LISP FROM NOTHING

LISP FROM NOTHING

NILS M HOLM

LISP From Nothing

Nils M Holm, 2020

Cover art, photographs, and illustrations by Nils M Holm

Print and Distribution:
Lulu Press, Inc.; Morrisville, NC; USA

For my mother.

CONTENTS

PREFACE

LISP has been a playground for tinkerers from the point of its inception more than 60 years ago. From the beginning it has attracted people who where fascinated by elegance and beauty. The first sketches of LISP interpreters, which will be displayed in this book, were works of art as much as they were mathematical models and practical programs.

Even when LISP evolved into a family of languages with “serious” applications in mind, it has always kept its playfulness. Questions like

What is the minimal LISP?
What is needed to implement LISP?
What are the exact semantics of LISP?

keep popping up in its wake. Lots of interesting books and papers, like [Baker1992], have explored these topics in great depth. Some features have been added to LISP, other have been deprecated. Different paths were taken, with SCHEME and COMMON LISP reflecting the extremes of minimalism and pragmatism.

At their core, though, there remains a small, common language that is LISP. Even if that core has diversified, COMMON LISP being a multi-namespace LISP and SCHEME a single-namespace one, the ideas remain widely compatible. Many people who use LISP enjoy thinking about this minimal core. However:

What exactly is the minimal LISP?

Most people would agree that it involves conses and atoms, the principle of the identity of atoms, anonymous functions, and conditional evaluation. Or, more LISPy:

CONS CAR CDR QUOTE ATOM EQ LAMBDA COND

It has been shown, over and over again, that this set of functions is basically sufficient to write a *metacircular* (self-interpreting) LISP [McCarth1981]. Sometimes additional forms like LABEL are added, but they are—strictly speaking—not necessary, as will be shown in this book.

On the more extreme side, some people claim that LAMBDA alone is enough to implement LISP, because LISP is based on lambda calculus [Church1941], and lambda calculus is a Turing-complete system. We shall see that this is not the case—even though a *lot* of the semantics of LISP can be expressed by function abstraction and application exclusively [Holm2016].

Then the superficial simplicity of metacircular interpreters glosses over some not-so-beautiful but necessary details; details like input and output (READ, PRINT), garbage collection, etc. Metacircular interpretation takes these as given and excludes them from the discussion. Without these preexisting facilities, though, what is the minimal bootstrappable LISP?

Can a compiler for LISP be written in the minimal dialect of LISP that is often chosen to demonstrate metacircular interpretation, a dialect of LISP that has atoms and lists as its only data types? Which additional functions and special forms are required? Can we get away without using numbers or strings?

Of course, we can! The remainder of this book will illustrate and develop several implementations of minimal LISP. It will start with McCarthy's first metacircular interpreter and then define a set of additional functions that will serve as a basis for bootstrapping a complete LISP system—complete with functions like ASSOC, MAPCAR, READ, PRINT, etc.

After bootstrapping the LISP system that system will be used to implement a compiler for the extended minimal LISP. Using the compiler, a more complete LISP interpreter can then be built, and the interpreter can be extended with a macro expander, hence

allowing to define special forms in LISP programs. This is what one of the later chapters of the book will demonstrate.

The discussion will usually go to great depths and explore all kinds of design decisions and implications. It will answer questions like

- Why is symbol identity expensive?
- Why is there sometimes no alternative to global variables?
- How does dynamic scoping interfere with tail recursion?
- Can a garbage collector be written in LISP?
- Why can (or cannot) LAMBDA bind recursive functions?

While discussing all these aspects of LISP, the book will also shed some light on what hacking was like in the early days of computing. If you enjoy hearing about magnetic core memory, punch cards, and hard-copy terminals, there might be some interesting trivia ahead!

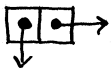
As usual, all code shown in the book (except for the most trivial examples) can be found on my homepage, **`t3x.org`**.

Enjoy the tour!

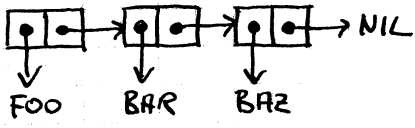
Nils M Holm, October 2020

INTENDED AUDIENCE

This is not an introduction to LISP. You will definitely enjoy this book more, if you know what an S-expression is, what



means, and how



is related to (FOO BAR BAZ). If you know what a compiler and an interpeter is and what they do, even better!

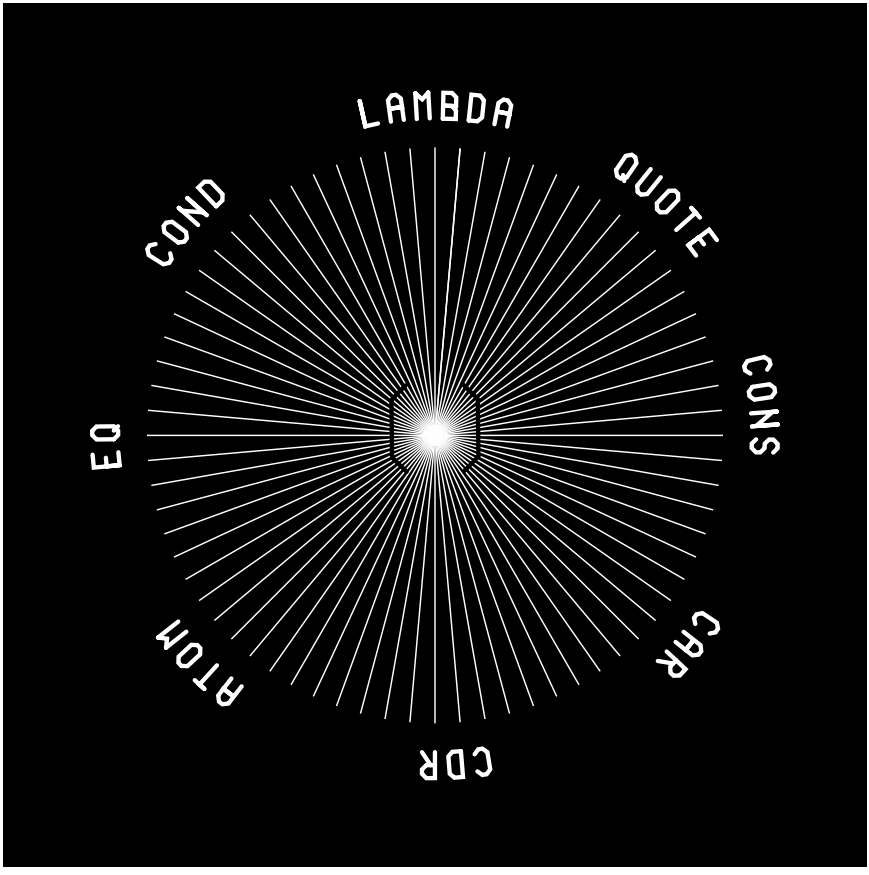
If you do not know any of the above and still want to read the book, go ahead, but be prepared to stumble across a lot of unfamiliar jargon! There is an introduction to purely symbolic LISP (page 241) and a glossary (page 249) in the appendix, for those who are really determind.

ACKNOWLEDGEMENTS

The following people contributed substantially to the formation of the human being that I am today and hence to the creation of this book. I am very grateful for the privilege of knowing you or having known you! Thanks to:

My mother, Hanne, my father, Günter †, and my aunt Gitty for making my childhood a pleasant one. My grandparents, Franz † and Lilo †, for being decent human beings and rôle models. Andrea for being my friend and inspiring me to become a vegetarian. Britta for being my first crush. Matthias Carlos José Maria for sharing my interest in trashy science fiction and letting me hang out at his place. Torsten S for keeping in touch over the decades. Andreas † for letting me play with the computers in his shop. Torsten L for marrying the girl I was into. Matthias for defending me when I slept in my office chair. Eric for sparring lessons and sharing cool demos on the computer. Benjamin and Tove for really caring about me. Dr. Benn for lending an ear. MJK for being an honest and helpful guy, even if his jokes can be a pain in the neck. Prof. Nagler for bringing me back to meditation practice. Nick for so many days spent together and being a loyal friend. Werner for lots of interesting discussions and his ability to agree to disagree. Gerrit for knowing that hypernormalization is all around us. My goddaughter Célina for sharing her interesting thoughts and being the sun in my sky.

Finally, my apologies to everyone who thinks they should be in this list, but are not! You are most probably right!



LET THERE BE LISP

WHEN LISP was first conceived in the years 1956 and 1957, it was a mathematical notation rather than a programming language, simply because there was not yet any existing implementation of the language. The notation used to write LISP programs on paper differed significantly from what LISP looks like today—and even from what it looked like two years later.

The *M-expression* notation (short for "*meta expression* notation") used square brackets for function application, function names outside of the brackets like ordinary mathematics, and some symbols that have their origin in the domain of mathematical logic. Programs were formulated differently. For instance, the AMONG function, which tests whether a list contains a specific S-expression, would be defined like this in the LISP 1 Programmer's Manual [McCarthy1960]:

```
among[x;y] =
  ~null[y] ∧ [equal[x;car[y]] ∨ among[x;cdr[y]]]
```

A literal translation to later LISP would look like this

```
(DEFINE ((AMONG (LAMBDA (X Y)
  (AND (NOT (NULL Y))
       (OR (EQUAL X (CAR Y))
           (AMONG X (CDR Y))))))))
```

but, of course, you would rather write it in the following way only a few years later:

```
(DEFINE ((AMONG (LAMBDA (X Y)
  (COND ((NULL Y) NIL)
        ((EQUAL X (CAR Y)) T)
        (T (AMONG X (CDR Y))))))))
```

In case you wonder, AMONG is indeed a lot like MEMBER, but just returns a truth value instead of a tail of the list argument Y. The focus shifted from mathematics to programming rather quickly in the first years of LISP, as can be seen in the early *LISP Programmer's Manuals*, and examples soon adopted a style that that can still be recognized today.

M-Expression	S-Expression
`foo`	`FOO`
`FOO`	`(QUOTE FOO)`
`foo[bar;baz]`	`(FOO BAR BAZ)`
`(FOO,BAR,BAZ)`	`(QUOTE (FOO BAR BAZ))`
`(FOO · BAR · BAZ)`	`(QUOTE (FOO BAR BAZ))`
`f[x] = y`	`(DEFINE ((F (LAMBDA (X) Y))))`
`λ[[x;y];z]`	`(QUOTE (LAMBDA (X Y) Z))` †
	`(FUNCTION (LAMBDA (X Y) Z))`
`~a`	`(EQ A NIL)`
	`(NOT A)`
`a ∨ b ∨ ···`	`(COND (A) (B) ...)`
	`(OR A B ...)`
`a ∧ b ∧ ···`	`(COND (A (COND (B) ...)))`
	`(AND A B ...)`
`[a → b; ···]`	`(COND (A B) ...)`
`a ∧ b ∨ c`	`(OR (AND A B) C)`
`a ∧ [b ∨ c]`	`(AND A (OR B C))`

Fig. 1 – M-expression and S-expression syntax († the FUNCTION operator is not present in LISP 1.)

M-expressions remained a popular pencil-and-paper notation for some time, though, even while programs were already submitted in *S-expression* ("*symbolic expression*") notation to the computer. Symbolic expression notation was originally used to represent

data, while M-expressions were intended to write programs. However, the first LISP interpreter accepted input in S-expression notation and that circumstance “froze” the language in a very early stage of its development [McCarthy1981].

Although M-expressions were never documented fully, a lot of examples are provided in the *LISP Programmer's Manuals* [McCarthy1960,1962], so a translation chart from M-expressions to S-expressions can be constructed from them (see figure 1). The chart uses LISP 1.5 syntax on the symbolic expression side. We will get back to that!

The following M-expression is an early specification of the EVAL function, which is the central part of the LISP interpreter. The expression is a corrected version of the one published in the first version of the LISP programmers manual [McCarthy1960]. The version in the manual contains a few trivial mistakes, like closing parentheses in wrong places, as well as one not-so-trivial one, which evaluates function arguments twice. This suggests that M-expressions were used in a rather informal way back in the days.

```
eval[x;e] =
  [atom[x] → assoc[x;e];
   atom[car[x]] →
     [eq[car[x];QUOTE] → cadr[x];
      eq[car[x];ATOM]
       → atom[eval[cadr[x];e]];
      eq[car[x];EQ]
       → eq[eval[cadr[x];e];
            eval[caddr[x];e]];
      eq[car[x];COND]
       → evcon[cdr[x];e];
      eq[car[x];CAR]
       → car[eval[cadr[x];e]];
      eq[car[x];CDR]
       → cdr[eval[cadr[x];e]];
```

```
    eq[car[x];CONS]
     → cons[eval[cadr[x];e];
            eval[caddr[x];e]];
    ;; The manual says evlis[cdr[x];e]
    ;; instead of cdr[x] below; see text
    T → eval[cons[assoc[car[x];e];
                  cdr[x]];e]];
eq[caar[x];LABEL]
  → eval[cons[caddar[x];cdr[x]];
         cons[list[cadar[x];car[x]];e]];
eq[caar[x];LAMBDA]
  → eval[caddar[x];append[pair[cadar[x];
         evlis[cdr[x];e]];e]]]
```

It is not remarkable that small errors creep up in pencil-and-paper code, especially since balancing square brackets in M-expressions appears to be harder than keeping track of parentheses in S-expressions (at least this is the experience of the author). Maybe this is another reason why S-expression notation gained so much popularity so quickly.

The above EVAL function is a straight-forward implementation of a minimal LISP system and with a few exceptions its inner working should be obvious to the present-day LISP programmer. There are some peculiarities, though.

(1) The ASSOC function of LISP 1 returned the value associated with a given key instead of the pair associating the key with the value, so ASSOC could be used to map atoms directly to their corresponding values.

(2) The PAIR function built a list of pairs from two existing lists by combining their elements pairwise. Note that a *pair* was a list of two elements in LISP 1 and *not* a *dotted pair*! Even the ASSOC function used a list of such “pairs” back then. A modern-day implementation of PAIR would look like this:

```
(DEFUN PAIR (A B)
  (MAPCAR (FUNCTION LIST) A B))
```

(3) LISP 1 was a *LISP-1* (note the dash!), i.e., a *single-namespace* LISP. Hence the MAPLIST function

```
maplist[x;f] =
  [null[x] → NIL;
   T → cons[f[x];maplist[cdr[x];f]]]
```

could be translated directly to the S-expression

```
(DEFINE ((MAPLIST (LAMBDA (X F)
  (COND ((NULL X) NIL)
        (T (CONS (F X) (MAPLIST (CDR X) F))))))))
```

and applied to values with

```
(MAPLIST (QUOTE CAR) (QUOTE (A B C)))
```

This would not be possible in later versions of LISP, which introduced multiple bindings per atom (e.g. for values and functions) and hence turned LISP into a LISP-N, a LISP with multiple namespaces. While the application would still work in the above way in modern LISP, the implementation of MAPLIST would need the FUNCALL operator in the application of F:

```
(FUNCALL F X)
```

Note that LISP 1.5 was a LISP-N, but did not have a FUNCALL operator! The LISP 1 version of MAPLIST still worked in LISP 1.5 [McCarthy1962], so LISP 1.5 had characteristics of both a LISP-1 and a LISP-N.

The EVCON and EVLIS functions used by EVAL implemented the evaluation of the COND special form and the evaluation of argument lists in function applications, respectively. They were part of the interpreter specification and not part of the LISP 1 language. Their implementations are given below.

```
evcon[c;e] =
  [eval[caar[c];e]
    → eval[cadar[c];e];
   T → evcon[cdr[c];e]]

evlis[x;e] =
  [null[x] → NIL;
   T → cons[eval[car[x];e];
            evlis[cdr[x];e]]
```

In the interpreter specification the argument x denotes the expression to be evaluated and e is the *environment* in which the function is to be evaluated. The environment is a list of pairs, as described above, suitable for submitting it to ASSOC. It supplies an initial set of *bindings* from *atoms* (*symbols*) to values.

The LABEL special form, as implemented by EVAL, was a clever construct for defining *anonymous recursive functions*. Anonymous functions ("*lambda functions*") cannot usually recurse (apply themselves), just because they are anonymous, so there is no name that can be used to invoke them. The LABEL form creates an anonymous function and invokes it *in situ* (on the spot) in an environment that contains its own definition, thereby allowing it to recurse. For instance, evaluating the expression

```
((LABEL FOO
   (LAMBDA (X)
     (COND ((NULL X) (QUOTE BAR))
           (T (FOO (CDR X))))))
 (QUOTE (A B C)))
```

in an environment e=NIL would result in the evaluation of

```
((LAMBDA (X)
   (COND ((NULL X) (QUOTE BAR))
         (T (FOO (CDR X)))))
 (QUOTE (A B C)))
```

in the environment

```
((FOO (LABEL FOO
        (LAMBDA (X)
          (COND ((NULL X) (QUOTE BAR))
                (T (FOO (CDR X))))))))
```

The mechanism can be emulated in modern LISP systems by using LETREC in SCHEME:

```
((letrec
   ((foo (lambda (x)
      (cond ((null? x))
            (else (foo (cdr x)))))))
   foo)
 (quote (a b c)))
```

or LABELS in COMMON LISP:

```
(FUNCALL
  (LABELS
    ((FOO (X)
       (COND ((NULL X) (QUOTE BAR))
             (T (FOO (CDR X))))))
    (FUNCTION FOO))
  (QUOTE (A B C)))
```

The minimal EVAL function presented in the LISP 1 manual interprets many simple LISP programs correctly, but it is merely a proof of concept or a formal specification, and not meant to be an actual implementation, as the following issues suggest:

(1) built-in functions, such as CONS and ATOM, are *special operators* and not functions, i.e. they must appear in the "function" position of a form. In all other positions they are undefined which means, in particular, that they cannot be passed to *higher-order functions*. The expression

```
((LAMBDA (F) (F (QUOTE (A)))) CAR)
```

would be undefined under the given implementation of EVAL. Quoting CAR would not help, either, because it is not bound in the environment e. Theoretically you could use *eta expansion* [Church1941], i.e. wrap a lambda function around the special operator, to make the above program work, though:

```
(QUOTE (LAMBDA (X) (CAR X)))
```

Note that lambda functions were not *self-quoting* unlike in many later single-namespace LISPs, like SCHEME [Scheme1975].

(2) Undefined expressions (like expressions with an undefined operator or a non-atomic S-expression in the operator position) cause an infinite loop in the interpreter. The operator is looked up in the environment, giving NIL. The operator is then replaced with NIL and the expression is resubmitted, causing the NIL in the operator position to be looked up in the environment, etc, ad infinitum.

However, one might very well argue that addressing these issues is optional in a formal specification and just assume that all

submitted programs are well-formed. Eta-expanded variants of the built-in functions could be supplied in the environment, thereby solving (1).

MINIMAL METACIRCULAR LISP

The LISP interpreted by EVAL understands the following operators in addition to the concept of function application:

CONS CAR CDR QUOTE ATOM EQ LAMBDA LABEL COND

With the exception of LABEL this is probably what most people would agree to be the most minimal implementation of LISP. Some people might argue that lists can be created in terms of LAMBDA, thereby eliminating CAR, CDR, CONS, and ATOM. However, this is only true, if lexical scoping is used, because a cons pair implemented by LAMBDA relies on closures that store the values of a pair internally. Closures were a rather esoteric concept back in the days of early LISP, though.

Note that neither QUOTE not EQ can be implemented in pure *lambda calculus* [Church1941] and hence not in LISP, either. So it is a myth that LISP can be implemented on top of LAMBDA alone—even if we assume the presence of lexical scoping. See [Holm2016] for an in-depth discussion of QUOTE. The short version is that there are no “constants” in lambda calculus, so QUOTE does not make any sense in it. The short version of the argument regarding EQ is this: how does the concept of *identity*, or “sameness”, apply to a system that has been designed to work on a sheet of paper? Of course, the lambda calculus form

$$(\lambda xx) \equiv (\lambda xx)$$

indicates that two terms are syntactically equivalent, but are they *the same*? It is impossible to say without a concept of identity, like locations in the memory of a computer.

LABEL can be replaced by LAMBDA, although in a cumbersome way (we will assume that lambda forms are self-quoting in the remainder of this text):

```
((LAMBDA (FOO)
   (FOO (QUOTE (A B C))))
 (LAMBDA (X)
   (COND ((EQ X NIL) (QUOTE BAR))
         (T (FOO (CDR X))))))
```

(Yes, this really works in a dynamically scoped LISP-1! We will get back to this detail; see page 197.)

Removing LABEL leaves us with the following set of operators for a minimal LISP system:

CONS CAR CDR QUOTE ATOM EQ LAMBDA COND

In addition, the system has to implement function application and recognize the symbol *NIL* as the "false" value and *T* as the canonical "true" value. This is usually done by supplying an association of the form (T T) (or (T . T)) in the initial environment. The symbol T then evaluates to itself while NIL evaluates to NIL, because there is no association for it.

Is this minimal LISP metacircular? I.e., is there an interpreter written with the above forms exclusively (plus function application) that can interpret itself? John McCarthy has provided a beautiful proof of concept in an addendum to the History of Programming Languages II proceedings [McCarthy1981]. Based on his work, here follows a metacircular interpreter for a minimal LISP that will even work on a modern COMMON LISP or SCHEME system.

The implementation shown here differs from the EVAL in the previous section in several ways.

(1) It is written in S-expression notation.

(2) It adds the operators CAAR, CADAR, CADDR, CADR, and CDAR for convenience. It should be obvious that this is a purely cosmetic addition because, for instance, any occurrence of (CADAR X) can be substituted by (CAR (CDR (CAR X))).

(3) It adds the reduction rule (LAMBDA ...) → (LAMBDA ...), i.e. lambda functions evaluate to themselves.

(4) It adds the reduction rule (NIL ...) → *UNDEFINED, i.e. forms with an undefined operator in the operator position reduce to a specific symbol named "*UNDEFINED".

(5) It implements T as a special symbol, so it does not have to be supplied in the initial environment.

(6) It uses LABEL, which should be understood to be syntactic sugar on top of LAMBDA, as shown above. The LABEL special form used here is *not* the one used in early LISP, though, but one that uses the syntax of modern-day LISP's LET or LETREC (although semantics differ!). The LABEL form used here can be thought of as follows:

```
(LABEL ((FOO FOO-VALUE)
        (BAR BAR-VALUE))
  EXPR)
```

is equivalent to

```
((LAMBDA (FOO BAR)
   EXPR)
 (QUOTE FOO-VALUE)
 (QUOTE BAR-VALUE))
```

The LABEL special form proposed here is even more trivial to implement than the original LABEL form, because its second argument already is in the shape of an environment, so it can just

be APPENDed to the front of the current environment to establish its bindings.

Note that the LABEL special form used here quotes its value expressions! This does not matter in this particular case, though, because LABEL is only used to bind lambda functions in the following program, which evaluate to themselves anyway.

The interpreter defines its own version of ASSOC (called LOOKUP) and its own version of APPEND (called APPEND2) in order to avoid name conflicts in modern LISP systems. For the same reason EVAL is called XEVAL.

Here it comes: a minimal metacircular LISP in a single LABEL:

```
(LABEL
   ; FIND VALUE OF X IN E
  ((LOOKUP
     (LAMBDA (X E)
       (COND ((EQ NIL E) NIL)
             ((EQ X (CAAR E))
               (CADAR E))
             (T (LOOKUP X (CDR E))))))

   ; EVALUATE COND
   (EVCON
     (LAMBDA (C E)
       (COND ((XEVAL (CAAR C) E)
                (XEVAL (CADAR C) E))
             (T (EVCON (CDR C) E)))))

   ; BIND VARIABLES V TO ARGUMENTS A IN E
   (BIND
     (LAMBDA (V A E)
       (COND ((EQ V NIL) E)
             (T (CONS
```

```
                (CONS (CAR V)
                      (CONS (XEVAL (CAR A) E)
                            NIL))
                (BIND (CDR V) (CDR A) E))))))

; SAME AS APPEND
(APPEND2
  (LAMBDA (A B)
    (COND ((EQ A NIL) B)
          (T (CONS (CAR A)
                   (APPEND2 (CDR A) B))))))

; EVALUATE EXPRESSION X IN ENVIRONMENT E
(XEVAL
  (LAMBDA (X E)
    (COND
      ((EQ X T) T)
      ((ATOM X)
         (LOOKUP X E))
      ((ATOM (CAR X))
        (COND
          ((EQ (CAR X) (QUOTE QUOTE))
            (CADR X))
          ((EQ (CAR X) (QUOTE ATOM))
            (ATOM (XEVAL (CADR X) E)))
          ((EQ (CAR X) (QUOTE EQ))
            (EQ (XEVAL (CADR X) E)
                (XEVAL (CADDR X) E)))
          ((EQ (CAR X) (QUOTE CAR))
            (CAR (XEVAL (CADR X) E)))
          ((EQ (CAR X) (QUOTE CDR))
            (CDR (XEVAL (CADR X) E)))
          ((EQ (CAR X) (QUOTE CAAR))
            (CAAR (XEVAL (CADR X) E)))
          ((EQ (CAR X) (QUOTE CADR))
```

```
                (CADR (XEVAL (CADR X) E)))
              ((EQ (CAR X) (QUOTE CDAR))
                (CDAR (XEVAL (CADR X) E)))
              ((EQ (CAR X) (QUOTE CADAR))
                (CADAR (XEVAL (CADR X) E)))
              ((EQ (CAR X) (QUOTE CADDR))
                (CADDR (XEVAL (CADR X) E)))
              ((EQ (CAR X) (QUOTE CONS))
                (CONS (XEVAL (CADR X) E)
                      (XEVAL (CADDR X) E)))
              ((EQ (CAR X) (QUOTE COND))
                (EVCON (CDR X) E))
              ((EQ (CAR X) (QUOTE LABEL))
                (XEVAL (CADDR X)
                       (APPEND2 (CADR X) E)))
              ((EQ NIL (CAR X))
                (QUOTE *UNDEFINED))
              ((EQ (CAR X) (QUOTE LAMBDA))
                X)
              (T (XEVAL (CONS (XEVAL (CAR X) E)
                              (CDR X))
                        E))))
          ((EQ (CAAR X) (QUOTE LAMBDA))
            (XEVAL (CADR (CDAR X))
                   (BIND (CADAR X) (CDR X) E)))))))

(XEVAL (QUOTE form) NIL))
```

The program evaluates any form inserted in the place of *form* in the final application of XEVAL and returns its *normal form* (value). For instance, substituting the following program for *form* in the above expression and then evaluating it will yield the value (A B C D E F):

```
(QUOTE
  (LABEL
    ((APPEND
      (LAMBDA (A B)
        (COND ((EQ A NIL) B)
              (T (CONS (CAR A)
                       (APPEND (CDR A) B)))))))
    (APPEND (QUOTE (A B C))
            (QUOTE (D E F))))))
```

Any S-expression that is a well-formed program of the minimal subset of LISP described here can be submitted for evaluation. What is particularly interesting about the evaluator is that it is written in precisely the language that it interprets, i.e. it is a *metacircular* interpreter. Inserting the code of the interpreter itself in the place of *form* and then the above example program in the place of the *form* in the embedded instance of the interpreter will still evaluate to the same value, (A B C D E F).

This can be tried in COMMON LISP or SCHEME by adding just a little bit of glue. In SCHEME, LABEL can be defined to be the same as LETREC, which works fine, because their syntax is identical and their semantics close enough:

```
(define-syntax label
  (syntax-rules ()
    ((label ((n f) ...) x)
     (letrec ((n f) ...) x))))
```

Then, there are some minor differences between SCHEME and generic LISP that have to be accounted for:

```
(define atom symbol?)  ; not the whole truth
```

```
(define t (quote t))
(define nil (quote ()))
(define eq eq?)
```

In COMMON LISP, a close-enough replacement of LABEL can be defined in terms of LABELS:

```
(DEFMACRO LABEL (B X)
  `(LABELS
     ,(MAPCAR
        (LAMBDA (A)
          (LET ((F (CADR A)))
            `(,(CAR A) ,(CADR F) ,(CADDR F))))
        B)
     ,X))
```

The following experiments will be conducted in SCHEME, because they are simpler in a single-namespace LISP. With the above definitions in place, the LABEL implementing the metacircular LISP can be wrapped up in a list resembling a lambda function:

```
(define evsrc
  (quote            ; note the QUOTE!
    (lambda (x e)
      ; insert the interpreter here and
      ; replace the final XEVAL application
      ; with this one:
      (xeval x e))))
```

A working version of XEVAL is then just one EVAL away:

```
(define xeval (eval evsrc))
```

(If your SCHEME system does not have an EVAL function, you will have to create a second instance of EVSRC, but without the QUOTE, and bind it to the name XEVAL.)

Given these definitions, you can then evaluate any minimal LISP program *expr* using the expression

---

(xeval (quote *expr***) nil)**

To evaluate an XEVAL interpreter evaluating the expression, use

---

(xeval `(,evsrc ',*expr* **nil) nil)**

---

and to evaluate an XEVAL interpreter evaluating XEVAL evaluating the expression, use

---

(xeval `(,evsrc '(,evsrc ',*expr* **nil) nil) nil)**

At this point there is *a lot of interpretation* going on:

- an inner XEVAL interpreting *expr*
- an outer XEVAL interpreting the inner XEVAL
- an outermost XEVAL interpreting the outer XEVAL
- SCHEME interpreting the outermost XEVAL
- the CPU interpreting the SCHEME system
- the laws of nature interpreting the CPU

Of course at each level of interpretation, there is some loss of performance and some reduction in available space. The whole thing is already becoming very impractical in the above example.

This restriction would not exist, if the LISP system compiled to machine code instead of interpreting abstract code. Then an inner EVAL would just be another identical instance of an outer EVAL: compiling a LISP compiler with itself would just generate another instance of itself. We will get back to this point soon.

SOME MISSING BITS AND PIECES

There are lots of beautiful minimal metacircular LISP systems out there. However, they all have one thing in common: they work only if you already have a functioning LISP system to interpret them.

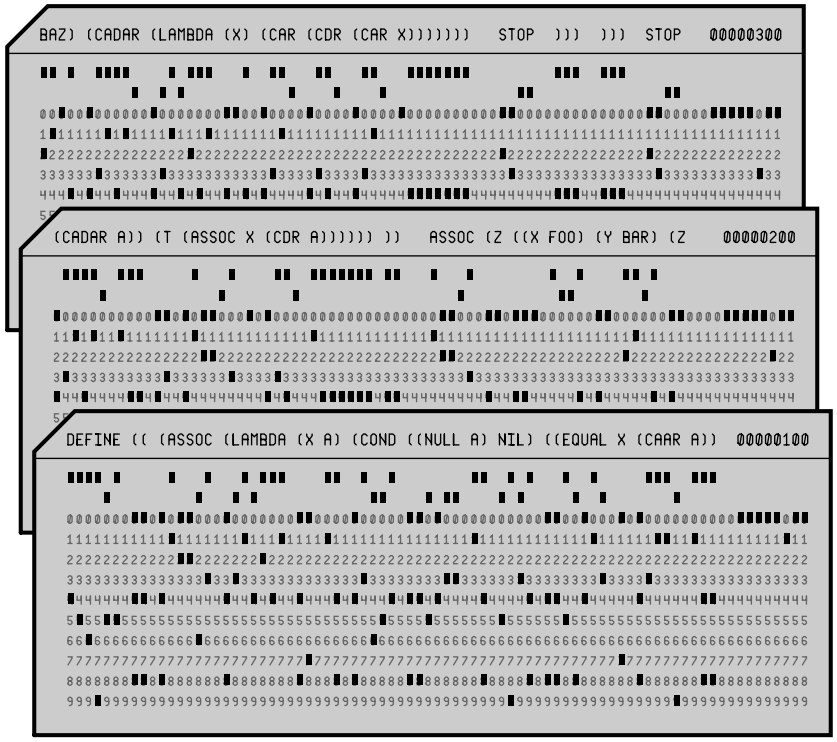
They silently assume that there already is

- a reader/parser (READ)
- a printer (PRINT)
- an unlimited CONS pool (i.e., a garbage collector)
- a pre-existing EVAL interpreting the interpreter

What if none of the above exists? How far away is a minimal metacircular LISP system under these circumstances? What does it take to bootstrap EVAL from nothing?

We shall see.

EVAL FROM NOTHING



AND LISP FOR FREE

Yes, these punch card images contain real code! You could punch them and feed them to a LISP system on an IBM 704 and the code would evaluate.

THE HOPL II paper on the history of LISP [McCarthy1981] displays the appearance of the LISP interpreter as a sudden event, but the LISP 1 manual lists nine people who have worked on different aspects of the LISP system, so the “sudden event” was probably just the appearance of APPLY, the part of the LISP system that gives meaning to forms. The parts that have been worked on by different individuals include [McCarthy1960]:

- APPLY
- READ
- PRINT
- the garbage collector
- algebra and floating point numbers
- the compiler
- the Flexowriter interface

As can be deduced from this list, the LISP 1 system was much more complex than the minimal EVAL presented in the previous chapter. The parts of it that are relevant in the course of the discussion in this text are: APPLY (or its cousin, EVAL), READ, PRINT, and the garbage collector.

The relationship between APPLY and EVAL was such that APPLY could be defined in terms of EVAL as follows:

```
(DEFINE ((APPLY (LAMBDA FN ARGS)
  (EVAL (CONS FN (APPQ ARGS)) NIL))))
```

where the APPQ function put an application of QUOTE around each member of ARGS, so, for instance,

```
(APPLY (QUOTE CONS) (QUOTE (A B)))
```

passed the following expression to EVAL:

```
(CONS (QUOTE A) (QUOTE B))
```

HACKING IN THE 1960'S

You may or may not have noticed the absence of the "Flexowriter" option from the discussion. The *Flexowriter* was one of the earliest devices that would qualify as a "terminal" these days. Back then it was called a teletypewriter or a Flexowriter, which was its official product name. We will return to the Flexowriter in a later chapter, because it did not play a big rôle in the early development of LISP.

When LISP was developed in the 1960's, programs were submitted to the computer on punch cards and programs printed their results on chain printers (or similar devices).

A *punch card* is a rectangular piece of card board, slightly larger than the palm of an average hand, with holes in it that indicate the characters that are stored on the card. Eighty characters fit on one card, but the rightmost eight columns were normally reserved for a serial number—in case you dropped a deck of cards.

Characters were not binary-encoded on punch cards, because that would punch too many holes in the card and hence make it floppy and prone to getting stuck in the reader. The encoding used instead was a 12-*channel* encoding (12 positions for holes per character) with three holes per character at most.

One line of code was usually stored on a card, so even small programs consisted of "decks" of cards, which were inserted in a card reader and read in sequence. The process was pretty fast unless a card got stuck. Even the first punch card readers could process 150 cards per minute while later models could easily read more than 1000 cards per minute. Note that most card readers read the cards "sideways", scanning 80 bits at a time, rather than lengthwise, character by character.

The meaning of the codes on a card depended on the *character set* in use. Every vendor had their own character set, which typically consisted of six-bit code points, so 64 characters were

available at most, but many character sets contained fewer characters than that. Even the same vendor sometimes had different character sets for the same machine, to be used for different purposes.

The IBM 704 used an encoding called BCDIC ("binary coded decimal interchange code"). The default IBM 704 BCDIC did not include any parentheses, though, so LISP most probably used the FORTRAN character set [Backus1956], which is shown in figure 2.

Ch	PC	CP	Ch	PC	CP	Ch	PC	CP	Ch	PC	CP
1	1	1	A	12+1	17	J	11+1	33	/	0+1	49
2	2	2	B	12+2	18	K	11+2	34	S	0+2	50
3	3	3	C	12+3	19	L	11+3	35	T	0+3	51
4	4	4	D	12+4	20	M	11+4	36	U	0+4	52
5	5	5	E	12+5	21	N	11+5	37	V	0+5	53
6	6	6	F	12+6	22	O	11+6	38	W	0+6	54
7	7	7	G	12+7	23	P	11+7	39	X	0+7	55
8	8	8	H	12+8	24	Q	11+8	40	Y	0+8	56
9	9	9	I	12+9	25	R	11+9	41	Z	0+9	57
	–	48	+	12	16	–	11	32	0	0	0
=	8+3	11	.	12+8+3	27	$	11+8+3	43	,	0+8+3	59
–	8+4	12	)	12+8+4	28	*	11+8+4	44	(	0+8+4	60

Fig. 2 – IBM 704 FORTRAN character set – "Ch" (Char) is the glyph associated with a code point, "CP" is the six-bit code point, and "PC" indicates the rows to be punched in the column representing the character on a punch card. A blank character would not have any holes.

On the output side of an IBM computer system you would typically find a *chain printer*, which was essentially a very large, very fast, and very loud typewriter. In this case "large" means easily a cubic meter in volume, "loud" means that a dot matrix printer sounds like whispering when compared to a chain printer, and "fast" means in the league of modern office laser printers.

However, chain printers could not use different font faces or font sizes. They printed monospace glyphs on continuous paper that often had green horizontal stripes to facilitate looking up values in the rows of tables. You could print boldface characters or underlining by using *overstriking*, though. To print the line

```
This is boldface text.
```

you would first print the line

```
This is boldface text.
```

and then print the line

```
        boldface
```

over it without advancing the paper. Underlining worked by printing underscore characters in the second pass. Note that the underscore character is absent from the FORTRAN character set, so this did not work in FORTRAN (or LISP).

Chain printers printed lines of text by rotating a chain of glyphs in front of the paper and striking the paper with hammers whenever the right characters appeared in the right positions. The printer had one hammer per column, so it would typically print an entire line in less than a full rotation of the chain. This worked amazingly well and was very fast—and very noisy.

You did not have to suffer from the noise, though, because you typically would not spend much time (if any) in the room containing the computer equipment. The process of developing programs was different back then, and in many cases did not involve touching any hardware. The job of a "programmer" was in its infancy. It was often mathematicians who wrote code.

