TUNIX Lisp

The Garbage-Collected Manual

Sven Michael Klose

Contents

Overview

TUNIX Lisp is a highly efficient Lisp interpreter, written in ANSI-C. It is designed for constrained environments, such as embedded systems, classic home computers, including 6502-based systems.

Features:

- Compacting mark-and-sweep garbage collector.[1]
- Lean I/O interface.[2]
- Supplementary compressed stack.[3]

This distribution builds executables for these platforms using the cc65 C compiler suite:

- Commodore C128
- Commodore C16
- Commodore C64
- Commodore Plus4
- Commodore VIC-20 (+27K)

It also compiles on regular Unixoids, using the GNU compiler toolchain or compatibles.

Differences to other dialects

The TUNIX Lisp dialect is very much like any other. Here are some things that raise an eyebrow when seeing them the first time:

Most other dialects	TUNIX Lisp
backquote sign “’	dollar sign ‘$’

[1] Planned to be made interruptible to some degree, optionally truly interruptible providing a copying garbage collector.
[2] Instead of providing a file number for each I/O operation an input and/or output channel must be selected beforehand, bridging the gap between plain standard I/O and multi-stream handling without making the API more complex from the start, supporting operation in maximally constrained environments.
[3] It holds byte-sized tags instead of larger return addresses. Also to support architectures with limited CPU stacks, like those with MOS-6502 CPUs.

Most other dialects	TUNIX Lisp
(RPLACA c v)	(SETCAR c v)
(RPLACD c v)	(SETCDR c v)
(MAKE-SYMBOL x)	(SYMBOL l)
(SYMBOL-VALUE s)	(VALUE s)
(FILTER f l)	(@ f l)
(LAMBDA (args . body))	(args . body)
#\A	\A

Because the backquote (‘) is not part of the charsets of old machines TUNIX Lisp intends to support, the dollar sign ($) is used as the abbreviation for QUASIQUOTE.

MEMBER and FIND are comparing with EQ instead of EQL as these functions are used internally as well and need to be fast. Use NEMBER-IF or FIND-IF together with EQL to match numbers by value.

LAMBDA is not around yet. Function expressions are quoted when used as arguments to other functions. That makes compiling them to ‘native’ function impossible, so something similar will have to be in later versions.

Symbols are strings

Symbols have a case-sensitive name and a value and they also serve as strings. They can be converted to and from character value lists:

```
1  (symbol '(\A \B \C)) -> "ABC"
```

Symbols may also be anonymous, with no name at all.

```
1  (symbol) ; Anonymous symbol that won't get re-used.
```

Memory consumption

Heap

Object allocation is fast, requiring bumping up the pointer to the top of the growing heap, and a boundary check to trigger garbage collection when the heap is full.

Data type	heap
cons	5
number (32 bit signed)	5
symbol (also string)	4-260

CPU stack, object stack, and tag stack

Alongside the CPU stack a separate garbage-collected object stack holds function arguments and objects that need to be relocated during garbage collection. An additional raw stack holds return tags of byte size instead of full return addresses, and raw pointers to built-in procedure's argument definitions.

Inevitable creation of list elements

APPLY copies all arguments but the last one.

User interface: The READ/EVAL/PRINT-Loop (REPL)

The REPL is the user interface. It prompts you for input by printing an asterisk '*' in its regular mode, except on Commodore 8-bit machines, to allow using the KERNAL's screen editor. After reading an expression it is evaluated and the result of that evaluation is output. Then it starts over, prompting you for the next expression.

REPLs can be nested, e.g. when an error occured, a new REPL is launched in debug mode. With that you can examine the environment, execute code step by step and present a correct alternative for a faulty expression.

Any REPL, no matter its mode, can be terminated using the built-in QUIT function, which takes a return value for the REPL as its argument. It may become the return value of a LOAD function when loading a file, or the alternative return value of a faulty expression in debug mode. Built-in function IGNORE interrupts evaluation of the current expression read by the active REPL to start over with reading and evaluating the next one.

