(((()))) 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, and 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 compatible.

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, but can be cleared up quickly:

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 printing. Then it starts over, prompting you for the next expression. It also processes input from files and helps debugging by providing convenient one-character commands. REPLs can be nested, e.g. when an error occured, a new REPL is launched in debug mode, where you can provide a replacement for a faulty expression, execute code step by step and examine symbols.

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 to return to the operating system TUNIX Lisp is running on.

Definiton of permanent symbols

FN and VAR assign expressions to a symbol which is then added to the universe (a list of symbols the garbage collector is starting off with). The difference between FN and VAR is that VAR evaluates its initialization argument.

```
; Define permanent, named function.
(fn welcome ()
  (out '"Hello World!")
  (terpri))

; Define permanent, named variable.
(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 on error unless ONERROR has been defined.

The debugger shows a description of the error, followed by the current top-level expression, and the erroraneous expression emphasized within it and the prompt, indicating wait for input. Here's an example, triggering an error by trying to call the undefined function CAUSE-ERROR:

```
1 * (some-undefined-function)
2 Debugger #1:
3 Error #5: Not a fun.
4 In :
5 (>>> some-undefined-function <<<)
```

The debugger takes commands like the regular REPL, e.g. "(print x)" will print evaluated X in the current context. It also knows single-character commands to make your life easier:

Command	Description
c	Continue program execution.
s	Step into user-defined procedure.
n	Execute current expression in whole.
pX	Evaluate and print expression X.

Debugger short commands

User-defined error handler ONERROR

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

Error handling related functions

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 expression or body of the user-defined function thas is evaluated, and the expression inside it is faulty.

```
; SKETCH! UNTESTED!
; Load missing functions on demand.
(fn onerror (n repl x)
  ; n:    Error code
  ; repl: Top-level expression or
  ;       body of user-defined function.
  ; x:    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))
              nil
         (!= (read)
           (when (and (cons? !)
                      (or (eq (car !) 'var)
                          (eq (car !) 'fn))
                      (eq (cadr !) x))
             (eval !)
             (return x)))))))
```

Error codes

ID (ERR_…)	Code	Description
TYPE	1	Unexpected object type.
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.
OUT_OF_HEAP	6	Out of heap. Cannot catch.

ID (ERR_…)	Code	Description
UNKNOWN_TYPE	7	Internal error.
NO_PAREN	8	‘)’ missing.
STALE_PAREN	9	Unexpected ‘)’.
CHANNEL	10	Channel not open.
USER	12	ERROR function was called.
INTERNAL	13	Internal interpreter error.

Built-in functions

General

Function	Description
(universe)	Return list of permanent symbols.
(gc)	Free unused objects.
(exit ?n)	Exit program or interpreter with code.

(universe): Return list of permanent symbols.

(gc): Free unused objects.

Triggers the garbage collector. It marks all objects linked to the 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.

Form	Type
(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.

Function	Description
(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.

Function	Description
(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.

```
(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]

[1] To be changing in the future.

Increment/decrement

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

```
(var x 23)
(++ x)      ; 24
(-- x)      ; 22
x           ; 23
```

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.
(open pathname)	Open file and return channel.
(err)	Return number of last error or NIL.
(eof)	Tell if read reached end of file.

Function	Description
(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.
(load pathname)	Load and evaluate file.

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): Open file and channel.

(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.

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

```
(fn hexdump (from len)
  (dotimes (j (max 1 (/ len 8)))
    (let nrow (min len 8)
      (dotimes (i (min len 8))
        (outhex (peek from) 2)
        (= from (++ from)))
      (= len (- len nrow)))
    (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.

Compiler

The simplicity, flexibility and transparency of a Lisp interpreter is cutting edge in every aspect. TUNIX Lisp is a bit more complicated as 16-bit indexed pointers and stacks larger than 256 bytes are required on 8-bit CPUs like the MOS 6502 and the required code space to handle the two bytes of a 16-bit word is eating away on precious heap size and performance. The situation can be improved a little for error-free applications by removing all sorts of error checking through setting the compile-time option “NAIVE”, but it still takes a working development to get an application to be almost free of errors. Achieving that is next to impossible in the first place due to the complexity inherent to computer programs.

The target machine: Bytecode format

Offset	Size	Description
0	1	Object TYPE_BYTECODE
1	1	Total size - 3
2	2	Argument definition
3	1	Local stack size / object list size
4	1	Code offset
5	?	Data (garbage-collected objects)
<5	?	Raw data
<5	1-220	Code

A bytecode function has a memory layout similar to that of a symbol, with a type, length and value slot. The length tells the number of bytes following. The value points to a regular argument definition list , e.g. ‘(first . rest)’. Addtionally the size of the local stack frame, which is subtracted from the stack pointer upon function entry, is given, followed by the length of a GCed object list and the size of a raw data array which contains jump destination offsets. All that is followed by the actual bytecode.

Instruction format

The highest bit of the first byte determines if the code is an assignment or a jump. Jumps also contain an index into a code offset table.

Jumps

