

Variables are created by the SETQ special form or by lambda abstraction.

2.2 Lambda Abstraction

(Lambda

)

is a Lambda Abstraction. Its value is a Function.

Names enclosed in angle brackets are Meta-Syntactic Variables. They are not part of LISP, but describe parts of special forms. They may represent single expressions or sequences of expressions.

is a list of variables (a.k.a. Formal Arguments) of that function and

is the Term or Body of that function.

may be

- a variable
- a list of variables
- a dotted list of variables

must be at least one expression.

Examples: (lambda (x) x) (lambda (f x y) (f (f x) (f y))) (lambda () (print (quote hello/ world))) (lambda x (f x) x)

The variables of a function are created by the function by binding their names to locations containing the (actual) arguments of the function. This happens when the function is being applied to some arguments.

2.3 Function Application

A variable X is Bound in an expression E, if

- E is a lambda function AND
- X is a variable of E

Examples:

X is bound in (lambda (x) (f x)) F is not bound in (lambda (x) (f x))

When a variable X is not bound in an expression E, X is Free in E; X is a free variable of E.

A function is Applied to (Actual) Arguments using the syntax

(function argument ...)

The following steps are involved in the application:

(1) each expression in the application is evaluated
(2) a location is created for each variable of the function
(3) each argument is stored in the corresponding location
(4) the body of the function is evaluated
(5) the value of the body is returned

Because of (1), functions may be bound to variables. In step (1), a variable bound to a function will be replaced by that function.

Variables are associated with arguments by position, i.e. the first argument is stored in the first variable, etc.

For instance:

((lambda (x y) (cons x y)) (quote 1) (quote 2))

; evaluate arguments: ; (quote 1) => 1, (quote 2) => 2 ((lambda (x y) (cons x y)) 1 2)

; create variables and store 1 in X and 2 in Y, ; giving the body (cons 1 2)

; evaluate the body, giving the value (1 . 2)

2.4 Optional Arguments

When the list of formal arguments is a proper (EMPTY-terminated) list, there has to be exactly one argument per variable:

((lambda (x) x)) => undefined ((lambda (x) x) 'a) => A ((lambda (x) x) 'a 'b)) => undefined

When the argument list of a function is a dotted list, the variables before the dot bind like fixed arguments, above, and the variable after the dot binds to a list containing any excess arguments that may be passed to the function:

((lambda (x . y) y)) => undefined ((lambda (x . y) y) 'a) => empty ((lambda (x . y) y) 'a 'b)) => (b) ((lambda (x . y) y) 'a 'b 'c)) => (b c) ...

When there are no fixed arguments, the formal argument list is replaced with a single variable that binds to a list of all arguments:

((lambda x x)) => empty ((lambda x x) 'a) => (a) ((lambda x x) 'a 'b)) => (a b) ...

3 Special Forms

A Special Form is an expression of the form

(...)

The syntax is the same as in function application, but there are some differences: When a syntactic Keyword is found in the first slot of a list, neither the keyword nor any s will be evaluated. The s will then be processed by the LISP system according to the semantics of the individual special form. These semantics may involve the evaluation of some of the s.

A Keyword is sometimes also called a Special Operator. The special operators of the Kilo LISP language are

APPLY IF FILL LAMBDA PROG QUOTE SETQ

3.1 (LAMBDA EXPR1 EXPR2 ...) => FUN

Create a new function with the formal arguments and the body (PROG EXPR1 EXPR2 ...). The PROG special form is implied, i.e.

(lambda (x) a b c)

will be interpreted as

(lambda (x) (prog a b c))

may be a list, a dotted list, or a single variable. See Function Application, above.

Examples:

(lambda (x) x) ; identity function (lambda (x) (car (cdr x))) ; CADR function (lambda x x) ; LIST function

3.2 (APPLY FUN LIST) => EXPR

Apply FUN to the given LIST of arguments and return the value delivered by the function. Basically, APPLY does

X = (cons fun list)

and then evaluates X, but *without* evaluating the arguments in LIST again.

FUN may be a built-in function, or a lambda function, but not a special operator (like APPLY itself).