Built-in function EXIT stops the program and returns to the topmost REPL. When passed an exit code, EXIT terminates the interpreter and returns to the operating system.

Definiton of permanent symbols

FN and VAR assign expressions to symbols which are also added to variable *universe*, a list of symbols the garbage collector starting off with to reach all used objects. The difference between FN and VAR is that VAR evaluates its initialization argument and FN assigns its argument list unevaluated.

```
1  ; Define permanent, named function.
2  (fn welcome ()
3    (out '"Hello World!")
4    (terpri))
5
6  ; Define permanent, named variable.
7  (var x nil)
```

If you are using a screen editor, the SOURCE function is rather useful. It returns a defining expression for any symbol.

Functions

Functions are lists starting with an argument definition followed by a list of expressions. The result of the last expression is returned.

The LAMBDA keyword is not around at the moment but it has to be to make the compiler work.

```
; Function with no arguments, returning symbol NIL.
(nil)

; Function with no arguments, returning number '3'.
(nil
  1
  2
  3)

; Function returning its argument.
((x)
  x)

; Functions returning their argument list.
(x
  x)
((first . rest)
  (cons first rest))
```

Anonymous functions can be used as the first element of an expression without quoting:

```
; Print number '100'.
(((x)
   (print (+ 1 x)))
 99)
```

Anonymous functions as arguments need to be quoted though:

```
; Add 1 to each number in list.
(@ '((n) (++ n)) '(l 2 3))
```

The QUASIQUOTE (short form "$") can be used to emulate read-only lexical scope by unquoting outer values:

```
; Make a function that adds X to its argument.
```

```
(fn make-adder (x)
  $((a)
     (+ a ,x)))
```

Argument type descriptions (and definitions)

Built-in functions have character-based and typed argument definitions. They are also used, padded with spaces, to describe arguments in this manual for all procedures (functions, macros and special forms).

Code	Type
x	anything
c	cons
l	list (cons or NIL)
n	number
s	symbol
a	memory address (positive number)
b	byte value

They may also have prefixes:

Prefix	Description
+X	any number of type X
?X	optional
'X	unevaluated

Input/output

TUNIX Lisp boils I/O down to its basics: one channel for input and one for output, initially wired to "standard I/O", like your terminal with screen and keyboard. Input and output can each be switched to other channels. If you launch a LOAD command to execute a Lisp file, the input channel is connected to that file until it's been read entirely, but in general a channel can be directed to another one anytime.

READing and PRINTing expressions

Expressions can be read and written using built-in functions READ and PRINT. Strings and chars have dedicated formats:

Type format examples	Description
(a . d)	"dotted pair" (must be quoted),
"string"	String. Escape is "\".
\A	Character value.

READ and PRINT also support abbreviations if compiled in:

Expression	Abbreviation
(quote x)	'x
(quasiquote x)	$x
(unquote x)	,x
(unquote-splice x)	,@x

Catching I/O errors and state

Character-based I/O

Input and output channel

An input and an output channel can be switched between open streams separately using functions SETIN and SETOUT. Symbols STDIN and STDOUT contain the standard I/O channel numbers.

```
; Switch to standard I/O channels.
(setin stdin)
(setout stdout)
```

The currently active channels numbers are in symbols FNIN and FNOUT.

New channels are created by OPEN to access files:

```
(fn user-defined-load (pathname)
  (with ((last-result  nil)
         (old-channel  fnin)
         (new-channel  (open pathname)))
    (unless (err)
      (setin new-channel)
      (while (not (eof))
        (= last-result (eval (read))))
      (setin old-channel)
      last-result)))
```

Debugging and advanced error handling

The debugger is invoked in case of an error unless ONERROR has been defined. Beatiful things can be done by handling errors automatically, but let's get our hands on the debugger first.

The debugger REPL

Variable	Description
b	List of symbols that are breakpointed.
r	Initial return value of current REPL.