```
1JJIIIII
```

- J: type
- 00: unconditional
- 01: If %0 is not NIL.
- 10: If %0 is NIL.
- 11: unused
- I: Index into raw data table

Return from function

```
00000000
```

Assignment:

Assigments include a destination on the stack, a function object and the arguments which are either on the object list or on the stack..

```
0?DDDDDD FFFFFFFF
```

- D: Destination on stack
- F: Function (index into object array)

Arguments:

Arguments are either indexes into the object array or into the stack.

```
EPIIIIII
```

- E: End of argument list flag
- P: 0: stack place 1: object
- I: Index into stack or object array

Passes

The TUNIX compiler has a micro-pass architecture, requiring only workspace for the current pass. It translates macro-expanded input into an assembly-style metacode made of assignments, jumps and jump tags. Function information is also gathered for the following optimization and code generation passes.

Transform to metacode:

- Compiler macro expansion
- Quote expansion
- Quasiquote expansion
- Argument renaming
- Function collection
- Scoping
- Lambda expansion
- Block folding
- Expression expansion

After that transformation the resulting code has to be cleaned from macro artifacts.

Cleaning up at least:

- Optimization

Finally the desired code can be generated, e.g. byte code or assembly language.

Code generation:

- Place expansion
- Code macro expansion

Compiler macro expansion

Expands control flow special forms (BLOCK, GO, RETURN, ?, AND, OR) to these assembly-level jump and tag expressions:

Metacode	Description
(%= d f +x)	Call F with X and assign result to D.
(%jmp s)	Unconditional jump.

Metacode	Description
(%jmp-nil s)	Jump if %0 is NIL.
(%jmp-t s)	Jump if %0 is not NIL.
(%tag s).	Jump destination.

Jump tags must be EQ.

Expansion of ?, AND and OR

```
(? (a)
   (b)
   (c)

(%= %0 a)
(%jmp-nil 1)
(%= %0 b)
(%jmp 2)
(%tag 1)
(%= %0 c)
(%tag 2)
```

BLOCK expansion

The BLOCK expander need to expand child blocks first so that deeper RETURNs with a clashing name have precedence:

```
(block nil
  ...
  (block nil
    ...
    (return nil)  ; Must return from the closer BLOCK NIL.
    ...))
```

By translating deeper BLOCKs' first, RETURN statements are resolved in the correct bottom-up order.

The last expression of a BLOCK always assigns to %O which is synonymous for return values from then on.

Block folding

BLOCKs have been expanded to %BLOCKs to hold the expressions together for this pass. They are now spliced into each other to get a single expression list for each function.

```
; Input code
(when (do-thing? x)
  (do-this)
  (do-that))

; After MACROEXPAND.
(? (do-thing? x)
   (block t
     (do-this)
     (do-that)))

; After COMPILER-MACROEXPAND.
(%= %0 (do-thing? x))
(%jmp-nil 1)
(%block
  (= %0 (do-this))
  (= %0 (do-that)))
(%tag 1)

; After BLOCK-FOLD.
(%= %0 (do-thing? x))
(%jmp-nil 1)
(do-this)
(%= %0 (do-that))
(%tag 1)
```

Quote expansion

Quotes are entries on the function's object list.

Quasiuote expansion

QUASIQUOTEs need to be compiled to code to apply LIST and APPEND instead.