The writing usually took place in your office, on a sheet of paper or a special "coding form", while sitting at a wooden desk. Coding forms divided the paper into columns and fields, and they were used mostly for the benefit of the data typists—typically a woman's job—who transferred your programs to punch cards. Some

institutions had a spare card punch for use by the people doing the programming, but submitting the sheets to the typist's office was much more common.

After picking up your deck of cards you would add a job description, usually on a blank card on top of the deck, and hand it to the operator who would then allocate a time slot for your job and submit it to the computer. The job description typically included instructions about which tape to mount on which tape drive, etc. Only one job could be run at a time, so you often had to wait for some hours, or even until the next day, before your job could be run. As soon as the program finished, you got your deck back in your inbox, together with a stack of fanfold paper from the chain printer. The inboxes were large and often located in an extra room where people went to pick up their results.

If you were lucky, the print-out contained the desired results. If you were not so lucky, it contained error messages of the compiler or an error description scribbled on a sheet of paper by the operator. Then you would have to fix the problem, whatever it was, and start from the beginning. People tried their best to write syntactically correct programs back then, because a single syntax error would ruin the entire job.

You could probably accelerate the process a little bit by inviting the data typist for lunch or using the spare key punch, if the institution let you do that. Punching cards was not an easy job, though, and you probably needed an introduction first. One might argue that using a key punch was quite a bit more complicated than using a modern text editor.

Deleting a line from a program was easy: you just tossed the corresponding card to the wastebasket. Adding a line meant to punch it, give it a proper serial number between the cards at the point where it was to be inserted, and put it into the deck.

Punching a card basically meant to type the program on a huge typewriter. Pressing a key immediately punched the corresponding

character. Because codes were designed to be unique, there was no easy way to fix a mistake. Once a code had been punched, it could not be corrected. If you had made a mistake, you had to toss the faulty card to the bin and start over. It was not *that* cruel, though. The card punches had a few nifty features that would make your life easier.

The most helpful feature was the ability to *copy* cards—or even fields or individual characters of a card [IBM1965]. This means that you could copy the faulty card up to the point of the mistake, correct it, and sometimes, if you were lucky, you could also copy the rest of the card. This would work, for instance, if you just had to replace a single character. Deleting characters sometimes worked by replacing them with blanks. It typically worked in LISP, because it is a free-form language without any "fields", where specific columns that had to be used for specific purposes.

If you had to insert text, though, you were most probably out of luck. Have a look at the following line:

```
(DEFINE ((ASSOC ((LAMDA (X A) (COND ((NULL A)
```

To insert the missing "B" in LAMBDA, you could copy the beginning of the line up the to missing character, but from that point on you had to re-type the line, because the key punch could only copy characters *to the same position* on a new card. That is:

```
(DEFINE ((ASSOC ((LAM     ← you could copy this part
                      B
                       DA (X A) (COND ((NULL A)
                       but you had to re-type this
```

Obviously, you typed *much* more carefully back in those days!

The entire program development cycle is displayed in figure 3. Note that hours could pass between the beginning and end of the process, and this description has glossed over a few additional things that could get in your way, like maintenance or the reader shredding a few cards of your deck.

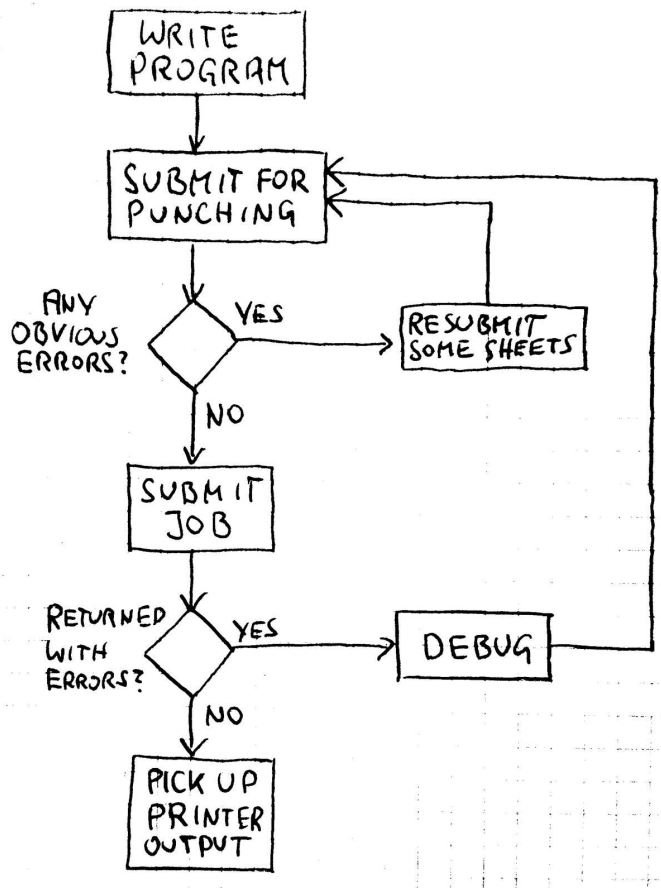


Fig. 3 – Program development cycle

In general, things that take a few seconds these days, like correcting a typo and recompiling the program, could easily take hours back then. Computers were huge industrial installations,

and you communicated with them through a proxy who operated the machine in your place. It was this process that Steve Russell most certainly used when he wrote the first version of APPLY [McCarthy1981]. Keeping this in mind may shed some new light on the effort that the creation of the first LISP system was.

BUILDING THE SMALLEST LISP

When implementing an evaluator for the minimal set of operators defined on page 20, there are some things that are often taken for granted in an existing LISP system, most prominently routines for reading and writing S-expressions (READ, PRINT) and an automatic memory management procedure delivering an unlimited supply of cons cells. The EVAL function, which is also listed among the prerequisites on page 28, is not necessary when building a compiler, which is what will be done in the remainder of this chapter. The compiler will eventually be able to compile its own code, thereby removing the cyclic dependency on EVAL (and creating a cyclic dependency on itself).

A *compiler* that compiles LISP from textual representation to low-level code is much more complex than the basic evaluator presented in the previous chapter. Hence a slightly extended dialect of LISP will be used to implement it. There will be several steps in the process developing the compiler:

- defining a new minimal LISP that is suitable for bootstrapping
- implementing the low-level parts that cannot be written in LISP
- implementing a compiler for the minimal LISP
- bootstrapping the full LISP system from the minimal LISP
- bootstrapping the LISP compiler

The *stage-0 compiler*, i.e. the compiler that is used to compile the LISP compiler initially, is written in SCHEME. However, we will not

discuss it here and start with the LISP version immediately. (The SCHEME code can be found in the appendix.) Once the LISP compiler is able to compile itself, the SCHEME version is no longer needed. The output of the LISP compiler can then be kept in order to bootstrap it in the future.

Things that cannot be implemented in LISP itself are mostly three things:

- a procedure for reading characters from a file or device
- a procedure for writing characters to a file or device
- a procedure for terminating the program

In addition, the system discussed here will implement the *garbage collector* in a more low-level language. This is not strictly necessary, but the implementation model chosen here leaves no other choice. A garbage collector written in LISP will be discussed in a later chapter.

The low-level section of the code will also contain procedures for (1) printing the value of the last expression in a program, although PRINT itself will be implemented in LISP; (2) routines for binding and unbinding arguments and controlling the flow of function application; (3) a halting and error reporting mechanism. The low-level part of the system will not be discussed in detail here, but the full code can be found in the appendix.

The low-level part of the LISP system will be written in the C programming language [K&R1988]. It could as well be written in assembly language, because it is very simple, but the C language was chosen for the sake of simplicity and portability.

The steps from the bootstrapping list at the beginning of this section will not be discussed in the given order here, because that would be too confusing. Instead, the minimal LISP system suitable for bootstrapping the compiler will be discussed first, the bootstrapping of the complete LISP system next, and the compiler and the low-level part of the runtime system last.

A BOOTSTRAPPABLE MINIMAL LISP

This is the set of built-in standard LISP functions and special forms that will be implemented by the LISP compiler:

CONS CAR CDR QUOTE ATOM EQ LAMBDA LABEL COND **PROGN SETQ**

These operators probably look quite familiar. The following special forms are added to the EVAL of the previous chapter:

(PROGN EXPR ...) evaluates a sequence of expressions and throws away their values, except for the value of the last expression, which it returns.

(SETQ ATOM EXPR) binds the given atom to the value of the given expression, i.e. it *changes* or *mutates* the binding of the atom.

The COND, LAMBDA, and LABEL forms will accept more than one expression in their bodies, i.e. PROGN is implied in them.

This new functionality basically exists to support an *imperative programming* style, the sequential modification of bindings of variables, which will be used in some places to implement the LISP system and the compiler. Could the compiler be written in a purely functional style? Most probably. Would it be beautiful and/or elegant? Most certainly not. Without support for macros, a lot of state would have to be passed around in the code, thereby making it unnecessarily complex. This is why mutable data structures will be used to build some parts of the system.

When a clause of a COND special form compiled by the compiler consists of a *predicate* alone, the clause will return the value of the predicate, *if it succeeds*. E.g.:

```
(COND (NIL) ((QUOTE FOO)))  ==>  FOO
```

This case pretty much falls out as a side effect of compiling LISP.

The LAMBDA special form will be extended to support LEXPRs—functions that collect all their arguments in a single list and bind that list to a single variable. When there is an atom in the place of the list of variables of LAMBDA, the resulting function will be an LEXPR:

(LAMBDA X X) ==> *lexpr*
((LAMBDA X X) 'FOO 'BAR 'BAZ) ==> **(FOO BAR BAZ)**

LEXPRs are not strictly required to bootstrap the compiler, but without them the LIST function would have to be a special kind of function of the type LSUBR, a primitive function with a variable number of arguments. Implementing LEXPRs is similarly complex, but adds a much more general feature to the language.

PROGN is not strictly necessary, either, but its functionality has to be present anyway for implementing the bodies of COND, LAMBDA, and LABEL. It also comes in handy for COND clauses of the form

((PROGN *do-something-useful* **NIL))**

which will do something useful and then fall through to the next clause.

In addition there will be a few new special forms that will be used internally to implement the LISP system in LISP:

*READC *WRITEC
*CAR *CDR *RPLACA *RPLACD
*ATOMP *SETATOM
*NEXT *POOL
*HALT

The *READC and *WRITEC functions implement character-based input and output, which will be necessary to implement READ and PRINT. The present implementation reads the standard input descriptor and writes to the standard output descriptor of Unix and Unix-derived environments. Since all I/O goes through these functions, this is easily changed, though. Both functions operate

on weird internal data structures that will be explained later in the code. *Do not use these functions in your LISP programs!*

The *CAR, *CDR, *RPLACA, and *RPLACD functions are *almost* like their standard LISP counterparts, but they do not perform any type checking, so they can operate on any kind of object, and not just on conses. This is a bad, bad idea in user-level programs, but will be used quite frequently in the implementation of the LISP system itself. For instance, using *RPLACD to change the CDR field of an atom will actually change the value bound to that atom due to the way atoms look like internally. Similarly, *RPLACA would change the name of the atom without touching the value! Fasten your seat belts! (No, we will not actually do this!)

The *ATOMP function finds out if a cell (see below) has its *atom tag* set. This does not mean that the cell is a LISP atom, though—the atom tag is a much more low-level concept that will be explained in detail in the next section. The *SETATOM function can be used to set or clear the atom tag of a cell. It is mostly used to create LISP atoms. Nothing stops LISP from being low-level! Your imagination is the limit!

The *POOL variable points to the *cell pool*, i.e. the memory pool from which conses and atoms will be allocated. *NEXT is a function that, given a cell, returns the address of the next cell in the pool. These functions are mostly used to locate data structures with known addresses inside of the cell pool. They are also essential to implementing a garbage collector in LISP—an exercise that will be postponed to a later chapter, though.

The *HALT function can be used to abort program execution and return control to the operating system in case of an error. It expects a message argument (an atom) that it will print before exiting.

BOOTSTRAPPING A COMPLETE LISP

This section will define the data structures of the LISP system and use the functions of the minimal LISP system to build a more complete LISP system, including the reader, the printer, the usual list processing functions, and some higher-order functions—basically all you need to build a compiler without having to bend over backwards.

The *symbol list* is a list containing all *atoms* (*symbols*) that are known to the LISP system. It is the mechanism that gives *identity* to atoms. Whenever the reader reads an atom, it looks it up in the symbol list and if the symbol already is in the list, it will return the symbol in the list instead of the one just read. When the symbol is not in the list, it will add it first. Because the reader always returns the same member of the symbol list when reading a symbol consisting of the same characters, symbols are equal in the sense of EQ.

The process of adding an atom to the symbol list is called "interning". It will be described later in this chapter.

The root of the symbol list is located in the third slot of the cell pool (see page 72). It is bound to the symbol *SYMLIS:

```
(SETQ *SYMLIS (*NEXT (*NEXT *POOL)))
```

*FUNTAG is a special value that corresponds to the address of the ninth slot in the cell pool. It is used to identify *function objects* and will be used by PRIN1.

```
(SETQ *FUNTAG (*NEXT
                (*NEXT
                  (*NEXT
                    (*NEXT
```

```
                    (*NEXT
                      (*NEXT *SYMLIS)))))))
```

The following variables represent characters that are cumbersome to use inside of LISP programs without confusing the programmer.

Note that **"foo"** is *not* a string, but a so-called *pseudostring*! It is an interned and self-quoting symbol, but unlike “ordinary” symbols its name may contain any kind of character. Pseudostrings are mostly used for printing and for comparing input to non-symbol characters.

Pseudostrings appeared in LISP 1.6 (a.k.a. PDP-6 LISP), an early precursor of MACLISP, but they had a different syntax: there were no delimiting quote characters, and special or reserved characters contained in pseudostrings had to be *slashified*, that is, prefixed with a slash [MIT1967]. For example:

```
HELLO/ WORLD/!
```

However, slashified notation becomes illegible quite easily, which is why the double-quote notation is chosen here.

Also note that the definition of *NL contains an actual newline character! Moving the ; NEWLINE comment up one line would bind the symbol to a string containing the comment!

```
(SETQ *LP "(") ; LEFT PAREN
(SETQ *RP ")") ; RIGHT PAREN
(SETQ *NL "
") ; NEWLINE
```

The functions ATOM, CAR, CDR, CONS, EQ, and *HALT will be inlined by the compiler, thereby making them special operators. The following definitions make them available as functions, too, so they can easily be passed to higher-order functions. For instance,

(MAPCAR CAR X) would not work without the below definition of CAR, because CAR would only be recognized in an operator position. Note that *HALT is renamed HALT.

```
(SETQ ATOM (LAMBDA (X) (ATOM X)))
(SETQ CAR (LAMBDA (X) (CAR X)))
(SETQ CDR (LAMBDA (X) (CDR X)))
(SETQ CONS (LAMBDA (X Y) (CONS X Y)))
(SETQ EQ (LAMBDA (X Y) (EQ X Y)))
(SETQ HALT (LAMBDA (X) (*HALT X)))
```

Here come CAAR and friends. Nothing special to see here.

```
(SETQ CAAR (LAMBDA (X) (CAR (CAR X))))
(SETQ CADR (LAMBDA (X) (CAR (CDR X))))
(SETQ CDAR (LAMBDA (X) (CDR (CAR X))))
(SETQ CDDR (LAMBDA (X) (CDR (CDR X))))
(SETQ CAAAR (LAMBDA (X) (CAR (CAR (CAR X)))))
(SETQ CAADR (LAMBDA (X) (CAR (CAR (CDR X)))))
(SETQ CADAR (LAMBDA (X) (CAR (CDR (CAR X)))))
(SETQ CADDR (LAMBDA (X) (CAR (CDR (CDR X)))))
(SETQ CDAAR (LAMBDA (X) (CDR (CAR (CAR X)))))
(SETQ CDADR (LAMBDA (X) (CDR (CAR (CDR X)))))
(SETQ CDDAR (LAMBDA (X) (CDR (CDR (CAR X)))))
(SETQ CDDDR (LAMBDA (X) (CDR (CDR (CDR X)))))
```

Using an LEXPR makes the LIST function trivial.

```
(SETQ LIST (LAMBDA X X))
```

NOT and NULL both check for equality to NIL, but (EQ X NIL) is faster, so you will see it rather often in the code.

```
(SETQ NULL (LAMBDA (X) (EQ X NIL)))
(SETQ NOT NULL)
```

RPLACA and RPLACD are just wrappers around the non-type-safe *RPLACA and *RPLACD functions.

```
(SETQ RPLACA
  (LAMBDA (X Y)
    (COND ((ATOM X)
           (HALT "RPLACA: EXPECTED CONS"))
          (T (*RPLACA X Y)))))

(SETQ RPLACD
  (LAMBDA (X Y)
    (COND ((ATOM X)
           (HALT "RPLACD: EXPECTED CONS"))
          (T (*RPLACD X Y)))))
```

REVERSE is a thin layer around RECONC. Because there is no module system and no lexical scoping, you may never redefine RECONC, or REVERSE will stop to work!

```
(SETQ RECONC
  (LAMBDA (A B)
    (COND ((EQ A NIL) B)
          (T (RECONC (CDR A)
                     (CONS (CAR A) B))))))

(SETQ REVERSE
  (LAMBDA (A)
    (RECONC A NIL)))
```

NREVERSE reverses a list in situ, destructively, without creating a new list, and returns it. You already do know why

```
(SETQ L (QUOTE (A B C)))
(NREVERSE L)
```

is a bad idea, don't you? If not try the above and then examine the value of L! What you probably meant was

```
(SETQ L (NREVERSE L))
```

except when L is a local variable and (NREVERSE L) is a tail application.

```
(SETQ NREVERSE
  (LAMBDA (A)
    (LABEL
      ((NRECONC
         (LAMBDA (A B)
           (COND ((EQ A NIL) B)
                 (T (SETQ *NRTMP (CDR A))
                    (*RPLACD A B)
                    (NRECONC *NRTMP A))))))
      (COND ((EQ A NIL) NIL)
            ((ATOM A)
              (HALT "NREVERSE: EXPECTED LIST"))
            (T (NRECONC A NIL))))))
```

Even APPEND can be implemented in terms of RECONC. Of course this means that when you redefine RECONC, APPEND will also stop to work.

NCONC finds the end of A and then changes its cdr part to B. NCONCing something to a constant is *self-modifying* code! E.g.:

```
(SETQ F
  (LAMBDA (X)
```

```
    (NCONC (QUOTE (A B C)) X)))
(F (QUOTE (D)))  ==>  (A B C D)
(F (QUOTE (E)))  ==>  (A B C D E)
(F (QUOTE (F)))  ==>  (A B C D E F)
```

The effect is caused by the RPLACD inside of NCONC.

```
(SETQ APPEND
  (LAMBDA (A B)
    (RECONC (REVERSE A) B)))

(SETQ NCONC
  (LAMBDA (A B)
    (LABEL
      ((LOOP (LAMBDA (A B)
         (COND ((ATOM (CDR A))
                  (RPLACD A B))
                (T (NCONC (CDR A) B))))))
      (COND ((ATOM A) B)
            (T (LOOP A B)
               A)))))
```

EQUAL's life is simple, if there are only atoms and lists:

```
(SETQ EQUAL
  (LAMBDA (A B)
    (COND ((EQ A B))
          ((ATOM A) NIL)
          ((ATOM B) NIL)
          ((EQUAL (CAR A) (CAR B))
            (EQUAL (CDR A) (CDR B))))))
```

Nothing special to see in the implementations of MEMBER and ASSOC, except that ASSOC uses the standard semantics instead

of the LISP 1 one, i.e., it returns the pair forming an association and not just the “value” (cdr/cadr part) of that pair.

Hence it does not matter whether associations have the form (A B) or (A . B). ASSOC will work fine in either case.

```
(SETQ MEMBER
  (LAMBDA (X A)
    (COND ((EQ A NIL) NIL)
          ((EQUAL X (CAR A)) A)
          (T (MEMBER X (CDR A))))))

(SETQ ASSOC
  (LAMBDA (X A)
    (COND ((EQ A NIL) NIL)
          ((EQUAL X (CAAR A)) (CAR A))
          (T (ASSOC X (CDR A))))))
```

MAPCAR maps a unary function (or LEXPR) over a list and MAPCAR2 maps a binary function (or LEXPR) over two lists and stops when reaching the end of the shorter one. E.g.:

```
(MAPCAR2 LIST '(A B C) '(1 2)) ==> ((A 1) (B 2))
```

In case you wonder about the funny variable names, like *F: they are used to avoid the *downward FUNARG problem* (page 189). A leading “*” character should never appear in programs, except in LISP system code and variables of higher-order functions.

```
(SETQ MAPCAR
  (LAMBDA (*F *A)
    (LABEL
      ((MAP (LAMBDA (A R)
         (COND ((EQ A NIL) (NREVERSE R))
               (T (MAP (CDR A)
                       (CONS (*F (CAR A)) R)))))))
```

```
     (MAP *A NIL))))

(SETQ MAPCAR2
  (LAMBDA (*F *A *B)
    (LABEL
      ((MAP (LAMBDA (A B R)
         (COND ((EQ A NIL) (NREVERSE R))
               ((EQ B NIL) (NREVERSE R))
               (T (MAP (CDR A)
                       (CDR B)
                       (CONS (*F (CAR A) (CAR B))
                             R)))))))
      (MAP *A *B NIL))))
```

REDUCE combines elements of a list left-associatively and RREDUCE combines them right-associatively. Both expect a neutral or base element B to be inserted in the leftmost/rightmost position. E.g.:

```
 (REDUCE CONS 'FOO '(A B C))
                        ==>  (((FOO . A) . B) . C)
(RREDUCE CONS 'FOO '(A B C))
                        ==>  (A B C . FOO)
```

```
(SETQ REDUCE
  (LAMBDA (*F *B *A)
    (LABEL
      ((RED (LAMBDA (A R)
         (COND ((EQ A NIL) R)
               (T (RED (CDR A)
                       (*F R (CAR A))))))))
      (RED *A *B))))

(SETQ RREDUCE
  (LAMBDA (*F *B *A)
```

```
(LABEL
   ((RED (LAMBDA (A R)
      (COND ((EQ A NIL) R)
            (T (RED (CDR A)
                    (*F (CAR A) R)))))))
   (RED (REVERSE *A) *B))))
```

WRITEC will print the first character of its argument, if its argument is an atom. There cannot be any type checking here, because WRITEC can also print first characters of *atom names*, which is something entirely different. Things get a bit messy here! All the ugly details follow immediately.

TERPRI prints a newline character.

```
(SETQ WRITEC
  (LAMBDA (C)
    (*WRITEC C)))

(SETQ TERPRI
  (LAMBDA ()
    (*WRITEC *NL)))
```

The next function will be PRIN1, which is like PRINT, but does not emit any final newline character. This is a good occasion for a short digression and have a look at data structures.

In a purely symbolic LISP system, like the one that is being defined here, there are only two data types: the *atom* (or *symbol*) and the *list* (or, more precisely, the *cons cell*). The implementation discussed here also has a “function” type, which will be explained later. All structures in the LISP system are made of cells internally.

A *cell* is a thing that has a “car” field, a “cdr” field and a “tag” field. If the cell is a cons cell, then the car and cdr fields point to other

cells. If the cell is an *atomic cell*, then only the cdr field points to another cell while the car field can hold any value whatsoever.

An *atom* is a cell that connects an *atom name* to another object (cell) called the *value* of the atom. See figure 4. The root cell in the figure, which represents the LISP atom in its entirety, connects the atom name to the value to which the atom is currently *bound*. The atom name consists of the characters $C_1 .. C_N$, each of them packaged in a separate atomic cell—a cell with its *atom tag* (A) set in its tag field. Like ordinary lists, the atom name ends with NIL.

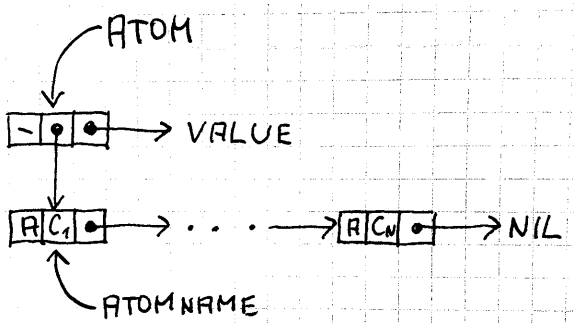


Fig. 4 – Atoms and atom names

Internally, (the root of) a LISP atom is a cell whose *car part* has its atom tag set. The construction of atoms will be explained later, when discussing the reader.

Remember that the *WRITEC function can print the first characters of both atoms and atom names? When its argument has its atom tag set, *WRITEC will just print its car part, assuming that it will contain the code point of a character. When the argument does not have its atom tag set, it will print the car part of its car part. Hence

```
(*WRITEC (QUOTE FOO))
```

and

```
(*WRITEC (*CAR (QUOTE FOO)))
```

will both print the character “F”.

The sole purpose of the *WRITEC function is to implement the PRIN1 and, subsequently, the PRINT function.

PRIN1 is very straight-forward, it has only a few simple internal functions: the PRC function traverses an atom name and prints its characters, the PR-ATOM function extracts and prints the name of an atom, and PR-MEM prints the members of a list, separated by blanks, but without the enclosing parentheses. It also prints dotted pairs, i.e., the tails of lists whose last member is not NIL, as in

```
(CONS (QUOTE A) (QUOTE B))  ==>  (A . B)
```

The PR function determines the type of the object to print and delegates output to PR-ATOM or PR-MEM, respectively. When attempting to print an atom name, it will print its first character in angle brackets, e.g.

```
(PRIN1 (*CAR (QUOTE FOO)))
```

will print “<F>”.

When printing a *function object*, PRIN1 will just output

```
<FUNCTION>
```

Special care has to be taken when printing NIL, because it is not an atom, but a special object without an atom name. Hence its characters will have to be output individually.

Question for the investigative minds: What would happen, if PRIN1 would just print (QUOTE NIL) instead?

```
(SETQ PRIN1
  (LAMBDA (X)
    (LABEL
      ((PRC (LAMBDA (X)
         (COND ((EQ X NIL))
```

```
            (T (*WRITEC X)
               (PRC (*CDR X))))))
      (PR-ATOM (LAMBDA (X)
        (PRC (*CAR X))))
      (PR-MEM (LAMBDA (X)
        (COND ((EQ X NIL))
              ((ATOM X)
                (PR ". ")
                (PR X))
              ((EQ (CDR X) NIL)
                (PR (CAR X)))
              (T (PR (CAR X))
                 (PR " ")
                 (PR-MEM (CDR X))))))
      (PR (LAMBDA (X)
        (COND ((EQ X NIL)
                (*WRITEC (QUOTE N))
                (*WRITEC (QUOTE I))
                (*WRITEC (QUOTE L)))
              ((*ATOMP X)
                (*WRITEC "<")
                (*WRITEC X)
                (*WRITEC ">"))
              ((ATOM X)
                (PR-ATOM X))
              (T (*WRITEC *LP)
                 (PR-MEM X)
                 (*WRITEC *RP))))))
     (PR X)
     X)))
```

The PRINT function prints its argument using PRIN1 and then emits a newline character. Note that traditional LISP does this the other way around: it first emits the newline and then the argument! We will get back to this later.

```
(SETQ PRINT
  (LAMBDA (X)
    (PRIN1 X)
    (TERPRI)
    X))
```

When a symbol is not yet *interned*, i.e. not yet added to the symbol list, it cannot be compared to another symbol using EQ. The only way to find out if an uninterned symbol is equal to another symbol (interned or not), is to compare the characters making up the names of the two symbols. This is what the SAMENAMEP function does.

The function SAME**P**NAMEP (note the additional “P”!) first appeared in MACLISP [Moon1974]. It compared the PNAME (*print name*) properties of two symbols. Back then (and even today) LISP symbols had *properties*, which were essentially flat lists of keywords and values. The PNAME property of a symbol was associated with the characters of the name that would be used to PRINT the symbol.

Different properties were used to store different values associated with a symbol. For instance, the EXPR property would link to a function associated with the symbol and the APVAL property would link to the value bound to the symbol. This mechanism probably started the evolution of LISP toward a LISP-N, a multi-namespace LISP. LISP 1.5, although it made extensive use of property lists, still behaved much like a LISP-1, but MACLISP already had all the characteristics of a LISP-N. E.g. FUNCTION had to be used to refer to a function binding in a variable context and FUNCALL had to be used to apply a function that was associated with the value property of a variable. In LISP 1.5 you could still apply functions bound as values to variables without using FUNCALL, in MACLISP this was no longer possible.

The LISP discussed here is a LISP-1, using a single namespace for everything, much like SCHEME [Scheme1991]. Hence an atom has only two “properties”, its name and its value. The SAMENAMEP function expects two *atom names*, i.e. atoms with their root cells removed, and returns T, if the chains of characters of the two names match. E.g.:

```
(SAMENAMEP (*CAR (QUOTE FOO))
           (*CAR (QUOTE FOO)))  ==>  T
```

Note that the function uses EQ to compare characters, which works fine, because identical characters have identical code points.

```
(SETQ SAMENAMEP
  (LAMBDA (X Y)
    (COND ((EQ X NIL) (EQ Y NIL))
          ((EQ Y NIL) NIL)
          ((EQ (*CAR X) (*CAR Y))
            (SAMENAMEP (*CDR X) (*CDR Y))))))
```

The INTERN function adds an atom to the symbol list *SYMLIS, thereby *interning* it, i.e. making it known to the LISP system as a unique symbol (see also page 43).

The INTERN function could be as simple as looking up an atom in a list (using SAMENAMEP to compare symbols) and adding it when it is not already present. However, this approach has a *time complexity* of $O(n^2/2)$ so, as the symbol list fills, adding new atoms will become prohibitively expensive. $O(n^2/2)$ means that adding n symbols will take $n^2/2$ time units. For instance, adding 10 symbols will take 50 units, but adding 1000 symbols will take 500,000 units.

Most more sophisticated data structures, like hash tables or sorted trees, will require some numeric computations, which cannot

(easily or cheaply) be done in purely symbolic LISP. This is why INTERN uses a simpler structure, a so-called *bucket list*.

A bucket list divides its members into sublists (called “buckets”) where all symbols in one bucket start with the same character. For instance:

```
((F FOO F)
 (Q Q QUOTE)
 (X X)
 (T T TERPRI)
 (A ATOM A)
 ...)
```

So in the ideal case, the time complexity of inserting a symbol into the bucket list is $O(n^2/2m)$, where m is the number of buckets. Because LISP symbols are mostly alphabetic (plus the occasional asterisk), a good estimate would be $O(n^2/54)$. The alert reader will notice that the m in $O(n^2/m)$ goes toward 1 as the number of atoms increases. In this particular case, though, the number of atoms in the system has exactly the right size for the bigger m to make a vast difference! Refer to the chapter on the science of LISP for a detailed assessment.

The INTERN function first finds the bucket for the first character of the given symbol. If that bucket exists, it looks up the symbol in the bucket. If that also exists, it returns the symbol found in the bucket.

If the symbol is not contained in the bucket, it is added to it and then returned. If no bucket for the given first character exists, a bucket just containing the first character and the new symbol is added to the list. In any case INTERN returns the interned symbol, which may or may not be identical to the symbol passed to it. (When a pre-existing symbol is added, the pre-existing symbol is returned instead of the one passed to INTERN.)

Note that the FIND function of INTERN is used to look up both buckets and members of buckets (i.e. it is similar to both ASSOC

and MEMBER). The M argument controls how symbols are looked up. When M=CAR, the function attempts to find a symbol in a flat list and when M=CAAR, it locates a bucket in an association list. The bucket list itself is an association list, but its members are flat lists.

Also note an interesting property of LABEL here: it evaluates and binds its arguments *sequentially*, from top to bottom, so FIRST and FIND can be first defined and later applied inside of the bindings part of the same LABEL. This property of LABEL was not specified in the evaluator of the previous chapter and it makes the dialect of LISP defined in it slightly incompatible to the one being implemented here! (Again, see the “science” chapter for details!)

```
(SETQ INTERN
  (LAMBDA (X)
    (LABEL ((FIRST
              (LAMBDA (X)
                (CONS (MKNAME (*CAR X) NIL)
                      NIL)))
            (FIND (LAMBDA (X M A)
              (COND ((EQ A NIL) NIL)
                    ((SAMENAMEP (*CAR X)
                                (*CAR (M A)))
                     (CAR A))
                    (T (FIND X M (CDR A))))))
            (F (FIRST X))
            (B (FIND F CAAR (CAR *SYMLIS))))
      (COND (B (LABEL ((V (FIND X CAR (CDR B))))
                 (COND (V)
                       (T (*RPLACD B
                                   (CONS X (CDR B)))
                          X))))
            (T (*RPLACA *SYMLIS
```

```
                        (CONS (CONS F (LIST X))
                              (CAR *SYMLIS)))
              X)))))
```

The MKNAME function creates a new *atom name* by attaching a single-character atom name to the front of an existing atom name. It is to atom names as CONS is to lists. Note that both arguments of MKNAME must be atom names! Hence the correct way to invoke the function is

```
(MKNAME (*CAR X) (*CAR Y))
```

where both X and Y are bound to symbols. For instance:

```
(CONS
  (MKNAME (*CAR (QUOTE F))
    (MKNAME (*CAR (QUOTE O))
      (MKNAME (*CAR (QUOTE B)) NIL)))
  NIL)                                    ==>  FOB
```

The outer CONS turns the atom name into an atom bound to the uninteresting value NIL. Delivering any real initial value will *not* work at this point, because interning the resulting atom may throw away that value! MKNAME is for creating *names*, not variables.

Note that the function first creates a cons cell with NIL in the place of the first character, then turns it into an atom cell, and finally stores the value of the character in the new first cell of the atom name! The code fragment

```
(LABEL ((N (CONS (*CAR C)) NIL))
  (*SETATOM N T))
```

would also work, but could have an unpleasant side effect: when a *garbage collection* would be triggered between binding N to its initial value and performing the *SETATOM operation, that garbage collection would mark a random cell “used”, namely the cell whose address would coincide with the code point of the new character. By first marking the cell atomic and *then* storing the code point, this case is precluded.

This might be over-engineering.

```
(SETQ MKNAME
  (LAMBDA (C A)
    (LABEL ((N (CONS NIL A)))
      (*SETATOM N T)
      (*RPLACA N (*CAR C)))))
```

PEEKC and READC are both functions that read a single character from the input device (or file) of the LISP system and return it. The difference between them is that READC “consumes” the character it delivers by requesting the next input character from the device, while PEEKC will not request a new character and hence deliver the same value over and over again when called multiple times. In other words, PEEKC performs a *look-ahead* operation on the input device.

The global variable *PEEKED is used to store a character that has been read by PEEKC. Both PEEKC and READC first check that variable and if it is bound to a value other than NIL, they return that value without actually performing a read operation.

PEEKC could easily be implemented as a low-level function, but in this case the LISP code is actually faster! A low-level peekc() function would not have access to the (LISP-level) INTERN function, so the delivered character would have to be interned each time when the low-level function returns. A high-level PEEKC function, like the below one, does this only one time per read or look-ahead, because the interned atom is then cached in a global variable.

Both functions return an interned atom with a single-character name corresponding to the character read. Note that *READC returns an atom name, so a cons cell has to be attached to its result in order to turn it into a complete atom.

If, for some reason, the input device should not be able to deliver a character, both functions return NIL.

```
(SETQ *PEEKED NIL)

(SETQ PEEKC
  (LAMBDA ()
    (COND (*PEEKED)
          (T (SETQ *PEEKED (*READC))
             (COND ((EQ *PEEKED NIL) NIL)
                   (T (SETQ *PEEKED
                        (INTERN (CONS *PEEKED
                                      NIL)))
                      *PEEKED))))))

(SETQ READC
  (LAMBDA ()
    (COND (*PEEKED
           (LABEL ((C *PEEKED))
             (SETQ *PEEKED NIL)
             C))
          (T (LABEL ((C (*READC)))
               (COND ((EQ C NIL) NIL)
                     (T (INTERN
                          (CONS C NIL)))))))))
```

The IMPLODE and MAKESYM functions both create a new symbol from the first letters of the atoms in a list. For example:

```
(IMPLODE (QUOTE (LISP IS SUPER PRETTY))) ==> LISP
(MAKESYM (QUOTE (F O O B A R))) ==> FOOBAR
```

The difference between these two is that the IMPLODE function interns the symbol that it returns, while MAKESYM returns an uninterned and hence *transient* symbol.

MAKESYM should be used to create symbols that are only used for printing and never compared to other atoms. For instance, the compiler will use it to generate symbols representing decimal numbers, which it will do quite often. Transient atoms do not clog the symbol list and their cells will get recycled when they are no longer used.

```
(SETQ MAKESYM
  (LAMBDA (N)
    (LABEL
      ((IMPL (LAMBDA (N A)
         (COND ((EQ N NIL)
                  (CONS A NIL))
               (T (IMPL (CDR N)
                        (MKNAME (*CAR (CAR N))
                                A)))))))
      (IMPL (REVERSE N) NIL))))

(SETQ IMPLODE
  (LAMBDA (X)
    (INTERN (MAKESYM X))))
```

EXPLODE is the inverse function of IMPLODE (as long as only single-character atoms are passed to IMPLODE). It creates a list of interned single-character atoms comprising the individual letters of the atom passed to it:

```
(EXPLODE (QUOTE FOOBAR))  ==>  (F O O B A R)
(EXPLODE (QUOTE LISP))    ==>  (L I S P)
```

```
(SETQ EXPLODE
  (LAMBDA (N)
    (LABEL
      ((MKATOM (LAMBDA (X)
         (INTERN (CONS (MKNAME X NIL) NIL))))
```

```
(EXPL (LAMBDA (N A)
  (COND ((EQ N NIL) (NREVERSE A))
        (T (EXPL (*CDR N)
                 (CONS (MKATOM N) A)))))))
(COND ((ATOM N) (EXPL (*CAR N) NIL))
      (T (HALT
          "EXPLODE: EXPECTED ATOM"))))))
```

The SYMBOLIC function returns a non-NIL value when the symbol passed to it represents a *symbolic character*, i.e. a character that may appear in a regular atom name. In other words, it returns truth for any character that READ will interpret as part of a symbol name. The function is not a true *predicate*, because it may return a value other than T or NIL. Hence it has no “P” suffix.