The debugger is the REPL in debug mode. It prints a status info before waiting for user input, so you know where the program execution has been interrupted. It has this format:

```
1 Debugger <number of nested debuggers>:
2 Error #5: <reason for break>
3 Rvalue: <last expression's (and debugger's) return value>
4 In:
5 <top-level expression with current one highlighted>
```

The debugger takes expressions like the regular REPL, plus some commands consisting of a single character to step through the code conveniently. If another error occurs, yet another debugger REPL will be invoked and the "number of nested debuggers" incremented.

The current expression is either the one that failed, or the one that will be evaluated next in cause the debugger stopped at a breakpoint (and no error number and description is shown).

The return value of the debugger will change with every expression you enter, except when using aforementioned short commands. In case of an error, that's the value you want to replace with a valid one before continuing program execution. Symbol *R* contains the return value when the debugger was invoked, should you want to see or use it again although you've replaced it already – just enter "*r*" and it'll be restored.

These are the available short commands:

Command	Description
c	Continue program execution.
s	Step into user-defined procedure.
n	Execute current expression in whole.
pX	Evaluate and print expression X. (No macros!)
bS	Set breakpoint on procedure S.
b	Print breakpoints.
dS	Delete breakpoint on procedure S.
d	Delete all breakpoints.

Command “p” evaluates the expression immediately following it. A macro expansion is *not* performed and it’ll *not* change the debugger’s return value.

Breakpoints

Global variable *B* is a list procedures’ names which, if called, will invoke the debugger.

You can modify *B* using the regular set of procedures:

```
1  ; Set breakpoint on procedure SUBSEQ.
2  (push 'subseq *b*)
3
4  ; Delete a breakpoint.
5  (= *b* (remove 'subseq *b*))
6
7  ; Delete all breakpoints.
8  (= *b* nil)
```

Inside the debugger REPL that’s inconvenient as every regular expression changes the debugger’s return value. Use short commands ‘b’ and ‘d’ instead.

```
1  bsubseq ; Set breakpoint on SUBSEQ.
2  dsubseq ; Delete breakpoint on SUBSEQ.
3  d       ; Delete all breakpoints.
```

User-defined error handler ONERROR

Function	Description
(onerror n x x)	User-defined error handler.
(ignore)	Break and continue with LOAD or REPL.

If defined, user-defined function ONERROR is called on errors, except for internal ones which halt the interpreter to avoid unexpected behaviour. Errors happening inside ONERROR will cause it to be called again. The handler must return a correct replacement value insted of calling QUIT or use IGNORE.

Arguments to ONERROR

ONERROR is called with the error code, the current REPL (top-level) expression, and the faulty expression within that:

```
; SKETCH! UNTESTED!
; Load missing functions on demand.
(fn onerror (errcode repl faulty)
  ; errcode: Error code.
  ; repl:    Top-level expression or
  ;          body of user-defined function.
  ; faulty:  The faulty expression in 'repl'.
  (? (== n 1) ; Not a function error.
     ; Evaluate matching definition in environment file.
     (with-infile f "env.lisp"
       (while (not (eof))
         (!= (read)
           (when (and (cons? !)
                      (or (eq (car !) 'var)
                          (eq (car !) 'fn))
                      (eq (cadr !) x))
             (eval !)
             (return x)))))))
```

Error codes

ID (ERROR_…)	Code	Description
TYPE	1	Unexpected object type.

ID (ERROR_…)	Code	Description
ARG_MISSING	2	One or more missing arguments.
TAG_MISSING	3	BLOCK tag couldn’t be found.
TOO_MANY_ARGS	4	Too many arguments.
NOT_FUNCTION	5	Object is not a function.
ARGNAME_TYPE	6	Argument name is not a symbol.
OUT_OF_HEAP	7	Out of heap. Cannot catch.
NO_PAREN	8	‘)’ missing.
STALE_PAREN	9	Unexpected ‘)’.
FILE	10	File error code in (ERR).
FILEMODE	11	Illegal mode for OPEN.
USER	12	ERROR function was called.
INTERNAL	14	Returned to operating system.