```
; From:
$(1 2 ,@x 4 5)

; To:
(append '(1 2) x '(4 5))
```

Function info collection

Creates function info objects with argument definitions. They are used by optimizing and code generating passes and, intially, are as simple as this:

```
(fn funinfo ()
  (@ list '(args)))
```

When extending the compiler, most things will revolve around this pittoresque, little thing as we'll see later.

Argument renaming pass

The following lambda-expansion might need to inline functions with argument names that are already in use. This pass solves that issue by renaming all arguments. [1]

```
; TODO example of shadowed arguments that would clash on
; a common list.
```

Lambda expansion

Inlines anonymous functions and moves their arguments to the FUNINFO of the top-level function, which is laying out the local stack frame in the process.

```
; TODO example of anonymous function first in expression.
```

Expression expansion

Breaks up nested function calls into a list of single statement assignments. After this the return value of any expression is in variable %0.

This expression

```
(fun1 arg1 (fun2 (fun3) (fun4)) (fun5))
```

becomes:

```
(%= %1 fun3)
(%= %2 fun4)
(%= %3 fun2 %1 %2)
(%= %4 fun5)
(%= %0 fun1 %0 %3 %4)
```

[1] A map of the original names must be created if debugging is in order.

Argument expansion

Checks arguments and turns rest arguments into consing expressions.

Optimization

Basic compression of what the macro expansions messed up at least, like remove assignments with no effect or chained jumps.

Place expansion

Expression	Description
(%S offset)	Offset into local stack frame.
(%D offset)	Offset into function data.

Here the arguments are replaced by %S or %O expressions to denote places on the stack or on the function's object list.

Code expansion

These are actually two passes:

- Collecting objects.
- Calculating jump destinations.

Internals

Heap object layouts

You can get the memory address of any object with RAWPTR. You can then use it to PEEK and POKE memory directly.

On 32-bit and 64-bit architecture pointers and numbers are four or eight bytes in size. The following tables show the layouts for 16-bit systems.

All objects start with a type byte:

Bit	Description
0	Type bit for conses
1	Type bit for numbers
2	Type bit for symbols
3	Type bit for built-in functions
4	Extended type (for symbols)
5	Unused
6	Unused
7	Mark bit for garbage collection

Cones

Offset	Description
0	Type info (value 1)
1-2	CAR value

Offset	Description
3-4	CDR value

Numbers

Offset	Description
0	Type info (value 2)
1-4	Long integer[1]

Symbols

Offset	Description
0	Type info (value 4 or 20)
1	Name length (0-255)
2-3	Pointer to next symbol for look-ups
4-x	Name (optional)

Adding the extended type bit turn a symbol to a special form. Arguments its function won't be evaluated then.

Built-in functions

Built-ins are symbols with a pointer to a descriptor of the built-in. It contains an ASCIIZ pointer to the name, another to the character-based argument definition and the address of its implementation.

Offset	Description
0	Type info (value 8)

[1] Will occupy eight bytes on 64-bit systems.

Offset	Description
1-2	Symbol value
3-4	Pointer to next symbol for look-ups
5	Name length (0-255)
(6-x)	Name (optional)

Ideas for the future

User-defined setters

```
(setcar x v)
(= (car x) v)
```

Multi-purpose operators

'+' to also append lists

```
; Same:
(append a b)
(+ a b)
```

Objects

Using SLOT-VALUE and abbreviating dot notation on associative lists. ASSOC should be a built-in function to avoid performance issues.

?: How about immutables?

Directory access

Function	Description
(mkdir s)	Create directory.
(opendir n s)	Open directory on channel.
(readdir n)	Read directory info.

Function	Description
(writedir n)	Write partial directory info.

(mkdir s): Create directory.

(opendir n s): Open directory on channel.

(readdir n): READ directory info.

(writedir n): Write partial directory info.

Exceptions

Catch stack.

Real-time applications

Interruptible GC with lower threshold to keep space for critical operations is a bad idea as a GC has to complete at some point.

Function	Description
(gc x)	GC with another root object.

GC could take an optional argument to specify another root than the universe to discard everything that is not part of an app.