The list bound to *SYMBOLS lists the characters that may appear in atom names. Note that sequences of decimal digits are atom names and not *numbers*, because there are no numbers in purely symbolic LISP.

```
(SETQ *SYMBOLS
      (QUOTE (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 * -
              0 1 2 3 4 5 6 7 8 9)))

(SETQ SYMBOLIC
  (LAMBDA (C)
    (MEMBER C *SYMBOLS)))
```

The READ function is by far the most complex function in the LISP system. It is a complete parser for *S-expressions* that translates their textual representation to internal structures of conses and atomic cells.

The parser uses one character of look-ahead, that is, it reads its input with PEEKC and consumes it with READC only after deciding which action to take. This is necessary, for example, when reading the texts of atoms inside of lists:

```
FOO)
```

Here an atom is constructed from the characters “F”, “O”, and “O”. Each character is consumed only after classifying it as a symbolic character. When the “)” is read via PEEKC, it is not consumed, but left in the input. Therefore, the caller of the atom parser will still find it in the input.

The following internal functions are used by READ:

SKIPC skips over *blank* characters (including newline characters) in the input and returns the first non-blank character.

RD-COMM reads a *comment* and discards it. It returns an uninteresting value and advances input past the newline character that ends the comment. Comments begin with a “;” character.

RD-ATOM reads an *atom* (*symbol*) and returns it. It is the “atom parser” described above. Note that the symbol NIL has to treated in a special way here, because it is not a “real” atom. It is a *special symbol* that has only an address, but no atom structure. Hence RD-ATOM will return NIL (the special symbol) when it collects the characters “N”, “I”, “L”.

RD-PSTR reads a *pseudostring*. It collects characters until it finds a delimiting “`"`” character, composes an interned symbol from them and returns it. When a *backslash* (“`\`”) character is found in a pseudostring, the subsequent character is included instead, so “`\\`” includes a backslash and “`\"`” includes a quote character. A read error (READC returning NIL) inside of a pseudostring results in an error and program termination.

Note that the mechanism of the backslash character is basically *slashification*, just using a different slash. The backslash has been chosen because it is also used to “escape” characters in strings in

SCHEME, so the LISP source code can be read by the SCHEME reader for the purpose of bootstrapping.

RD-LIST reads and collects members of a *list* until it encounters a closing parenthesis. It returns a list containing the collected members. As in RD-PSTR, a read error while parsing a list will result in program abortion. RD-LIST uses the global temporary variable *READ-TMP. See the chapter on “hacks and kludges” for details about why this is a bad idea and why nothing can be done about it.

RD-OBJ classifies the text that follows in the input based on the current look-ahead character and dispatches parsing accordingly to one of the above functions. There are a few interesting cases here:

(1) When a read error occurs at this level, READ will just return NIL. This means that there is no *end of file* detection; it will have to be handled by user code. We will get back to this!

(2) The function expands the notation ′FOO to (QUOTE FOO). This is not much used in this book, but the mechanism is there.

(3) Extra right *parentheses* will just be ignored. This was done by most early LISP systems and some modern LISP systems still do this. In the early days it was good practice to append a punch card containing just a lot of closing parentheses to a program, so that a missed parenthesis at the end of the code would not ruin the job.

(4) A *funny* (unknown) input character will cause an error and immediate program termination. Error handling could really use some improvement! Note that a TAB character (ASCII HT, code point 9) is a funny character, so you should better make sure that it does not appear in your program text!

```
(SETQ READ
  (LAMBDA ()
    (LABEL
```

```
((SKIPC (LAMBDA (C)
   (COND ((EQ " " C)
          (READC)
          (SKIPC (PEEKC)))
         ((EQ C *NL)
          (READC)
          (SKIPC (PEEKC)))
         (T C))))
 (RD-COMM (LAMBDA (C)
   (COND ((EQ C *NL))
         (T (RD-COMM (READC))))))
 (RD-ATOM (LAMBDA (C A)
   (COND ((SYMBOLIC C)
          (READC)
          (RD-ATOM (PEEKC) (CONS C A)))
         (T (COND ((EQUAL A (QUOTE (L I N)))
                   NIL)
                  (T (IMPLODE
                       (NREVERSE A))))))))
 (RD-PSTR (LAMBDA (C A)
   (COND ((EQ C NIL)
          (HALT "UNTERMINATED STRING"))
         ((EQ C "\"")
          (READC)
          (LIST (QUOTE QUOTE)
                (IMPLODE (NREVERSE A))))
         ((EQ C "\\")
          (READC)
          (SETQ C (READC))
          (RD-PSTR (PEEKC)
                   (CONS C A)))
         (T (READC)
            (RD-PSTR (PEEKC)
                     (CONS C A))))))
 (RD-LIST (LAMBDA (C A)
```

```
      (COND ((EQ C NIL)
             (HALT "UNTERMINATED LIST"))
            ((EQ *RP C)
             (READC)
             (NREVERSE A))
            (T (SETQ *READ-TMP
                 (RD-OBJ (SKIPC (PEEKC))))
               (RD-LIST (SKIPC (PEEKC))
                        (CONS *READ-TMP A))))))
    (RD-OBJ (LAMBDA (C)
      (COND ((EQ C NIL) NIL)
            ((SYMBOLIC C)
             (RD-ATOM C NIL))
            ((EQ C *LP)
             (READC)
             (RD-LIST (SKIPC (PEEKC)) NIL))
            ((EQ C "'")
             (READC)
             (LIST (QUOTE QUOTE)
                   (RD-OBJ (SKIPC (PEEKC)))))
            ((EQ C "\"")
             (READC)
             (RD-PSTR (PEEKC) NIL))
            ((EQ C ";")
             (RD-COMM (READC))
             (RD-OBJ (SKIPC (PEEKC))))
            ((EQ C *RP)
             (READC)
             (RD-OBJ (SKIPC (PEEKC))))
            (T (HALT "FUNNY CHARACTER"))))))
  (RD-OBJ (SKIPC (PEEKC))))))
```

That's it!

A pretty complete purely symbolic LISP system bootstrapped from a minimal LISP system in just a few hundred lines of code. Now all we need is a compiler and some runtime support to actually build it!

THE LOW-LEVEL INTERFACE

The interface to the low-level part of the LISP system is a tedious description of a set of rather simple C functions and macros. A detailed understanding of the inner workings of these procedures is optional, but not hard to obtain. The complete listing can be found in the appendix. At least some superficial knowledge will be necessary, though, because the compiler will generate calls to the following procedures. Note that some details of the following descriptions may not make much sense right now. They will be clarified later.

`void halt(char *s, cell n);`

Print an error message (s) and *terminate* the calling program. When n is not equal to *LIMIT (which is not a valid LISP object), it will print as a LISP object after the error message. This function will be used to report *runtime errors*.

`cell cons3(cell a, cell d, cell t);`

Allocate a new cell from the freelist and return it. The fields of the cell will be initialized with the given CAR value (a), CDR value (d) and tag bits (t). When the freelist is empty, a garbage collection will be performed. When the freelist is still empty after the collection, the program will terminate through halt().

`cell mkfun(int k);`

Create a *function object* for the compiled function located at the label k and return it. The internal structure of a function object will be explained later.

`cell readc(void);`

Read a single character from the input device and return it as a single-character atom name (*not* atom!).

`void writec(cell n);`

Write the first character of the atom or atom name n to the output device.

`void print(cell n);`

Print the S-expression n followed by a newline character. This function duplicates the implementation of PRINT. It is used to print objects in error messages and the normal forms of the final expressions in programs.

`label(x)`

Mark a destination for goto() .

`goto(x)`

Jump to a destination marked by label() .

`symbolic()`

Return a non-zero value, if the current expression (the content of the EXPR register) is a symbol (but not NIL).

`atomic()`

Return a non-zero value, if the current expression is an atom (i.e.: symbolic() or NIL).

`push()`

Push the the EXPR register to the LISP stack.

`cell pop(void);`

Remove a value from the LISP stack and return it.

`newframe()`

Push a new call frame to the LISP stack.

`toframe()`

Push the current expression to the current call frame (the call frame on top of the stack).

`void bind(cell v);`

Bind the values in the current call frame to the variables in the list v. Store the outer values of the variables in the call frame. The concept of "outer values" will be covered later in this text.

`int restore(cell p);`

Unbind all variables whose outer values are stored in the call frame p, thereby restoring their outer values. This function will be used for tail call optimization, where the frame of the caller (rather than the current frame) is restored.

`unbind()`

Unbind all variables bound in the current call frame via restore().

`int apply(int n);`

Push the return address n to the current call frame and return the label contained in the current expression (which must be a function object). If $n = -1$ then the function application is a tail application, which will not return. In this case perform tail call optimization.

`int retn(void);`

Remove the current call frame and return the address stored in it.

`u(c)`

Short for toupper(c).

`car_err()`

Report a non-NIL atom in CAR and terminate program.

`cdr_err()`

Report a non-NIL atom in CDR and terminate program.

MEMORY ORGANIZATION

The memory of a compiled LISP program will be organized in units called *cells* (see also page 51). A cell consists of three fields called *car*, *cdr*, and *tag* and it is referenced by using its offset in the *cell pool*. The cell pool consists of three arrays (also called car, cdr, and tag) that hold the values of the respective fields. So, for instance, car[x] is the car value of the cell x, cdr[x] is its cdr value, and tag[x] its tag value.

Because cells are referred to by their offsets, the C type of a cell is a small unsigned integer (“unsigned short int”). For instance, the CAAR function could be implemented like this (but it isn’t):

```
cell caar(cell x) { return car[car[x]]; }
```

On a typical system, the car and cdr fields are 16 bits wide and the tag field is 8 bits wide (although only 3 bits are used), giving a total of 40 bits per cell.

The tag field contains a set of three tag bits (flags) that can be set or reset. These bits are called *atomtag*, *marktag*, and *travtag*. When the atom tag of a cell is set, the cell is atomic (part of an atom) and otherwise it is a cons cell. The other tag bits are only used during garbage collection and will be discussed in a later chapter.

The cell pool of the system can hold up to $2^{16} - 1$ cells (given an “unsigned short int” size of at least 16 bits). The last cell is reserved; this will also be discussed in the context of the garbage collector.

At the bottom of the cell pool there are some slots that are preallocated for some special objects that are listed in figure 5.

Addr	Slot Name	Description	Ref
0	**`nil`**	the NIL object	
1	**`true`**	the T atom	
2	**`symlis`**	the symbol list	Y
3	**`oblist`**	the object list	Y
4	**`stack`**	the runtime stack	Y
5	**`expr`**	the current expression	Y
6	**`expr2`**	second argument expression	Y
7	**`tmp`**	temporary value	
8	**`funtag`**	function tag	
9	**`tname`**	the atom name of the T atom	
10	**`frelis`**	the freelist	Y

Fig. 5 – Special slots of the cell pool

The NIL object is located in slot 0, which allows code that checks for NIL to work efficiently. The car and cdr field of NIL point to NIL, so car[NIL] = cdr[NIL] = NIL. The value of the T atom is T itself, so T does not have to be quoted in LISP programs. SYMLIS is the (bucket) list of symbols and OBLIST is a list of literal objects known to the system. It will be explained later in the code.

STACK is the runtime stack of the LISP system. It will be used for storing return addresses, arguments, and intermediate results in function applications.

EXPR and EXPR2 bind to values during evaluation of LISP expressions. You may think of them as *registers*. They will also be explained in the code. The car and cdr fields of TMP are used to protect temporary values from the garbage collector.

FUNTAG is just a slot number without any associated object. It is used to tag function objects. This will also be discussed in the code.

FRELIS is the *freelist*, i,e. a list of unused cons cells that link to each other via their cdr fields. When FRELIS is NIL, the freelist is empty and a garbage collection has to be performed.

Note that some slots in the pool merely store *references* to the respective objects themselves in their car fields. For instance, the runtime stack is a linked list that does not really begin at the STACK slot. The car field of the STACK slot only contains a pointer to the stack itself, so when an object is pushed to the stack (by consing that object to it), the STACK slot can be updated accordingly (see figure 6).

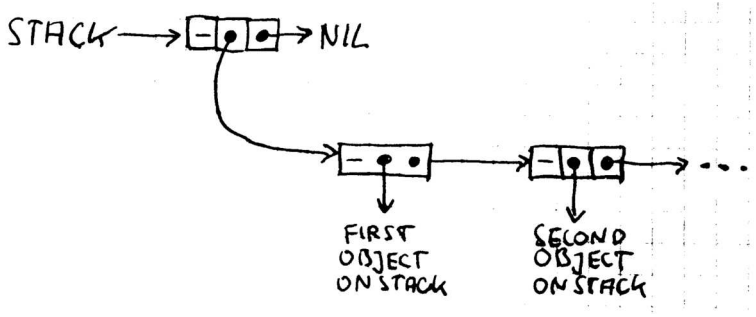


Fig. 6 – The runtime stack slot (STACK)

Slots that contain references to objects are marked with REF=Y in figure 5. The ref() macro is used to refer to the values of such slots. For instance, ref(stack) is the list implementing the runtime stack. A value x can be pushed to the runtime stack with the expression

ref(stack) = cons3(x**, ref(stack), 0)**

The value of an atom is referred to using the val() macro. For instance, the value of the atom T would be retrieved by the expression

```
val(true)
```

The symbols “true” and “nil” are used to represent T and NIL in the compiler output.

Of course val(x) is basically the same as cdr[x] and ref(x) is the same as car[x], but the alternative names tend to make the code more readable.

THE LISP COMPILER

The remainder of this chapter will discuss a compiler for the LISP system developed here so far. The compiler will be written in purely symbolic LISP and it will be able to compile itself. Let us call it “LISCMP” (pronounced “liscomp”) for the lack of a better name and following the tradition of packing an acronym in a six-character filename. I would have preferred to call the compiler “LISCOM”, but that name was already taken [Golden1970].

Naturally, there are some compromises to be made when writing a compiler in a purely symbolic language with a very basic runtime environment.

(1) The compiler will just read a program from standard input on a Unix system and write its output to standard output.

(2) It will expect the source code of the LISP system itself (as described in the previous section) to precede the code of the program to compile. This means that it will recompile the LISP system itself every time it compiles a program. There is no separate compilation.

(3) The low-level library source code must precede the compiler output in the output file, i.e. the compiler must append its output to a copy of the low-level runtime code file. This means that the low-level library will be recompiled every time.

(4) Because there are no lower-case characters in the LISP system discussed here, its output will have to be converted to

lower case before submitting it to a C compiler. A simple SCHEME program for downcasing the output can be found in the appendix.

These restrictions can easily be mitigated by using a simple Unix shell script that works as follows:

- concatenate the LISP system source code and the input program and save the result in a temporary file “run.lisp”
- copy the low-level runtime library to another temporary file, “run.c”
- compile the file “run.lisp” with LISCMP and append its output to “run.c”
- compile the file “run.c” with a C compiler
- (optionally) run the compiled program

Such a script will be discussed more in depth at the end of this chapter. An sample script can be found in the appendix.

With these technicalities out of the way, the compiler will compile LISP to an executable binary. Its output is not very compact, but reasonably fast. On the author’s trusty old 750MHz notebook computer, it self-compiles in two seconds (plus another seven seconds for the C compiler step). It is definitely suitable for writing some serious LISP applications, the first one being the compiler itself!

THE COMPILER SOURCE CODE STARTS HERE

Some numbers will have to be used in the compiler, but all of them will be either constants or counters. Therefore a very simple representation for numbers will suffice, like lists of decimal digit symbols, e.g.:

```
(1 2 3)  ; THE NUMBER 123
```

Single symbols could be used for constants, like (QUOTE 123) representing the number 123, but, to keep things consistent, list notation will be used throughout.

LIMIT is the address of the first cell that is *outside* of the cell pool. The maximum pool size is 65535 cells, so the highest address is 65534. This is because a cell is represented by an "unsigned short int" and

```
(unsigned short) 65535 + 1
```

might very well be undefined.

When the LIMIT constant changes, it will have to be adjusted in the low-level part of the runtime library as well.

```
(SETQ LIMIT (QUOTE (6 5 5 3 5)))
```

NIL-ADDR is the address of the NIL object and T-ADDR is the address of the T atom (see the figure on page 72).

```
(SETQ NIL-ADDR (QUOTE (0)))
(SETQ T-ADDR (QUOTE (1)))
```

The compiler will generate an initial cell pool image containing literal objects like atoms, quoted S-expressions, and lists of function variables. These objects will have to be located *somewhere* in the pool. ADDR is the first address (minus one) to be used for such objects. The address should be low, but not so low that it interferes with the special slots at the bottom of the pool. The actual first address to be used is 100; the counter will be incremented before use.

Note that the compiler does not actually *write* directly to any pool addresses—that would interfere badly with the LISP code currently running! It just generates code to set up the pool when the compiled program is started.

```
(SETQ ADDR (QUOTE (9 9)))
```

The SUCC (*successor*) function returns the successor of a decimal digit, e.g. (SUCC 2) ==> 3. Note that (SUCC 9) is zero.

The INCR function increments the number represented by a list of digits, e.g.:

```
(INCR (QUOTE (1 2 9)))  ==>  (1 3 0)
```

The arguments of the internal LOOP function are:

X = inverted input number
C = carry flag, initially true, thereby adding 1 to X
Y = output number

```
(SETQ SUCC
  (LAMBDA (X)
    (CADR (ASSOC X (QUOTE ((0 1) (1 2) (2 3)
                           (3 4) (4 5) (5 6)
                           (6 7) (7 8) (8 9)
                           (9 0)))))))

(SETQ INCR
  (LAMBDA (X)
    (LABEL
      ((LOOP (LAMBDA (X C Y)
         (COND ((EQ X NIL)
                  (COND (C (CONS (QUOTE 1) Y))
                        (T Y)))
               (C (SETQ INCR-TMP (SUCC (CAR X)))
                  (LOOP (CDR X)
                        (EQ INCR-TMP (QUOTE 0))
                        (CONS INCR-TMP Y)))
               (T (LOOP (CDR X)
```

```
                        NIL
                        (CONS (CAR X) Y)))))))
      (LOOP (REVERSE X) T NIL))))
```

The NUMBER function converts a list of digits to a transient symbol representing a number.

```
(SETQ NUMBER MAKESYM)
```

CELLS is a list of properties of cells that will form the literal objects (including symbols) to be contained in the initial cell pool image of the program being compiled. Each entry in the list will have the form

```
(addr tag car cdr)
```

where “addr” is the address of the cell in the cell pool, “tag” is either T (indicating an atomic cell) or NIL (indicating a cons cell), and “car” and “cdr” are numbers indicating the addresses of other cells (or literal values in atomic cells).

The entire initial cell pool of the generated program will be described by entries of the above form.

```
(SETQ CELLS NIL)
```

SYMLIS will be bound to a bucket list of atoms that are being read during compilation. It is a separate symbol list that is maintained by the compiler. This list will be used to build the initial symbol list (*SYMLIS) of the binary emitted by the compiler.

SYMLIS will not just contain atoms, but also the addresses where the atoms will be stored, i.e. each bucket will have the form

```
(LETTER (ATOM . ADDR) (ATOM . ADDR) ...)
```

The initial SYMLIS contains just the symbol T:

```
((T (T 1)))
```

Remember that numbers are lists! Hence (T 1) = (T . (1)).

```
(SETQ SYMLIS (LIST (LIST T (QUOTE (T 1)))))
```

OBLIST is a list of addresses where constants (lists quoted by QUOTE as well as lists of function variables) are stored. Atoms are not contained in the OBLIST, they are kept in SYMLIS instead. OBLIST is used for building the OBLIST (see pages 72 and 107) of the initial cell pool image.

Note that there are *two* object lists and two symbol lists in memory when the compiler runs: one holding the objects and symbols used by the compiler itself and one *built by the compiler* that will be used by the compiled program when it executes. The one built by the compiler is contained in the variables OBLIST, SYMLIS and CELLS.

```
(SETQ OBLIST NIL)
```

The code emitted by the compiler uses *labels* to mark portions of the code. LBL binds to the number of the most recently used label and the MAKE-LABEL function generates the next label to be used. The label 0 is reserved for the beginning of the program.

```
(SETQ LBL (QUOTE (0)))

(SETQ MAKE-LABEL
  (LAMBDA ()
    (SETQ LBL (INCR LBL))
    LBL))
```

MAKE-CELL allocates a new cell to be used at runtime (by advancing ADDR) and adds the properties of the cell to the CELLS list. The new cell will have the properties car=A, cdr=D, and tags=TG. MAKE-CELL returns the address of the allocated cell.

Note that "allocation" really only means to advance a number here. No actual space is being reserved for the cell—although space *is* being reserved for its description in CELLS.

```
(SETQ MAKE-CELL
  (LAMBDA (A D TG)
    (SETQ ADDR (INCR ADDR))
    (SETQ CELLS (CONS (LIST ADDR TG A D) CELLS))
    ADDR))
```

MAKE-ATOM adds an atom to the CELLS list by first allocating an atomic cell for each of its characters and then a cons cell for its root. It also sets up the car/cdr links between the cells properly. For instance, the symbol BAZ would create something like the following CELLS entries:

```
ADDR ATOM CAR CDR
(103 NIL  102 0)
(102  T   'B' 101)
(101  T   'A' 100)
(100  T   'Z' 0)
```

```
(SETQ MAKE-ATOM
  (LAMBDA (X)
    (LABEL
      ((LOOP (LAMBDA (X P)
         (COND ((EQ X NIL)
                (MAKE-CELL P NIL-ADDR NIL))
```

```
                (T (LOOP (CDR X)
                         (MAKE-CELL (CAR X)
                                    P T)))))))
      (LOOP (NREVERSE (EXPLODE X)) NIL-ADDR))))
```

The ADD-ATOM function creates a new ATOM (using MAKE-ATOM) *and* adds it to the initial symbol list of the emitted binary. The algorithm of ADD-ATOM is basically a duplication of the INTERN function (page 58), which should be no surprise, as they manipulate almost the same structure. ADD-ATOM processes already interned symbols, though, so it can use EQ in the place of SAMENAMEP.

The FIRST function is not embedded in ADD-ATOM to save the cost of binding it each time ADD-ATOM is called. In INTERN it was embedded, because INTERN is part of the LISP system, where it is important not to clutter the namespace with helper functions. In a user-level program, like the compiler, we can easily make sure that FIRST is a unique name.

```
(SETQ FIRST
  (LAMBDA (X)
    (INTERN (CONS (MKNAME (*CAR X) NIL) NIL))))

(SETQ ADD-ATOM
  (LAMBDA (X)
    (LABEL ((F (FIRST X))
            (B (ASSOC F SYMLIS)))
      (COND (B (LABEL ((V (ASSOC X (CDR B))))
                 (COND (V (CDR V))
                       (T (LABEL
                            ((A (MAKE-ATOM X)))
                            (RPLACD B
                              (CONS (CONS X A)
                                    (CDR B)))
                            A)))))
```

```
            (T (LABEL ((A (MAKE-ATOM X)))
                 (SETQ SYMLIS
                       (CONS
                         (CONS
                           F (LIST (CONS X A)))
                         SYMLIS))
               A))))))
```

MAKE-OBJECT recursively creates cell descriptions (for use in the CELLS list) for all cells that make up a given data object (atom or quoted list). It returns the address of the root cell of that object. For T and NIL, it just returns their fixed addresses.

```
(SETQ MAKE-OBJECT
  (LAMBDA (X)
    (COND ((EQ X NIL) NIL-ADDR)
          ((EQ X T) T-ADDR)
          ((ATOM X) (ADD-ATOM X))
          (T (MAKE-CELL (MAKE-OBJECT (CAR X))
                        (MAKE-OBJECT (CDR X))
                        NIL)))))
```

The ADD-OBJECT function is similar to MAKE-OBJECT, but also adds the root address of the given object to the OBLIST. Because atoms are kept in the symbol list instead of the object list, processing of atoms is delegated to the ADD-ATOM function. Like MAKE-OBJECT, ADD-OBJECT returns fixed addresses for T and NIL.

Note that ADD-OBJECT does not perform any *deduplication* for two reasons: (1) it is expensive, especially in a purely symbolic language, where there are no numbers and hence no hash tables, sorted, trees, or similar structures. (2) It would break self-modifying code (see the chapter about hacks and kludges for details).

Also note that deduplication of *atoms* is inherent in their nature.

```
(SETQ ADD-OBJECT
  (LAMBDA (X)
    (COND ((EQ X NIL) NIL-ADDR)
          ((EQ X T) T-ADDR)
          ((ATOM X) (ADD-ATOM X))
          (T (SETQ OBLIST (CONS (MAKE-OBJECT X)
                                OBLIST))
             (CAR OBLIST)))))
```

The variable-argument EMIT function prints all of its arguments sequentially, with no blanks in between, and then appends one final newline character. It is used to emit C code. For instance,

```
(EMIT "FOO = " (NUMBER (QUOTE (1 2 3))) ";")
```

would print

```
FOO = 123;
```

```
(SETQ EMIT
  (LAMBDA X
    (LABEL
      ((LOOP (LAMBDA (X)
         (COND ((EQ X NIL)
                (TERPRI))
               (T (PRIN1 (CAR X))
                  (LOOP (CDR X)))))))
      (LOOP X))))
```

BLOCKCOM compiles a *block* of code. A block of code is a list of expressions that will evaluate in sequence at run time, like the *bodies* of LAMBDA, LABEL, or PROGN, and the *consequents* (conditionally evaluated expressions) of COND.

Most functions that compile code receive two arguments: the expression X to compile and the TA flag indicating a *tail application*.

When TA is not NIL, the block compiled by BLOCKCOM appears in a so-called *tail position*, i.e. in a position in a program where a function application would be a tail application that never returns to the caller. This will be discussed later in detail.

When a block appears in a tail position, then the last expression in the block is also in a tail position (but the other expressions in the block are not). Therefore, only the last application of EXPRCOM (which compiles an expression) in BLOCKCOM passes TA along, while all other applications in BLOCKCOM set it to NIL.

```
(SETQ BLOCKCOM
  (LAMBDA (X TA)
     (COND ((EQ X NIL))
           ((EQ (CDR X) NIL)
             (EXPRCOM (CAR X) TA))
           (T (EXPRCOM (CAR X) NIL)
              (BLOCKCOM (CDR X) TA)))))
```

CONDCOM compiles the clauses of the COND special form. For a given clause

```
(predicate expr1 ...)
```

it first compiles the predicate with EXPRCOM, which leaves the result in REF(EXPR), the EXPR register. It then compiles the conditional block (expr1 ...) wrapped in an “if” statement at C level. Generally,

```
(COND (A B ... C) (D E ... F) ...)
```

compiles to

```
A;
if (ref(expr) != NIL) {
        B;
        ...
        C;
}
else {
    D;
    if (ref(expr) != NIL) {
        E;
        ...
        F;
    }
    else {
        ...
        if (ref(expr) != NIL) {
            ...
        }
        else {
            ref(expr) = NIL;
        }
    }
}
```

Note that compiling an expression (like A or B, above) always leaves its value in REF(EXPR). Of all expressions in the example above, only C and F are in tail positions. The BLOCKCOM function takes care of this detail.

When a COND clause has an empty consequent, then the predicate is *not* in a tail position, because its truth still value has to be checked when it returns. A clause consisting only of a predicate P compiles to

```
P;
if (ref(expr) != NIL) { } else { ... }
```

which automatically leaves the value of P in REF(EXPR) when it is not NIL. Hence predicate-only clauses need no special code in the compiler. Their evaluation falls out as a by-product of the algorithm used by CONDCOM.

When CONDCOM finds a clause with a predicate of T, it just compiles the consequent of that predicate and ignores all subsequent clauses. The program

```
(COND (T FOO) (BAR BAZ))
```

will compile in the same way as just FOO.

There is a special case where T is the predicate in a predicate-only clause: the clause (T) compiles to

```
ref(expr) = true;
```

in order to make (COND (T)) deliver the expected value.

In the days of ancient LISP some programmers had developed the habit of leaving out a predicate of the form T in the last clause of COND. E.g.:

```
(COND ((FOO X) (BAR X))
      (T (BAZ X)))
```

was written as

```
(COND ((FOO X) (BAR X))
      ((BAZ X)))
```

This looks like a clever trick to save the evaluation of one atom. However, the predicate of COND is *never* in a tail position, so the application of BAZ in the above case cannot be tail call optimized when the T is missing. Also, some compilers optimize away predicates of the form T anyway, as LISCMP does.

The difference did not matter much when tail call optimization was not widespread, but as soon as the optimization became widely available, the habit became a bad habit.

```
(SETQ CONDCOM
  (LAMBDA (X TA)
    (COND ((EQ X NIL)
           (EMIT "REF(EXPR) = NIL;"))
          ((EQ T (CAAR X))
           (COND ((EQ NIL (CDAR X))
                  (BLOCKCOM (CAR X) TA))
                 (T (BLOCKCOM (CDAR X) TA))))
          (T (EXPRCOM (CAAR X) NIL)
             (EMIT "IF (REF(EXPR) != NIL) {")
             (BLOCKCOM (CDAR X) TA)
             (EMIT "} ELSE {")
             (CONDCOM (CDR X) TA)
             (EMIT "}")))))
```

VARSYMP returns T if its argument denotes an atom that can be used as a variable. All atoms except for T and NIL can be variables.

BTW, no, the clause ((ATOM X)) is not suffering from the habit explained above! It is the predicate of a predicate-only clause.

```
(SETQ VARSYMP
  (LAMBDA (X)
    (COND ((EQ X NIL) NIL)
          ((EQ X T) NIL)
          ((ATOM X)))))
```

SETQCOM compiles the SETQ special form. As in early LISP only one *variable* and one expression are accepted. When the first argument of SETQ is not a variable, and error will be reported. (SETQ V X) forms compile to

```
X;
val(v) = ref(expr);
```

where *v* is the cell containing the root of the atom V. SETQCOM creates a description of the the atom V and adds it to the CELLS list, if it does not already exist. Therefore it can introduce new variables.

```
(SETQ SETQCOM
  (LAMBDA (X)
    (COND ((NOT (VARSYMP (CADR X)))
           (HALT "SETQ: EXPECTED SYMBOL")))
    (EXPRCOM (CADDR X) NIL)
    (EMIT "VAL("
          (NUMBER (ADD-ATOM (CADR X)))
          ") = REF(EXPR);")))
```

FUNCOM compiles a *lambda function*. Generally, the form

```
(LAMBDA VARS EXPR1 ...)
```

compiles to

```
goto(skip);
label(fun);
bind(VARS);
EXPR1;
...
unbind();
k = retn(); break;
label(skip);
ref(expr) = mkfun(fun);
```

The *skip* and *fun* labels are generated by FUNCOM. The *skip* label is used to skip over the function body and the *fun* label is used to mark the compiled code of the function.

The code of the generated function first *binds* the arguments in the current call frame to the function variables listed in VARS. It then evaluates the expressions in the body of LAMBDA, unbinds the arguments again and returns to the caller, leaving the value returned by the function in REF(EXPR).

After skipping over the body, the emitted program will create a new function object and store it in the EXPR register.

There are a lot of things going on under the hood, so let us have a closer look. A lot of technical details will follow; if you are not interested in those, feel free to skip ahead to the summary at the end of this description! (Page 93)

The attentive reader may have noticed the “break” after the call to retn(). This is the way in which compiled LISP code will transfer control to labels. Because the C language has no (portable) computed “goto”, the following construct is used:

```
for (k=0;;) switch(k) {
case 0:
    /* compiled LISP code follows here */
}
```

Control can then be passed to a specific “case” by setting k to its value and issuing a “break”. Voila: computed “goto”. [Feeley2004]

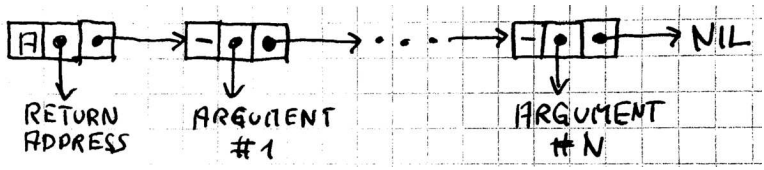


Fig. 7 – Initial call frame with argument list

The retn() function returns the value of a label that immediately follows the call to a function. By setting k (the “instruction pointer”) to that value and doing a “break”, control is passed back to the

caller. Retn() fetches the value of k from the current call frame. A LISCMP *call frame* is displayed in figure 7.

A *call frame* is a weird structure, because it looks like an atom name, but has regular cons cells in its tail. This does not matter much, though, because it is normally invisible to the programmer (but you *can* make it visible with *CAR and *CDR).

The atomic cell at its head contains the *return address* of the current call, which is exactly the k extracted by retn().

Initially the cdr part of the atomic cell is just a list of values that are passed to the function being applied:

$$(value_1 \ \cdots \ value_N)$$

The bind() function then replaces this list (destructively!) with an association list of the form

$$((var_1 \ . \ ovalue_1) \ \cdots \ (var_N \ . \ ovalue_N))$$

where each $ovalue$ is the "outer value" of the associated function variable var from VARS. At the same time bind() changes the bindings of the variables in VARS.

E.g., given the variables VARS=$(x\ y\ z)$ and their (outer) bindings $x = ox,\ y = oy$, and $z = oz$ and an argument list $(a\ b\ c)$, bind() performs these steps for each tuple consisting of a member of $(x\ y\ z)$ and a corresponding member of $(a\ b\ c)$:

- create a pair (association) containing an atom from VARS and the value that is currently bound to that atom, e.g.: (x . ox)
- bind that atom to the corresponding value in the call frame, e.g.: x=a
- replace the value in the call frame with the association created above, e.g.: ((x . ox) b c)

After performing these steps the variables of the function are bound to the arguments passed to the function, and the "outer"

values of the variables are safely stored away in the call frame. The call frame is then in the state displayed in figure 8.

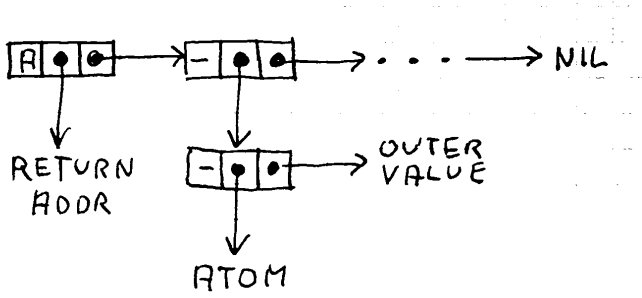


Fig. 8 – Call frame with saved outer values

In the above example, the new bindings of the variables would be $x = a$, $y = b$ and $z = c$, and the tail of the call frame would be the association list ((x . ox) (y . oy) (z . oz)).

The *outer value* of a variable is the value that the variable had before the application of the currently evaluating function took place. For instance, the "inner value" (or just "value") of X in the function FOO is INNER in the following example, and its outer value is OUTER:

```
(LABEL ((X (QUOTE OUTER))
        (FOO (LAMBDA (X) X)))   ; X = INNER
  (FOO (QUOTE INNER)))
```

Inner and outer values arise whenever the binding of a variable is changed temporarily. The inner value temporarily replaces the outer value. Here is another example:

```
(LABEL ((FOO (QUOTE OUTER)))
  (LABEL ((FOO (QUOTE INNER)))
    (PRINT FOO))                    ;  INNER
  FOO)                            ==>  OUTER
```

This expression will print the inner value of FOO which at that point replaces the outer value of FOO, but it will return the outer value, which is restored when exiting the inner LABEL.

At the end of a function body unbind() restores the bindings saved in the call frame by bind(), thereby binding all function variables to their outer values again.

There is one minor complication, though! When the lambda function is an LEXPR (a *list expression* where VARS a single variable that will be bound to the entire list of arguments), then the call frame as modified by bind() has to have a slightly different shape, which is displayed in figure 9.

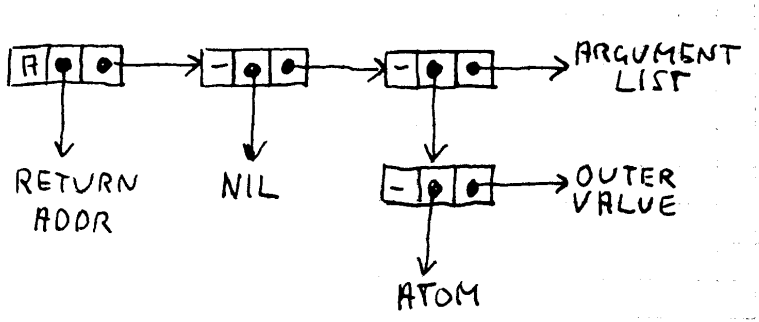


Fig. 9 – LEXPR call frame with saved outer value

Here the original argument list is being preserved, because it has to be bound in its entirety to the only argument of the function. It is preceded by a single association linking that variable to its outer value, and then there is a NIL atom between the return address and that association. This NIL atom serves as an indicator to unbind(), which will then only restore the value of the single variable.

After unbinding/restoring the variables of a function, the function will return by extracting the return address from the call frame and jumping to that address. The retn() function, which delivers the return address, also removes the call frame from the stack.

The mkfun() function, finally, creates a *function object* for the given function. A function object is an ordinary LISP value that can be passed to functions and printed. It is also the only object that can be *applied* to some values, thereby *calling* the function. The internal structure of a function object is shown in figure 10.

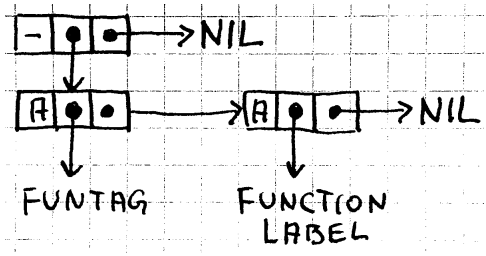


Fig. 10 – Function object

There is one atomic cell pointing to FUNTAG (see page 72), which is used as a marker for function objects, and another atomic cell carrying the label of the compiled function. The cons cell at the root of the structure merely turns the object into a regular atom, so ATOM will return T for it.