Built-in functions

General

Function	Description
(gc)	Free unused objects.
(exit ?n)	Exit program or interpreter with code.

(gc): Free unused objects.

Triggers the garbage collector. It marks all objects linked to variable *UNIVERSE*, compacts the heap and relocates all pointers.

(exit ?n): Exit program or interpreter with exit code.

When called without arguments the program is stopped and control is returned to the top-level REPL. When called with a number that number is the exit code for the interpreter which will terminate immediately.

Definitions

Form	Type
(var 'name x)	Define symbol with value.
(fn 'name 'args'+body)	Define function.
(special 'name 'args'+body)	Define special form.

Function	Description
(source s)	Return defining expression for a symbol.

(fn ‘name ’args’+body): Define permanent, named function.

(special ‘name ’args’+body): Make special form.

Special forms are functions that take their arguments unevaluated, e.g. QUASIQUOTE and MACRO, so you don’t have to quote arguments of that function manually.

(var ’name init): Define permanent, named variable.

(source s): Return defining expression for a symbol.

Evaluation and flow control

Function	Description
(quote ’x)	Return argument unevaluated.
(apply f +x)	Call function with list of arguments.
(funcall f +x)	Call function with explicit arguments.
(eval x)	Evaluate expression.
(? cond +x)	Evaluate expression conditionally.
(and +x)	Logical AND. Evaluate until NIL.
(or +x)	Logical OR. Evaluate until not NIL.
(block ’s +x)	Named block with expression list.
(return x ?’s)	Return from named block with value.
(go ’s)	Jump to tag in named block.

(quote x)

Returns argument unevaluated. Suppresses replacing symbols by their values on evaluation.

```
; Define variable X, containing the string "What a day!".
(var x "What a day!")
x          -> "What a day!"
(quote x) -> x
'x         -> x  ; Short form.
```

(apply fun . args): Apply function.

Calls function FUN. Unlike the rather straightforward FUNCALL, which takes its arguments as provided, APPLY expects the last element of ARGS to be a list, which is then appended to the previous elements:

```
(fn list x
  x)

(apply list '(10 11))   -> (10 11)
(apply list 1 2 '(3 4)) -> (1 2 3 4)
```

The reason for this is that it takes away the need to do that kind of concatenation oneself repeatedly, which would happen a lot otherwise.

(funcall f +x): Call function.

Basically calls function F with the list of arguments X.

```
; Basically the same:
(funcall 'print "Hello world!")
(print "Hello world!")

(funcall (?
           (eq color 'red)    print-red
           (eq color 'yellow) print-yellow
           (eq color 'green)  print-green)
         "Hello world!")
```

(eval x): Evaluate expression.

Evaluates expression X and it's subexpressions.

(? x +x): Conditional evaluation

Returns the second argument if the first one evaluates to non-NIL. Otherwise the process is repeated starting with the third argument, unless there is only one argument left which is then the default.

```
(? nil
   1)   -> nil
(? nil
   1
   2)   -> 2
(? nil
   1
   2
   3)   -> 3
(? t
   1
   2)   -> 1
(? t)   -> nil
```

(and +x)

Evaluates all arguments in order unless one evaluates to NIL. The value of the last evaluation is returned.

```
(and 1 2 nil) -> nil
(and 1 2)     -> 2
```

(or +x)

Evaluates all arguments unless one evaluates to non-NIL. The value of the last evaluation is returned.

```
(or 1 nil) -> 1
(or nil 2) -> 2
```

(block name . body), (return x block-name), (go tag)

Evaluates the list of expressions in BODY, returning the value of the last unless a RETURN from the block has been initiated. The name of the block passed to RETURN has to match. It is NIL, if not specified.

```
(block foo
  'a
  (return 'b foo)
  'c) -> b
```