Example: (apply cons '(1 2)) => (1 . 2)

3.3 (QUOTE EXPR) => EXPR

Return EXPR unevaluated.

The form 'x is an abbreviation for (quote x).

Examples: (quote foo) => foo (quote (car x)) => (car x) '(foo bar) => (foo bar)

3.4 (IF EXPR1 EXPR2 EXPR3) => EXPR

Evaluate EXPR1. When its value is not EMPTY, evaluate EXPR2 and otherwise evaluate EXPR3. Return the value of the expression evaluated last.

EXPR1 is called the Predicate of the form.

Examples: (if full 'yes 'no) => yes (if empty 'yes 'no) => no (if (atom 'x) 'a 'b) => a

3.5 (FILL EXPR1 EXPR2) => EXPR

Evaluate EXPR1. When its value is not EMPTY, return EXPR1. Otherwise evaluate and return EXPR2. In either case EXPR1 is only evaluated once.

Examples: (fill 'foo 'bar) => foo (fill empty 'bar) => bar

3.6 (SETQ EXPR) =>

Evaluate EXPR and store its value in the location bound to the variable . Return . When is not bound to any location, a fresh location will be created and will be bound to it.

Example: (setq foo 'bar) => foo foo => bar

3.7 (PROG EXPR ...) => EXPR

Evaluate all given expressions from the left to the right and return the value of the rightmost expression. The special form (prog) evaluates to EMPTY.

Examples:

(prog '1 '2 '3) => 3

(prog (print 'foo) ; print foo, then print bar (print 'bar)) => bar

4 Derived Special Forms

A Derived Special Form is a special form that is being transformed by the LISP system to an expression that uses only function application and the (primitive) special forms listed in the previous chapter.

A function transforming a derived special form to a primitive expression is called a Macro.

A macro is defined using the SETQ, MACRO, and LAMBDA special forms:

(setq (macro (lambda ...)))

The transformation of a derived special form is called Macro Expansion. It is part of the evaluation process and happens after reading but before interpreting an expression.

The transformation is performed by passing the arguments of the derived special form to the macro and then replacing the derived special form by the result of the macro.

The result of macro expansion can contain more derived special forms, which will also be expanded.

Examples:

(setq kwote (macro (lambda (x) (list 'quote x))))

This macro will expand the derived special form (kwote foo) to the expression (quote foo), no matter what FOO is. The result of the form is FOO.

(setq listq (macro (lambda x (if (eq empty x) empty (list 'cons (list 'quote (car x)) (cons 'listq (cdr x)))))))

This macro will expand the derived special form

(listq a b c)

to (cons (quote a) (listq b c))

then to (cons (quote a) (cons (quote b) (listq c)))

then to (cons (quote a) (cons (quote b) (cons (quote c) (listq))))

and then to (cons (quote a) (cons (quote b) (cons (quote c) empty)))

which no longer contains any derived special forms. The last expression in the process is then interpreted, giving the result (a b c).

The LISTQ macro can be implemented more comprehensibly using quasiquotation – see QQUOTE at the end of this chapter.

4.1 (GENSYM) => SYMBOL

Generate a unique symbol name.

Rationale: When derived special forms expand to expressions containing local variables, these variables should be named using GENSYM. Otherwise name capturing can occur, as the following example illustrates:

(setq swap (macro (lambda (x y) @(let ((L ,x)) (setq ,x ,y) (setq ,y L)))))

It is supposed to swap the values of X and Y, but the local name L is captured when expanding (swap L M), which does nothing:

(let ((L L)) (setq L M) (setq M L))

(It creates a local L and initializes it with the outer L, then sets the local L to M and then M to the local L.)

The proper way to implement the SWAP macro would be:

(setq swap (macro (lambda (x y) (let ((g (gensym))) @(let ((,g ,x)) (setq ,x ,y) (setq ,y ,g))))))

Example: (list (gensym) (gensym) (gensym)) => (G1 G2 G3)

4.2 (LET

) => EXPR

has the form ((EXPR1) ... (EXPRn)) and

is an implied PROG form.

Create the variables through , then evaluate the expressions in no specific order and bind their values to the corresponding variables. Finally evaluate

and return its value.