The FUNTAG is used by the PRINT function to identify function objects and during function application to make sure that the applied object really is a function.

SUMMARY

That was a lot of prose to describe a short function. Most of the complexity is hidden away in the low-level functions, which can be found in the appendix. The big picture, though, is just this: a function

- saves outer values of function variables
- binds function variables to arguments
- evaluates the function body

- unbinds variables, thereby restoring their outer values
- removes its own call frame and returns to the caller

The *outer value* of a variable is the value that the variable has outside of the function.

The value returned by a function is the value that is left in the EXPR register when evaluating the last expression in the function body.

Note that the last expression in the body of a function is *always* in a tail position, regardless of the context in which the function appears.

```
(SETQ FUNCOM
  (LAMBDA (X)
    (LABEL
      ((SKIP (MAKE-LABEL))
       (FUN  (MAKE-LABEL)))
      (EMIT "GOTO(" (NUMBER SKIP) ");")
      (EMIT "LABEL(" (NUMBER FUN) ");")
      (EMIT "BIND("
            (NUMBER (ADD-OBJECT (CADR X))) ");")
      (BLOCKCOM (CDDR X) T)
      (EMIT "UNBIND();")
      (EMIT "K = RETN(); BREAK;")
      (EMIT "LABEL(" (NUMBER SKIP) ");")
      (EMIT "REF(EXPR) = CONS3(CONS3(FUNTAG, "
            "CONS3(" (NUMBER FUN)
            ", NIL, ATOMTAG), ATOMTAG), NIL, "
            "0);" ))))
```

FUNAPP compiles a function application. In general, each expression of the form

(*fun* $expr_1 \cdots expr_N$**)**

will be compiled to code like this:

newframe();
$expr_N$**;**
toframe();
...
$expr_1$**;**
toframe();
fun;
apply(*ret***);**
label(*ret***);**

The newframe() function pushes an empty and incomplete call frame (NIL) to the stack and toframe() conses the current expression (EXPR) to the front of the call frame. Hence the call frame after evaluating the *arguments* $expr_N$ through $expr_1$ will look like this:

($expr_1 \cdots expr_N$**)**

I.e., it will contain the values of the arguments passed to the function *fun*. FUNAPP evaluates function arguments right-to-left, but this is merely an implementation detail and should not be replied upon. You might as well evaluate left-to-right and append argument values to the call frame destructively.

The apply() function attaches the return label *ret* to the call frame on top of the stack (thereby making the frame complete) and transfers control to the label stored in the function object *fun* (which is stored in EXPR at that point). See the description of FUNCOM, further above, for details on function objects and transfer of control.

There is one special case to consider here: when the function application compiled by FUNAPP is in a *tail position*, then there is no need for *fun* to return to the label *ret*. This is signaled to the

apply() function by passing -1 to it instead of a valid label (which must be positive).

When performing a tail application, apply() fetches the return label from the call frame of the caller and inserts it in the frame of the callee. It then removes the caller's frame and unbinds its variables, so that the number of frames on the stack stays the same. Hence a function application in a tail position does not allocate any additional space on the stack.

Here is what apply(-1) does. Given the program

```
(SETQ F (LAMBDA (X Y) (ATOM (G X Y))))
(SETQ G (LAMBDA (X Y) (H Y X)))  ; <--- WE ARE
(SETQ H (LAMBDA (X Y) X))        ;       HERE
(F (QUOTE FOO) (QUOTE BAR))
```

here is the stack at the point where apply() is being called inside of the function G:

```
( (BAR FOO)       ; H'S STACK FRAME
  (ret_F FOO BAR) ; G'S STACK FRAME
  ...)
```

The arguments to be passed to H are already on the stack, but the return address is still missing from the new call frame. Under the incomplete call frame of H there is the complete call frame that was used to invoke G inside of F. It will be used to return to F eventually, because the application of G in F is not a tail application.

The application of H in G is a tail application, though, so apply() takes the return address of the caller's frame (ret_F), attaches it to the incomplete call frame of H, and then removes the caller's frame:

```
( (ret_F BAR FOO)  ; H'S STACK FRAME
  ...)
```

The effect of this manipulation is that H will eventually return to F instead of G. The call frame of G (created by F) is removed and its variables are unbound by apply() when calling H and not by retn() when returning from G to to F (which will never happen).

The practical effect of this optimization is that tail-recursive functions will run in constant space. A tail-recursive function is a function in which all recursive applications are in tail positions. This is the case, for instance, in ASSOC and MEMBER (page 49) and many other functions of the LISP system.

```
(SETQ FUNAPP
  (LAMBDA (X TA)
    (EMIT "NEWFRAME();")
    (LABEL
      ((LOOP (LAMBDA (A)
         (COND ((EQ A NIL))
               (T (EXPRCOM (CAR A) NIL)
                  (EMIT "TOFRAME();")
                  (LOOP (CDR A)))))))
      (LOOP (REVERSE (CDR X))))
    (EXPRCOM (CAR X) NIL)
    (COND (TA (EMIT "K = APPLY(-1); BREAK;"))
          (T (LABEL ((RET (MAKE-LABEL)))
               (EMIT "K = APPLY("
                     (NUMBER RET)
                     "); BREAK;")
               (EMIT "LABEL("
                     (NUMBER RET) ");")))))))
```

LABCOM compiles the LABEL special form. In principle a LABEL special form is an in-situ lambda function application, but there are some differences. That is, the following two programs are *almost* equivalent:

```
(LABEL                              ((LAMBDA (F G)
  ((F (LAMBDA (X) (G X)))             (F (QUOTE FOO)))
   (G (LAMBDA (X) X)))               (LAMBDA (X) (G X))
  (F (QUOTE FOO)))                   (LAMBDA (X) X))
```

One significant difference is that the left program works and the right one does not (but with a small modification it will!). We will return to that at the end of this description!

What the form

```
(LABEL ((v1 a1)
        ...
        (vN aN))
  x1 ...)
```

basically does is to bind each v_i to the value of the corresponding a_i, evaluate the expressions $x_1 \cdots$ with those bindings in effect, and then unbind the variables $v_i \cdots$ again. It looks like a function application and even uses the same mechanism as function application: it pushes a call frame to the stack, stores outer values in it, and restores the outer values at the end.

There are two little differences, though: (1) LABCOM uses a call to pop() instead of retn() to remove the call frame and (2) it stores NIL instead of a return address in the call frame, because LABEL will not return anywhere.

One major difference between LABEL and in-situ function application is that LABEL is guaranteed to evaluate the arguments a_i in top-to-bottom order: a_i will always evaluate before a_{i+1}. In addition it will always bind a_i to v_i *before* evaluating a_{i+1}. Therefore the following program works fine:

```
(LABEL ((X (QUOTE (BAZ)))
        (Y (CONS (QUOTE BAR) X))
        (Z (CONS (QUOTE FOO) Y)))
  Z)
     ==>  (FOO BAR BAZ)
```

In an in-situ lambda function application this would not work, because it would *first* evaluate all arguments a_i and *then* bind them to the corresponding variables v_i. I.e., when the value of Y would be computed, X would still be unbound (or, rather, bound to its outer value).

Another major difference is that the last expression in LABEL is *not* in a tail position. This is the reason why the initial example in the description of LABCOM works when using LABEL and does not work when using a LAMBDA function and in-situ application! Have another look at this program:

```
((LAMBDA (F G)
   (F (QUOTE FOO)))
 (LAMBDA (X) (G X))
 (LAMBDA (X) X))
```

Here the application (F (QUOTE FOO)) is a tail application. This means that F and G are unbound before transferring control to F. So when F applies G, G is no longer bound to the desired value!

This means that the program could be fixed by making the application of F a non-tail application, and indeed:

```
((LAMBDA (F G)
   ((LAMBDA (X) X)
    (F (QUOTE FOO))))
 (LAMBDA (X) (G X))
 (LAMBDA (X) X))          ==>  FOO
```

This is why LABCOM sets the tail position argument to NIL when calling BLOCKCOM. The advantage of LABEL is that it can define local recursive (or interdependent) functions. Its disadvantage is that it is not tail-recursive. We will further explore this detail the science chapter!

```
(SETQ LABCOM
  (LAMBDA (X)
```

```
(EMIT "NEWFRAME();")
(LABEL
  ((LOOP (LAMBDA (X)
     (COND
       ((EQ X NIL))
       (T (COND ((NOT (VARSYMP (CAAR X)))
                 (HALT
                   "LABEL: EXPECTED SYMBOL")))
          (SETQ LABCOM-TMP
                (NUMBER (ADD-ATOM (CAAR X))))
          (EMIT "REF(EXPR) = CONS3("
                LABCOM-TMP ", VAL("
                LABCOM-TMP "), 0);")
          (EMIT "TOFRAME();")
          (EXPRCOM (CADAR X) NIL)
          (EMIT "VAL("
                (NUMBER (ADD-ATOM (CAAR X)))
                ") = REF(EXPR);")
          (LOOP (CDR X)))))))
  (LOOP (CADR X)))
(EMIT "REF(EXPR) = NIL;")
(EMIT "TOFRAME();")
(BLOCKCOM (CDDR X) NIL)
(EMIT "UNBIND();")
(EMIT "POP();")))
```

The CHECKTAG and SETTAG functions emit code for checking and setting the various tag bits of a cell. They implement the special forms *ATOMP, *MARKP, *TRAVP, *SETATOM, *SETMARK, and *SETTRAV. The code emitted for these forms is almost identical, only the tag bits differ.

For instance, (*ATOMP X) will compile to

```
X;
ref(expr) = tag[ref(expr)] & atomtag? true: nil;
```

(*SETATOM X T) will compile to

```
X; tag[ref(expr)] |= atomtag;
```

and (*SETATOM X NIL) will compile to

```
X; tag[ref(expr)] &= ~atomtag;
```

Code for the other tag bits will be compiled accordingly.

Note that *SET... forms are special forms and their second arguments will *not* be evaluated; they must be either T (to set a tag bit) or NIL (to clear it).

```
(SETQ CHECKTAG
  (LAMBDA (X)
    (EXPRCOM (CADR X) NIL)
    (LABEL
      ((F (CADR (ASSOC (CAR X)
                  (QUOTE ((*ATOMP ATOMTAG)
                          (*MARKP MARKTAG)
                          (*TRAVP TRAVTAG)))))))
      (EMIT "REF(EXPR) = (TAG[REF(EXPR)] & "
            F ")? TRUE: NIL;"))))

(SETQ SETTAG
  (LAMBDA (X)
    (EXPRCOM (CADR X) NIL)
    (LABEL
      ((F (CADR (ASSOC (CAR X)
                  (QUOTE ((*SETATOM ATOMTAG)
                          (*SETMARK MARKTAG)
                          (*SETTRAV
                           TRAVTAG))))))
      (EMIT "TAG[REF(EXPR)]"
            (COND ((CADDR X) " |= ") (T " &= ~"))
            F ";"))))
```

Some inline code emitted by the compiler needs two arguments, like that implementing CONS or EQ. The TWOARGS function evaluates two argument expressions and puts their values in the EXPR and EXPR2 registers. The generated code looks like this:

```
argument 2;
push();
argument 1;
ref(expr2) = pop();
```

Evaluating the second argument will put its value in the EXPR register, which will then be pushed to the stack. Evaluating the first argument will also put its value in EXPR, so at the end the first argument value will be in EXPR and the second one in EXPR2.

```
(SETQ TWOARGS
  (LAMBDA (X)
    (EXPRCOM (CADDR X) NIL)
    (EMIT "PUSH();")
    (EXPRCOM (CADR X) NIL)
    (EMIT "REF(EXPR2) = POP();")))
```

EXPRCOM compiles the given expression by delegating compilation to one of the functions above. Some cases are handled directly by EXPRCOM, though. For instance, it inlines most of the built-in functions, like CAR, CONS, etc. For single-argument functions, it evaluates the argument first and then inlines the function code. For instance, (CAR X) will compile to

```
X;
if (symbolic(ref(expr))) car_err();
ref(expr) = car[ref(expr)];
```

Two-argument functions will have their arguments placed in the EXPR and EXPR2 registers, as described in the TWOARGS function above, and then the inline code of the function will follow.

The CONS function, for example, will then generate the following inline code:

```
ref(expr) = cons3(ref(expr), ref(expr2), 0);
```

The QUOTE special form just loads the cell address of the desired object into the EXPR register. It also adds the object to the CELLS list.

There is one interesting case in EXPRCOM and this is the special case for the in-situ application of *lambda functions*. The expression

```
((LAMBDA (X) X) (QUOTE FOO))
```

will generate a regular function call, and not a *tail application*, even if the application appears in a tail position. This hack is a workaround for the following case

```
((LAMBDA (Y)
   ((LAMBDA (Y)
      ((LAMBDA (X) (LIST Y X))
       (QUOTE WORK)))
    (QUOTE DOES)))
 (QUOTE DOESNT))
```

If the inner lambda function application would be a tail application, the program would evaluate to (DOESNT WORK), while it clearly should evaluate to (DOES WORK). How does this happen?

In the outermost scope, Y Is bound to DOESNT and in the middle scope, it is bound to DOES. When the innermost function would be tail-applied, its application would unbind the variables in the call frame of its caller, thereby restoring the binding of Y to DOESNT.

By catching this special case and making the function application a regular application, the issue is resolved.

The drawback of the fix is, of course, that in-situ lambda function applications cannot evaluate in constant space. This is not as bad as it sounds, though! For example,

```
((LAMBDA (X) (X X)) (LAMBDA (X) (X X)))
```

still evaluates in constant space! Can you explain why?

```
(SETQ EXPRCOM
  (LAMBDA (X TA)
    (COND ((EQ NIL X)
           (EMIT "REF(EXPR) = NIL;"))
          ((EQ T X)
           (EMIT "REF(EXPR) = TRUE;"))
          ((EQ '*POOL X)
           (EMIT "REF(EXPR) = 0;"))
          ((EQ '*LIMIT X)
           (EMIT "REF(EXPR) = "
                 (NUMBER LIMIT) ";"))
          ((ATOM X)
           (EMIT "REF(EXPR) = VAL("
                 (NUMBER (ADD-ATOM X)) ");"))
          ((EQ 'COND (CAR X))
           (CONDCOM (CDR X) TA))
          ((EQ 'LABEL (CAR X))
           (LABCOM X))
          ((EQ 'LAMBDA (CAR X))
           (FUNCOM X))
          ((EQ 'PROGN (CAR X))
           (BLOCKCOM (CDR X) TA))
          ((EQ 'QUOTE (CAR X))
           (EMIT "REF(EXPR) = "
                 (NUMBER (ADD-OBJECT (CADR X)))
                 ";"))
          ((EQ 'SETQ (CAR X))
           (SETQCOM X))
          ((EQ 'ATOM (CAR X))
           (EXPRCOM (CADR X) NIL)
           (EMIT "REF(EXPR) = ATOMIC()? "
```

```
        "TRUE: NIL;"))
((EQ 'CAR (CAR X))
  (EXPRCOM (CADR X) NIL)
  (EMIT "IF (SYMBOLIC()) CAR_ERR();")
  (EMIT "REF(EXPR) = CAR[REF(EXPR)];"))
((EQ 'CDR (CAR X))
  (EXPRCOM (CADR X) NIL)
  (EMIT "IF (SYMBOLIC()) CDR_ERR();")
  (EMIT "REF(EXPR) = CDR[REF(EXPR)];"))
((EQ 'CONS (CAR X))
  (TWOARGS X)
  (EMIT "REF(EXPR) = CONS3(REF(EXPR), "
        "REF(EXPR2), 0);"))
((EQ 'EQ (CAR X))
  (TWOARGS X)
  (EMIT "REF(EXPR) = REF(EXPR) == "
        "REF(EXPR2)? TRUE: NIL;"))
((EQ '*HALT (CAR X))
  (EXPRCOM (CADR X) NIL)
  (EMIT "HALT(ATOMNAME(REF(EXPR)), "
        (NUMBER LIMIT) ");"))
((MEMBER (CAR X)
         '(*ATOMP *MARKP *TRAVP))
  (CHECKTAG X))
((EQ '*CAR (CAR X))
  (EXPRCOM (CADR X) NIL)
  (EMIT "REF(EXPR) = CAR[REF(EXPR)];"))
((EQ '*CDR (CAR X))
  (EXPRCOM (CADR X) NIL)
  (EMIT "REF(EXPR) = CDR[REF(EXPR)];"))
((EQ '*DUMP (CAR X))
  (EXPRCOM (CADR X) NIL)
  (EMIT "DUMP(REF(EXPR));"))
((EQ '*LOAD (CAR X))
  (EXPRCOM (CADR X) NIL)
```

```
        (EMIT "LOAD(REF(EXPR));"))
      ((EQ '*NEXT (CAR X))
        (EXPRCOM (CADR X) NIL)
        (EMIT "REF(EXPR)++;"))
      ((MEMBER (CAR X)
               '(*SETATOM *SETMARK *SETTRAV))
        (SETTAG X))
      ((EQ '*READC (CAR X))
        (EMIT "REF(EXPR) = READC();"))
      ((EQ '*RPLACA (CAR X))
        (TWOARGS X)
        (EMIT "CAR[REF(EXPR)] = "
              "REF(EXPR2);"))
      ((EQ '*RPLACD (CAR X))
        (TWOARGS X)
        (EMIT "CDR[REF(EXPR)] = "
              "REF(EXPR2);"))
      ((EQ '*WRITEC (CAR X))
        (EXPRCOM (CADR X) NIL)
        (EMIT "WRITEC(REF(EXPR));"))
      ((NOT (ATOM (CAR X)))
        (FUNAPP X NIL))
      (T (FUNAPP X TA)))))
```

The PROLOG function emits the head of the run() function, which will contain the compiled LISP code. The variable k serves as an *instruction pointer*. The switch in an infinite loop implements computed "goto", as explained earlier (page 89). The setup() function will be described further below.

```
(SETQ PROLOG
  (LAMBDA ()
    (EMIT)
    (EMIT
```

```
 "/******** LISCMP OUTPUT FOLLOWS ********/")
(EMIT)
(EMIT "VOID RUN(VOID) {")
(EMIT "INT K;")
(EMIT "SETUP();")
(EMIT "FOR (K=0;;) SWITCH (K) {")
(EMIT "CASE 0:")))
```

EMIT-OBLIST creates a *spine* for the *object list* OBLIST by adding descriptions for it to the CELLS list (see page 78). The spine of a list is the chain of cons cells that connects the individual elements of a list.

Up to this point the addresses of literal objects were stored in a list structure that is part of the compiler. However, the purpose of the OBLIST is to protect those literals at the run time of the generated program. Hence a new spine for the list must be created that is itself part of the initial cell pool of the program.

EMIT-OBLIST also emits code to store the root cell of the new spine of OBLIST in the special slot OBLIST of the cell pool (see page 72). Note that “OBLIST” refers to two different concepts here: (1) to the list of objects itself and (2) the special slot that points to it.

The object list is necessary because references to literal objects, like LISP lists, cannot be marked in compiled code, because the garbage collector cannot examine compiled programs. This is why copies of such references are stored in the OBLIST, which will be marked “used” during garbage collection, thereby protecting the literal objects from being recycled.

Note that atoms (symbols) are not contained in the OBLIST, because they are stored in a separate structure, the SYMLIS (symbol list).

```
(SETQ EMIT-OBLIST
  (LAMBDA ()
    (LABEL
      ((LOOP (LAMBDA (OL A)
         (COND ((EQ OL NIL)
                  (EMIT "REF(OBLIST) = "
                        (NUMBER A) ";"))
               (T (LOOP (CDR OL)
                        (MAKE-CELL
                          (CAR OL) A NIL)))))))
      (LOOP OBLIST NIL-ADDR))))
```

The CHAR function is used to create C language representations of character code points *and* make sure that they are upper-case. Because this is purely symbolic LISP, which has no ability to examine or modify characters (except via slow association list look-up), CHAR delegates part of its task to the low-level runtime library.

It identifies newline, apostrophe, and backslash characters and emits the proper sequences for them, but passes all other characters to the u() macro, which is shorthand for toupper(). CHAR creates transient symbols of the form U('*char*') so it does not pollute the SYMLIS.

```
(SETQ CHAR
  (LAMBDA (C)
    (COND ((EQ C *NL) "'\\N'")
          ((EQ C "'") "'\\''")
          ((EQ C "\\") "'\\\\'")
          (T (MAKESYM
               (LIST "U" "(" "'" C "'" ")"))))))
```

INIT-CELLS traverses the CELLS list (page 78) and emits a cell pool initialization line for each cell allocated in the initial cell pool of the compiled program. The emitted code will set up the initial cell pool when the compiled program is being run.

Each initialization line has the form

car[x**] =** a**; cdr[**x**] =** d**; tag[**x**] =** f**;**

where x is the address of the cell to initialize, a is its car value, d its cdr value, and f its tag bits (which will be either 0 or *atomflag*).

The output of INIT-CELLS will be written to the setup() function, which may have a size of several thousand lines, even for moderately-sized LISP programs.

```
(SETQ INIT-CELLS
  (LAMBDA ()
    (LABEL
      ((INIT (LAMBDA (N)
         (LABEL
           ((X (NUMBER (CAAR N)))
            (A (COND ((ATOM (CAR (CDDAR N)))
                      (CHAR (CAR (CDDAR N))))
                     (T (NUMBER
                          (CAR (CDDAR N))))))
            (D (NUMBER (CADR (CDDAR N))))
            (F (COND ((CADAR N) "ATOMTAG")
                     (T "0"))))
           (EMIT "CAR[" X "] = " A "; "
                 "CDR[" X "] = " D "; "
                 "TAG[" X "] = " F ";"))))
       (LOOP (LAMBDA (N)
         (COND ((EQ N NIL))
               (T (INIT N)
                  (LOOP (CDR N)))))))
      (LOOP CELLS))))
```

The EPILOG function emits the tail of the run() function, which contains the compiled LISP code. The tail looks like this:

```
return;
}}
```

The “return” statement will return from run() and hence terminate program execution. The braces delimit the “switch” statement and the run() function itself.

EPILOG also generates the setup() function, which initializes the OBLIST and SYMLIS slots and initializes all cells used in the initial cell pool.

The LABEL in EPILOG removes all addresses from the SYMLIS structure before allocating cells for it and emitting code to assign it to the SYMLIS slot. I.e., the (outer) MAPCAR in the LABEL performs the following transformation on each bucket of the symbol list:

$$(char\ \ (atom_1\ .\ addr_1)\ \ \ldots\ \ (atom_N\ .\ addr_N))$$
$$\rightarrow$$
$$(char\ \ atom_1\ \ \ldots\ \ atom_N)$$

This is done because the SYMLIS maintained by the compiler contains addresses of symbols, but the SYMLIS of the emitted program contains only symbols (see page 56).

```
(SETQ EPILOG
  (LAMBDA ()
    (EMIT "RETURN;")
    (EMIT "}}")
    (EMIT)
    (EMIT "VOID SETUP(VOID) {")
    (LABEL
      ((Y (MAPCAR (LAMBDA (X)
                    (CONS (CAR X)
```

```
                              (MAPCAR CAR (CDR X))))
                    SYMLIS)))
      (EMIT "REF(SYMLIS) = "
            (NUMBER (MAKE-OBJECT Y))
            ";"))
    (EMIT-OBLIST)
    (INIT-CELLS)
    (EMIT "}")))
```

The DUMP-POOL function appends a comment containing a more readable representation of the initial cell pool to the emitted C program.

```
(SETQ DUMP-POOL
  (LAMBDA ()
    (EMIT "/******** POOL DUMP FOLLOWS ********")
    (MAPCAR (LAMBDA (X)
              (PRIN1 (NUMBER (CAR X)))
              (PRIN1 " ")
              (PRIN1 (COND ((CADR X) "A")
                           (T "-")))
              (PRIN1 " ")
              (PRIN1 (COND ((ATOM (CADDR X))
                            (CADDR X))
                           (T (NUMBER
                                (CADDR X)))))
              (PRIN1 " ")
              (PRIN1 (NUMBER (CADDR (CDR X))))
              (TERPRI))
            CELLS)
    (EMIT
     "**********************************/")))
```

The LISCMP function is the entry point and outer *loop* of the LISP compiler. It reads a LISP program, expression by expression, and emits corresponding C code. It also emits the setup code needed by the emitted program.
Note that every input program *has to* finish with the atom

```
*STOP
```

If it does not, the compiler will keep reading indefinitely, because there is no other means of end-of-file detection. Also note that the symbol *STOP inside of a list will not cause the compiler to stop. The symbol has to appear at the top level, outside of all lists.

```
(SETQ LISCMP
  (LAMBDA ()
    (PROLOG)
    (LABEL
      ((LOOP (LAMBDA (X)
         (COND ((EQ (QUOTE *STOP) X))
               (T (EXPRCOM X NIL)
                  (LOOP (READ)))))))
      (LOOP (READ)))
    (EPILOG)
    (DUMP-POOL)))
```

Run the LISP compiler.

```
(LISCMP)
```

The compiler driver (a shell script on Unix) should provide the final *STOP symbol, but including it here will not hurt!

```
*STOP
```

USING LISCMP

Compiling programs with LISCMP is not as straight-forward as submitting a LISP expression to the metacircular evaluator, but a little bit of glue in the form of a Unix shell script can make it *almost* as easy to use!

Without such a script, the steps involved in program compilation with LISCMP are as follows. (Note: we will use three intermediate files here, called “run.lisp”, “run.c”, and “run”. These files are transient and should not be used for anything else!)

(1) Copy the file “lisp.lisp” to “run.lisp”. This is the source code of the LISP system, and it has to be recompiled each time you compile a LISP program. On a Unix system, type

```
cp lisp.lisp run.lisp
```

(2) Append your LISP code to the file “run.lisp”. For instance:

```
echo "(LIST 'FOO 'BAR 'BAZ)" >>run.lisp
```

(3) Append an end-of-file (*STOP) marker:

```
echo "*STOP" >>run.lisp
```

(4) Copy the file “lisrun.c” to “run.c”. This is the low-level part of the runtime library. It has to be recompiled every time. On a Unix system:

```
cp lisrun.c run.c
```

(5) Compile “run.lisp” with LISCMP, convert it to lower case, and append it to “run.c”:

```
liscmp <run.lisp | dd conv=lcase >>run.c
```

The conv=lcase option of the DD command converts input to lower case. If you are not on a Unix system: a SCHEME program performing this conversion can be found in the appendix.

(6) Compile “run.c” with your local C compiler:

```
cc -o run run.c
```

The program “run” is the executable version of your LISP program!

Quite a bit of work, isn’t it? A shell script really does help a lot here. The one in the appendix would work this way ($ is the Unix prompt, ^D (control-D) is the end-of-file marker):

```
$ sh lcg -
(LIST 'FOO 'BAR 'BAZ)
^D
COMPILE
RUN
(FOO BAR BAZ)
$
```

The script is also contained in the LISCMP source code package!

BOOTSTRAPPING LISCMP

You may have noticed above that this process assumes the existence of a “liscmp” executable, i.e. a LISCMP compiler compiled with itself.

There are two ways to obtain a LISCMP executable.

(1) compile liscmp with the lcg script:

```
echo '*stop' | sh lcg liscmp.lisp
```

and then rename the file “run” “liscmp”. This looks like magic, because the lcg script uses LISCMP to compile LISCMP. What the script actually does is to use an existing SCHEME interpreter on your system and a version of LISCMP that is written in SCHEME to generate the initial compiler, the so-called *stage-0 compiler*. (You may have to choose a different SCHEME interpreter in the lcg script!)

If you run the above command again, LISCMP will actually self-compile. The lcg script will check for a “liscmp” executable and use it in the place of the SCHEME version if it exists.

(2) The other way to obtain a LISCMP executable is even simpler. The LISCMP package contains a pre-compiled compiler named “boot.c”, which is the output of LISCMP compiling itself, but before submitting it to the C compiler. Just finish its compilation to obtain a working stage-0 binary:

```
cc -o liscmp boot.c
```

Finally, you should perform the so-called *triple test* on the compiler to make sure it works properly. After generating the stage-0 binary, compile LISCMP with itself

```
echo '*stop' | sh lcg liscmp.lisp
```

Save the intermediate C code in a temporary file (_stage_1) and rename the compiler output “liscmp”. This is your stage-1 compiler, which has been compiled with itself.

```
mv run.c _stage_1
mv run liscmp
```

Finally compile the compiler with itself one more time, giving a stage-2 compiler. The intermediate code generated for the stage-2 compiler should be identical to the output generated for the stage-1 compiler.

```
echo '*stop' | sh lcg liscmp.lisp
diff run.c _stage_1
```

If it is, the triple test has been passed!

HACKS & KLUDGES

The typeface used here is that of the Wanderer Continental typewriter. The heading was typed on a real typewriter, photographed, blown up, and converted to Postscript.

The Continental is older than LISP 1, but I suspect that some of them were still in use in the 1960's.

THERE ARE some obvious shortcomings with the compiler presented in the previous chapter. It does some weird end-of-file detection, and in case of erroneous input programs it sometimes prints a helpful message and sometimes its code just runs into an undefined expression, like taking the car part of an atom, and then it aborts with a not-so-helpful message. It also consumes a *lot* of memory for such a little compiler. Let us have a closer look at some of these issues.

LISCMP is a proof of concept and not a production compiler. It pretty much expects its input program to be syntactically correct and performs *very* little error checking. Erroneous programs may result in

- *compiler error messages* and termination of compilation
- the compiler itself aborting with an internal error
- the generation of code that will fail at *runtime*
- the generation of code with undefined behavior

Can these behaviors be provoked? The first kind can easily be triggered. There are two spots in the LISCMP source code where the compiler applies HALT in order to exit with an error message. This happens when

- A SETQ form has a non-variable as its first argument
- A LABEL binding has a non-variable in the place of a variable

Remember that variables are (unquoted) atoms *except for T and NIL* (see page 87). So expressions like

```
(SETQ NIL (QUOTE FOO))
```

or

```
(LABEL ((T (QUOTE FOO))) T)
```

are not well-formed. Because the compiler tests these cases, they will cause an error message to print and compilation to be aborted

gracefully when they appear in a program. For instance, the program

```
(SETQ (FOO) (QUOTE BAR))
```

will abort compilation with the error message

```
*** SETQ: EXPECTED VARIABLE
```

No location information is supplied and not even the object that is found in the place of a variable is being reported. The error handling mechanism, like the rest of the compiler, is quite rudimentary. And this is the *most* helpful output you can expect from the compiler in case of an error.

Making the compiler itself abort is easy. Whenever there is a (non-NIL) atom in a place where the compiler wants to apply CAR or CDR to a list, a runtime error will be reported. The following program is missing the outer parentheses around the binding part of LABEL:

```
(LABEL (FOO (QUOTE BAR)) FOO)
```

When LISCMP attempts to compile this program, it will abort with the message:

```
*** CAR: EXPECTED LIST: FOO
```

because the atom FOO is found in the place where the compiler expected the first list containing a variable/argument pair. Of course, the message is not very helpful, because it describes how the *compiler* failed and not how compilation failed. At least it prints the offending atom, which may serve as a clue as to the source of the error.

In fact the above is at the same time and instance of the third kind of error, because the compiler itself is in this case a program that fails at run time. Failure at run time is easily triggered by expressions like

```
(CDR (QUOTE FOO))
```

The following errors will be caught by the LISCMP low-level runtime library:

- running out of cells
- taking the car or cdr of a non-nil atom
- passing the wrong number of arguments to a function
- application of a non-function

Taking the car/cdr of an atom and applying a non-function to some values will helpfully report the offending object, the other errors will just cause a message to print.

Running out of cells is easily provoked by an expression like this one:

```
((LAMBDA (X) (X X)) (LAMBDA (X) (X (X X))))
```

This is almost the famous $\Omega \equiv ((\lambda x.\, xx)(\lambda x.\, xx))$ [Church1941], but with an additional application of x that turns the inner application of x into a regular function application instead of a tail application. Compiling and running the expression should fail very quickly.

In fact you can observe how quickly it fails by putting the garbage collector in “verbose mode”. When the LISCMP runtime finds a command line argument (no matter which one), it will activate garbage collector messages. This is how it can be done with the “lcg” compiler driver script:

```
$ sh lcg - verbose-gc
((LAMBDA (X) (X X)) (LAMBDA (X) (X (X X))))
^D
COMPILE
RUN
64358 CELLS RECLAIMED
7 CELLS RECLAIMED
```

```
0 CELLS RECLAIMED
*** OUT OF CELLS
```

Verbose garbage collections can often provide a rough estimate about the complexity and maximum space requirement of a program. For instance, evaluating Ω will show you that it computes in constant space:

```
$ sh lcg - verbose-gc
((LAMBDA (X) (X X)) (LAMBDA (X) (X X)))
^D
COMPILE
RUN
64358 CELLS RECLAIMED
64207 CELLS RECLAIMED
64208 CELLS RECLAIMED
64208 CELLS RECLAIMED
64208 CELLS RECLAIMED
...
```

When a program terminates normally with verbose GC enabled, it will also print the maximum number of cells allocated during evaluation. It is a good rule of thumb that at least half of the pool should be free after each garbage collection to assure optimum performance. At 60% allocation performance will start to degrade visibly and soon GC will become a bottleneck. The memory usage of LISCMP itself will be examined later in this chapter.

Runtime errors due to wrong argument counts in applications and applications of non-functions are also easily provoked:

```
((LAMBDA () T) (QUOTE FOO))
(NIL)
```

This leaves the fourth kind of error, which is *undefined behavior*. This kind of error really should not happen, but when it does, the result of running the compiled program is unpredictable. Errors of this kind are hard to locate and therefore really should be detected

at compile time. However, leaving this kind of error unchecked in a proof of concept is common practice, because catching all the cases would blow up the code significantly without offering much insight to the reader. Even McCarthy's minimal evaluator enters an infinite loop when an undefined variable appears in the place of the function in an application.

There are quite a few spots in LISCMP where an obvious error is not caught. For example, the arguments of lambda functions are not checked in any way. Why should they? They can be lists, empty lists, or atoms! But what happens when NIL or T appear in an list of function variables, or when the same name appears twice? For instance, which values do the following expressions have?

```
((LAMBDA (T) T) (QUOTE FOO))
((LAMBDA (X X) X) (QUOTE FOO) (QUOTE BAR))
((LAMBDA (NIL) (CDR NIL)) (QUOTE FOO))
```

The first one will either bind FOO to the atom T or it will not for some reason. If you have studied the compiler source code thoroughly, you should know which one it is!

In fact the answer is much weirder than the options given above. The program *will* bind FOO to T, but it will still not change the value of T!

```
((LAMBDA (T)        T)   (QUOTE FOO))   ==>  T
((LAMBDA (T) (*CDR T)) (QUOTE FOO))   ==>  FOO
```

We know that the *CDR of an atom is the value bound to it. So the above indicates that T is indeed bound to FOO, but at the same time the first program still evaluates to T. T is bound to FOO and not bound to FOO at the same time! Welcome to the wonderful world of undefined behavior!

The reason for this unfortunate result is that LISCMP will assume that the value of T never changes and hence inline it. T is an ordinary symbol though, so its binding can change—and *will*

change in the above program. The value will never be used, though, because the compiler inlines the *presumed* value of T, which is T.

Now for the second case, which should be simple:

```
((LAMBDA (X X) X) (QUOTE FOO) (QUOTE BAR))
```

gives FOO, right? Arguments evaluate from the right to the left, so BAR evaluates first and then FOO evaluates, so the value of the expression should be FOO.

Maybe you looked closer than that, though! While arguments *evaluate* from the right to the left, they *bind* from the left to the right. This detail is hidden in the bind() function in the low-level part of the library, though, and who wants to study some C code? Because of the order of binding,

```
((LAMBDA (X X) X) (QUOTE FOO) (QUOTE BAR))  ==>  BAR
```

Or, more correctly,

```
((LAMBDA (X X) X) 'FOO 'BAR)  ==>  undefined
```

because you really should not rely on implementation details like this one!

The last case is also really weird:

```
((LAMBDA (NIL)       NIL ) 'FOO)  ==>  NIL
((LAMBDA (NIL) (CDR  NIL)) 'FOO)  ==>  FOO
((LAMBDA (NIL) (CDDR NIL)) 'FOO)  ==>  NIL
```

The first of the above expressions should not be unexpected at this point. The compiler inlines the value of NIL, so binding NIL to FOO has no visible effect.

The second expression also makes sense, because NIL will be "bound" to FOO (although it is a cons cell and not an atom), and (CDR NIL) = (*CDR NIL), so this one actually returns the value that is being "bound" to NIL in the function application.

The third one should induce a mild headache, though, because (CDR NIL) = FOO and hence (CDDR NIL) = (CDR **FOO**), which is the CDR of an atom and should be an error!

The reason for this unexpected result is tail call optimization. The application of CDDR is in a tail position, so before control is transferred to CDDR, NIL is being unbound from FOO.

Why does this not happen with (CDR NIL)? Because the compiler inlines CDR, but not CDDR.

While it might be fun to chase and explain the reasons for undefined behavior, this is something that is best avoided from the beginning. The argument part of LAMBDA should be an atom or a list of *unique* symbols in the sense of VARSYMP (page 87). Feel free to improve LISCMP!

READER ODDITIES

LISCMP does not do any *end of file* (*EOF*) detection, because it reads its input programs with READ, which just delivers NIL when no characters are available from its input source. In principle this is easily fixed, but it also poses an interesting question:

What exactly is the "end of file"?

Modern operating systems record exact file sizes in their file systems, so the end of the file is reached when attempting to read the $n+1^{st}$ character from an n-character file. Not so long ago, though, file sizes were recorded in "blocks", so the EOF was not so easy to determine. When processing files of characters, there was usually some kind of convention for indicating the EOF, like filling the last block of a file with a character that could not appear inside of the content of the file. The CP/M operating system, for instance, used the ASCII SUB character (code point 26), also known as control-Z [DR1976].