Blocks of name NIL are used for loops. For the purpose of just butting up expressions use T instead to make RETURNs for name NIL drop through.

```
(macro progn body
  $(block t ; We don't want to catch returns.
     ,@body))
```

BLOCK also handles jumps initiated by GO. A jump destination, the “tag”, must be the same symbol passed to GO unquoted. It is an error if the tag cannot be found in any of the parent blocks in the current function. If no expression follows the tag, NIL is returned.

```
; Print "1" and "3".
(block nil
  (print 1)
  (go jump-destination)
  (print 2)
  jump-destination
  (print 3))
```

Equality

Function	Description
(eq a b)	Test if objects are the same.
(eql a b)	Test if numbers are the equal or EQ.
(equal a b)	Test if trees are EQL.

(eq a b): Test if objects are the same.

Tests if two objects are the very same.

Numbers usually are not as they are not looked-up for reuse like symbols. Use EQL instead.

(eql a b): Test if numbers are the equal or EQ.

Like EQ except for numbers: their true values are compared using function == instead.

(equal a b): Test if trees are EQL.

Like EQL but traversing down conses, allowing to compare lists and trees (lists of lists).

Predicates

Function	Test on…
(not x)	NIL
(atom x)	not a cons
(cons? x)	cons
(symbol? x)	symbol
(number? x)	number
(builtin? x)	built-in function
(special? x)	special form

All predicates except NOT and SYMOL? return their argument instead of T when true.

TODO: Impressive example where it's advantagous.

(not x): NIL

Returns T on NIl and NIL otherwise.

(atom x): not a cons

Returns its argument if it's an atom, except for NIL for which T is returned.

(cons? x): cons

Returns its argument if it is a cons, NIL otherwise.

(symbol? x): symbol

Returns its argument if it is an atom. T is returned for NIL. And NIL is returned for conses.

(number? x): number

Returns its argument if it is a number or NIL otherwise.

(builtin? x): built-in function

Returns its argument if it is a built-in or NIL otherwise.

(special? x): special form

Returns its argument if it is a special form or NIL.

Symbols

Function	Description
(symbol l)	Make symbol with name from char list.
(= 's x)	Set symbol value.
(value s)	Get symbol value.

(symbol l): Make symbol with name from char list.

(= 's x): Set symbol value.

(value s): Get symbol value.

This is what evaluation is doing with symbols.

Conses

Function	Description
(car l)	Return first value of cons or NIL.
(cdr l)	Return second value of cons or NIL.
(setcar c x)	Set first value of cons.
(setcdr c x)	Set second value of cons.

A ‘cons’ points to two other objects, called ‘car’ and ‘cdr’ for historical reasons. They could also be called ‘first’ and ‘second’, ‘first’ and ‘rest’ or ‘head’ and ‘tail’. However: they are just two object pointers packed together.

(car l): Return first value of cons or NIL.

(cdr l): Return second value of cons or NIL.

(setcar c x): Set first value of cons.

Returns the cons.

(setcdr c x): Set second value of cons.

Returns the cons.

Lists

This functions are around because the interpreter needs them internally.

Function	Description
(length l)	Return length of list.
(@ f l)	Run elements through function.
(butlast l)	Copy list but not its last element.
(last l)	Return last cons of list.
(member x l)	Return list starting with X.
(remove x l)	Copy list except element X.

(butlast l): Copy list but not its last element.

```
1  (butlast '(1 2 3)) ; (1 2)
```

(last l): Return last cons of list.

Return the last cons of a list, not the object it contains.

```
(last '(1 2 3)) ; (3)
```

(length l): Return length of list.

```
(length nil)        ; 0
(length '(1 2 3)))  ; 3
```

(member x l): Return cons containing X.

```
(member 'b '(a b c)) ; '(b c)
```

Uses EQ as the predicate, so numbers will most probably not be found..

```
(member 2 '(1 2 3)) ; NIL
```

Use MEMBER-IF to use EQL with numbers.