```
(gc 'appstart)
; Put APPSTART back in the universe.
(var appstart appstart)
```

Or one could save the current list of definitions and throw everything out that appeared later:

```
(var *old-defs* (universe))
(load "app.lisp")
(gc *old-defs*)
```

Bielefeld DB

Function	Description
(db-open a)	Open database.
(db-add s x)	Add expression with string key.
(db-find s)	Find ID by key.
(db-read n)	READ by ID.
(db-close n)	Close database.

(db-open a): Open database.

(db-add s x): Add expression with string key.

(db-find s): Find ID by key.

(db-read n): READ by ID.

(db-close n): Close database.

Embedded database to the rescue the day for large data sets.

Defining built-ins

Function	Description
(mkbuiltin s a)	Add built-in function.

Submit to your fantasy.

Compressed lists

Conses which only store the CAR if the CDR is the next object on the heap. This can be done at allocation time but would make the list's CDRs immutable.

The disadvantage is that extra checks are required to access a CDR.

Fragmented heap

To support static memory allocation, e.g. for native code.

Processes

Sharing the heap they forked off from. With pipelining.

Wanted

- Math lib for lists of decimals of arbitrary length.

Glossary

Anonymous Function

A function without a name, often used as a parameter to higher-order functions. These functions do not require quoting when used directly but must be quoted when passed as arguments.

Argument Definition

The specification of parameters that a function, macro or special form takes. Arguments can be defined with character codes indicating their types and may include prefixes for optional or unevaluated arguments.

Built-in Function

A function that is implemented within the Lisp interpreter itself rather than being defined by the user. These functions typically offer basic operations and access to system-level features.

Bytecode A form of intermediate code that is more

abstract than machine code but less abstract than high-level source code. TUNIX Lisp compiles functions into bytecode for efficient execution.

Car and Cdr

Historical terms referring to the first and second elements of a cons cell, respectively. `car` returns the first element, and `cdr` returns the second element of a cons cell.

Channel

An abstraction for input/output operations, allowing for switching between different streams of data. Channels are managed using functions like `setin` and `setout`.

Cons Cell

A fundamental data structure in Lisp, consisting of two parts: the `car` and the `cdr`. Cons cells are used to build lists and other complex data structures.

Dot Notation

A syntactical convenience in TUNIX Lisp for accessing elements of lists and objects. For example, `x.y` is shorthand for `(slot-value x 'y)`.

Garbage Collection

The process of automatically identifying and reclaiming memory that is no longer in use by the program. TUNIX Lisp uses a compacting mark-and-sweep garbage collector.

Heap

A region of memory used for dynamic allocation of objects. The heap grows as new objects are created, and garbage collection is triggered when memory runs low.

Interpreter

A program that executes instructions written in a programming language without requiring them to be compiled into machine code. TUNIX Lisp includes an interpreter for running Lisp code.

Lambda

A keyword used in many Lisp dialects to define anonymous functions. In TUNIX Lisp, the keyword is omitted, and function expressions are quoted instead.

List

A sequence of elements, typically linked together using cons cells. Lists are a primary data structure in Lisp and can be manipulated using various built-in functions.

Mark-and-Sweep

A garbage collection algorithm that marks active objects and sweeps away those that are not reachable from the root set. TUNIX Lisp uses a compacting version of this algorithm.

Quasiquote

A feature that allows for partially quoted expressions, enabling easy construction of lists with evaluated and unevaluated parts. In TUNIX Lisp, quasiquote is represented by the dollar sign `$`.

Procedure

A general term that refers to any callable entity, including functions, macros, and special forms. Procedures are fundamental building blocks in Lisp, allowing for modular and reusable code.

REPL (Read-Eval-Print Loop)

An interactive programming environment that reads user input, evaluates it, prints the result, and then waits for more input. The REPL is a key feature of the Lisp programming experience.

Special Form

A construct that is evaluated in a unique way, often involving special syntax or behavior that cannot be replicated by regular functions. Examples include `quote`, `?`, and `lambda`.

Symbol

A basic data type in Lisp used to represent identifiers. Symbols can have names and values, and are often used as variable names. In TUNIX Lisp symbols are also used as strings.

Universe

The root set of all symbols known to the garbage collector. This list of symbols is where garbage collection begins, ensuring that all active objects are retained.

Variable

An identifier associated with a value. Variables can be defined using the `var` special form and can be assigned new values using the = function.