Then what would the end-of-file be on a strip of paper tape or a deck of punch cards? Paper tape, being a potentially infinite

sequential medium (you can always glue some more tape to the end of it!), could only use some kind of convention to indicate the EOF. Punch cards are not only a potentially infinite input source, they are also block-oriented. A completely unpunched card will read as 72 blanks (and a sequence number). Most cards have a number of trailing blanks, so where exactly is the end of input?

Early LISP systems (LISP 1 and LISP 1.5) used a *stop card* [McCarthy1960] to indicate the end of input (see figure 11). The stop card mechanism is similar to the approach used by LISCMP. It used the atom STOP outside of any lists to indicate the end of input. On the stop card the atom was followed by lots of closing parentheses plus another appearance of the word STOP, just in case there had been any open lists. Extra closing parentheses were just ignored by the system. Note that 10 closing parentheses at the end of the code were considered to be “a lot”! Programs were simpler back then!

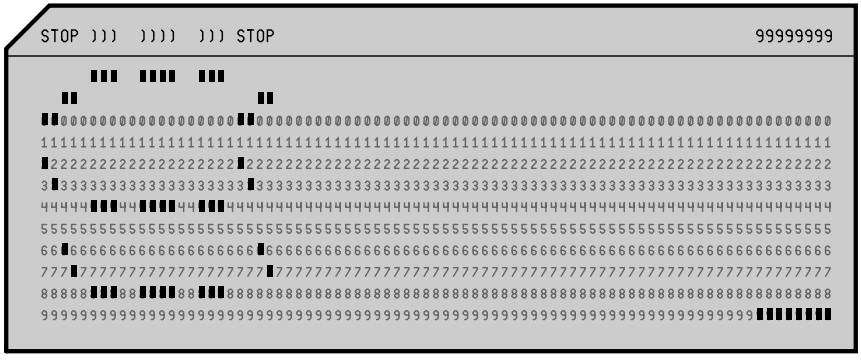


Fig. 11 – Stop card – note the blanks between the groups of parentheses! Cards with long sequences of identical characters were prone to getting shredded and stuck in the reader.

Of course, the LISCMP low-level runtime library could handle all kinds of end of file conventions under the hood. It does detect the EOF after all, or READC and PEEKC could not deliver NIL when running out of characters.

The obvious way to signal the EOF to a LISP program would be to deliver something that cannot normally be returned by a program: a special object that does not appear under any other circumstances. Such an object, SCHEME calls it the “EOF object” [Scheme1991], would be a special object, like T or NIL. It would allocate a special slot at the bottom of the cell pool (see page 72) and only be returned by READ (and maybe READC and PEEKC) in the case of an exhausted input source. The EOF object could be made visible by binding it to a name or by introducing a predicate to test it.

This modification would not be very complicated. LISCMP simply chooses the easiest and most portable way of EOF detection. Note that the method used by LISCMP would even work if READC and PEEKC would *not* indicate the EOF!

PSEUDOSTRINGS

Pseudostrings were briefly described as “interned, self-quoting symbols that can contain any kind of character” earlier in this text, which is true, but conveniently glosses over a very ugly kludge.

Pseudostrings can be compared with EQ, exploded and imploded, read and printed, just like ordinary symbols. E.g.:

```
(EQ "FOO BAR" "FOO BAR")    ==>  T
(EXPLODE "HELLO, WORLD!")   ==>  (H E L L O ,
                                     W O R L D !)
(IMPLODE '(X " / " Y)       ==>  X / Y
```

One thing that becomes obvious immediately is that many pseudostrings cannot be *read back* once printed. When printing the pseudostring "A B C" and then reading it back, the reader would just deliver A. When reading back "(X)", the result will be a list instead of a symbol. When printing "X/Y" and then reading it back, the compiler would collapse with an error message due to an unknown input character. So pseudostrings break the symmetry of READ and PRINT.

(BTW, other objects break this symmetry, too. For example, printing (LAMBDA (X) X) will output <FUNCTION>, which is also un-READ-able.)

However, this is still not the ugly kludge that was announced above. The really ugly detail is about the way in which pseudostrings are “self-quoting”.

Many objects are *self-quoting* in modern LISP, i.e. they do not require QUOTE in order to be interpreted as literal data objects. In purely symbolic LISP, this is basically the NIL object and maybe also T. In other LISP systems, this would also include numeric literals, character literals, maybe vectors and other compound data objects, etc.

The self-quotation is normally done during evaluation of an expression. The evaluator knows which objects are self-quoting and interprets them accordingly. However, FOO and "FOO" are the same type of object from the evaluator’s perspective, so how can it know which one to quote and which not?

The proper solution would be to add another type for pseudostrings and let the evaluator handle this type differently. LISCMP already has one type other than atoms and lists and that is the type of the function object. It uses a special *type tag* (FUNTAG) to identify function objects. Such a type tag could also be added for pseudostrings.

However, pseudostrings are used mostly for printing in LISCMP, so a much simpler—and really ad-hoc—solution is used instead: the *reader* wraps an application of QUOTE around pseudostrings. You can easily make them visible:

```
(QUOTE ("FOO" "BAR" "BAZ"))
  ==>  ((QUOTE FOO) (QUOTE BAR) (QUOTE BAZ))
```

Note that no evaluation except for the QUOTE in the input program is going on here. The QUOTE around each member of the list is added by READ. In case you do not believe it, try this:

```
$ sh lcg -
(READ)
^D
("FOO" "BAR" "BAZ")
```

Or, have a look at the source code! (Page 66)

Of course, there is another obvious solution to the problem and that is to make pseudostrings entirely normal (unquoted) symbols and let the user quote them if they need to. This is what LISP 1.6 and early MACLISP did [MIT1967]. Later MACLISP used real strings instead of pseudostrings [Moon1974].

DOTTED PAIRS

Finally, the READ function of LISCMP does not read *dotted pairs*, thereby causing another break in the symmetry of READ and PRINT:

```
$ sh lcg -
(CONS (QUOTE A) (QUOTE B))
^D
COMPILE
RUN
(A . B)

$ sh lcg -
(QUOTE (A . B))
^D
COMPILE
*** FUNNY CHARACTER
```

This is a deliberate design decision, because it makes sure that all LISP programs read by the reader at least consist of proper lists (as opposed to *dotted lists)*.

The advantage of this constraint is that decomposing forms with CAR, CDR, and friends becomes much less error-prone. For

example, testing the second argument of SETQ for being a proper variable symbol involves the test

```
(VARSYMP (CADR X))
```

where X is bound to the SETQ form being examined. Even if the SETQ form is syntactically completely wrong, like

```
(SETQ)
```

the test still works, because

(CAR NIL) = (CDR NIL) = NIL

Each end of a list really branches into an infinite tree of NILs (see figure 12), so every missing piece of a form, no matter how deeply nested, will be replaced by NIL.

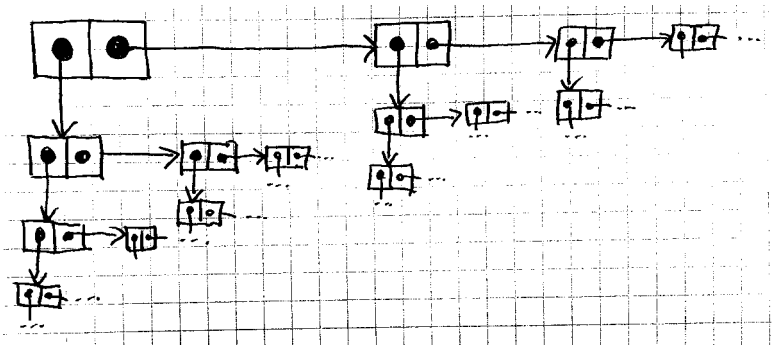


Fig. 12 – Infinite tree of NILs

Even a form like

```
(LABEL)
```

will not cause the compiler any trouble. It will even compile properly, because both the bindings part and the body of the form are NIL! (It should not compile, though!)

If the reader allowed dotted pairs, things would be much more complicated, because then the compiler would have to be prepared for input like

```
(SETQ . FOO)
```

Not that the compiler does do much consistency checking as it is, but removing dotted pairs from the input syntax of READ makes its life a whole lot easier!

WOULD IT RUN ON AN IBM 704?

LISCMP is not exactly an optimizing compiler. In particular, the code generated by it is not very compact. So how does its size compare to the original LISP system on the IBM 704 Electronic Data Processing Machine? Would it fit on such a machine?

The IBM 704 had a memory capacity of 4096 to 32,768 machine words with 36 bits per word and a 15-bit address space ($2^{15} = 32,768$). So a cons cell would fit in one machine word, because 36 bits provided enough space for two 15-bit pointers and six tag bits. The 704 at the M.I.T. had the maximum memory configuration, and the LISP 1 system allocated 12,000 words [McCarthy1960], so around 20,000 cells were available to LISP programs.

BTW, the memory on the 704 was *core memory*, where each bit was stored in a tiny, magnetized iron toroid called a “core”. The machine had $32,768 \times 36$ bits of memory [IBM1954], so there were 1,179,648 tiny little cores in its memory subsystem. Even if cores are very tiny, they take up *some* space and their weight accumulates.

Figure 13 shows a tiny part of the “modern” core plane of a late DEC PDP-11 computer. Core memory was not so densely packed in the IBM 704. The rack containing its core memory was huge: about 2×3 meters with a depth of almost one meter. One rack contained 16,384 words of storage and the necessary driver logic, so the machine at the M.I.T. had two such racks.

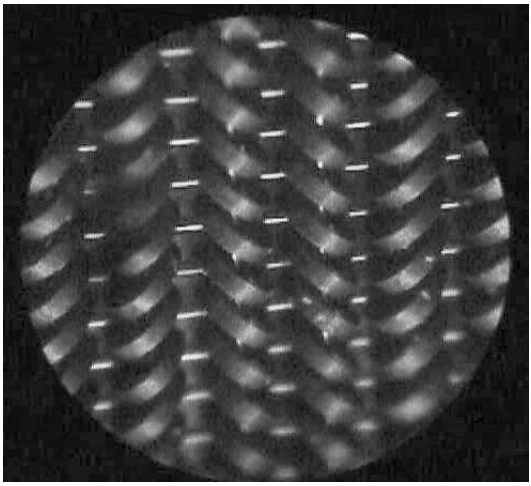


Fig. 13 – Core plane (50x magnification) – the picture shows a tiny section of a "modern" DEC G114B UNIBUS core plane from the late 1970's.

Core memory has a lot of interesting properties. It is *non-volatile*, so in principle a machine could be powered down and the content of its working memory would not be lost (but in practice it did not work all the time). Reading core memory was *destructive*, because it involved writing a zero to the desired storage location and then measuring which bits had flipped.

There were three wires running through each core, called X, Y, and SENSE. The X and Y lines were used to select a core in the core matrix and change its polarity. The SENSE line was used to sense the pulse that a core emitted when it flipped its polarity. It emitted a much weaker pulse when its polarity stayed the same. So in order to read the SENSE line, the core had to be brought to a known state, thereby overwriting it—usually with zero.

A typical read operation in a core memory system used a mechanism called "write after read". It meant to write zero to the cores of a machine word, extract its value and immediately write the value back to the location. Of course this meant that a read operation took twice as long as a write operation. Therefore, early

computers often had instructions that worked directly on memory locations, so the instruction could read its operand, leave the operand cores undefined while computing its result, and then write the result back to the location.

Modern systems do not have these restrictions and memory is cheap and physically small these days, so saving memory is no longer a big issue. On the other hand code density was much tighter on old machines, because instructions would often fit in a single machine word. Therefore sizes of compiled code are hard to compare. A naive approach would be to take 32,768 times 4.5, because 36 bits equal 4.5 bytes. This would give a maximum executable size of $32,768 \times 4.5 = 147,456$ bytes.

But then instructions on a 386 computer can be as short as one byte and as long as 15 bytes [Intel1986], while they were mostly 4.5 bytes on the 704, so this is really a tough exercise. Let's make the available capacity on the 704 computer $150,000$ bytes.

It does not matter much, though, because the code (text) size of the LISCMP LISP system alone (excluding the compiler) is already about $70,000$ bytes plus $65,535 \times 5 = 327,675$ bytes for cons cells. Adding the compiler gives a text size of about $175,000$ bytes, so no matter how tiny the cell pool would be, there is no way to fit LISCMP on something as "small" as an IBM 704. "Small" is a funny thing to say about a computer whose average installation weighs around 10 metric tons and requires a room the size of a large apartment with reinforced floors and industry-level air conditioning.

The text size of the LISCMP executable could probably be brought down significantly by making it emit assembly language instead of C. Experience with other compilers suggests a decrease of 50% to 75%. Let us meet in the middle and make that 67%, which would give around $60,000$ bytes. Still much too much, because that leaves too few cells in the freelist for compiling more than the

most trivial programs. LISCMP needs about $27,000$ cells ($\tilde{}240$K bytes) to compile itself!

So, no matter how hard we try, LISCMP requires a much bigger machine than the IBM 704. But then wide parts of the first LISP systems were written in assembly language, while LISCMP is written almost completely in LISP, down to some pretty low level, so the entire comparison probably does not make much sense to begin with.

But why does LISCMP consume so much memory? An educated guess would be that building the initial OBLIST and SYMLIS allocates most of the cells. Indeed, replacing the ADD-ATOM and ADD-OBJECT functions of the compiler with

```
(SETQ ADD-ATOM (LAMBDA (X) NIL-ADDR))
(SETQ ADD-OBJECT (LAMBDA (X) NIL-ADDR))
```

reduces the memory footprint from 27,000 to 4,200 cells—small enough to actually fit on an IBM 704, but then the compiler would no longer build a pool image. Maybe a solution could be advised that writes the image to mass storage, but then LISCMP does not have any functions for accessing permanent storage, so this is not something that can be added ad-hoc.

So it *is* impressive that a complete LISP system—much more complete than LISCMP, including things like arithmetic functions, etc—fit in 12,000 words of 36-bit memory back in the days or early LISP!

TEMPORARY VARIABLE ISSUES

Sometimes a *temporary variable* is used for swapping the values of two variables (e.g. NREVERSE, page 47) or to avoid computing the value of an expression twice. For an instance of the latter, see the INCR function on page 77. It stores the successor of (CAR X) in the temporary variable INCR-TMP, because it appears twice in the subsequent function application:

```
                    (SETQ INCR-TMP (SUCC (CAR X)))
                    (LOOP (CDR X)
                          (EQ INCR-TMP (QUOTE 0))
                          (CONS INCR-TMP Y))
```

INCR-TMP is a global (top-level) variable, and global temporary variables are a bad idea in any language, because they can change at any time in any context without any obvious hints to be discovered in the source code.

To further explore this matter, have a look at the following program, which makes use of the INCR function:

```
(SETQ COUNTER (QUOTE (0)))

(SETQ NEXT
  (LAMBDA (X)
    (SETQ TMP COUNTER)
    (SETQ COUNTER (INCR COUNTER))
    TMP))

(MAPCAR NEXT (QUOTE (X X X X X)))
```

The idea is to set up a counter function (NEXT) which takes a dummy variable and returns a number that increases each time NEXT is applied. MAPCAR then generates a list of the first five numbers of the resulting sequence. If the variable INCR-TMP of INCR was also called TMP, the program would not work, because NEXT saves the old value of COUNTER in TMP and then applies INCR, which changes TMP to a different temporary value.

The source code of the INCR function is not readily available at the point where NEXT is being defined, so the issue will most probably escape the attention of the programmer. It is impossible to keep track of all the temporary variables called “TMP”.

Because TMP is global, its binding is never replaced by any local variables, so it has no *outer value*. Hence its value is not reset to any previous (outer) value when INCR is left, so NEXT would use the value assigned to TMP in INCR.

What can be done about this? The approach used here is to always prefix the name TMP with the name of the function in which it appears (e.g. INCR-TMP), which would be unique in every program. Of course this approach requires some discipline on the part of the programmer.

So which alternatives exist? A global *stack* could be created for outer values of global variables. E.g.:

```
(SETQ *GLOBALS NIL)

(SETQ PUSH
  (LAMBDA (X)
    (SETQ *GLOBALS
          (CONS (CONS X (*CDR X))
                *GLOBALS))))

(SETQ POP
  (LAMBDA ()
    (*RPLACD (CAAR *GLOBALS) (CDAR *GLOBALS))
    (SETQ *GLOBALS (CDR *GLOBALS))))
```

The (outer) value of a global variable could then be saved by PUSHing the variable to the stack, and the outer value of the most recently saved variable could be restored by applying POP. (Remember that *RPLACD applied to an ATOM changes the *binding* of that atom.) The clause in INCR would then look like this:

```
(PUSH (QUOTE TMP))
(SETQ TMP (SUCC (CAR X)))
(LOOP (CDR X)
      (EQ TMP (QUOTE 0))
      (CONS TMP Y))
(POP)
```

While this solution solves the problem technically, it creates a new one, which is potentially bigger than the original one. Note that the application of LOOP in the above example is no longer tail-recursive, because POP is applied after LOOP returns! This may

not be a problem as long as the loop iterates only a few times, but is not very satisfying as a general solution.

Also, this approach does not free the programmer from the task of adding extra code *and* it adds the possibility of unbalanced appearances of PUSH and POP. When the programmer forgets to insert an application of POP, the wrong value will be restored by a subsequent POP application, which would lead to bugs that are very hard to locate.

What about local temporary variables? E.g.:

```
(LABEL ((TMP (SUCC (CAR X))))
  (LOOP (CDR X)
        (EQ TMP (QUOTE 0))
        (CONS TMP Y)))
```

While this solution would free the programmer from keeping track of global variables, it does not solve the problem of tail recursion, because the last expression in the body of LABEL is not in a tail position (see page 99). As shown on the same page, an in-situ application of LAMBDA would not make the application of LOOP tail recursive, either.

The best option that allows temporary values *and* tail recursion seems to be a global temporary variable and a naming convention that precludes collisions. Not beautiful, but practicable.

The reason why this is a limitation of dynamic scoping will be explained later, in the chapter about the science of LISP.

SEPARATION OF OBLIST AND SYMLIS

The list of symbols (SYMLIS) and the list of objects (OBLIST) are two different structures in the LISP system being discussed here. There are some good and some not-so-good reasons for doing this.

The good reason is that deduplication is expensive in purely symbolic LISP. In a more complete programming language, you would use *hashing* in order to avoid duplication, but there are no numbers in LISCMP, so hashes are hard to compute.

Note that this does not imply that other languages can compute things that LISCMP cannot compute! LISCMP is Turing-complete and can therefore compute anything you like. However, it cannot always do this efficiently. To compute hashes, you would first have to implement arithmetic functions based upon lists and then use those to calculate hash values. This approach might very well be *slower* than using some less advanced method, like bucket lists. Bucket lists, though, are not *very* efficient, only slightly more so than flat lists. Interning symbols is the greatest bottleneck of LISCMP by far!

So the simplest solution is not to deduplicate the OBLIST at all. Each literal object is just added to the list, which means that the list will contain many, many instances of objects like (X), because there are many functions with a single argument called X.

You might think that deduplicating OBLIST would save so much memory that garbage collection pressure would drop to such a degree that it would compensate for the additional cost of hashing. Not a bad idea! However, LISCMP is designed to be simple, and adding this method would increase its complexity quite a bit. This is probably how C compilers mutated from 64KB programs into multi-megabyte monstrosities over the years!

The other reason not to deduplicate OBLIST is rather obscure. Have a look at the following program:

```
(SETQ ADD
  (LAMBDA (X)
    (NCONC (QUOTE (X)) (LIST X))))
```

This is a self-modifying program, just like the one shown on page 47. Each time it is called, it concatenates its argument

destructively to the list literal (X), thereby modifying the object in OBLIST and thereby modifying the program that refers to the object. In modern dialects of LISP, the modification of list literals is either undefined or an error, but in LISCMP the behavior of the above is as expected—which is not to say that it is a good idea!

Now imagine OBLIST deduplication to be performed while compiling the above program. It would compile fine and even work as expected the first time the function was applied. Evaluating

```
(ADD (QUOTE Y))
```

would modify the object in ADD to (X Y) and return it. The next time ADD would be applied in the same way, though, it would fail, because the wrong number of arguments is being supplied to it. This is because the list of variables of ADD is (X) and the literal list inside of it is also (X), so deduplicating OBLIST unifies them and hence running

```
(ADD (QUOTE Y))
```

turns the programs into

```
(SETQ ADD
  (LAMBDA (X Y)
    (NCONC (QUOTE (X Y)) (LIST X))))
```

So maybe it is a good idea to discourage self-modifying code after all!

Not deduplicating OBLIST can be justified, but the whole purpose of SYMLIS is that every atom contained in it must be unique. Deduplication is inherent in its purpose. Therefore LISCMP maintains two lists, one for (non-atomic) objects and one for atoms.

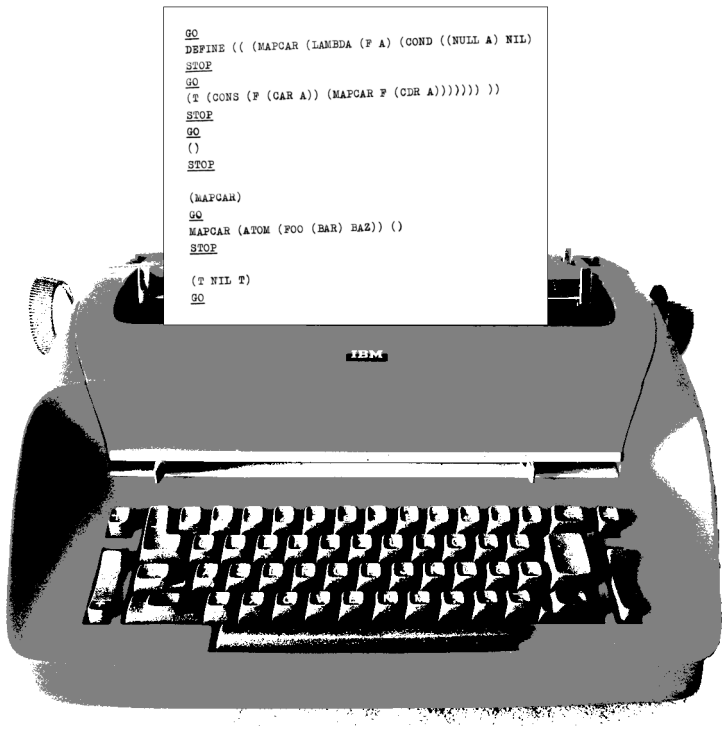
BTW, OBLIST is built completely while compiling a program. Once the program has been compiled, the list is fixed and will no longer change at run time. SYMLIS, on the other hand, will grow at run

time whenever calling INTERN or any function that uses INTERN internally, like READ or IMPLODE.

When using a tree-walking interpreter to evaluate LISP, like the one discussed in the first chapter, no OBLIST is required, because a function like

```
(SETQ FOO (LAMBDA () (QUOTE (HELLO WORLD))))
```

would be represented by exactly the above S-expression. The literal object (HELLO WORLD) is contained in the expression, so it is protected from garbage collection as long as the enclosing function is referenced somewhere. There is no need to keep an extra reference to the literal in a separate safe location.



READ, EVAL, PRINT, LOOP

Okay, the image shows an IBM Selectric and not a real Friden Flexowriter. You have to admit that the 1970's spaceship console design looks cool, though!

IN 1959, the Friden Calculating Machine Company produced a very interesting machine called the *Flexowriter*. The Flexowriter was basically a very huge, very sturdy, and very loud heavy-duty typewriter. What distinguished it from other typewriters was that it allowed you to punch texts to paper tape and then automatically print letters from paper tape at 11 characters per second.

So the Flexowriter was more than a typewriter, it was a small-scale printer. Once you had your text on paper tape, you could print as many copies as you liked to.

Paper tape (or punched tape) can be imagined as a very long and narrow strip of paper with holes punched into it, so paper tape is like a very long punched card, but with fewer channels. Typically it had five or six channels, depending on the character set in use, and it was stored either on reels or as fan-folded stacks of paper.

What was even cooler about the Flexowriter was that there was a specific model, the Flexowriter FTM [Friden1959], that had a communication interface. So you could attach two Flexowriters to each other and whatever you typed on one end would print both locally and on the other end. You could even read paper tape on one machine, send the characters stored on it over the communication interface and have a letter automatically printed on the remote machine. So the Flexowriter was not only a typewriter and a printer, it was also a *teletypewriter*.

It was only a matter of time before someone had the idea to place a computer on the remote end of the Flexowriter, so characters you typed were sent to the computer for processing and output sent by the computer would print on the Flexowriter. This idea started a whole new way of working with a computer. Instead of coding on a sheet of paper, submitting your code for punching, then submitting a deck of punch cards for execution, and finally picking up your results from your inbox, you could just type in a program and get a result immediately.

Even at 11 characters per second, printed by a very bulky and very noisy machine, the great novelty of this mode of operation was *interactivity*. For the first time in the history of computing, the programmer interacted with the computer rather than exchanging letters with it.

This mode of operation would remain an exception for a while, though, because *time sharing*, although it already had been invented, was not in widespread use. In a time sharing system, multiple users would "share" the resources of the machine. For instance, while one user was typing, the computer could perform some calculation for another user at the same time. This concept was quite new in 1960, though, so usually only one user could interact with the computer at a time. While an interactive session was in progress, no *batch jobs* could be executed, so Flexowriter interaction collided with the normal flow of operation.

The "LISP-Flexo" system was rather limited at the beginning, for example it did not include the LISP compiler [McCarthy1960]. Most programs were still submitted on decks of punch cards.

However, the Flexowriter changed how people thought about interaction with the computer, and *hard-copy terminals*, like the Flexowriter, soon started to replace batch job processing. Then *glass TTYs*—teletype devices that "printed" on a screen rather than paper—started to replace hard-copy terminals, and then "smart" *terminals*—terminals that could move the cursor about the screen—started to replace glass TTYs.

In the beginning of the interactive age, though, "terminals" were basically typewriters with a communication cable attached. Whatever you typed on the keyboard was submitted to the computer, character by character, and whatever the computer responded was printed on the same typewriter.

Methods of correcting errors in your input were limited, because the typewriter could not erase a character once it had typed it. You could not "move the cursor", because there was no cursor. There

was just a blank spot at the position where the next character would be typed.

The Flexowriter *did* have a backspace key, though, which allowed you to erase the most recently typed characters. “Erasing” was typically visualized by re-printing the removed character, so typing (LAMDA, then erasing the DA, inserting the B and typing DA again looked like this:

(LAMDAADBDA

No actual underlining was done, though, it only serves to illustrate the process. The underlined characters were printed in order to signal that the corresponding character was removed from the input.

```
GO
DEFINE (( (MAPCAR (LAMBDA (F A) (COND ((NULL A) NIL)
STOP
GO
(T (CONS (F (CAR A)) (MAPCAR F (CDR A)))))) ))
STOP
GO
()
STOP

(MAPCAR)
GO
MAPCAR (ATOM (FOO (BAR) BAZ)) ()
STOP

(T NIL T)
GO
```

Fig. 14 – A dialog on the LISP-Flexo system – system output is underlined; the Flexowriter did not actually do this, though, it only serves to clarify the dialog

Figure 14 shows a sample dialog with the LISP-Flexo system. The interpreter would print “GO” to prompt the programmer to type a program of the form

operator **(***operand* $\cdots$**)** *environment*

where the operator could be any LISP operator: a built-in function like CONS, a lambda function, or a special form operator like DEFINE. The *operands* would be to supplied as a list, which would be interpreted as a list of *quoted* objects. The *environment*, finally, would be an association list supplying additional functions or other values to be used in the computation.

So, for instance, the expression

```
(CONS (QUOTE FOO) (QUOTE BAR))
```

would be entered as

```
CONS (FOO BAR) ()
```

This is why some textbooks (like [Maurer1972]) preserved this style of printing LISP programs for a long time, even when glass TTYs had taken over the world of computing and the notation that still is in use today already had become much more common.

The LISP-Flexo system would keep prompting until all three parts of a program had been entered. As soon as you typed “carriage return” (CR), it would print “STOP” to indicate that it was no longer ready to receive input. It would then process the input, print a result, if any, and issue another “GO” prompt.

Note that LISP 1.5 omitted the “environment” part and assumed NIL, so it prompted only for an operator and its operands.

Since the READ function read input directly through the Flexowriter, there was no need to press CR after entering a list. The READ function would recognize the outermost parenthesis and the system would print “STOP” immediately. The same happened when you typed a blank after an atom. The READ function would return immediately and the system would start to process its input.

This method still worked in PDP-6 LISP [MIT1967] and MACLISP,

but there was no longer a GO/STOP prompt. An expression was simply typed at a teletypewriter or (glass) TTY and as soon as a blank after an atom or the final parenthesis of a list was received, the LISP system started to evaluate immediately and then printed the result.

Because the position of the first output character would still be in the line of the input, the system would print a newline character first and then the result. This is probably why the PRINT function of most LISP system still does a TERPRI *before* printing its argument. (LISCMP does not follow this tradition, though.)

BTW, the DEFINE function of early LISP had only one single argument and that was a list of atom/value pairs. To define the function AMONG from chapter 1, for example, you would typically write something like

```
DEFINE (( (AMONG (LAMBDA (X Y) (COND ((NULL Y)
NIL) ((EQUAL X (CAR Y)) T) (T (AMONG X (CDR
Y)))))) ))
```

The double parentheses at the beginning and end of the argument of DEFINE, although common style, are a bit misleading, though. The form

```
DEFINE (( (A B) (C D) ... ))
```

actually passes one single list of definitions to DEFINE:

```
DEFINE ( ((A B) (C D) ...) )
```

INTERPRETING LISP

In this chapter a simple interpreter for the LISP language accepted by LISCMP will be developed. In fact, the interpreter will only accept a subset of that language, though, because some semantics differ. For instance, the *CDR of an atom is not its value in interpreted LISP, because a different binding strategy is used. So the *CAR, *CDR, *RPLACA, and *RPLACD functions

are missing. Other functions that are primarily intended for internal use in LISCMP, like *ATOMP, *SETATOM, *READC, and *WRITEC, or even READC and WRITEC, will be missing as well.

LISINT, the LISP interpreter, will be compatible to LISCMP only at a rather abstract level. Following the tradition of *metacircular evaluation*, it will pass evaluation of *built-in* or *primitive* functions down to the level of LISCMP. E.g., to evaluate an application of NCONC, LISINT will evaluate the arguments of the function and then use LISCMP's compiled NCONC function to perform the actual operation. This is exactly what McCarthy's evaluator and the XEVAL evaluator of chapter 1 do.

Like XEVAL, LISINT will use *deep binding*, which is implemented by association lists. Note that LISCMP uses *shallow binding*, where symbols link directly to values, which results in much faster look-up, but also in a more complex strategy for saving outer values.

Unlike the simple evaluator from the first chapter, LISINT will implement *tail call optimization*, *mutation*, and sequential evaluation of LABEL, so that it becomes more compatible to LISCMP. The core of the interpreter is still very similar to the initial implementation, though.

The implementation is also very inefficient. Like LISCMP it merely serves as a proof of concept: a workable LISP interpreter can be written in a compiled minimal dialect of LISP.

THE INTERPRETER SOURCE CODE STARTS HERE

The first thing that differs between the XEVAL interpreter from chapter 1 and LISINT is that LISINT maintains two *environments*. This is necessary in order to support *mutation* of *global variables.* XEVAL has only three methods for *binding* variables to values:

- by including them in the initial environment
- by function application
- by LABEL

The first case is static and in the other two cases the scope of the bindings is limited to the dynamic extent of the corresponding construct, namely the bodies of LAMBDA and LABEL. The modified environment is passed as a parameter to a recursive application of XEVAL and as soon as XEVAL returns, the modified bindings no longer exist. So how can

```
(SETQ FOO (QUOTE BAR))
```

work at the global level, where its effect is supposed to persist after its application?

To make this possible in LISINT, there is a *global environment* and a *local environment*. The global environment holds bindings of symbols that are not function variables and the local environment holds bindings of function variables. When SETQ does not find its first argument in the local environment, it will try the global one.

Of course this means that variables will have to be looked up in both environments as well, in the local one first and in the global one as a fall-back, because local bindings can take precedence over global ones, but not vice versa.

The BINDING function locates the first binding of the atom X in the environment E. It returns the binding upon success and NIL, if the atom is not contained as a key in the association list E.

The LOOKUP function first looks up X in the environment E, which is passed to it as an argument. If a binding is found in E, it then returns the value to which X is bound in E. If X is not in E, it searches the global environment ENV and returns the value associated with X in ENV, or NIL if X is not in ENV.

The variable ENV will be bound to an initial global environment later in the code.

```
(SETQ BINDING
  (LAMBDA (X E)
    (COND ((EQ NIL E) NIL)
          ((EQ X (CAAR E))
            (CAR E))
          (T (BINDING X (CDR E))))))

(SETQ LOOKUP
  (LAMBDA (X E)
    (LABEL ((B (BINDING X E)))
      (COND (B (CADR B))
            (T (LABEL ((B (BINDING X ENV)))
                 (COND (B (CADR B))))))))))
```

The EVPROG function evaluates the bodies of LABEL, LAMBDA, and PROGN as well as the consequent expressions of COND. It evaluates each expression except for the last one in a body in a non-tail context and only the last one as a tail application.

To implement tail application EVPROG uses two versions of the local environment: the regular local environment E and the *outer environment* OE. The outer environment contains the outer bindings of the values in the regular environment. Because LISINT uses deep binding, a new environment is created by attaching new bindings to an existing environment. For instance, binding FOO to INNER in the existing environment

```
((FOO OUTER))
```

will give

```
((FOO INNER) (FOO OUTER))
```

Because variables are looked up left-to-right, the binding (FOO OUTER) will become inaccessible. It is being *shadowed* by the inner binding.

In this case the outer environment would be the original one,

```
((FOO OUTER))
```

By replacing the regular environment by the outer one, the inner binding is removed. This is why EVPROG passes OE on to EVAL3 in a tail application. When EVAL3 performs a tail call, it will use OE in the place of E.

Note that EVPROG passes E in the place of OE to EVAL3 when evaluating an expression in a non-tail position. In this case performing tail call optimization will have no effect, because E and OE are the same environment.

This mechanism is used by many functions that are called by EVAL3.

```
(SETQ EVPROG
  (LAMBDA (P E OE)
    (COND ((EQ P NIL))
          ((EQ NIL (CDR P))
            (EVAL3 (CAR P) E OE))
          (T (EVAL3 (CAR P) E E)
             (EVPROG (CDR P) E OE)))))
```

EVCON evaluates the COND special form. It uses the same mechanism as EVPROG to perform tail and non-tail application.

Note the use of a temporary variable! The predicate of each clause of COND is evaluated and its value is bound to the symbol EVCON-TMP. When there are no consequents in the clause, the value of EVCON-TMP is just returned, which implements *predicate-only clauses* properly.

If the value of the predicate was not bound to a variable, it would have to be evaluated twice:

```
((EVAL3 (CAAR C) E E)
  (COND ((EQ NIL (CDAR C))
         (EVAL3 (CAAR C) E E))
        (T (EVPROG (CDAR C) E OE))))
```

In this case, any effect of the predicate of the clause would be observed twice, which would be unexpected. For instance,

```
(COND ((PRINT (QUOTE FOO))))
```

would print FOO two times. Alternatives to global temporary variables have been discussed on page 136, with the conclusion that none of them are viable.

```
(SETQ EVCON
  (LAMBDA (C E OE)
    (SETQ EVCON-TMP (EVAL3 (CAAR C) E E))
    (COND ((EQ NIL C) NIL)
          (EVCON-TMP
            (COND ((EQ NIL (CDAR C)) EVCON-TMP)
                  (T (EVPROG (CDAR C) E OE))))
          (T (EVCON (CDR C) E OE)))))
```

The BIND function binds the variables in the list V to the arguments in the list A and adds the new bindings to the outer environment OE. It returns the new environment. Note that BIND *evaluates* arguments in the inner environment E, but *adds* bindings to the outer environment OE. This reflects the timing in tail applications compiled by LISCMP: first arguments are evaluated and *then* the outer environment is restored. In non-tail applications, OE equals E, so BIND still works as expected.

Note that each argument A is evaluated in the same environment E, so the bindings are established *in parallel*. First all A's are evaluated and then each of them is bound (in an unspecific order) to the corresponding V. This means that

```
((LAMBDA (X Y)
   ((LAMBDA (X Y)
      (LIST X Y))
    (CONS X Y)
    (CONS Y X)))
 (QUOTE 1)
 (QUOTE 2))
```

will return ((1.2) (2.1)), because at the time when (CONS Y X) is evaluated, the binding of X to (CONS X Y) has not yet been established *and* at the time when (CONS X Y) is evaluated, the binding of Y to (CONS Y X) has not yet been established. There is no observable order. We might as well assume that the bindings are being established in parallel.

There is one special case in BIND where V is a symbol. In this case BIND computes all arguments in A, giving a new list of evaluated arguments, and binds that list to V. This special case implements LEXPRs. Incidentally the code also implements SCHEME-style “improper” formal argument lists [Scheme1991], like (A B . C), which are not specified by LISCMP!

This does not matter though, because improper (dotted) lists cannot be read by READ, so this part of the implementation never becomes visible to the programmer.

```
(SETQ BIND
  (LAMBDA (V A E OE)
    (COND ((EQ V NIL) OE)
          ((ATOM V)
            (CONS (LIST V (MAPCAR
                            (LAMBDA (X)
                              (EVAL3 X E E))
                            A))
                  OE))
```

```
          (T (CONS (LIST (CAR V)
                         (EVAL3 (CAR A) E E))
                   (BIND (CDR V) (CDR A)
                         E OE))))))
```

BINDSEQ is like BIND, but establishes bindings *sequentially*. Given the list

B = **((**v_1 a_1**)** $\cdots$ **(**v_N a_N**))**

it evaluates each a_{i+1} in an environment where the value of a_i is already bound to v_i. Hence it implements the binding part of LABEL. Consequently a version of the example program given in the description of BIND that uses LABEL instead of LAMBDA would evaluate as follows:

```
(LABEL ((X (QUOTE 1))
        (Y (QUOTE 2)))
  (LABEL ((X (CONS X Y))
          (Y (CONS Y X)))
    (LIST X Y)))          ==>  ((1 . 2) (2 1 . 2))
```

because X is already bound to the value of (CONS X Y) when (CONS Y X) is being evaluated.