(remove x l): Copy list except element X.

```
(remove 'b '(a b c)) ; '(a c)
```

Uses EQ as the predicate, so REMOVE-IF must be used together with EQL to match numbers.

(@ f l): Filter list by function

```
(@ ++ '(1 2 3)) ; (2 3 4)
```

Also handles dotted pairs, filtering the last atom if it is not NIL.

```
(@ ++ '(1 2 . 3)) ; (2 3 . 4)
```

Numbers

Comparing

Function	Description
(== n n)	equal
(> n n)	greater than
(< n n)	less than
(>= n n)	greater than or equal
(<= n n)	less than or equal

Arithmetics

Function	Description
(+ n n)	Add numbers.
(- n n)	Subtract rest of numbers from first.
(* n n)	Multiply numbers.
(/ n n)	Divide first number by rest of numbers.
(% n n)	Modulo of numbers.

These operators take two arguments instead of variadic lists.[1]

Increment/decrement

Function	Description
(++ n)	Increment (add 1).
(– n)	Decrement (take 1).

```
1 (var x 23)
2 (++ x)      ; 24
3 (-- x)      ; 22
4 x           ; 23
```

[1] To be changing in the future.

Bit manipulation

Function	Description
(bit-and n n)	AND
(bit-or n n)	Inclusive OR.
(bit-xor n n)	Exclusive OR.
(bit-neg n)	Flip all bits.
(» n nbits)	Shift right.
(« n nbits)	Shift left.

I/O

Function	Description
(read)	Read expression.
(print x)	Print expression.
(load name)	Load and evaluate file.
(open name mode)	Open file and return channel.
(err)	Return number of last error or NIL.
(eof)	Tell if read reached end of file.
(setin n)	Set input channel.
(setout n)	Set output channel.
(in)	Read char.
(out x)	Print char or plain symbol name.
(terpri)	Step to next line.
(fresh-line)	Open line if not on a fresh one.
(close n)	Close a channel.

Variable	Description
last-in	Last input char.
last-out	Last output char.
fnin	Input channel number.
fnout	Output channel number.

(read): Read expression.

(print x): Print expression.

(open pathname mode): Open file and channel.

Opens file at PATHNAME for reading or writing. MODE must be a symbol. Returns the channel number or NIL.

Mode	Description
r	Read mode
w	Write mode

Illegal modes cause an ERROR_FILEMODE.

(err): Return number of last I/O error or NIL.

(eof): Tell if last read reached end of file.

(setin channel): Set input channel.

(setout channel): Set output channel.

(in): Read char.

(out x): Print char or plain symbol name.

(terpri): Step to next line.

(fresh-line): Open line if not on a fresh one.

(close channel): Close a channel.

(load pathname): Load and evaluate file.

Time

Constant	Description
+bps+	Number of bekloppies per second.

Variable	Description
start-time	Number of bekloppies at start.

Function	Description
(time)	Current bekloppie count.

Constant +bps+ contains the number of bekloppies per second.

On CBM machines it's currently set to 50, no matter if it's a NTSC or PAL machine. Please contribute some auto-detection to file "cbm-common.lisp".

On Unices +BPS+ is 1000 and counting starts with launching TUNIX Lisp.

Function TIME return the current count of bekloppies. It depends on the actual machine TUNIX Lisp is running on when the counting from 0 started.

Raw machine access

Function	Description
(rawptr x)	Get address of object in memory.
(peek a)	Read byte from memory.
(poke a b)	Write to memory.
(sys a)	Calls machine code subroutine.

(rawptr a): Read byte from memory.

(peek a): Read byte from memory.

(poke a b): Write to memory.

(sys a): Calls machine code subroutine.