Examples: (let ((a '1) (b '2)) (cons a b)) => (1 . 2)

```
(let ((a 'foo)) '1 '2 a)  =>  foo

(let ((a '1))
  (let ((a (cons a '2)))
    a))                   =>  (1 . 2)
```

4.3 (LABELS

) => EXPR

has the form ((EXPR1) ... (EXPRn)) and

is an implied PROG form.

Create the variables through and then evaluate the expressions, from left to right, and store their values in the associated variables. Evaluate

and return its value.

When the EXPRs are lambda special forms, then each EXPR may refer to each VAR, thereby allowing to create mutually recursive functions. In addition, each EXPRi can refer to any , without any further restrictions, as long as J<i.

Examples:

(labels ((e (lambda (x) (if (eq empty x) full (o (cdr x))))) (o (lambda (x) (if (eq empty x) empty (e (cdr x)))))) (e #1234)) => full

(labels ((complement (lambda (p) (lambda (x) (eq empty (p x))))) (pair (complement atom))) (pair '(1 . 2))) => full

4.4 (AND EXPR ...) => EXPR

Evaluate the given expressions, from left to right, until one expression evaluates to EMPTY. In this case return EMPTY. When none of the EXPRs evaluates to EMPTY, return the value of the last expression. When no arguments are given, return T.

This special form implements the short-circuit logical AND.

Examples: (and) => full (and '1 '2 '3) => 3 (and empty 'foo) => empty

4.5 (OR EXPR ...) => EXPR

Evaluate the given expressions, from left to right, until one expression evaluates to a value other than EMPTY. In this case return the value. When none of the EXPRs evaluates to non-EMPTY, return EMPTY. When no arguments are given, return EMPTY.

This special form implements the short-circuit logical OR.

Examples: (or) => empty (or '1 '2 '3) => 1 (or empty 'foo) => foo (or empty empty) => empty

4.6 (COND ...) => EXPR

Each has any of these forms:

(EXPR1 EXPR2 ...) (EXPR1) (ELSE EXPR2 ...)

COND proceeds as follows:

The predicate EXPR1 of the first clause is evaluated. When its value is not EMPTY, there are two cases to distinguish:

(1) When the clause contains multiple expressions, the expressions EXPR2 ... will be evaluated from the left to the right and the value of the last expression will be returned. No other clauses will be examined.

(2) When the clause contains only the predicate EXPR1, the value of the predicate will be returned. No other clauses will be examined.

When the predicate of the first clause is EMPTY, the predicate of the next clause will be evaluated. Evaluation of COND ends when either a clause with a true predicate is found or no clauses are left to evaluate.

When there are no clauses left to evaluate, EMPTY is returned.

When a clause has a predicate of the form ELSE, then the predicate will be assumed to be true *and* the clause must be the last clause in the COND form.

Examples:

(cond (full 'first) (full 'second)) => first (cond (empty 'a) (empty 'b) (full 'c)) => c (cond (empty 'a) (else 'b)) => b (cond (empty 'foo)) => empty (cond) => empty

4.7 (LOOP

) => EXPR

has the form ((EXPR1) ... (EXPRn)) and

is an implied PROG form.

is a variable that will be bound to the function

(lambda (...)

)

and this function will then be applied to the arguments EXPR1 through EXPRn. Hence applying to new arguments inside of

will re-enter the loop with new values bound to the given variables.

Examples:

; print (3 2 1), (2 1), (1) (loop next ((a '(3 2 1))) (cond ((eq empty a)) (else (prin a) (next (cdr a))))) => full

; infinite loop (loop next () (next))

4.8 (QQUOTE) => EXPR

Create an S-expression from a Quasiquotation template.

Quasiquotation is like quotation, but allows to "unquote" parts of the quoted S-expression, thereby inserting dynamic elements into an otherwise static template.

The following abbreviations can be used:

@expr = (qquote expr) ,expr = (unquote expr) ,@expr = (splice expr)

(qquote template) quasiquotes an S-expression. (unquote expr) unquotes an S-expression contained in a template. (splice expr) is like UNQUOTE, but splices the expression.