No outer environment is passed to BINDSEQ, because sequential bindings are never established in a tail application context.

```
(SETQ BINDSEQ
  (LAMBDA (B E)
    (COND ((EQ B NIL) E)
          (T (BINDSEQ
               (CDR B)
               (CONS (LIST (CAAR B)
                           (EVAL3 (CADAR B) E E))
                     E))))))
```

The MODIFY function implements the *mutation* of bindings. It changes the currently visible binding of V to refer to the value X. First it looks up V in the local environment E. If a binding for V is found in that environment, it replaces its value with X. Otherwise it tries the same procedure with the global environment ENV.

When the symbol is not found in either E or ENV, then it is added to ENV. This part is what enables SETQ to establish *top-level* (*global*) *bindings*.

```
(SETQ MODIFY
  (LAMBDA (V X E)
    (LABEL
      ((B (BINDING V E)))
      (COND
        (B (RPLACA (CDR B) X))
        (T (LABEL
             ((B (BINDING V ENV)))
             (COND (B (RPLACA (CDR B) X))
                   (T (SETQ ENV
                        (NCONC (LIST (LIST V X))
                               ENV)))))))))
    V))
```

The SAVE function saves the current configuration of the cell pool to the file named “LISINT”. It uses the internal LISCMP function *DUMP, which has been kept a secret until now, mostly because it is not really needed. Dumping and loading images is a useful feature, though! We will get back to this soon!

An image can be saved at any time by typing (SAVE) at the LISINT prompt.

```
(SETQ SAVE (LAMBDA () (*DUMP (QUOTE LISINT))))
```

The EVAL3 function *evaluates* the expression X in the union of the local environment E and the global environment ENV. It returns the normal form of X or *UNDEFINED if X has no normal form (but still terminates). It basically works in the same way as XEVAL in the first chapter, but there are some differences.

(1) EVAL3 interprets a lot more forms as special forms and passes them to the corresponding LISCMP operators for evaluation, like REVERSE, APPEND, ASSOC, IMPLODE, READ, PRINT, etc.

(2) All special forms work as specified by LISCMP. In particular LABEL binds its arguments sequentially.

(3) EVAL3 has mutation and a global environment, so expressions may have persistent effects. (Of course, input and output are also effects.)

(4) EVAL3 performs tail call optimization, so expressions like Ω evaluate in constant space.

(5) LAMBDA special forms evaluate to *LAMBDA special forms. This is necessary in order to implement (4), as will be shown below.

Earlier in this text (page 103) it has been explained in detail why in-situ lambda function applications may not be tail applications. This limitation does not apply to functions that are bound to a variable, though. I.e.

```
(FOO (QUOTE BAR))
```

may be a tail application (depending on its context), but

```
((LAMBDA (X) ...) (QUOTE BAR))
```

may never be a tail application.

EVAL3 (and XEVAL), however, evaluate the atom in the operator position of a function application, replace it with its value, and resubmit the form, thereby turning (F X) into ((LAMBDA ...) X). So how can EVAL3 distinguish the two cases?

Whenever an expression of the form

```
(LAMBDA ...)
```

is found, the evaluator replaces the symbol LAMBDA with *LAMBDA. Note that this does not happen in expressions of the form

```
((LAMBDA ...) ...)
```

because the evaluator does not evaluate the head of the form in this case. In expressions like

```
(SETQ FOO (LAMBDA ...))
```

or

```
(LABEL ((FOO (LAMBDA ...)))
  ...)
```

though, the lambda form will be evaluated and hence the operator *will* be replaced. Therefore, atoms always bind to functions of the form (*LAMBDA ...) while functions that are applied in-situ will remain in their original form without the operator symbol replaced.

The interpreter only evaluates applications with *LAMBDA operators in their functions as *tail applications*, while all other function applications will be regular applications.

Tail application is the only place in the interpreter code where OE is used in the place of E. It adds new bindings to OE, thereby creating a new E and passes the old OE along as the new OE.

In regular (non-tail) applications, E is extended with new bindings and then used as the new E *and* OE.

```
(SETQ EVAL3
  (LAMBDA (X E OE)
    (COND
      ((EQ X T) T)
      ((ATOM X)
```

```
   (LOOKUP X E))
((ATOM (CAR X))
  (COND ((EQ (CAR X) (QUOTE QUOTE))
         (CADR X))
        ((EQ (CAR X) (QUOTE ATOM))
         (ATOM (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE EQ))
         (EQ (EVAL3 (CADR X) E E)
             (EVAL3 (CADR (CDR X)) E E)))
        ((EQ (CAR X) (QUOTE CAR))
         (CAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CDR))
         (CDR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CAAR))
         (CAAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CADR))
         (CADR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CDAR))
         (CDAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CAAAR))
         (CAAAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CAADR))
         (CAADR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CADAR))
         (CADAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CADDR))
         (CADDR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CDAAR))
         (CDAAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CDADR))
         (CDADR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CDDAR))
         (CDDAR (EVAL3 (CADR X) E E)))
        ((EQ (CAR X) (QUOTE CDDDR))
         (CDDDR (EVAL3 (CADR X) E E)))
```

```
((EQ (CAR X) (QUOTE CONS))
  (CONS
    (EVAL3 (CADR X) E E)
    (EVAL3 (CADR (CDR X)) E E)))
((EQ (CAR X) (QUOTE NOT))
  (EQ (EVAL3 (CADR X) E E) NIL))
((EQ (CAR X) (QUOTE NULL))
  (EQ (EVAL3 (CADR X) E E) NIL))
((EQ (CAR X) (QUOTE RPLACA))
  (RPLACA (EVAL3 (CADR X) E E)
          (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE RPLACD))
  (RPLACD (EVAL3 (CADR X) E E)
          (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE REVERSE))
  (REVERSE (EVAL3 (CADR X) E E)))
((EQ (CAR X) (QUOTE NREVERSE))
  (NREVERSE (EVAL3 (CADR X) E E)))
((EQ (CAR X) (QUOTE APPEND))
  (APPEND (EVAL3 (CADR X) E E)
          (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE NCONC))
  (NCONC (EVAL3 (CADR X) E E)
         (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE EQUAL))
  (EQUAL (EVAL3 (CADR X) E E)
         (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE MEMBER))
  (MEMBER (EVAL3 (CADR X) E E)
          (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE ASSOC))
  (ASSOC (EVAL3 (CADR X) E E)
         (EVAL3 (CADDR X) E E)))
((EQ (CAR X) (QUOTE IMPLODE))
  (IMPLODE (EVAL3 (CADR X) E E)))
```

```
            ((EQ (CAR X) (QUOTE EXPLODE))
              (EXPLODE (EVAL3 (CADR X) E E)))
            ((EQ (CAR X) (QUOTE COND))
              (EVCON (CDR X) E E))
            ((EQ (CAR X) (QUOTE PROGN))
              (EVPROG (CDR X) E E))
            ((EQ (CAR X) (QUOTE LABEL))
              ((LAMBDA (NE)
                 (EVPROG (CDDR X) NE NE))
               (BINDSEQ (CADR X) E)))
            ((EQ (CAR X) (QUOTE PRIN1))
              (PRIN1 (EVAL3 (CADR X) E E)))
            ((EQ (CAR X) (QUOTE PRINT))
              (PRINT (EVAL3 (CADR X) E E)))
            ((EQ (CAR X) (QUOTE TERPRI))
              (TERPRI))
            ((EQ (CAR X) (QUOTE READ))
              (READ))
            ((EQ (CAR X) (QUOTE SETQ))
              (MODIFY (CADR X)
                      (EVAL3 (CADDR X) E E) E))
            ((EQ (CAR X) (QUOTE SAVE))
              (SAVE))
            ((EQ (CAR X) (QUOTE LAMBDA))
              (CONS (QUOTE *LAMBDA) (CDR X)))
            ((EQ NIL (CAR X))
              (QUOTE *UNDEFINED))
            (T (EVAL3 (CONS (EVAL3 (CAR X) E E)
                            (CDR X))
                      E OE))))
      ((EQ (CAAR X) (QUOTE *LAMBDA))
        (EVPROG (CDDAR X)
                (BIND (CADAR X) (CDR X) E OE)
                OE))
      ((EQ (CAAR X) (QUOTE LAMBDA))
```

```
((LAMBDA (NE)
   (EVPROG (CDDAR X) NE NE))
 (BIND (CADAR X) (CDR X) E E))))))
```

EVAL is just EVAL3 with an empty local environment.

```
(SETQ EVAL (LAMBDA (X) (EVAL3 X NIL NIL)))
```

The symbol ENV is bound to the initial global environment. This environment can be modified by SETQ at the top level when the interpreter is running. Most definitions in the initial environment are just redefinitions of special forms. This was explained in the compiler code on page 44. Basically these are versions of built-in special forms that can be used as values.

Note that *LAMBDA is used in the place of LAMBDA in these definitions, because this is how SETQ would store functions in the environment. You would never use *LAMBDA in a regular program outside of ENV, though, because it would enable tail call optimization in places where it would alter the semantics of the language and hence introduce undefined behavior.

Then there are definitions like the one of MAPCAR, which duplicate the implementation of a LISCMP function instead of just delegating evaluation to LISCMP.

Delegation to the lower level of evaluation does not work with *higher-order functions*, though, because the function objects of the two levels are incompatible. LISINT creates functions of the form (*LAMBDA ...), but LISCMP expects function atoms composed of a FUNTAG pointer and a machine code label (see page 93). Hence an environment entry of the form

```
(MAPCAR (*LAMBDA (F A) (MAPCAR F A)))
```

would eventually result in a non-function object being passed to LISCMP's MAPCAR. Higher-order functions have to be interpreted at the level of LISINT itself.

```
(SETQ ENV (QUOTE
  ((ATOM (*LAMBDA (X) (ATOM X)))
   (EQ (*LAMBDA (X Y) (EQ X Y)))
   (CAR (*LAMBDA (X) (CAR X)))
   (CDR (*LAMBDA (X) (CDR X)))
   (CAAR (*LAMBDA (X) (CAAR X)))
   (CADR (*LAMBDA (X) (CADR X)))
   (CDAR (*LAMBDA (X) (CDAR X)))
   (CAAAR (*LAMBDA (X) (CAAAR X)))
   (CAADR (*LAMBDA (X) (CAADR X)))
   (CADAR (*LAMBDA (X) (CADAR X)))
   (CADDR (*LAMBDA (X) (CADDR X)))
   (CDAAR (*LAMBDA (X) (CDAAR X)))
   (CDADR (*LAMBDA (X) (CDADR X)))
   (CDDAR (*LAMBDA (X) (CDDAR X)))
   (CDDDR (*LAMBDA (X) (CDDDR X)))
   (CONS (*LAMBDA (X Y) (CONS X Y)))
   (LIST (*LAMBDA X X))
   (NOT (*LAMBDA (X) (NOT X)))
   (NULL (*LAMBDA (X) (NULL X)))
   (RPLACA (*LAMBDA (X Y) (RPLACA X Y)))
   (RPLACD (*LAMBDA (X Y) (RPLACD X Y)))
   (REVERSE (*LAMBDA (X) (REVERSE X)))
   (NREVERSE (*LAMBDA (X) (NREVERSE X)))
   (APPEND (*LAMBDA (X Y) (APPEND X Y)))
   (NCONC (*LAMBDA (X Y) (NCONC X Y)))
   (EQUAL (*LAMBDA (X Y) (EQUAL X Y)))
   (MEMBER (*LAMBDA (X Y) (MEMBER X Y)))
   (ASSOC (*LAMBDA (X Y) (ASSOC X Y)))
   (IMPLODE (*LAMBDA (X) (IMPLODE X)))
   (EXPLODE (*LAMBDA (X) (EXPLODE X)))
   (PRIN1 (*LAMBDA (X) (PRIN1 X)))
   (PRINT (*LAMBDA (X) (PRINT X)))
   (TERPRI (*LAMBDA () (TERPRI)))
```

```
(READ (*LAMBDA () (READ)))
(MAPCAR
  (*LAMBDA (*F *A)
    (LABEL
      ((MAP (LAMBDA (A R)
         (COND ((EQ A NIL) (NREVERSE R))
               (T (MAP (CDR A)
                       (CONS (*F (CAR A))
                             R)))))))
     (MAP *A NIL))))
(MAPCAR2
  (*LAMBDA (*F *A *B)
    (LABEL
      ((MAP (LAMBDA (A B R)
         (COND ((EQ A NIL) (NREVERSE R))
               ((EQ B NIL) (NREVERSE R))
               (T (MAP (CDR A)
                       (CDR B)
                       (CONS (*F (CAR A)
                                 (CAR B))
                             R)))))))
     (MAP *A *B NIL))))
(REDUCE
  (*LAMBDA (*F *B *A)
    (LABEL
      ((RED (LAMBDA (A R)
         (COND ((EQ A NIL) R)
               (T (RED (CDR A)
                       (*F R (CAR A))))))))
      (RED *A *B))))
(RREDUCE
  (*LAMBDA (*F *B *A)
    (LABEL
       ((RED (LAMBDA (A R)
          (COND ((EQ A NIL) R)
```

```
                    (T (RED (CDR A)
                            (*F (CAR A) R)))))))
          (RED (REVERSE *A) *B)))))))
```

Herald the interpreter.

```
(TERPRI)
(PRINT "LISP INTERPRETER")
```

The following LABEL implements the REPL, the *read eval print loop*. It could be more minimal, if it did not write the GO and STOP messages. Here is a minimal REPL:

```
(LABEL ((LOOP (LAMBDA (X)
           (LOOP (PRINT (EVAL (READ)))))))
  (LOOP NIL))
```

This version is particularly funny, because it applies the functions READ, EVAL, PRINT, LOOP in exactly this order. Some people suggest that this is how the REPL got its name in the first place. This may or may not be the case, but there were interpreter loops before there was tail call optimization (which is needed for the above) and they were usually described like this [MIT1967]:

```
(PROG NIL
  A  (TERPRI)
     (PRINT (EVAL (READ)))
     (GO A))
```

Note that the REPL program below loads an *image file* before entering the interactive loop and saves a new image before exiting. An image file is basically a copy of the entire cell pool, including the SYMLIS, OBLIST, and all definitions that have been typed at the interpreter prompt.

So when saving an image, everything that has been added to the memory of the interpreter, all variables, constants, and functions

will be saved to the file. When the interpreter is started for the next time, the image will be loaded and all definitions will immediately be available again.

The internal *LOAD function of LISCMP loads an image file.

Note that each program must have its own image file. For instance, two different versions of LISINT cannot share image files. Therefore, when you modify LISINT, you should change the image file name!

```
(LABEL
  ((LOOP (LAMBDA ()
     (PRINT (QUOTE GO))
     (SETQ *EXPR (READ))
     (PRINT (QUOTE STOP))
     (COND ((EQ *EXPR (QUOTE *STOP)))
           (T (TERPRI)
              (PRINT (EVAL *EXPR))
              (LOOP))))))
  (*LOAD (QUOTE LISINT))
  (LOOP)
  (SAVE))
```

To run LISINT, just compile it with the LISCMP compile-and-go script. The script will automatically run the interpreter when done:

```
$ sh lcg lisint.lisp
COMPILE
RUN

LISP INTERPRETER
IMAGE NOT LOADED
GO
```

The interpreter is ready to accept expressions at that point. To exit enter *STOP at the GO prompt. Sending a keyboard interrupt or

typing an unknown character, like "." or "%" will terminate the interpreter without updating the image file.

If you want to keep a LISINT binary, copy the file "run" to "lisint" after compiling the interpreter.

PROGRAMS WRITING PROGRAMS

Once an interpreter for LISP exists, a *macro expander* is only a couple of definitions away. Macro expansion adds another level of abstraction to LISP, because it allows to mechanically rewrite expressions before evaluating them. A macro expander is a program writing (or, rather, transforming) a program.

Technically a *macro* or *FEXPR* (*form expression*) is an ordinary LISP function that is bound in a separate name space. After reading an expression, the LISP system checks if any operators in the expression are bound in the macro namespace. If it finds such an operator, it applies the associated macro function to the arguments of the operator *without evaluating them first*. That is, the macro function receives the external form (S-expressions) of its arguments and not their values. The result of the function then *replaces* the expression containing the macro name as operator.

Macro transformation takes place after reading an expression, but before evaluating it. Of course the transformation is itself an instance of evaluation! This sounds more confusing than it is, though.

For example, given a macro Q that implements QUOTE, the following transformation would take place (here → means "expands to"):

```
(LIST (Q FOO) (Q BAR))
  →  (LIST (QUOTE FOO) (QUOTE BAR))
```

Note that Q cannot be implemented as a LISP function, because all LISP functions are applied by value, i.e. their arguments are evaluated before the function is applied to them. So

```
(SETQ Q (LAMBDA (X) X))
```

would not work, because X already has been evaluated when Q is applied to it. The above merely implements the identity function. For different reasons (it being a constant function)

```
(SETQ Q (LAMBDA (X) (QUOTE X)))
```

would not work. It would always return the atom X. But what about this?

```
(SETQ Q
  (LAMBDA (X)
    (LIST (QUOTE QUOTE) X)))
```

For any given x, this function would return (QUOTE x). However, x would still be evaluated before applying Q to it and the function would merely return a list object that *looks like* an application of QUOTE.

And this is exactly how macro expansion works: If the above Q function would be bound in the macro namespace, the interpreter would apply that function to its argument *without* evaluating it, and replace the application of Q with the value returned by Q. So (Q x) becomes (QUOTE x) and is *then* evaluated.

This is why LISP is so cool! Everything that looks like a LISP program *is* a LISP program as soon as it is being evaluated. A list resembling a LISP program is a LISP program as soon as it is passed to EVAL. This property is known as *homoiconicity*, the use of the same representation for code and data (from Greek “homo”, meaning “same” and “icon”, meaning “image”). Because of this property the macro expander can rewrite parts of an expression with ordinary list manipulation functions, like CAR, CDR, CONS, LIST, and APPEND.

The following modifications have to be made to LISINT to turn it into a LISP interpreter with macros, let us call it LISINT/M.

The variable MACROS is bound to the macro namespace, which is an association list and initially empty.

The ADDMACRO function just adds a new pair of the form

(*macro-name macro-function***)**

to head of the namespace. Macro redefinitions are made by adding a new association that shadows any existing one. Feel free to improve the function so that is replaces existing definitions instead of filling the namespace with dead macros.

```
(SETQ MACROS NIL)

(SETQ ADDMACRO
  (LAMBDA (N X)
    (SETQ MACROS
      (CONS (LIST N X) MACROS))
    N))
```

The following case has to be added to the (ATOM (CAR X)) branch of EVAL3. The location does not matter much, but since macros are not defined very often, it is a good idea to put it way down the list, for example after the case for SETQ.

This case implements the special form

(MACRO *name function***)**

which adds a new macro to the LISINT/M system.

```
            ((EQ (CAR X) (QUOTE MACRO))
              (ADDMACRO (CADR X) (CADDR X)))
```

EXPAND expands all macros in the expression X. Recursive macros are expanded recursively.

(1) If X is an atom, it is just returned, because atoms cannot be macro applications.

(2) If X is a list of the form (QUOTE ...), then it is just returned, because quoted objects cannot be macro applications. Without this case, expressions of the form

(QUOTE (*macro-name* **...))**

would be expanded, which would most probably not reflect the intention of the programmer.

(3) If X is a list, its car part (operator) is looked up in the macro namespace. If there is no such macro, EXPAND is mapped over X, thereby expanding all macro applications in subexpressions of X.

(4) If a list with a macro name in the first position is found, then a new list representing a function application is created as follows:

(*macro-name* $arg_1 \cdots arg_N$**)**
$\rightarrow$ **(***macro-function* **(QUOTE (**$arg_1 \cdots arg_N$**)))**

That is, the macro function associated with the given macro name is applied to the *quoted* list of macro arguments. For instance, the transformation of the Q macro application

```
(Q (FOO BAR BAZ))
```

would proceed as follows: the name Q is found in the macro namespace. It is associated with the function

```
(LAMBDA (X)
  (LIST (QUOTE QUOTE) (CAR X)))
```

(Note that macro functions have only one variable that will be bound to a list containing all arguments of the macro application. Hence the function extracts the first argument with CAR!)

The name Q in

```
(Q (FOO BAR BAZ))
```

is then replaced with the macro function function and the argument list (the *cdr* part of the above expression) is quoted:

```
((LAMBDA (X)
   (LIST (QUOTE QUOTE) (CAR X)))
 (QUOTE ((FOO BAR BAZ))))
```

The resulting expression is submitted to EVAL for evaluation. It returns

```
(QUOTE (FOO BAR BAZ))
```

which replaces the original application of Q. Therefore Q implements the QUOTE special form. Because it is *derived* from the (primitive, or built-in) QUOTE special form, it is also called a *derived special form*.

Before returning the expression returned by EVAL, it is expanded once more in order to transform recursive macros. In the case of the Q macro nothing will happen in this step, but macro functions may return expressions containing further macro applications, which would be expanded in this final step. We will get back to this immediately.

```
(SETQ EXPAND
  (LAMBDA (X)
    (COND ((ATOM X) X)
          ((EQ (CAR X) (QUOTE QUOTE)) X)
          (T (SETQ EXPAND-TMP
                   (ASSOC (CAR X) MACROS))
             (COND (EXPAND-TMP
                     (EXPAND
                       (EVAL
                         (CONS
                           (CADR EXPAND-TMP)
                           (LIST
                             (LIST (QUOTE QUOTE)
                                   (CDR X)))))))
                   (T (MAPCAR EXPAND X)))))))
```

The EVAL function itself is modified to expand its argument before passing it to EVAL3. Note that EVAL and EXPAND are mutually recursive: EVAL calls EXPAND to expand macros and EXPAND calls EVAL in order to evaluate applications of macro functions.

```
(SETQ EVAL
  (LAMBDA (X)
    (EVAL3 (EXPAND X) NIL NIL)))
```

Finally note that LISINT/M should use the image file name "LSINTM" (or, at least something other than "LISINT"), because LISINT and LISINT/M cannot share image files!

DERIVED SPECIAL FORMS

The Q macro used as an example above is probably one of the simplest derived special forms, expressing QUOTE in terms of QUOTE. Using the syntax added to LISINT in the previous section it would look like this:

```
(MACRO Q
  (LAMBDA (X)
    (LIST (QUOTE QUOTE) (CAR X))))
```

Note, again, that macros receive a quoted list of *all* arguments in their single variable. Hence the macro application

```
(Q (FOO BAR))
```

would deliver ((FOO BAR)) to Q in the variable X. Early LISP systems typically passed the entire special form, including the macro name, to FEXPRs [McCarthy1962] [MIT1967], but LISINT/M omits the macro name. The macro probably knows what it is called and omitting its name saves one CDR operation

for accessing each argument, which makes macro functions more readable.

The next example, the LISTQ macro, is a *recursive macro* that builds a fresh list of quoted objects. E.g.

```
(LISTQ FOO BAR BAZ)
```

is equal to

```
(LIST (QUOTE FOO) (QUOTE BAR) (QUOTE BAZ))
```

In case you wonder what the difference to just

```
(QUOTE (FOO BAR BAZ))
```

is, consider the *self-modifying* example on page 47! When NCONCing something to a quoted list, the quoted list will be modified. When NCONCing to a list created by LISTQ, a fresh list is modified every time. I.e. the example on page 47 would not be self-modifying if it used (LISTQ A B C) in the place of (QUOTE (A B C)).

The LISTQ macro rewrites (LISTQ) to NIL and

(LISTQ a_1 a_2 $\cdots$**)**

to

(CONS (QUOTE a_1**) (LISTQ** a_2 $\cdots$**))**

The result of the second case contains an application of LISTQ itself, so this is a *recursive macro*. Because macros expand recursively, this process continues until (LISTQ) expands to NIL. For example: (again, $\rightarrow$ means “expands to”):

```
(LISTQ FOO BAR BAZ)
  →   (CONS (QUOTE FOO)
            (LISTQ BAR BAZ))
  →   (CONS (QUOTE FOO)
            (CONS (QUOTE BAR)
                  (LISTQ FOO)))
```

```
→   (CONS (QUOTE FOO)
           (CONS (QUOTE BAR)
                 (CONS (QUOTE BAZ)
                       (LISTQ))))
→   (CONS (QUOTE FOO)
           (CONS (QUOTE BAR)
                 (CONS (QUOTE BAZ)
                       NIL)))
```

Note that rewriting terms (expressions) was quite awkward without *quasiquotation*! In modern LISP, the consequent of the general case of the COND in LISTQ could be written as

```
`(CONS (QUOTE ,(CAR A)) (LISTQ ,(CDR A)))
```

but back in the days of LISP 1.5 or LISP 1.6 you had to use CONS, LIST, and APPEND to create new expressions [MIT1967].

```
(MACRO LISTQ
  (LAMBDA (A)
    (COND ((NULL A) NIL)
          (T (LIST (QUOTE CONS)
                   (LIST (QUOTE QUOTE)
                         (CAR A))
                   (CONS (QUOTE LISTQ)
                         (CDR A)))))))
```

When looking at the relationship between LISTQ and LIST, it becomes quite clear that a non-recursive LISTQ derived special form can be devised which simply maps x to (QUOTE x) via MAPCAR:

```
   (LISTQ  A  B  C)
→ (LIST  'A 'B 'C)
```

This is interesting, because it shows that the full LISP language is available at macro expansion time:

```
(MACRO LISTQ
  (LAMBDA (A)
    (CONS (QUOTE LIST)
          (MAPCAR (LAMBDA (X)
                    (LIST (QUOTE QUOTE) X))
                  A))))
```

The LET special form of modern LISP is implemented by a macro that performs a rather complex transformation. It maps the special form

(LET ((v_1 a_1**)** ⋯ **(**v_N a_N**))** x ⋯**)** to **((LAMBDA (**v_1 ⋯ v_N**)** x ⋯**)** a_1 ⋯ a_N**)**

That is, it implements (almost) a parallel-binding variant of LABEL. Its macro extracts the variables v_i and arguments a_i from the special form using MAPCAR and then uses LIST and APPEND to create an equivalent function application.

Like COMMON LISP's LET, this macro is not intended for binding functions, but for different reasons! See page 99.

```
(MACRO LET
  (LAMBDA (A)
    (LABEL ((VS (MAPCAR CAR (CAR A)))
            (AS (MAPCAR CADR (CAR A))))
      (APPEND
        (LIST (APPEND
                (LIST (QUOTE LAMBDA) VS)
                (CDR A)))
        AS))))
```

MACRO EXPANSION IN THE COMPILER

Note that macro expansion only works in the LISINT/M interpreter, but cannot work in the LISCMP compiler, because LISCMP cannot interpret LISP programs at compile time. This is not a general limitation of LISP, though!

It is possible to combine LISCMP and LISINT/M in order to create a compiler that is capable of macro expansion. To create a macro-expanding compiler from nothing, the following steps are required:

- bootstrap a LISP compiler (done)
- write a LISP interpreter using the LISP compiler (done)
- embed the interpreter in the compiler

There is one minor complication, though! When a function is compiled to machine code, it is not available at the level of the interpreter. So how can the following program be compiled?

```
(SETQ DOUBLE (LAMBDA (X) (LIST X X)))

(MACRO QUADRUPLE
  (LAMBDA (X)
    (LIST (QUOTE QUOTE)
          (DOUBLE (DOUBLE (CAR X))))))

(QUADRUPLE FOO)
```

The problem is that the compiler will compile DOUBLE to machine code and then the interpreted QUADRUPLE function will attempt to use the compiled DOUBLE function, but DOUBLE is not available at the level of the interpreter.

One possible solution to this problem would be to compile *and* interpret all expressions submitted to the LISP system. In this case there would be a compiled version of DOUBLE that would be used in compiled programs and an interpreted version that would be used during macro expansion.

Many early LISP systems had both a compiler and an interpreter. In fact LISP 1 already had both [McCarthy1960]. The myth that LISP is an “interpreted language” is just that—a myth.

Of course maintaining two implementations of the same language in one system is prone to all kinds of trouble. There will be minor incompatibilities between the implementations that are founded in the inherent differences between interpretation and compilation. See, for example, the chapter on the “Peculiarities of the Compiler” in [Moon1974].

Modern LISP systems typically compile all expressions to machine code, so the dichotomy described here ceased to exist. However, this seamless integration is something that cannot be done in a purely symbolic LISP system, because the only data type is the cell. How is the compiler supposed to emit machine code to memory in such an environment?

So this is where the limits of the minimal, purely symbolic LISP are finally reached. It is pretty impressive though, what has been achieved on the way:

- a library of standard LISP functions
- a compiler
- an interpreter
- a macro expander

and all this is based on only four special forms and five functions:

ATOM CAR CDR COND CONS EQ LABEL LAMBDA QUOTE

(plus eleven more low-level functions for bootstrapping the LISP system).

What is particularly impressive about LISP is that the code is very *readable* all along, even if bootstrapped from such a minimal set of operations!

SCIENCE!
SCIENCE!

If you have not read Alfred Bester's "Tiger! Tiger!", go grab a copy!

FROM THE beginning, LISP had a particularly scientific flair to it. This may have been a characteristic of the pioneer days of computing in general or it may have had its origins in the mathematical background of LISP.

LISP was originally invented to solve a very specific problem, and that was the manipulation of *symbolic* (mathematical) *expressions* [McCarthy1958], a discipline that would later be known as "computer algebra". A computer algebra system (CAS) is a program that performs all kinds of term rewriting on the symbolic representation of mathematical expressions. For instance, the system may recognize terms of the form

```
(POWER X Y)
```

(meaning x^y) and be able to differentiate them with respect to x, giving

```
(TIMES Y (POWER X (DIFFERENCE Y 1)))
```

(meaning yx^{y-1}). Note that no numeric calculation is going on here, the operation is purely symbolic! In fact the LISP system presented in this book would be sufficient to implement a very basic variant of such a system.

LISP 1 already had a built-in SMPLFY function that could simplify mathematical terms of the above form. There was a lot of interest in symbolic computation systems back in those days, and many such systems were developed in different locations. At the M.I.T., the first one was probably a system named SAINT, the "Symbolic Automatic Integrator". The development eventually culminated in MACSYMA, "Project MAC's Symbolic Manipulator" [Moses2012], which quickly became one of the most popular computer algebra system of its time. MACSYMA was written in MACLISP which was a removed descendant of the original LISP 1:

```
LISP 1  →  LISP 1.5  →  LISP 1.6  →  MACLISP
                                     (PDP-6 LISP)
```

At the time when MACSYMA was developed, LISP had gained a lot of momentum, though, so it was no longer limited to its original domain. LISP systems were used for all kinds of purposes, ported to all kinds of systems, and rewritten in different programming languages. (See, for example, [Nordstrom1970].)

While LISP evolved quickly past its original purpose it still kept its mathematical inclination. Even LISP itself became a subject of scientific reasoning. Due to the declarative nature of LISP programs it is easy to prove certain properties of a program, which was a rather difficult task in most other languages of that time.

In *declarative programming* style, the computer is not instructed to *do* certain things, but the relation between the input and the output of a function is being described—or declared—instead.

For instance, the following variant of the MAPLIST function may be considered to be declarative:

```
(SETQ MAPLIST
  (LAMBDA (F A)
    (COND ((NULL A) NIL)
          (T (CONS (F A) (MAPLIST F (CDR A)))))))
```

It maps the value $A = \text{NIL}$ to NIL and every other value of A to $F(A)$ consed to the application of MAPLIST to the rest of A. A few things become obvious immediately:

(1) The value of F is never manipulated in any way, hence it can be considered to be a constant.

(2) Only CDR and NULL are being applied to A, hence the length of A can only decrease.

(3) When the length of A reaches null, the program terminates.

(4) Due to (1), (2), and (3) the MAPLIST program must terminate for any proper list A (as long as F terminates).

Of course this reasoning glosses over the fact that (CAR A) is undefined if A is an atom and F must be a unary function or LEXPR, but the general course of the proof is sound.

PROOF OF LABEL IN TERMS OF LAMBDA

Here is a more elaborate proof. An earlier chapter in this text (page 20) claimed that LABEL can be expressed in terms of LAMBDA. The following macro mechanically rewrites a special form named LETN ("nested LET"), which is equivalent to LABEL, to another expression that contains only applications of lambda functions, but still implements LABEL.

The transformation of the LETN special form is as follows:

(LETN ((v_1 a_1**)** $\cdots$ **(**v_N a_N**))** x $\cdots$**)** $\rightarrow$ **(LET ((**v_1 a_1**))** $\cdots$ **(LET ((**v_N a_N**))** **((LAMBDA () **x $\cdots$**))) **$\cdots$**)**

That is, it transforms LETN to a set of nested applications of the LET special form, so that each a_i is evaluated and bound to the corresponding v_i *before* a_{i+1} is evaluated. Hence LETN implements sequential binding. The in-situ lambda function application wrapped around the body of the special form prevents tail call optimization, just as in LABEL. Therefore LETN is equivalent to LABEL.

```
(MACRO LETN
  (LAMBDA (A)
    (COND ((NULL (CAR A))
           (LIST (APPEND
                   (QUOTE (LAMBDA ()))
                   (CDR A))))
          (T (LIST (QUOTE LET)
                   (LIST (CAAR A))
```

```
                (APPEND
                  (LIST (QUOTE LETN)
                        (CDAR A))
                  (CDR A)))))))
```

Reasoning similar to the examination of MAPLIST, above, can be applied to the LETN macro as well. The base case of the proof rewrites the LETN form with an empty binding section:

(LETN () X ...) $\rightarrow$ **((LAMBDA () X ...))**

and the induction step is as follows:

(LETN ((A B) (C D) ...) X ...)
$\rightarrow$ **(LET ((A B)) (LETN ((C D) ...) X ...))**

So the LETN macro eventually rewrites each binding of LETN to a separate, nested application of LET. It rewrites its complete own special form in terms of LET, and we already known from the previous chapter that LET expands to LAMBDA:

(LET ((v_1 a_1**)** $\rightarrow$ **((LAMBDA (**v_1 $\cdots$ v_N**)** x $\cdots$**)**
$\cdots$ a_1 $\cdots$ a_N**)**
(v_N a_1**))**
x $\cdots$**)**

Therefore, LABEL can be completely expressed in terms of LAMBDA. The LABEL special form is, strictly speaking, optional for bootstrapping a minimal implementation of (dynamically scoped) LISP. This is a mere technicality, though, because for all practical purposes an implementation of basic LISP functions, like READ, PRINT, or even MAPCAR, would be very hard to write or even to decipher without LABEL. For instance, compare the very simple example

```
(LETN
  ((F (LAMBDA (X) (COND ((NULL X)) ((G X)))))
   (G (LAMBDA (X) (F (CDR X)))))
  (F (QUOTE (1 2 3 4 5))))
```

to its expanded form,

```
((LAMBDA (F)
   ((LAMBDA (G)
      ((LAMBDA () (F (QUOTE (1 2 3 4 5))))))
    (LAMBDA (X) (F (CDR X)))))
 (LAMBDA (X) (COND ((NULL X)) ((G X)))))
```

and then imagine READ or PRINT being written in this way.

BTW, many LISP systems provide a MACROEXPAND function that expands macro applications without evaluating the expanded form, so there is no need to expand macro applications manually. Implementing a version of MACROEXPAND for LISINT/M is left as an exercise to the reader. (Try it, it is not that hard!)

The alert reader may have noticed the injection *dynamically scoped* in the above discussion, and a user of modern LISP may object to the conclusion, because COMMON LISP's "LABELS" or SCHEME's "LETREC" cannot be expressed in terms of LAMBDA alone. This is true, and we will explore this detail further after having a closer look at different ways of binding symbols.

BINDING STRATEGIES

In modern (COMMON) LISP variables have *indefinite extent* and *lexical scope* [CLtL1984]. Indefinite extent means that once a variable has been created it exists forever (or until the garbage collector can prove that it is not longer accessible). Lexical scope means that each variable is only visible inside of a certain textual (lexical) region.

Using COMMON LISP nomenclature, variables in early LISP systems had *indefinite scope* and *dynamic extent*—a combination that is commonly referred to as *dynamic scope*. Indefinite scope means that variables are visible everywhere and not just in a limited textual region. Dynamic extent means that bindings of

variables are a function of time: they are established at some point, remain active for a while, and are then destroyed.

In the following, “lexical scoping” will refer to the combination of indefinite extent and lexical scope and “dynamic scoping” will refer to the combination of indefinite scope and dynamic extent.

The following program finds out what kind of scoping a LISP system uses:

```
((LAMBDA (X)
   ((LAMBDA (F)
      ((LAMBDA (X)
         (CAR (F)))
       (QUOTE (DYNAMIC))))
    (LAMBDA () X)))
 (QUOTE (LEXICAL)))
```

When running this program in SCHEME or COMMON LISP, it will return LEXICAL, and when running it in an older LISP system, like MACLISP or FRANZ LISP, it will return DYNAMIC. In fact, to run it in MACLISP, FRANZ LISP, or COMMON LISP, some minor modifications have to be made to the program due to the LISP-N nature of the later dialects of LISP:

```
((LAMBDA (X)
   ((LAMBDA (F)
      ((LAMBDA (X)
         (CAR (FUNCALL F)))
       (QUOTE (DYNAMIC))))
    (FUNCTION (LAMBDA () X))))
 (QUOTE (LEXICAL)))
```

According to the definition of the APPLY function in Appendix B of [McCarthy1960] and examples in the secondary literature (e.g. [Weissman1967]), the program should work in LISP 1.5 when FUNCALL is omitted, but the author could not get it to work in LISP 1.5 version 1960-03-01. In that version in-situ application of

LAMBDA and LABEL special forms does not seem to work at all. Even more interestingly, the expression

```
(LAMBDA (X) X) (FOO)
```

does work when passed directly to EVALQUOTE, but

```
DEFINE (( (TEST (LAMBDA (Q) ((LAMBDA (X) X)
                             (QUOTE FOO)))) ))
TEST (NIL)
```

does not work. Maybe this as a peculiarity of that specific version or maybe the implementation did not follow the specification.