Example: Print memory dump

```
1  (fn hexdump (from len)
2    (dotimes (j (max 1 (/ len 8)))
3      (let nrow (min len 8)
4        (dotimes (i (min len 8))
5          (outhex (peek from) 2)
6          (= from (++ from)))
7        (= len (- len nrow)))
8      (terpri)))
```

Error handling

Function	Description
(quit ?x)	Return from debugger REPL
(exit)	Stop program and go to top-level REPL.
(error x)	Issue a user error.
(onerror n x x)	User-defined error handler.
(ignore)	Break and continue with LOAD or REPL.
(debug)	Raises a SIGTRAP signal for debugging.
(debugger)	Invoke debugger with next instruction.

(quit ?x): Return from debugger REPL.

(exit): Stop program and go to top-level REPL.

(error x): Issue a user error.

(onerror n x x): User-defined error handler.

(ignore): Break and continue with LOAD or REPL.

(debug): Raises a SIGTRAP signal for debugging.

(debugger): Invoke debugger with next instruction.

Quasiquoting

Form		Description
(quote x)	’x	Return X unevaluated.
(quasiquote x)	$x	Unevaluated if not unquoted.
(unquote x)	,x	Insert into QUASIQUOTE.
(unquote-splice x)	,@x	Splice into QUASIQUOTE.

(quote x) | ’x: Return X unevaluated.

(quasiquote x) | $x: Unevaluated if not unquoted.

(unquote x) | ,x: Insert into QUASIQUOTE.

(unquote-splice x) | ,@x: Splice into QUASIQUOTE.

Macro system

Variable	Description
macros	Simple list of macro names.

Function	Description
(macro s a ?+x))	Add macro function to *macros*.
(macro? x)	Test if symbol is in *macros*.
(macroexpand x)	Expand expression.

A macro is a special form which is replaced by the expression it returns during 'macro expansion'. Macros are expanded in the REPL between READ and EVAL, if the value of MACROEXPAND is a user-defined function.

Defined macros are kept in associative list *MACROS*. Predicate MACRO? checks if a symbol is a defined macro in that list.

Macros are defined with special form MACRO:

```
; Evalue list of expressions BODY and return the result
; of the last.
(macro progn body
  $(block t
     ,@body))
```

Macros can also be used inside other macros, so contructs of increasing complexity can be made from simpler components.

```
; Evalute BODY if CONDITION is true.
(macro when (condition . body)
  $(? ,condition
      (progn
        ,@body)))
```

```
6
7  ## (macro s a +x)): Define macro.
8
9  Special form, adding macro S with arguments A and body B to
10 \*MACROS\*.
11
12 ## (macro? x): Test if symbol is in \*macros\*.
13
14 Predicate to test if a symbol denotes a macro in \*MACROS\*.
15
16 ## (macroexpand x): Expand expression.
17
18 # Environment
19
20 The environment contains a widely accepted set of functions
21 and macros known from most other implementations of the Lisp
22 programming languages.
23
24 ## Local variables
25
26 | Macro           | Desscription                        |
27 |-----------------|-------------------------------------|
28 | (let n init +b) | Block with one local variable.
29 | (with inits +b) | Block with many local variables.
30
31 ### (let n init +b): Block with one local variable.
32
33 ### (with inits +b): Block with many local variables.
34
35 ## Control flow macros
36
37 | Macro                | Description                      |
38 |----------------------|----------------------------------|
39 | (prog1 +b)           | Return result of first.          |
40 | (progn +b)           | Return result of last.           |
41 | (when cond +b)       | Evaluate if condition is true.   |
42 | (unless cond +b)     | Evaluate if condition is false.  |
43 | (while (cond x) +b)  | Loop while condiiton is true.    |
44 | (dolist (i init) +b) | Loop over elements of a list.    |
45
46 ### (prog1 +b): Return result of first.
47
48 ### (progn +b): Return result of last.
49
50 ### (when cond +b): Evaluate if condition is true.
51
52 ### (unless cond +b): Evaluate if condition is false.
53
54 ### (while (cond x) +b): Loop while condiiton is true.
55
56 ### (dolist (i init) +b): Loop over elements of a list.
```

```
57
58  ## Lists
59
60  | Function      | Description                          |
61  |---------------|--------------------------------------|
62  | (list +x)     | Return list evaluated.               |
63  | (list? x)     | Test if argument is NIL or a cons.   |
64  | (cadr l)...   | Nested CAR/CDR combinations.         |
65  | (carlist l)   | Get first elements of lists.         |
66  | (cdrlist l)   | Get rest elements of lists.          |
67  | (copy-list x) | Copy list.                           |
68  | (copy x)      | Copy recursively.                    |
69  | (find x l)    | Find element X in list.              |
70
71  ### (list +x): Return list evaluated.
72
73  ### (list? x): Test if argument is NIL or a cons.
74
75  ### (cadr l)...: Nested CAR/CDR combinations.
76
77  ### (carlist l): Get first elements of lists.
78
79  ### (cdrlist l): Get rest elements of lists.
80
81  ### (copy-list x): Copy list.
82
83  ### (copy x): Copy recursively.
84
85  ### (find x l): Find element X in list.
86
87  ## Loops
88
89  | Macro                        | Description              |
90  |------------------------------|--------------------------|
91  | (dolist (iter init) . body) | Loop over list elements. |
92  | (dotimes (iter n) . body)   | Loop N times.            |
93
94  ### (dolist (iter init) . body): Loop over list elements.
95
96  ### (dotimes (iter n) . body): Loop N times.
97
98  ## Stacks
99
100 | Macro      | Description                      |
101 |------------|----------------------------------|
102 | (push x l) | Destructively push onto stack L. |
103 | (pop l)    | Destructively pop from stack L.  |
104
105 Stacks are lists to which elements are pushed to or popped
106 off the front.
107
```

```
### (push x l): Destructively push onto stack L.