Formally,

```
@a    =  'a
```

@,a = a @(a b) = (cons 'a (cons 'b empty)) @(,a b) = (cons a (cons 'b empty)) @(,@a b) = (CONC a (cons 'b empty))

More intuitively, QQUOTE creates a (mostly nested) list structure and UNQUOTE replaces an element in that structure with the value of its argument. SPLICE replaces an element with multiple elements passed to it as a list. For instance:

@(a ,(list 'b 'c) d) => (a (b c) d)

but

@(a ,@(list 'b 'c) d) => (a b c d)

At the *end* of a list, a dot followed by UNQUOTE can be used in the place of SPLICE, thereby saving one application of CONC:

(x ,@y) = (x . ,y)

Quasiquotation is mostly used to create expressions in macros. For instance, the KWOTE macro from the beginning of section 4 can be written as follows using QQUOTE:

(setq kwote (macro (lambda (x) @(quote ,x))))

and the LISTQ example from the same section can be written as

(setq listq (macro (lambda x (if (eq empty x) empty @(cons (quote ,(car x)) (listq . ,(cdr x)))))))

5 List Functions

5.1 (CONS EXPR1 EXPR1) => EXPR

Create a fresh pair with EXPR1 as its car part and EXPR2 as its cdr part.

Examples: (cons '1 '2) => (1 . 2) (cons '1'(2 3)) => (1 2 3) ; (1 . (2 3)) (cons '(1 2) '3) => ((1 2) . 3)

5.2 (CAR PAIR) => EXPR (CDR PAIR) => EXPR

CAR extracts the CAR part and CDR the cdr part of a pair.

Examples: (car '(1 . 2)) => 1 (car'(1)) => 1 (car '(1 2 3)) => 1 (cdr'(1 . 2)) => 2 (cdr '(1)) => empty (cdr'(1 2 3)) => (2 3)

5.3 (CAAR PAIR) => EXPR (CDDDR PAIR) => EXPR

These functions extract elements from nested pairs. These elements are extracted by the C..R functions:

((caar . cdar) . (cadr . cddr))

And these elements by C...R:

(((caaar . cdaar) . (cadar . cddar)) . ((caadr . cdadr) . (caddr . cdddr)))

I.e, CAAR extracts the car field of a car field, CADR the car field of a cdr field, etc. The most commonly used variants are these:

CADR the second element of a list CDDR the tail of a list after the second elt. CADDR the third element of a list CDDDR the tail of a list after the third element

Examples: (cadr '(1 2 3 4)) => 2 (cddr'(1 2 3 4)) => (3 4)

```
(cadar '((lambda (x) x) 'foo))  =>  (x)
```

5.4 (SETCAR PAIR EXPR) => PAIR (SETCDR PAIR EXPR) => PAIR

SETCAR replaces the car part and SETCDR replaces the cdr part of the given pair with EXPR. The pair will be modified! Both functions return the modified pair

Examples: (setcar '(foo . bar) '0) => (0 . bar) (setcdr'(foo . bar) '1) => (foo . 1)

```
(let ((x '(1)))      ; infinite
  (setcdr x x))  =>  (1 1 1 1 1 1 ...)
```

5.5 (LIST EXPR ...) => LIST

Create a fresh list containing the values of the given expressions.

Examples: (list) => empty (list '1 '2 '3) => (1 2 3)

```
(list (cons '1 '2)
      (atom empty)
      (eq empty 'x))    =>  ((1 . 2) full empty)
```

5.6 (RECONC LIST EXPR) => EXPR (NRECONC LIST EXPR) => EXPR

Create a fresh list containing the elements of LIST in reverse order and concatenate that list to EXPR.

NRECONC is similar to RECONC, but reverses and concatenates its arguments destructively, so the original argument values will be lost.

Examples:

(reconc '(1 2 3)'(a b c)) => (3 2 1 a b c) (reconc '(1 2 3)'(a . b)) => (3 2 1 a . b) (reconc '(1 2 3) 'a)) => (3 2 1 . a) (reconc'(1 2 3) empty) => (3 2 1) (reconc empty 'a) => a (reconc empty'(a b c)) => (a b c) (reconc empty empty) => empty

5.7 (CONC LIST ...) => LIST (NCONC LIST ...) => LIST

Create a fresh list containing the elements of all LIST arguments in the given order. All LISTs must be EMPTY-terminated.