The latter does not seem very improbable, because the semi-formal specification of LISP 1.5 also defines lexical scoping via the FUNCTION special form, but this feature was never actually implemented [Barnett2020].

Dynamic scoping was the standard scoping mechanism in all LISP systems until SCHEME implemented lexical scoping as its default [Scheme1975]. Lexical scoping is the default binding mechanism in pretty much all LISP systems since around 1984, when COMMON LISP was first documented [CLtL1984].

You may have noticed that the above program uses the literals (DYNAMIC) and (LEXICAL) and takes their car part in the body of F instead of passing DYNAMIC and LEXICAL to the functions and just returning the variable X. This is necessary, because *tail call optimization* can interfere with the dynamic scoping mechanism. The program

```
((LAMBDA (X)
   ((LAMBDA (F)
      ((LAMBDA (X)
         (F))
       (QUOTE DYNAMIC)))
    (LAMBDA () X)))
 (QUOTE LEXICAL))
```

would indeed return LEXICAL when compiled with LISCMP, but not when interpreted by LISINT. The effect is due to tail call optimization, and (CAR (F)) turns the application of F into a non-tail application. (This has been explained more in detail on page 103). This is one of the reasons why dynamic scoping is considered to be difficult: because it relies on exact timing, it is easy to introduce subtle effects when using different mechanisms for optimizations like tail call optimization. Finding the reason for the different results of LISCMP and LISINT is left as an exercise to the interested reader.

The style used in the above program was not very popular in the early days of LISP, exactly because dynamic scoping made it hard to keep track of the extent of a specific binding. For the sake of clarity here is an alternative version of the program, this time using the LET macro from the previous chapter:

```
(LET ((X (QUOTE (LEXICAL))))
  (LET ((F (LAMBDA () X)))
    (LET ((X (QUOTE (DYNAMIC))))
      (CAR (F)))))
```

Under *lexical scoping*, the binding of X in the body of F is identical to the binding of X that was created in the outer LET. The binding of X that is created in the inner LET is an independent binding that has no relationship to the outer X, so the inner and outer X are two different variables. Therefore the X in F still refers to the value (LEXICAL) when F is applied.

In other words: the value of X in F is a function of *location*. The X of F appears in a spatial context where X is bound to (LEXICAL), hence it inherits that binding. The binding persists, locally, inside of the function, even when a new binding for X (and hence a new variable) is introduced later.

Under *dynamic scoping*, there is only one variable called X and its binding changes dynamically. First it is bound to (LEXICAL) in the

outer LET and later it is bound to (DYNAMIC) in the inner LET, and this is the value it has *at the time* when F is applied.

Here the binding of a global variable X changes dynamically from (LEXICAL) to (DYNAMIC) (and then back to (LEXICAL) when leaving the innermost scope).

Both lexical and dynamic scoping behave equally with regard to function variables. For instance,

```
(LET ((X (QUOTE 1)))
  (PRINT X)
  (LET ((X (QUOTE 2)))
    (PRINT X)
    (LET ((X (QUOTE 3)))
      (PRINT X))
    (PRINT X))
  (PRINT X))
```

will print 1, 2, 3, 2, 1 even in a dynamically scoped LISP system. (Because LET is equal to LAMBDA, X is a function variable.) In this example, the spatial and temporal aspects of the bindings coincide, so both mechanisms provide the expected result. It does not matter if there are independent instances of X or just one instance that changes dynamically.

Dynamic scoping works fine unless it does not, so it took a while to discover all the problems associated with it and even longer until a solution became widely available.

There is really only one point where dynamic scoping gets in the way, though, and this is when *free variables* are used in combination with *higher-order functions*.

FIRST-CLASS FUNCTIONS

A *first-class data object* is an ordinary data object that can be passed to functions and returned by functions. LISP functions are first-class objects.

A function can contain two kinds of variables: bound ones and free ones. A *variable* is *bound* in a function, if it appears in the list of variables of that function. For instance, both X and Y are bound in the function

```
(LAMBDA (X Y) (F X Y))
```

(And, yes, this is probably where the term “binding” comes from! The concept of bound and free variables is much older than LISP and appeared in mathematical logic long before the first computer was invented.)

A *free variable* is a variable that appears in a function, but is *not* bound in that function. The variable F is a free variable in the above example.

Free variables in combination with dynamic scoping were the reason why limited use was made of higher-order functions in the early days of LISP,

The MAPCAR and REDUCE functions, for instance, accept a function as their first argument. A function accepting a function as an argument or returning a function to the caller is called a *higher-order function*. In early LISP, higher-order functions typically received a function argument and did something with it, like mapping it over a list or using it to combine elements of a list.

No higher-order functions in early LISP returned functions as values, and it is easy to demonstrate why. In SCHEME, which is lexically scoped, you might write a higher-order function named CONSER that returns a function that conses something to its argument:

```
(define conser
  (lambda (x)
    (lambda (y)      ; <-- this function
      (cons x y))))  ;     will be returned
```

```
(define foo-conser (conser (quote foo)))

(foo-conser (quote bar))  ==>  (foo . bar)
```

A similar function can be defined in any modern LISP, like COMMON LISP. But what happens in dynamically scoped LISP?

```
(SETQ X (QUOTE OOPS))

(SETQ CONSER
  (LAMBDA (X)
    (LAMBDA (Y)
      (CONS X Y))))

(SETQ FOO-CONSER (CONSER (QUOTE FOO)))

(FOO-CONSER (QUOTE BAR))  ==>  (OOPS . BAR)
```

When the outer lambda function of CONSER returns, it unbinds its variable, X, thereby restoring its outer value, OOPS. Changing the value of the outer (global) variable X later will also change the value that FOO-CONSER returns, because the global X and the X in FOO-CONSER share the same binding.

This problem is known as the *upward FUNARG problem* [Weizenbaum1968], and it is the reason why functions returning functions were not exactly popular in the early days of LISP.

The “upward” FUNARG problem is characterized by a function that is passed “up” to its caller, thereby unbinding its free variables. There is also a “downward” FUNARG problem where free variables of a function are *rebound* when passing it “down” as an argument to another function. The following example illustrates this problem. It will evaluate to (X X X) under lexical scoping, but to ((A B C) (B C) (C)) under dynamic scoping.

```
(LABEL
  ((MAPLIST (LAMBDA (F X)
     (COND ((NULL X) NIL)
           (T (CONS (F X)
                    (MAPLIST F (CDR X))))))))
```

```
((LAMBDA (X)
   (MAPLIST (LAMBDA (Y) X)
            (QUOTE (A B C))))
 (QUOTE X)))
```

Here the variable X, which is free in (LAMBDA (Y) X), will be rebound to (A B C) (and then (B C) and then (C)) when MAPLIST is applied to it its arguments. This happens because one of the arguments of MAPLIST is also named X, so the two variables share the same binding.

John McCarthy himself notes in [McCarthy1981] that these problems arose, because he had not understood Church's *lambda calculus* [Church1941] properly. One cannot blame him! Church's treatises on lambda calculus are packed with information and not exactly an easy reading, and the superficial simplicity of the calculus is surprisingly hard to implement on a computer, at least when done efficiently. So it may have been a good thing in the end that LISP was dynamically scoped in the beginning. It simplified its implementation significantly, and thereby probably accelerated its adoption beyond the M.I.T.

Implementing lexical scoping *inefficiently* is pretty straight forward, though. The following pages list a modified version of McCarthy's metacircular EVAL that implements lexical scoping. The differences to the version in chapter 1 will be highlighted in the listing. Note that the interpreter is no longer metacircular in order to make its code more comprehensible.

```
(LABEL
  ((LOOKUP
     (LAMBDA (X E)
       (COND ((EQ NIL E) NIL)
             ((EQ X (CAAR E))
               (CADAR E))
             (T (LOOKUP X (CDR E))))))
```

```
(EVCON
  (LAMBDA (C E)
    (COND ((XEVAL (CAAR C) E)
           (XEVAL (CADAR C) E))
          (T (EVCON (CDR C) E)))))
```

In a lexically scoped system, every function has its own local *environment*. Function arguments are evaluated in the environment of the caller (CE), but added to the environment of the callee (E).

```
(BIND
  (LAMBDA (V A E CE)
    (COND ((EQ V NIL) E)
          (T (CONS (LIST (CAR V)
                         (XEVAL (CAR A) CE))
                   (BIND (CDR V)
                         (CDR A) E CE))))))
```

The EVLAB function rewrites LABEL special forms in the following way before passing them back to XEVAL:

```
(LABEL ((v1 a1)        →   ((LAMBDA (v1 ··· vN)
        ···                     (SETQ v1 a1)
       (vN aN))                  ···
  x1 ···)                        (SETQ vN aN)
                                 x1 ···)
                            NIL1 ··· NILN)
```

As can be seen in the expanded form, the expressions of LABEL (a_i) still evaluate in top-to-bottom order and a_{i+1} in only evaluated after binding v_i to a_i. Hence the LAMBDA expression on the right side implements LABEL. The expanded construct does a tail

application in the last expression of its body, though, which LABEL does not do. This is not a problem in a lexically scoped system, as will be explained later in this chapter.

The above transformation is necessary in order to implement recursive (and mutually recursive) functions under lexical scoping. We will get back to this!

The rather unintuitive composition of nested APPEND and LIST in the body of EVLAB would look like this if we had quasiquotation:

```
`((LAMBDA ,VS ,@(SETUP VS AS) ,@XS) ,@NS)
```

Note that LABEL creates a cyclic structure in lexically scoped LISP! Printing a recursive function in naive implementations of LISP (like the one described here) will not terminate and eventually crash the system.

```
(EVLAB
  (LAMBDA (B XS E)
    (LABEL
      ((VS (MAPCAR CAR B))
       (AS (MAPCAR CADR B))
       (NS (MAPCAR (LAMBDA (X) NIL) VS))
       (SETUP (LAMBDA (VS AS)
         (COND ((NULL VS) NIL)
               (T (CONS (LIST (QUOTE SETQ)
                              (CAR VS)
                              (CAR AS))
                        (SETUP (CDR VS)
                               (CDR AS))))))))
      (XEVAL (APPEND
               (LIST
                 (APPEND
                   (LIST (QUOTE LAMBDA))
                   (APPEND
                     (LIST VS)
```

```
                        (APPEND (SETUP VS AS)
                                XS))))
                NS) E))))
```

EVLIS evaluates a multi-expression body XS in the environment E. It basically implements PROGN. Multi-expression bodies are needed in order to evaluate the expanded forms of LABEL.

```
(EVLIS
  (LAMBDA (XS E)
    (COND ((NULL (CDR XS))
           (XEVAL (CAR XS) E))
          (T (XEVAL (CAR XS) E)
             (EVLIS (CDR XS) E)))))
```

EVSET implements SETQ. There is only one active environment at any given point of time, so it just looks up the association of an atom and replaces its value.

```
(EVSET
  (LAMBDA (X E)
    (RPLACA (CDR (ASSOC (CAR X) E))
            (XEVAL (CADR X) E))))

(XEVAL
  (LAMBDA (X E)
    (COND
      ((EQ X T) T)
      ((ATOM X)
         (LOOKUP X E))
      ((ATOM (CAR X))
        (COND
          ((EQ (CAR X) (QUOTE QUOTE))
```

```
    (CADR X))
  ((EQ (CAR X) (QUOTE ATOM))
    (ATOM (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE EQ))
    (EQ (XEVAL (CADR X) E)
        (XEVAL (CADR (CDR X)) E)))
  ((EQ (CAR X) (QUOTE CAR))
    (CAR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CDR))
    (CDR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CAAR))
    (CAAR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CADR))
    (CADR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CDAR))
    (CDAR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CADAR))
    (CADAR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CADDR))
    (CADDR (XEVAL (CADR X) E)))
  ((EQ (CAR X) (QUOTE CONS))
    (CONS (XEVAL (CADR X) E)
          (XEVAL (CADDR X) E)))
  ((EQ (CAR X) (QUOTE COND))
    (EVCON (CDR X) E))
  ((EQ (CAR X) (QUOTE SETQ))
    (EVSET (CDR X) E))
  ((EQ (CAR X) (QUOTE LABEL))
    (EVLAB (CADR X) (CDDR X) E))
  ((EQ NIL (CAR X))
    (QUOTE *UNDEFINED))
```

The following new case turns an expression of the form

(LAMBDA *variables expression* ⋯**)**

into a *closure* (function) of the form

(*CLOSURE *variables environment expression* ⋯**)**

The resulting closure carries with it a copy of the environment that was in effect at the time of its creation. The *expressions* will evaluate in this environment when the closure is later applied. Note that the environment is inserted *before* the expressions, because there is a variable number of expressions.

A closure is called a “closure”, because is *closes over* the free variables in its body, which is a fancy term for memorizing the bindings of free variables.

```
((EQ (CAR X) (QUOTE LAMBDA))
  (APPEND (LIST (QUOTE *CLOSURE)
                (CADR X)
                E)
          (CDDR X)))
(T (XEVAL (CONS (XEVAL (CAR X) E)
                (CDR X))
          E))))
```

Closures evaluate in the environment stored in them, thereby reestablishing any bindings that were in effect when the closure was created. (The caddar part of a function application is the saved environment of the applied closure, and the cdddar part is the multi-expression body of the closure.)

```
((EQ (CAAR X) (QUOTE *CLOSURE))
  (EVLIS (CDDR (CDAR X))
         (BIND (CADAR X)
               (CDR X)
               (CADR (CDAR X))
               E)))
```

In the original LISP 1, expressions of the form

```
((F X) Y)
```

were unexpected, but they are common in lexically scoped LISP, so the following case will catch these.

```
        (T (XEVAL (CONS (XEVAL (CAR X) E)
                        (CDR X))
                  E)))))

  ;; DOWNWARD FUNARG PROBLEM, EXPECTED: (X X X)
  (EXPR1 (QUOTE
           (LABEL
             ((MAPLIST (LAMBDA (F X)
                (COND ((EQ X NIL) NIL)
                      (T (CONS (F X)
                               (MAPLIST F
                                 (CDR X))))))))
             ((LAMBDA (X)
                (MAPLIST (LAMBDA (Y) X)
                         (QUOTE (A B C))))
              (QUOTE X)))))

  ;; UPWARD FUNARG PROB., EXPECTED: (FOO . BAR)
  (EXPR2 (QUOTE
           (LABEL
             ((CONSER
               (LAMBDA (X)
                 (LAMBDA (Y)
                   (CONS X Y)))))
             ((CONSER (QUOTE FOO))
              (QUOTE BAR))))))

 (LIST (XEVAL EXPR1 NIL)
       (XEVAL EXPR2 NIL)))
```

The example programs (EXPR1, EXPR2) in the LABEL evaluate to the expected results, so neither the upward nor the downward FUNARG problem exists in this implementation. However, the implementation uses *deep binding*, which is a very expensive ($O(n)$) method for looking up variables. Looking up the values of variables is one of the most frequent operations in a LISP system, though, and hence the $O(1)$ shallow binding method is preferred whenever possible. Its is possible to combine shallow binding with lexical scoping, but this approach is much more complex and beyond the scope of this text. See, for example, [Holm2016].

RECURSIVE FUNCTIONS AND DYNAMIC SCOPE

Here is a final note on recursive functions and binding strategy. There are in fact special forms that do work under dynamic scoping, but do not work under lexical scoping. One of them has been used throughout this entire text, probably without giving too much thought to it.

We have seen (and proven) that LABEL is in fact equal to nested LAMBDA, and LABEL is used to define local *recursive functions*. Therefore LAMBDA can also be used to bind recursive functions:

```
((LAMBDA (F)
   ; DISABLE TAIL CALL OPTIMIZATION
   ((LAMBDA () (F (QUOTE (1 2 3 4 5))))))
 (LAMBDA (X)
   (COND ((NULL X))
         (T (F (CDR X))))))  ==>  T
```

This cannot work in lexically scoped LISP, though, because (LAMBDA (X) ...) will close over F *before* F is bound to the function by (LAMBDA (F) ...). So how would LABEL be implemented in lexically scoped LISP?

The problem with the above program in a lexically scoped system is that there are two variables named F. One global one (with an

unknown value) that is being closed over, and a local one in the topmost function in the example. In order to make recursion work, there must not be more than one variable named F.

So what LABEL would have to do is to create a variable F, close over it, and *then* modify it. In SCHEME, LABEL can be defined as a macro that expands as follows (this is basically the SCHEME version of the expansion outlined on page 191):

```
(LABEL ((v1 a1)        →   ((lambda (v1 ··· vN)
        ···                    (set! v1 a1)
       (vN aN))                ···
  x ···)                       (set! vN aN)
                               x ···)
                             #f ··· #f)
```

The program

```
(label ((f (lambda (x)
             (cond ((null? x))
                   (else (f (cdr x)))))))
  (f (quote (1 2 3 4 5))))
```

would then expand to

```
((lambda (f)
   (set! f (lambda (x)
             (cond ((null? x))
                   (else (f (cdr x))))))
   (f (quote (1 2 3 4 5))))
 #f)
```

Here a single variable named F is created by the outer LAMBDA and bound to #F (SCHEME's "false" value). *The same* variable F is then being closed over by the lambda function and then *the same* F is modified to bind to the function, thereby making it recursive. The variable F in the environment of the recursive function and the variable F in the outer function *share the same binding*, because they are in the same lexical scope. The same

variable is bound in the outer function and free in the inner function.

Unlike the dynamically scoped LABEL the SCHEME version of LABEL or even SCHEME's own recursive binding form LETREC are tail recursive. That is, under lexical scoping the following function will evaluate in constant space for any size of list that is passed to it.

```
(SETQ F
  (LAMBDA (X)
    (LABEL ((G (LAMBDA (Y) (H Y)))
            (H (LAMBDA (Z) (F (CDR Z)))))
      (COND ((NULL X))
            (T (G X))))))
```

Remember that dynamically scoped LABEL cannot evaluate in constant space, because tail application would unbind G and H in (G X), so H would be unbound in G. (See also page 99.)

Under lexical scoping, tail application is possible, because the variables G and H are local to F in this case. Their bindings are independent of any other binding of the symbols G and H in different locations and hence they do not have to be "unbound" at the end of their scope. Because no timing issues arise, tail call elimination is much simpler and can be applied without any restrictions. Even in-situ applications of lambda functions can be tail applications, because free variables of lambda functions cannot be changed from outside of the function and are never "unbound". The concept of unbinding symbols does not even exist under lexical scoping, because bindings in lexically scoped systems have indefinite extent.

Here is the SCHEME variant of the above program with the LABEL special form expanded according to the scheme (no pun intended!) outlined earlier:

```
(define f
  (lambda (x)
    ((lambda (g h)
       (set! g (lambda (y) (h y)))
       (set! h (lambda (z) (f (cdr z))))
       (cond ((null? x)
              (else (g x)))))
     #f #f)))
```

The in-situ application of (LAMBDA (G H) ...) can safely be tail call optimized here, because (LAMBDA (Y) ...) closes over H, and neither G nor H are ever unbound anyway.

TEMPORARY VARIABLES REVISITED

Because all binding constructs can be tail call optimized under lexical scoping, the temporary variable problem (page 134) does not exist, either. Any binding construct, like LABEL, LABELS, LETREC, LET, LET*, or LAMBDA can be used to introduce local variables. Under lexcial scoping the example from page 137 would work *and* evaluate in constant space:

```
(LABEL ((TMP (SUCC (CAR X))))
  (LOOP (CDR X)
        (EQ TMP (QUOTE 0))
        (CONS TMP Y)))
```

Lexical scoping has made *some* things more complicated, but it has also made *a lot of things* simpler and more consistent!

Finally note that *local recursive* functions in dynamically scoped systems offer another way to introduce temporary variables that can be combined with (internal) tail call optimization. The following example illustrates this approach. Using the NEXT function from page 135, it creates a list of the form ((0) (0) (1) (1) (2) (2) ...):

```
(SETQ DOUBLES
  (LAMBDA (X)
    (LABEL ((V NIL)
            (D (LAMBDA (X R)
                 (COND ((NULL X)
                        (NREVERSE R))
                       (T (SETQ V (NEXT NIL))
                          (D (CDR X)
                             (CONS V
                               (CONS V R))))))))
      (D X NIL))))
```

Here the local variable V is a true local variable, because it will be unbound at the end of the LABEL. (We cannot name it TMP, because NEXT uses this name for a global variable!) At the same time the internal function D will be tail call optimized, because it evaluates *inside* of the LABEL, i.e. no recursive call in D leaves the LABEL.

Obviously this approach only works in functions that recurse internally, like D. In fact it would also work in INCR (page 77) and NREVERSE (page 47). Functions that recurse at the top-level, like ASSOC or MEMBER, can be rewritten to recurse internally, but at the cost of adding another binding. In fact this additional cost exists even in functions that already do recurse internally, which is why a global temporary variable may still be preferred (like in NREVERSE, because it is a system-level function).

GARBAGE COLLECTION

The *automatic memory manager*, which is commonly also called the *garbage collector*, is the mechanism that recycles the cells of conses and atoms as soon as it can prove that these cells can no longer be accessed by any LISP program in the system. The mechanism thus creates the illusion of an infinite pool of cells. For instance, the program

```
(LABEL ((F (LAMBDA ()
             (CONS (QUOTE FOO) (QUOTE BAR))
             (F))))
  (F))
```

will—in theory—run forever and allocate an infinite number of cons cells. Even if it does not really run forever, it will exhaust a cons pool of 65,535 cells in a fraction of a second on a modern computer system. However, it will keep running and keep allocating cells. This is because the cells allocated by the program are neither bound to any variable nor pushed to the stack, so they can be recycled during the next garbage collection. On the author's old 750MHz notebook computer, the above program will trigger almost 500 garbage collections per second.

So what exactly does the garbage collector (GC) do? Whenever CONS (or IMPLODE, EXPLODE, READ, LAMBDA, etc) allocate cells, these cells are removed from a list called the *freelist*. The freelist is a linked list that connects all unused cells in the cell pool via their cdr fields. Like in all (proper) lists, its last cdr field is NIL. So when the freelist is NIL, a garbage collections is initiated.

There are various methods for garbage collection—for a more detailed discussion, see [JL1996]. A rather simple approach to GC is the family of *mark & sweep* algorithms, which divide the task into two phases:

(1) a "mark" phase, in which used cells are marked;

(2) a "sweep" phase, in which the freelist is rebuilt.

In the first phase the garbage collector marks all cells that are currently in use by the system by setting a tag bit in them. The marking phase is basically a tree traversal. It starts at the root of an object and uses depth-first traversal to set the *MARK tags* of all cells in the tree. The roots of the trees to mark are locations like

- the symbol table
- the object list
- the EXPR registers
- the runtime stack

These locations are called the *GC roots* of the system. In the LISCMP runtime library the GC roots are exactly the special slots listed in the table on page 72.

In the second phase, the GC will scan the cell pool from start to end and

- reset the MARK bit of cells that have been marked, and
- add all other cells to the freelist.

Cells that cannot be reached through any GC root can no longer be accessed by any LISP program in memory, so the GC algorithm has proven that the cells that are still unmarked after the first phase can safely be recycled.

The automatic memory management system of LISCMP is part of the low-level runtime library. There is no reason, though, why a garbage collector should not be written in LISP.

A GARBAGE COLLECTOR IN LISP

The garbage collector presented in the following can—again, in theory—replace the GC of LISCMP. In fact it could, if it were not for a single issue: LISCMP allocates arguments of function applications in the cell pool, and the sweep phase will iterate over each cell in the pool, doing one function call per iteration. So there will be more function applications than cells in the freelist and therefore running the garbage collection function would trigger a garbage collection.

This issue could be addressed by separating the runtime stack of the LISP system from its cell pool [Holm2019], i.e. allocating the

cells of the stack in a fixed vector rather than in the cell pool itself. Then then GC function would not allocate any cells from the pool and hence it could actually be used to perform the task of garbage collection in the system.

This is not how LISCMP works, though, and the modification outlined above is not trivial, so the discussion of the GC algorithm will be limited to theory at this point.

The MARK function marks all atomic cells and cons cells of a given tree N. It does so by traversing the tree in depth-first order and setting the MARK bit of the tag field of each visited cell.

A few new functions and variables have to be added to the LISP system in order to make GC work. They will be introduced in the following pages.

The *SETMARK and *SETTRAV functions set or clear the MARK and TRAV (traverse) bits of a cell, just like *SETATOM sets or clears the ATOM bit. The *MARKP and *TRAVP predicates check the MARK and TRAV bits and return T, if the corresponding bit is set. The bits will be explained after a short detour.

Depth-first traversal is a recursive algorithm that typically requires a stack to keep track of unvisited nodes of a tree. At each node, when the car branch is followed, the cdr branch has to be memorized for later traversal. This is unfortunate, because it means that traversing a tree requires an amount of memory that cannot be known before the traversal takes place.

The *Deutsch/Schorr/Waite* (DSW) graph marking algorithm is a clever algorithm that uses the space in the tree itself to save addresses of yet-to-be-visited nodes [SW1967]. The basic idea is that the algorithm *reverses pointers* of nodes. Normally a node points to two children, but when descending into a branch, the child will be modified to point to the parent. When returning to the child after traversing its branches, the original pointer is restored. See figure 15 for an illustration.

The DSW algorithm needs three cells of storage outside of the pool and two additional bits of storage per cell. There were six tag bits in the IBM 704 implementation of LISP and there are eight tag bis in LISCMP, so there is plenty of space left.

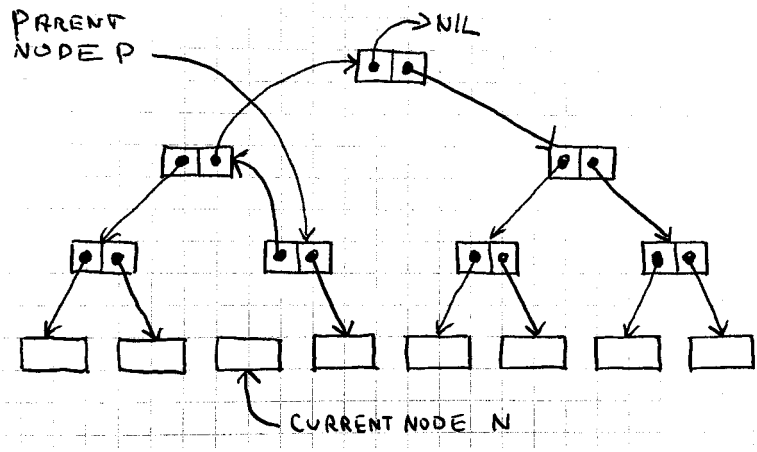


Fig. 15 – Pointer reversal in the D/S/W algorithm

Of the three additional cells, one is used for swapping values and the other two are used for pointing to the currently visited node and its parent.

The algorithm starts with the node pointer N pointing to the root of the tree to mark and the parent pointer P set to NIL, because the root has no parent. In the following a “node” is the same as a cell, but emphasizes the tree nature of the structure in which it is contained.

The *TRAV tag* of a cell indicates that the cdr branch of the cell still has to be traversed. Initially all TRAV and MARK tags of all cells are cleared.

(1) When the current node N is already marked and the parent P is NIL, the MARK function exits.

(2) When the current node N is already marked and has its TRAV tag set, the function clears its TRAV tag and proceeds with the cdr branch of N.

(3) When N is marked and does not have its TRAV tag set, the function returns to the parent and moves P to the parent of the parent.

(4) When the current node N is an atom, the MARK tag of N is set. The car branch of N will not be marked, and the algorithm proceeds immediately with the cdr branch.

(5) When the current node is a cons cell, both its MARK and TRAV tags will be set and the function will proceed with its car branch.

When the algorithm finishes, all nodes of the tree will have their MARK tags set and their TRAV tags cleared. Verifying this is left as an exercise to the reader.

The MARK function basically consists of the five cases outlined above. It does a lot of value shuffling using SETQ, *RPLACD, and *REPLACD. The “starred” variants of RPLACA, CAR, etc have to be used, because the function also processes atomic cells.

```
(SETQ MARK
  (LAMBDA (N)
    (LABEL
      ((P NIL)  ; PARENT
       (X NIL)  ; TEMPORARY
       (LOOP (LAMBDA (N)
         (COND
           ((*MARKP N)
             (COND
               ((EQ NIL P))              ; (1)
               ((*TRAVP P)               ; (2)
                  (SETQ X (*CDR P))
```

```
                  (*RPLACD P (*CAR P))
                  (*RPLACA P N)
                  (*SETTRAV P NIL)
                  (LOOP X))
               (T (SETQ X P)              ; (3)
                  (SETQ P (*CDR X))
                  (*RPLACD X N)
                  (LOOP X))))
         ((*ATOMP N)                      ; (4)
            (SETQ X (*CDR N))
            (*RPLACD N P)
            (SETQ P N)
            (*SETMARK P T)
            (LOOP X))
         (T (SETQ X (*CAR N))             ; (5)
            (*RPLACA N P)
            (SETQ P N)
            (*SETMARK P T)
            (*SETTRAV P T)
            (LOOP X))))))
      (LOOP N))))
```

The *freelist* is made accessible to a LISP program through the eleventh special value slot. (*CAR *FRELIS) is the first free cell in the list. The expression (*RPLACA *FRELIS NIL) would trigger a garbage collection immediately.

Do not do anything funny with the feelist!

```
(SETQ *FRELIS
  (*NEXT (*NEXT (*NEXT (*NEXT
    (*NEXT (*NEXT (*NEXT (*NEXT
      (*NEXT (*NEXT *POOL)))))))))))
```

*ROOTLIM is the first slot in the cell pool after the special value slots holding the *GC roots*. At the same time it is the first slot of that part of the pool from which conses and atomic cells will be allocated.

```
(SETQ *ROOTLIM (*NEXT *FRELIS))
```

The GC function performs a garbage collection. First it sets the freelist to NIL, so is will not be marked “used” in the mark phase. This is redundant here, because the freelist will always be NIL when a GC is triggered, but on some systems garbage collections may be started by other events (like applying a user-visible GC function), so clearing the freelist first is good practice.

The MARK-ROOTS function then iterates over the special value slots at the bottom of the cell pool and marks all GC roots. It uses the built-in *NEXT function to proceed to the next cell in the pool and stops when reaching *ROOTLIM.

COLLECT-FREE rebuilds the freelist, as described earlier in this section.

The COLLECT-FREE function uses the same mechanism as MARK-ROOTS to iterate over all cells in the pool (except for the special value slots, which are known to be in use). It stops when it reaches the special value *LIMIT, which points to the first cell in memory that is *not* part of the pool. The *LIMIT constant has to be generated by the compiler. It is the reason why there can only be $2^{16} - 1$ cells in the pool: the last slot is reserved for *LIMIT, which cannot hold any cell, because it is not accessible by the garbage collector.

```
(SETQ GC
  (LAMBDA ()
    (LABEL
```

```
((MARK-ROOTS
  (LAMDBA (N)
    (COND ((EQ N *ROOTLIM))
          (T (MARK N)
             (MARK-ROOTS (*NEXT N))))))
 (COLLECT-FREE
   (LAMBDA (N)
     (COND ((EQ N *LIMIT))
           ((*MARKP N)
              (*SETMARK N NIL)
              (COLLECT-FREE (*NEXT N)))
           (T (*RPLACD N (*CAR *FRELIS))
              (*RPLACA *FREELIS N)
              (COLLECT-FREE (*NEXT N)))))))
(*RPLACA *FRELIS NIL)
(MARK-ROOTS *POOL)
(COLLECT-FREE *ROOTLIM)))))
```

The CONS3 function allocates a cell, sets up its type (cons or atomic), and links its car and cdr fields to the supplied objects A and D. The variable AT indicates allocation of an atomic cell.

Remember that *SETATOM is a special form that expects T or NIL as its second argument, so (*SETATOM N AT) would *not* work!

The final application of *RPLACD in the function will return N.

```
(SETQ CONS3
  (LAMBDA (A D AT)
    (COND ((EQ NIL (*CAR *FRELIS))
           (GC)
           (COND ((EQ NIL (*CAR *FRELIS)))
                 (HALT "OUT OF CELLS"))))
    (LABEL ((N (*CAR *FRELIS)))
      (COND (AT (*SETATOM N T))
            (T  (*SETATOM N NIL)))
```

```
      (*RPLACA *FRELIS (*CDR (*CAR *FRELIS))))
      (*RPLACA N A)
      (*RPLACD N D)))
```

Here is CONS from nothing!

```
(SETQ CONS
  (LAMBDA (A D)
    (CONS3 A D NIL)))
```

THE PERFORMANCE OF LISP

It did not matter much how long your LISP program took to complete when it was scheduled to run at some point of the day by an operator. Of course, if it took much longer than the estimated time to completion, which was noted in the job instructions, the operator would terminate the job and notify you of its unsuccessful execution. Job run times would mostly be a problem when there were more jobs than available slots in the schedule.

The IBM 704 operated at a cycle time of 12 microseconds [IBM1954], which corresponds to a clock frequency of 83.3 kHz. That is *kilohertz*, not MHz, or GHz! Instructions would take anywhere between two and ten cycles to complete (with most instructions taking fewer cycles), so the machine would process some 30,000 instructions per second. Some more complex jobs, like recompiling the LISP system, could easily take hours to complete.

This is why LISP used *image files* from the beginning. The LISP code of the system was not read in, parsed, and compiled each time the system came up. Instead, at the end of a each job the system would be instructed to dump an image of the core memory

of the machine to magnetic tape. This image would then contain all definitions of LISP functions (even compiled ones) and other objects that were added to the system from punch card. When the next job was submitted, the image would be read back in from tape, which was significantly faster than reading code from punch card and compiling it. All functions defined in the previous jobs would then be immediately available.

Damaging an image was a bad thing, because it meant that it had to be rebuilt, which could take a long time. Some functions, like those typed in interactively, could be lost for good. This is why there were two kinds of job: TST (test) jobs, which would never update the image, and SET jobs, which would update the image, if no errors occurred during job execution [McCarthy1960].

Of course there could be different images and even different versions of the same image. For example, there were images containing the LISP compiler or the Flexowriter interface (but not both, because they would not fit in working storage together).

When the teletypewriter and later the terminal became the predominant mode of communication with the computer, response times of the system became more visible to the programmer and hence the interest in the efficiency and effectiveness of LISP implementations grew.

Lots of *benchmark programs* were written in the 1970's and 1980's in order to measure certain performance characteristics of LISP systems. Many of these programs required numeric operations, but some of them worked fine in purely symbolic LISP. A long-lasting example of the latter category is the BOYER benchmark test [Gabriel1985], which was considered to be a "realistic" test, because it measured how well a LISP system performed the task of term rewriting, which was something that many people would typically do with LISP in those days. The BOYER program is rather complex, though, so a much simpler test shall be discussed in the following.

The LTAK program (which appears as TAKL in [Gabriel1985]) is a variant of the *Takeuchi function* [McCarthy1979] that does not use any arithmetic operations. The only arithmetic operations that appear in the original TAK function are subtraction by one and the "less-than-or-equal" comparison. When using base-1 numbers (i.e. numbers that correspond to the number of elements in a list), subtraction by one is CDR and comparison is easily implemented by traversing two lists in parallel and finding out which one ends first.

The LTAK function in the below program implements the Takeuchi function using lists as numbers and NOT-LONGER implements the "less-than-or-equal" operation. NTIMES runs (TAK 18 12 6) ten times, because execution time of a single computation of LTAK is very short on modern hardware. The variables SIX, TWELVE, and EIGHTEEN are bound to lists of the respective lengths and then passed as arguments to LTAK.

```
(LABEL
  ((SIX (QUOTE (1 2 3 4 5 6)))
   (TWELVE (APPEND SIX SIX))
   (EIGHTEEN (APPEND SIX TWELVE))
   (LTAK (LAMBDA (X Y Z)
     (COND ((NOT-LONGER X Y) Z)
           (T (LTAK (LTAK (CDR X) Y Z)
                    (LTAK (CDR Y) Z X)
                    (LTAK (CDR Z) X Y))))))
   (NOT-LONGER (LAMBDA (A B)
     (COND ((EQ NIL A))
           ((EQ NIL B) NIL)
           (T (NOT-LONGER (CDR A) (CDR B))))))
   (NTIMES (LAMBDA (N)
     (COND ((NULL N))
           (T (PRINT (LTAK EIGHTEEN TWELVE SIX))
              (NTIMES (CDR N))))))))
```

```
(NTIMES (QUOTE (1 2 3 4 5 6 7 8 9 10))))
```

LTAK basically measures the performance of function application, which is an essential part of LISP. It benefits a lot from tail call optimization.

Here are some results for the TAKL benchmark test from the 1980's (all taken from [Gabriel1985]). Times are specified for *one iteration*, not ten, as shown in LTAK the code.

The program completed in 9.87 seconds under INTERLISP on a VAX 11/780, which was a one-MIPS machine, i.e. it would process one million instructions per second. This was a fairly typical machine in the 1980's.

The TAKL program took 0.3 seconds under Portable Standard LISP (PSL) on a *Cray-1*. This was a "supercomputer" normally used for number-crunching and one of the fastest machine that you could get your hands on back in the days.

On the other end of the spectrum, the program ran for 45 seconds on a *Xerox Dolphin*, a low-end LISP machine that was considered to be so slow that it was barely useful at all [HOPL1993].