~~~lisp
(var x '(2))
(push 1 x) ; '(1 2)
x ; '(1 2)
```

(pop l): Destructively pop from stack L.

```
(var x '(1 2))
(pop 1 x) ; '(1)
x ; '(2)
```

Queues

Function	Description
(make-queue)	Make queue.
(enqueue c x)	Add object X to queue C.

(make-queue): Make queue.

(enqueue c x): Add object X to queue C.

Sets

Function	Description
(unique x)	Make list a set.
(adjoin x set)	Add element to set.
(intersect a b)	Elements in both.
(set-difference a b)	B elements that are not in A.
(union a b)	Unique elements from both.
(set-exclusive-or a b)	Elements that are not in both.

Function	Description
(subseq? a b)	Test if A is subset of B.

(unique x): Make list a set.

(adjoin x set): Add element to set.

(intersect a b): Elements in both.

(set-difference a b): B elements that are not in A.

(union a b): Unique elements from both.

(set-exclusive-or a b): Elements that are not in both.

(subseq? a b): Test if A is subset of B,

Associative lists

Function	Description
(acons alist c)	Add key/value to associative list.
(assoc x l)	Return list that start with X.

A list of lists where the first element of each list is the key and the rest is the value.

(acons alist c): Add key/value to associative list.

(assoc x l): Return list that start with X.

Target information

Constant +TARGET+ identifies the target machine, which is one of:

- c128

- c16
- c64
- pet
- plus4
- unix
- vic20

Optional features

Compressed conses

When enabled by compile-time option COMPRESSED_CONS, storing the CDR of a cons can be spared if that is following immediately on the heap. Since that makes compressed conses immutable (you cannot use SETCDR on them), compression is performed if the garbage collector was called by the program and not the allocator. The GC is also called to compress conses if the available heap left is smaller than the number of bytes specified by compile-time option GC_AFTER_LOAD_THRESHOLD, which is 2048 by default.

If you want to make sure that all conses that can be are compressed you have to call the garbage collector twice.

Compile-time option VERBOSE_COMPRESSED_CONS is set, the GC will print a ‘C’ to the currently active output channel.