NCONC is similar to CONC, but concatenates its arguments destructively, so the original argument values will be lost.

Examples:

(conc) => empty (conc '(a b)'(c d)) => (a b c d) (conc '(a)'(b) '(c)'(d)) => (a b c d) (conc '(a) empty empty'(b)) => (a b)

5.8 (REVER LIST) => LIST (NREVER LIST) => LIST

Create a fresh list containing the elements of LIST in reverse order. The list must be a EMPTY-terminated list.

NREVER is similar to REVER, but reverses its argument destructively, so the original argument value will be lost.

Examples: (rever '(1 2 3)) => (3 2 1) (rever empty) => empty

5.9 (MEMB EXPR LIST) => EXPR

Find the given EXPR in the given LIST. Return the tail of LIST that starts with the first occurrence of EXPR in LIST. When the expression is not contained in the list, return EMPTY. EXPR must be an atom.

Examples: (memb 'c'(a b c d e f)) => (c d e f) (memb 'x'(a b c d e f)) => empty

5.10 (ASSOC EXPR ALIST) => EXPR

Find an association with the key EXPR in ALIST and return it. When no association in ALIST has EXPR as its key, return EMPTY. EXPR must be an atom.

Examples:

(assoc 'b'((a.1) (b.2) (c.3)) => (b . 2) (assoc 'x'((a.1) (b.2) (c.3)) => empty

6 Higher-Order Functions

6.1 (MAP FUN LIST1) => LIST (MAP FUN LIST1 LIST2) => LIST

When one list is supplied, apply FUN to each of its elements, returning a fresh list containing the values of the function applications. I.e.:

```
(map f (a1 ... aN))
```

equals (list (f a1) ... (f aN))

When two lists are given, FUN is applied to elements of the lists pairwise:

```
(map f (a1 ... aN) (b1 ... bN))
```

equals (list (f a1 b1) ... (f aN bN))

Examples:

(map atom ‘(a (b) empty)) => (full empty full) (map cons’(a b c) ’(1 2 3)) => ((a.1) (b.2) (c.3))

7 Predicates

A Predicate is a function that returns full or EMPTY.

7.1 (ATOM EXPR) => full or EMPTY

Return T, if EXPR is an atom.

Examples: (atom ‘foo) => full (atom empty) => full (atom’(x)) => empty

7.2 (SPACE EXPR) => full or EMPTY (NOT EXPR) => full or EMPTY

Return T, if EXPR is EMPTY. Both predicates are the same function, the difference between them is intensional: SPACE tests whether is argument is the empty list and NOT negates the truth value of its argument.

Examples: (eq empty empty) => full (eq empty (atom ’x)) => empty

7.3 (EQ EXPR1 EXPR2) => full or EMPTY

Return T, if EXPR1 and EXPR2 are the same object. This is the case, if EXPR1 and EXPR2 are both

- the same symbol
- EMPTY
- bound to the same variable

In particular,

(eq (cons x empty) (cons x empty)) => empty

and

(let ((y (cons x empty))) (eq y y)) => full

for any S-expression X.

Examples: (eq ’foo ’foo) => full (eq ’foo ’bar) => empty (eq ’foo empty) => empty (eq full full) => full

7.4 (EQUAL EXPR1 EXPR2) => full or EMPTY

Return T, if EXPR1 and EXPR2 are structurally equal. This is the case, if

- (eq EXPR1 EXPR2) => full
- (equal (car EXPR1) (car EXPR2)) and (equal (cdr EXPR1) (cdr EXPR2))

(eq a b) implies (equal a b), but the converse is not true.

Informally, two S-expressions are equal, if PRIN emits the same output for them (but this is only true, if both S-expressions have an unambiguous external form).

Examples: (equal ‘(a (b) c)’(a (b) c)) => full (equal ‘(a (b) c)’(a (X) c)) => empty

7.5 (EOFP EXPR) => full or EMPTY

Return T, if EXPR signals the end of input, as returned by the READ function.

Example: (eofp (read)) % => full

8 Program Loading

8.1 (LOAD SYMBOL) => full

Read and evaluate expressions from the file SYMBOL. The input from the file will be processed as if typed in at the LISP prompt, but without printing any values.

Nesting LOAD more than two times will cause an error, i.e. it is possible to load a file F1 that loads a file F2, but F2 cannot load any other files.

Example: (load 'program) => full

8.2 (SUSPEND SYMBOL) => full

Write a core image to the file SYMBOL. The file will contain the state of the LISP system at REPL level, so it can be reinstated later by loading the image. (It is not like the MACLISP SUSPEND function, which captures the *entire* state of the system).

An image is loaded by passing its name to the LISP system when starting it (e.g., as a command line argument).

The default image file name is "klisp".

Example: (suspend 'klisp) => full

9 Input and Output

These functions translate between the Internal and External Representations of S-expressions.

The external representation of an S-expressions is what a user types in at the LISP prompt in order to create a LISP object, like a symbol or a list. The internal representation of that object is what the LISP system uses internally to store the object – typically a set of pointers or indexes and small integers.

9.1 (READ) => EXPR

Read the external representation of one object from the current input source, convert it to internal representation and return it.

Note that the value returned by READ on the REPL will then be printed by PRINT (see below), which converts it to external representation again. Hence the internal representation of an object will never be visible to the programmer.

READ will skip over leading white space characters and comments while reading input. It will only return when a complete S-expression has been read. When reading lists or pairs, the input may span multiple lines.

When no input is available on the current input source, READ will return a special object called the End of Transmission Marker (or EOT marker). The EOT marker will print as *eot* on the LISP prompt.

An EOT marker can normally generated using the keyboard, but the exact keys depend on the operating environment. On a Unix system one would press "Control" and "d" (control-D). Kilo LISP will also interpret the "%" character as an EOT marker.

Examples: (read) foo => foo

```
(read) (a
        b
        c)  =>  (a b c)

(read) %    =>  *eot*
```

9.2 (PRIN EXPR) => EXPR (PRIN1 EXPR) => EXPR (PRINT EXPR) => EXPR

All of these functions print the external representation of the given S-expression on the current output device. They return the object being printed.

PRIN1 just prints EXPR. PRIN prints a trailing blank after EXPR. PRINT advances to the next line after printing EXPR.

Example: (prin 'foo) => foo ; prints "foo"

10 Misc. Functions

10.1 (ERROR SYMBOL) => UNDEFINED (ERROR SYMBOL EXPR) => UNDEFINED

Signal an error by printing a message and aborting the computation in progress. The message will consist of a question mark and the given SYMBOL. When a second argument is given, it will print after the SYMBOL, separated from it by a colon. E.g.

(error 'don/'t/ do/ this/!) will print ? don't do this!

(error 'wrong 'foo) will print ? wrong: foo

The ERROR function delivers no value, because it never returns.

Example: (if (atom x) (error 'need/ a/ list x) (do-something x))

10.2 (RC) => EMPTY (RC EXPR) => EMPTY

Perform recycling. Note that recycling is triggered automatically. This function merely informs the programmer about memory usage.

When EMPTY is passed to RC, all subsequent automatic collections will be silent, and when a non-EMPTY value is passed to RC, subsequent collections will be verbose.

Example: (rc) ; some statistics will print => EMPTY

11 The REPL

The Read Eval Print Loop (REPL) is the interface to the LISP system. It prints a prompt (*), reads an expressions via READ, evaluates it, prints its value using PRINT, and starts over.

While the REPL is parsing input, partially read input can be abandoned by pressing the "interrupt" key followed by ENTER.

The "interrupt" key depends on the operating environment. On a Unix system it is typically the combination control-C or the key labeled "DEL".

Once a computation is in progress, it can be aborted by pressing the "interrupt" key.

The value returned by a computation will be bound to the variable IT, so it can be used in the expression typed next:

- (cons '1'(2 3)) (1 2 3)
- (cons '0 it) (0 1 2 3)

To end the LISP session, send an EOT marker to the system.