LISCMP compiles the LTAK program in four seconds (mostly due to the C compiler step and because it also has to recompile the entire runtime library) and then runs ten iterations in 1.0 seconds, giving 0.1 seconds per iteration. This is about three times as fast as PCL on a Cray-1 in the 1980's. However, this says more about the development of hardware in the past 30 years than about the capabilities of LISCMP.

See figure 16 for a comparison of LISCMP to contemporary LISP systems. For the record, all tests were performed on a 64-bit I5-M520 CPU at 750 MHz. LISCMP output was compiled with the CLANG C compiler (version 3.4.1) at optimization level one (–O1).

As can bee seen in the figure, LISCMP performs 2.5 times slower than the slowest of the other compilers, but still 2.5 times faster

than the fastest interpreter, which does not sound too bad for a compiler that barely performs any optimizations at all.

System	Time
LISINT	440.00 s
MIT SCHEME 9.2 Interpreter	9.60 s
CHICKEN SCHEME 4.13 Interpreter	5.40 s
S9FES SCHEME Interpreter 5/2018	4.40 s
ECL COMMON LISP 16.1.3, interp.	2.50 s
LISCMP	1.00 s
CHICKEN SCHEME 4.13 Compiler	0.40 s
MIT SCHEME 9.2 Compiler	0.09 s
ECL COMMON LISP 16.1.3, compiled	0.07 s

Fig. 16 – Performance of LISP systems – "Time" is time per 10 LTAK iterations

Of course, one might wonder how much the performance of the C compiler that is used as a back end contributes to the performance of LISCMP (but then, other compilers, like ECL, compile to C as well). Figure 17 summarizes the performance of LISCMP at different *optimization levels* of the C compiler.

Opt. Level	Bootstrap time total	LISCMP Run Time	Binary Size
-O3	27.9s	1.8s	416 KB
-O2	27.9s	1.8s	415 KB
-O1	5.9s	1.9s	172 KB
-O0	4.7s	3.7s	236 KB

Fig. 17 – Effect of C code optimization on LISCMP – Opt. Level is the optimization flag passed to the CLANG 3.4.1 compiler. Bootstrap Time is the total time spent compiling LISCMP (including the LISCMP pass itself). LISCMP Run Time is the time spent in the LISCMP pass, which also serves as a measure for the effect of the optimization level. Binary Size is the size of the stripped ELF binary emitted by the compiler.

The performance of LISCMP does not improve much beyond optimization level one, but time spent in the C compiler increases significantly. So does the size of the generated binary. At level zero, where only the most basic optimizations are performed, the total compilation speed is highest, but LISCMP performance drops by almost 100%. So while C code optimization does have an effect on LISCMP code quality, the output does not benefit much from the higher optimization levels offered by CLANG.

Note that LISINT needs 44 seconds for a single iteration of LTAK—as much time as the unfortunate Dolphin workstation. There are three factors that contribute to this result:

- simplicistic design of the interpreter
- deep binding strategy
- large number of garbage collections

LISINT/M does not take much longer, because macro expansion is done only once per program and there are no macros in LTAK anyway.

The dominant factor causing the bad performance of LISINT is garbage collection pressure. LISINT performs 27,467 garbage collections while computing one iteration of LTAK while the machine code generated by LISCMP performs only 68 collections. The speed difference between LISCMP and LISINT can be explained mostly by this factor alone. LISCMP is about 440 times faster than LISINT and LISINT performs

$$\frac{27,467}{68} \approx 404$$

times as many garbage collections. Ignoring garbage collections, the compiled program is still about 35 times faster than the interpreted one, but this is a reasonable ratio between compiled and interpreted code.

OPTIMIZING LISCMP

LISCMP originally took almost 10 seconds to self-compile, excluding the C compiler step. The reader was quickly identified as the bottleneck of the process. More specifically, it was the *interning* of symbols that took most of the time. Each character that is read by READC is being interned, so the INTERN function is called very frequently. In the first implementation, it used a flat list to store symbols:

```
(SETQ INTERN
  (LAMBDA (X)
    (LABEL
      ((FIND (LAMBDA (SYML)
         (COND ((EQ NIL SYML) NIL)
               ((SAMENAMEP (*CAR X)
                           (*CAR (CAR SYML)))
                 (CAR SYML))
               (T (FIND (CDR SYML)))))))
      (LABEL ((Y (FIND (CAR *SYMLIS))))
        (COND (Y)
              (T (*RPLACA *SYMLIS
                          (CONS X (CAR *SYMLIS)))
                 (CAAR *SYMLIS)))))))
```

This approach is simple and workable, but has a time complexity of $O(\frac{1}{2}n^2)$ (see also page 57), which means that interning the about 1,300 symbols in the LISP and LISCMP source code takes around 850,000 comparisons. Replacing the above algorithm with one using a *bucket list* reduces the constant factor of the complexity from $\frac{1}{2}$ to about $\frac{1}{54}$ (assuming 27 buckets for the alphabetic characters plus the asterisk). This should get the

number of comparisons for inserting all 1,300 symbols down to the region around 31,000. There are a lot of vague statements in this paragraph, so how does this translate to compiler run times?

There are two places in the code where the INTERN algorithm is being used: (1) the LISP system itself, where it interns symbols read by READ, generated by IMPLODE and EXPLODE, and characters read by READC and PEEKC, and (2) the compiler, where it is used to build the symbol list of the initial cell pool image.

Replacing the INTERN algorithm in the compiler alone reduces the run time of self-compilation by about 20%. Replacing it in the compiler and in the LISP runtime library itself reduces the run time by 60%, bringing it down to about 4 seconds.

Another interesting observation is that PEEKC is actually faster when implemented in LISP (see page 60), because it can cache the most recently read symbol. You could not do this with a peekc() function written in C, because it would have to call the INTERN function, which is written in LISP, and in the present implementation of LISCMP this is not possible.

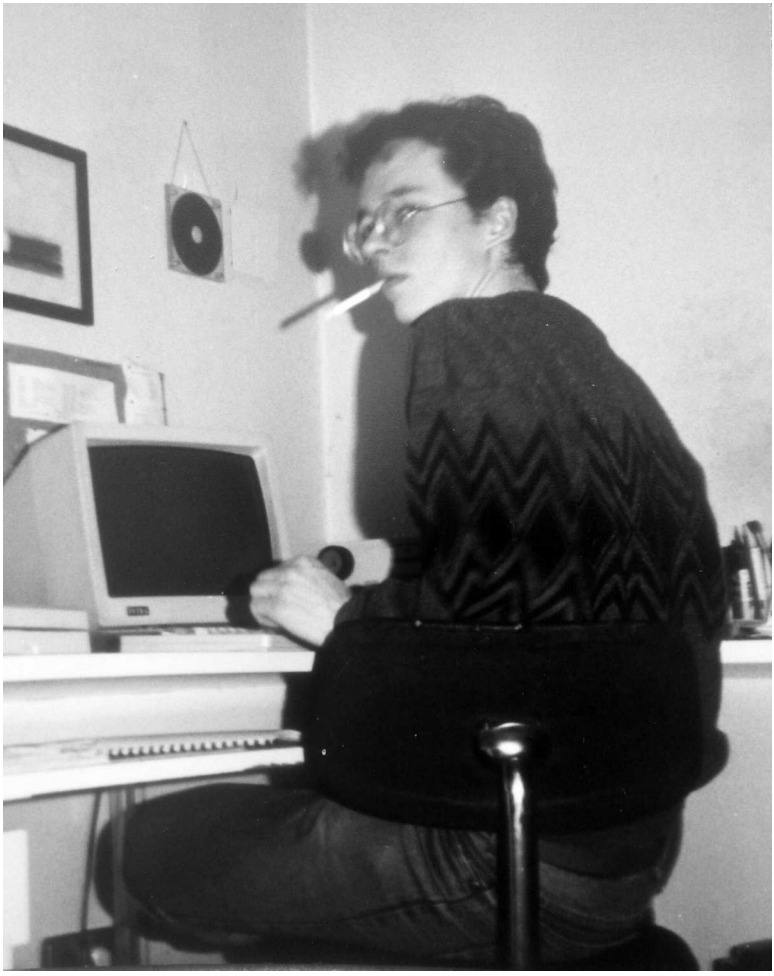
When implementing PEEKC at LISP level, a symbol cached by PEEKC is not interned again when peeked once more or finally consumed by READC. Thus the number of applications of INTERN is reduced significantly. This hack alone decreases the run time of LISCMP by another 50%, bringing it down to the final 2 seconds.

Interning symbols was a bottleneck in many early LISP systems, and many systems chose exactly the route taken by LISCMP: A flat list was fine as long as programs were small, but soon the READ function started to dominate the run times of programs. The bucket list was a quick fix that bought some time, but bringing down the constant factor of an $O(n^2)$ algorithm still leaves complexity quadratic, so in the long run READ will start to dominate run times again.

This is where the story of LISCMP ends. The largest program that it is currently compiling is itself, and the bucket list is a reasonable solution for programs in this size range.

LISP evolved far past this size of programs, though, and eventually the OBLIST (symbols were stored in the OBLIST in most LISP systems) was replaced by a more advanced data structure, like a hash table. To reflect this change the symbol was renamed OBARRAY. This change took place at some time between 1967 and 1974, when transitioning from PDP-6 LISP (a.k.a LISP 1.6) [MIT1967] to MACLISP [Moon1974].

HACKING



IN THE GOLDEN AGE

Photograph on previous page: Hacker in his natural habitat, 1980's.

COMPUTER SYSTEMS evolved quickly in the early days of LISP. While the design of the IBM 704 was based on vacuum tubes, a slightly later design, the IBM 7090, was based on discrete transistors. LISP 1.5 was implemented on a 7090.

The 7090 had a cycle time of 2.18 microseconds [IBM1959], giving a clock frequency of about 450 kHz. With most instructions completing in two cycles, the machine could process about 225,000 instructions per second. In modern-day jargon, you would call it a 0.2-MIPS machine.

Like the 704, the IBM 7090 was a 36-bit machine and like the 704 it had an address space of 15 bits, so it could address 32,768 words of core memory. Therefore a complete cons cell, with tag bits and all, could still be stored in one machine word. For all practical purposes, the 7090 was a really fast 704.

However, *memory* has always been a central consideration when choosing computers for LISP systems, and LISP outgrew the selected hardware repeatedly during its history. The 15-bit address space of the early IBM machines was tight to begin with. Even in LISP 1, individual features of the system had to be selected for loading, because storage was limited [McCarthy1960].

The next iterations of LISP systems, LISP 1.6 (a.k.a. PDP-6 LISP) and MACLISP, were implemented on the Digital Equipment Corporation (DEC) PDP-6 and PDP-10 computers. These machines were good choices for implementing LISP, because they had a 36-bit machine word size, but also an address space of 18 bits, meaning that they could address as much as 256K words of magnetic core storage while a cons cell would still fit in a single machine word.

Of course 2×18 bits left no space for tag bits, so a different approach had to be chosen for storing meta information. Most of the tag bits were used for *typing*, i.e. knowing if a given machine word stored a cons cell or an atom (and which kind of atom). One way to solve this problem was to store different kinds of cells in

different memory regions, so that some bits of the address of a cell could serve as a type indicator. This approach was call BIBOP ("big pag of pages") due to the paged memory used in the PDP-10 architecture [Steele1977]. The "big bag" was a region in the middle of the memory pool of LISP. Memory was allocated from the top and from the bottom of the pool, leaving a big bag of not-yet-allocated pages in its middle.

The instruction timing of the KA10 processor of early PDP-10 machines was highly diverse, and timing was specified per instruction rather than giving a general cycle length. A core memory read-write cycle was one microsecond, though, and most instructions took around two microseconds to complete [DEC1968], so an average clock frequency of about 500 kHz can be estimated, giving a speed of about 0.25 MIPS. So the speed improvement over the 7090 may not have been huge, but the eight times larger address space opened lots of new possibilities and reduced garbage collection pressure on existing programs.

Later models of the PDP-10 were one-MIPS machines, sometimes also called one-MIPS, one-megabyte machines [HOPL1993], because 256K words were roughly equal to one megabyte. The PDP-10 was very popular for running LISP until the memory barrier was reached again, in the late 1970's

Some artifacts from the 36-bit era have survived in LISP to this day. For example the, names CAR and CDR stem from the subroutines that extracted the car and cdr part of a cons cell from a register on the IBM machines. Instructions stored in 36-bit registers contained two fields called "address" and "decrement", so it was natural to use these fields to store pointers to car and cdr fields in LISP cons cells. The subroutines for extracting the fields were named CAR ("content of address field of register") and CDR ("content of decrement field of register").

These names may have remained popular, because they easily compose to names for accessing elements of nested lists, like

CAAR, CADDR, etc., but then other names remained popular, too, so maybe names just stuck once they had been introduced. Some of the early function names in LISP probably had their roots in the fact that the IBM and DEC machines of that age used *sixbit encodings*, so six characters would fit in a single machine word. Therefore only one machine word was needed to store names like RPLACA, RPLACD, SUBLIS, NCONC, etc.

The predominant mode of communication with the computer in the 1970's was the *hard-copy terminal* in the beginning, but it was quickly replaced by "glass TTYs", then cursor-addressable *terminals* like the LSI ADM-3A, and finally by more sophisticated terminals like the DEC VT100, which supported scrolling regions, underlining, boldface and reverse characters, etc.

Input and output were transmitted character by character between terminal and computer, and visually deleting characters from the input was supported on screen-based terminals. The LISP *reader* (READ) still terminated as soon as it found a delimiting character after an atom or a final closing parenthesis in a list. Hence a typical session on early LISP looked like this (again, system output is underlined for clarification):

```
*(SETQ FOO (QUOTE BAR))
BAR *FOO
BAR *(CONS FOO (QUOTE BAZ))
(BAR . BAZ) *(TIMES 5 7)
43 *
```

The MACLISP prompt character was a single asterisk. Evaluation of an expression started as soon as a delimiter was found and PRINT then advanced to a new line and printed the result, followed by another prompt character. Note that $5 \times 7 = 43$ is not a mistake, because the default numeric base on MACLISP was octal.

Printing the value of an expression on the same line as the prompt character was a technique to save paper on hard-copy terminals, but it stuck for a while even when glass terminals became the standard mode of communication. Later versions of LISP used a more screen-oriented mode of interaction.

The popularity of the PDP-10 was not just due to its technical nature, although there were lots of reasons to love it: a beautiful, highly orthogonal instruction set, a very high-level macro assembly language, and an overall architecture that looked like it was *designed* to implement LISP.

What was maybe as important, though, was that the PDP-10 was a hands-on machine. The IBM mainframes were stored away in a remote place and operated by designated personnel. Often they were rented rather than bought, and when the hardware malfunctioned, a technician in a suit would appear, go to the remote place, and fix it.

Things were different with the PDP-10, which was typically operated by the people who used it. Repairing, modifying, connecting, and rewiring the machines was done by self-taught people, sometimes in collaboration with technicians from DEC. The “computer” was no longer some far-away abstract machine, it was something you could touch, alter, and fix—and more than often you had to.

The DEC PDP-10 was a small computer when compared to the IBM 704 or 7090, but it was still a bulky and highly complex machine. Its complexity was pretty obvious, too. It was not hidden away in neatly arranged arrays of small microchips, it was exposed on boards full of integrated circuits, transistors, resistors, capacitors, potentiometers, etc. “Memory” was still stored in racks, but a rack was just the size of a large refrigerator and could store up to 256K words of magnetic core. Up to 16 of those racks could be connected to a single PDP-10, but the capacity of the early models was limited to 256K words due to the 18-bit address space.

Hard disks were the size of a washing machine and had 14-inch platters. Their capacity varied from a few million words to several hundred million words. The popular DEC RP06 drive could store 178 million machine words, corresponding to 810 megabytes, which was a decent size for a multi-user system of those days. The platters of some disk subsystems were organized in removable cartridges (called “packs”), so they could be removed, stored, or transferred to different machines.

The architecture of the PDP-10, like all DEC machines of that time, was well documented, and documentation either shipped with the machine or was readily available upon request. The PDP-10 was a very hackable machine in every imaginable way.

In the late 1970’s, though, memory grew tight once again. LISP hit the 256K-word barrier, and this time there was no obvious solution. Digital Equipment created a new PDP-10 architecture with a 30-bit memory bus that offered up to 32 “sections” of “local” 18-bit addresses. The new model could address up to 8,192K words of memory, but a cons cell would no longer fit in a single machine word, so the entire memory management part of the LISP system had to be rewritten. Although it solved the problem of memory shortage, this change was not well received.

DEC finally discontinued the PDP-10 and established its new flagship product, the VAX. Originally intended as a “virtual address extension” for the 16-bit PDP-11 minicomputer, the VAX quickly evolved into a whole family of new machines. Unfortunately, none of them were really suitable for running LISP. The VAX had a 32-bit address space, variable instruction length, and a very complex instruction set. For instance, it had an *instruction* for evaluating polynomial equations! [DEC1987] Although the VAX had a large address space, it retained the 512-byte page size of the PDP-10, which could cause problems for large monolithic programs, which LISP systems usually were.

Due to the small page size, there could be up to 2^{23} (eight million) page table entries. Of course no machine had 4,096 megabytes of memory back then, but even in smaller systems chunks of memory could have so many page table entires that the VAX started to page out parts of the page table and then the system basically stood still. The hard disks were busy with paging, but the program barely ran at all. This only happened when one single program used lots of memory, but this was what LISP typically did.

Even in the 1970's, when the PDP-10 was still an adequate machine for LISP programming, there were already plans to build special hardware for LISP. The *LISP machine* was a general purpose computer with special microprogramming for effective evaluation of LISP programs. Its word size was typically large enough to contain a pointer and some corresponding tag bits and it ranged from 24 to 40 bits.

Technically, the LISP machine had a few interesting features:

- parallel type checking
- concurrent / incremental garbage collection
- stack caching for faster function calls
- CDR coding for compact list representation
- bitmap displays and graphical user interfaces

The most interesting feature was, of course, that the machine ran LISP as its operating system and programming language, and even the most low-level parts of it were written in LISP.

Parallel *type checking* removed the dilemma of fast versus type-safe code. On conventional hardware the application of a built-in LISP function, like CAR, either had to make sure that its argument had the proper type or it had to rely on the proper type being supplied. The former variant was slow, because type checking had to be performed before the operation could be carried out, and the

latter was not safe. Parallel type checking did both in parallel and generated an exception afterwards when a type error occurred.

Concurrent garbage collection runs in parallel with the program that allocates from the memory pool, so there are no breaks while the garbage collector does its work. Incremental GC means that the pool is divided into "generations" of objects of different ages, and "minor" collections on young generations are performed first. A major collection is only triggered when all young generations run low on space. A major collection could still cause delays, but it happened infrequently.

Caching the top elements of the stack in fast memory increased the performance of languages that do a lot of function calls, like LISP does.

CDR coding was a technique where a list was represented by just placing its elements in consecutive storage locations. When the cdr field in a list was modified with RPLACD, an "invisible pointer" was stored in the cdr field. It would point to some distant location, but behave like an ordinary machine word. That is, reading the machine word containing the invisible pointer actually read the machine word pointed to and returned the value of that word. The indirection was performed at hardware level and was invisible to the programmer.

The greatest novelty of the LISP machine, however, was that it was a *single-user machine*. Computational resources were not shared among users, like on the later IBM mainframes or DEC mini computers. The user had the entire machine at their disposal and communicated with it through a graphical user interface not unlike those that are used these days—although some people would argue that the user interface of the LISP machines was way ahead of everything that is in use today. This is not an improbable claim given the tight integration of the hardware, the development tools, and the programming language.

LISP machines were very popular in the late 1970's and through the 1980's, and they are highly valued by many LISP programmers to this day. For details about LISP machine architecture see, for example, [WR1990]; for a bird's eye view of of the development of LISP machines see [Phillips1999] or [HOPL1993].

The LISP language itself evolved substantially in this era, culminating in dialects like LISP Machine LISP (later ZETALISP), INTERLISP, and SPICE LISP. Each of these languages had its own dedicated hardware, but for some of them ports to conventional computers existed. This time marked the climax of the diversification of LISP, which then came to an abrupt end with the effort to create one common language that would include the most essential features of all of the above plus some more. Thus, in the 1980's, COMMON LISP was born.

The development of computer hardware continued at a swift pace. In the 1980's computer *workstations*, which previously had been considered to be toys rather than "serious" computers, became sufficiently powerful to compete with mini computers like the VAX. LISP was ported to workstations, which were much cheaper than the LISP machines, but started to offer comparable performance. At the same time, the integration level of hardware increased, and the development of custom hardware became more expensive. LISP machines were no longer a sustainable business model.

The *Personal Computer* also appeared in the early 1980's, but did not become a serious alternative to the workstation until the 1990's. When it did, it quickly started to replace the workstation, though, and LISP became just one programming language among many. Because PCs were low on memory and had insufficient processor speeds in the beginning, LISP only filled a small niche in the ecosystem of PC software tools—a rôle that it retained in the years to come.

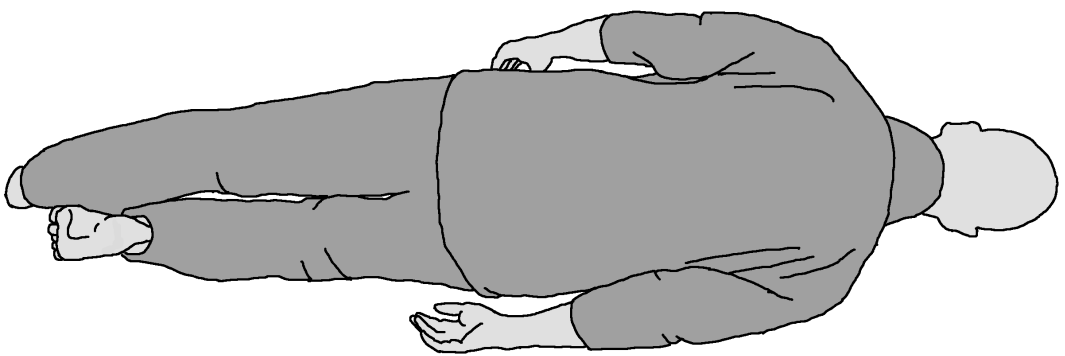
The diversity of the LISP family was gone. Only COMMON LISP was left and then there were a few more or less wide-spread niche

dialects, like SCHEME for teaching and programming language theory and AutoLISP for programming computer aided design software. All other dialects of LISP soon vanished in obscurity.

However, 60 years after its conception LISP still attracts people with a creative mind and scientific inclination. Even today, many people develop their own, often more or less half-baked, dialect of LISP as a college project, out of curiosity, or for all kinds of practical purposes. Interesting new features are invented, discovered, explored, and implemented. Although most of these projects never develop into something that will be perceived by a broader audience, there still is a vibrancy about the community of LISP users that is not found in many other places.

The golden days of LISP are long gone, but its spirit is still alive.

I’LL SEE MYSELF OUT!



THE LISP compiler and interpreter presented in this book are quite unpolished and minimal, but with a little bit of effort, they can be improved considerably. Here is some inspiration!

WARM-UP

The MAPCAR and MAPCAR2 functions (page 49) really should be one function (MAPCAR) that finds out the number of arguments passed to it and dispatch evaluation accordingly. Rewrite MAPCAR as an LEXPR so it can do this.

Writing a truly variable-argument MAPCAR as in COMMON LISP is a different thing though. You will need an APPLY special form that can apply a function to a single list containing its arguments, e.g.:

```
(APPLY CONS (QUOTE (FOO (BAR))))  ==>  (FOO BAR)
```

Adding APPLY to the compiler is not that hard: the second argument of APPLY is evaluated and pushed to the stack to be used as an argument list, then the first argument of APPLY is evaluated, leaving its value in REF(EXPR), and finally apply() is called.

EASY

Error checking and reporting is a point where the shortcomings of the system are most obvious. For instance, forms like

```
(COND FOO)
(LABEL BAR BAZ)
```

should generate more helpful error messages than

```
*** CAR: EXPECTED LIST: FOO
```

This is easily fixed by adding a few clauses to CONDCOM and LABCOM, but then there are quite a few other places where the compiler just runs into errors and aborts. Locating them all and

adding clauses to guard them is a tedious task, but it would improve the usability of the compiler and interpreter a lot.

Making *HALT (and HALT) two-argument functions and allowing them to report the offending part of an S-expression would also make the life of the programmer easier.

EXCEPTION HANDLING

At the moment the LISINT interpreter simply exits when you type something funny. A simple *exception handling* mechanism could improve that situation a lot. A CATCH/THROW form would be easy to implement and could be used to recover after encountering an error in the interpreter. Basically CATCH and THROW work like this:

```
(SETQ BAR
  (LAMBDA ()
    (THROW EXCEPTION
           (QUOTE EXCEPTION-VALUE))
    NEVER-REACHED))

(CATCH EXCEPTION
  SOMETHING
  (BAR)
  SOMETHING-ELSE)   ==>  EXCEPTION-VALUE
```

The program would set up a catch frame for the exception EXCEPTION and evaluate SOMETHING in it. It would then apply BAR, which would throw the exception, thereby returning to the *end* of the catch frame and make it return EXCEPTION-VALUE. The expressions NEVER-REACHED and SOMETHING-ELSE would never be evaluated.

A *catch frame* is set up by generating an exit label and pushing the exception symbol, the label, and the current stack to another stack (the exception stack). When an exception is thrown, the symbol is searched on the exception stack and all frames that do not match

the symbol are discarded. The matching frame is also removed, the reference to the runtime stack contained in it is restored, and a jump to the exit label is performed. When no catch frame has the requested symbol, an error is reported and the system shuts down.

Figuring out the details of the implementation is left as a fun exercise to the reader.

While discussing exception handling, it would make sense to intercept keyboard interrupts as well, so that sending SIGINT would not just terminate LISINT. This can be done via exception handling or by just adding a special slot that contains a “keyboard interrupt flag” that indicates whether SIGINT has been sent to the LISINT process.

NOT-SO-EASY

The LISP interpreters shown in this book all use *deep-binding*, i.e. they look up variable symbols in an association list, which is a linear-time operation. Interpretation could probably be sped up by some significant factor by implementing *shallow-binding*, where each symbol links directly to its value.

However, just using the value fields of existing symbols in the interpreter will not work. The following example will illustrate the problem: in LISCMP the symbol MAPCAR is bound to a function atom that points to the compiled function MAPCAR. When defining a MAPCAR function in LISINT (see page 161), a symbolic expression would be bound to the value field of the *same* symbol MAPCAR, which means that it could no longer be used by the LISCMP binary implementing LISINT. But LISINT does use MAPCAR to implement LEXPRs in the BIND function (page 152). Of course this problem affects *all* atoms shared between LISINT and LISCMP, and not just MAPCAR.

So a completely distinct set of symbols would have to be used in the interpreter. There are several ways to implement this. For

example, you could write an *INTERN function that is like INTERN, but accepts a symbol list as an additional argument and interns the given symbol in the given symbol list. This would certainly be the most efficient approach.

A different possibility would be to create an own atom type for the interpreter, something like

```
(*ATOM-TAG* NAME VALUE)
```

where *ATOM-TAG* is a unique S-expression in the sense of EQ. You can create it using

```
(SETQ *ATOM-TAG* (LIST (QUOTE *ATOM-TAG*)))
```

because all cons cells have a unique address in the cell pool.

No matter how the separate set of symbols is implemented, though, there is one even harder problem: the interpreter will have to handle two sets of symbols and use the right set in the right context.

For instance, you could implement your own READ function in LISINT and make it generate references to your own set of symbols. This creates another problem, though: for instance, it would generate a reference to your own CONS symbol when it reads the word CONS, while LISCMP compares atoms to its own (distinct) version of the CONS symbol so

```
(EQ X (QUOTE CONS))
```

would evaluate to NIL at the level of the LISCMP binary, even if the value of X is your own CONS symbol.

So *some* symbols have to be shared between the compiler and the interpreter. Which ones? How can this be implemented?

In order to avoid the whole mess you could also implement property lists of symbols [McCarthy1960] and just store the values bound in the interpreter under a separate property. This would

mean to rewrite the entire binding mechanism of LISCMP, but it might very well be the easiest solution.

GARBAGE COLLECTION

A *garbage collector* in LISP has a lot of advantages. For instance, it can be fine-tuned or even replaced with another implementation in LISP. In order to make the garbage collector from the science chapter work, some changes have to be made to the LISP system.

First of all, a separate *runtime stack* has to be set up so that function calls will no longer cause any allocation of cons cells. This can be done, for instance, by pre-allocating the top 1000 cells of the cell pool and exclude them from the garbage collection (by moving *LIMIT), so that they will never appear in the freelist.

The stack cells in the pool can then be treated as a vector and a stack pointer register can be set up to point to the topmost element allocated on the stack. Because allocation and deallocation would then be a simple move of the stack pointer, no *consing* would happen during function application—no cons cells need to be allocated.

Of course the *call frame* format also would have to change so that it puts individual arguments directly on the stack rather than collecting them in a list (which would again cause consing). LEXPRs will cause some trouble here, but maybe consing can be tolerated in their case.

Eventually, the LISP garbage collector needs to be *bootstrapped* as well. When the LISP system is first bootstrapped, there is no LISP-level GC function, so the one in the low-level runtime library has to be used. So there needs to be some kind of hook that allows to switch to the LISP implementation later.

There are a lot of nitty-gritty details to be addressed here. Tail call optimization, for instance, will be much more complex. This is definitely not an easy project.

MORE EFFICIENT INTERPRETATION

The LISP interpreters discussed here are very simplicistic and use an ad-hoc scheme for interpretation. Maybe the performance of the interpreters could be improved by using a proper *abstract machine*, like the SECD machine [Landin1964].

A simple abstract machine can be built on top of the LISP language discussed in this book, but would it be more efficient that the ad-hoc approach? What else can be done to reduce the overhead caused by the large number of COND clauses in the EVAL functions of the interpreters?

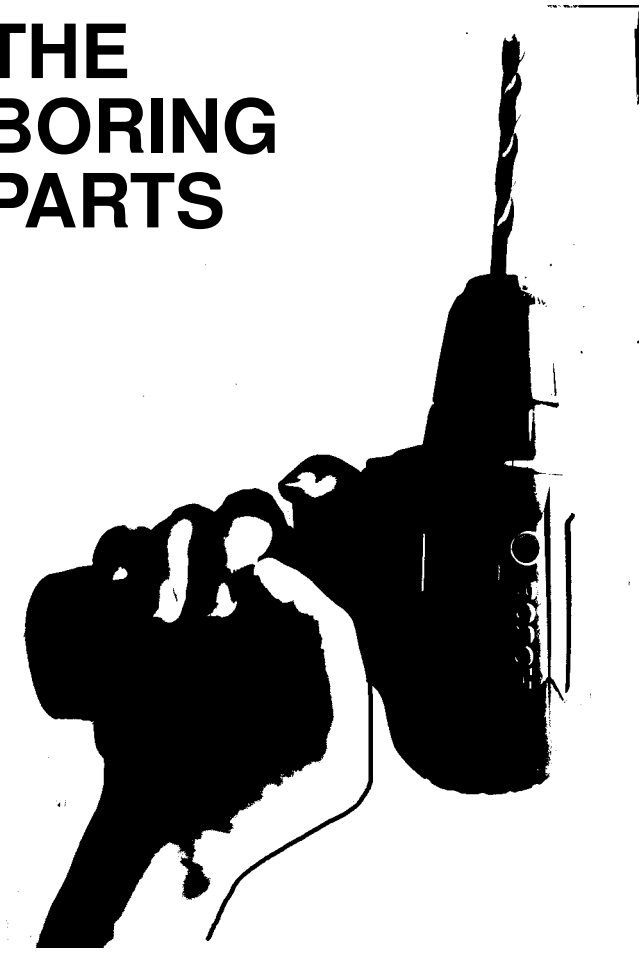
WAY BEYOND

Finally, it might be a fun exercise to explore a minimal LISP that can be used for bootstrapping an interactive *compiler*.

What does a LISP language have to provide in order to bootstrap a version of LISINT that compiles *directly* to native code? You will most probably need fixnums and byte vectors, but

What exactly is the minimal interactive LISP
that can bootstrap to binary code?

THE BORING PARTS



A LISCMP MICRO MANUAL

SYMBOLIC EXPRESSIONS (S-EXPRESSIONS)

An **atom** or **symbol** is any combination of these characters.

A B C D E F G H I J K L N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 0 – *

An **S-expression** is either an atom or a list.

A **list** is a set of S-Expressions that is enclosed in parentheses. The empty list () is also written NIL. Lists can contain lists.

```
(FOO BAR)
(FOO (BAR) BAZ)
(LAMBDA (X) (LIST X))
()
```

COMMENTS

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

```
; These are comments.
; Any characters are
; allowed here.
; Comments are ignored
; by the LISP system.
```

UNREADABLE FORMS

A form that contains characters other than those listed in the previous sections are **unreadable**, The LISP system will report an error when encountering them.

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

EXPRESSIONS

An **expression** is an S-expression with a meaning.

X $\rightarrow$ 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 functions, function application, and special forms.

VARIABLES

A **variable** is a symbol that is bound to an S-expression.
Variables evaluate to the S-expression to which they are bound. The S-expression to which a variable is bound is called the **value** of the variable. Variables are created by the SETQ special form or by function application.

VARIABLE → VALUE

References to unbound symbols (symbols that are not bound to any value) are undefined.

UNBOUND → *UNDEFINED

A **constant** is a variable that is bound to its own symbol.

CONSTANT → CONSTANT

OBJECTS

An **object** is a **quoted** S-expression. An S-expression is quoted by putting it in the place of the x in the expression (QUOTE x).

(QUOTE FOO) → FOO
(QUOTE (FOO BAR))
→ (FOO BAR)
(QUOTE (QUOTE X))
→ (QUOTE X)

Quotation can be abbreviated with an apostrophe:
'x = (QUOTE x).

'FOO → FOO
'(1 2 3) → (1 2 3)

The objects T and NIL do not have to be quoted.

NIL → NIL
T → T

NIL represents both logical falsity and the empty list. T denotes truth, but any value except for NIL is regarded as "true".

ELEMENTARY FUNCTIONS

The expression (F X) means "F applied to X" and (F X Y) means "F applied to X and Y", i.e. they are applications of functions to values. Here are some elementary functions:

(ATOM X) Return T, if X is an atom, else return NIL. Examples:
(ATOM ′ X) → T
(ATOM T) →T
(ATOM NIL) → T
(ATOM ′ (A)) → NIL

(CONS X Y) Create a new list from the S-expressions X and Y. Examples:
(CONS ′ A ′ (B C)) → (A B C)
(CONS ′ (A) ′ (B)) → ((A) B)
(CONS ′ A NIL) → (A)

(CAR X) Return the head / first element / car part of a list X. Examples:
(CAR ′ (A B C)) → A
(CAR ′ (A)) → A
(CAR NIL) → NIL
(CAR ′ A) → *UNDEFINED

(CDR X) Return the tail / rest / cdr part of a list X. Examples:
(CDR ′ (A B C)) → (B C)
(CDR ′ (A)) → NIL
(CDR NIL) → NIL
(CDR ′ A) → *UNDEFINED

(EQ X Y) Return T, if X and Y are the same object. Examples:
(EQ ′ FOO ′ FOO) → T
(EQ ′ FOO ′ BAR) → NIL
(EQ T T) → T
(EQ NIL NIL) → T

(EQ ′A ′(A)) → NIL
(EQ ′(A) ′(B)) → NIL
(EQ ′(A) ′(A)) → *UNDEFINED

LAMBDA FUNCTIONS

A **lambda function** is a function of zero, one, or multiple variables that computes a value. It has the general form:

(LAMBDA (X) (CONS X X))
(LAMBDA (X Y) Y)

(LAMBDA <variables> <body>)

where <variables> is a list of symbols and the <body> is at least one expression. The value of the last expression is the value of the function. For example,

(LAMBDA (X Y) (CAR X) (CDR Y))

is a function of the variables X and Y that first computes (CAR X) (and discards its value) and then computes (CDR Y) and returns its value.

(LAMBDA () (QUOTE FOO))

is a function of no variables that always returns FOO.

FUNCTION APPLICATION

An S-expression of the form $(F\ A_1 \cdots A_n)$ is a **function application**.

(CONS ′FOO ′BAR)
((LAMBDA (X) X) ′FOO)
(F ′1 ′2 ′3)

It applies the function F to the **arguments** $A_1 \cdots A_n$ and evaluates to a value. Each argument is an expression.

The following steps are involved in an application:

(1) the function part (F) is evaluated (hence functions can be named by SETQ; see below)

(2) each argument (A_i) in the application is evaluated

(3) each argument is **bound** to the corresponding variable; a variable that is bound to a value will evaluate to that value; variables are paired with arguments by position

(4) with the bindings in effect the body of the function is evaluated

(5) the variables are unbound from the arguments, thereby restoring their previous bindings

(6) the value of the body is returned

Here is an example. Boldface text indicates objects, i.e. **(X)** is the list containing X and not the application of the function X.

```
((LAMBDA (X Y) (CONS X Y))      ; evaluate arguments:
 (QUOTE 1)                      ; (QUOTE 1)   ==> 1
 (QUOTE (2)))                   ; (QUOTE (2)) ==> (2)

((LAMBDA (X Y) (CONS X Y))      ; bind X to 1
 1 (2))                         ; and  Y to (2)

(CONS X Y)  =  (CONS 1 (2))     ; evaluate the body

                                ; unbind X and Y

→ (1 2)                         ; return the result
```

SPECIAL FORMS

(QUOTE <object>)	Return the given object.
(LAMBDA <vars> <body>)	Return a function with the given given variables and expressions (<body>).
(PROGN <expr> ...)	Evaluate the given expressions in sequence and return the value of the last one. Example: (PROGN (PRINT ′ HELLO-) (PRINT ′ WORLD) NIL)

(SETQ <var> <expr>)

Bind the given variable to the value of the given expression. Example:

```
(SETQ DOUBLE
   (LAMBDA (X)
      (CONS X (CONS X NIL))))
```

(COND
(<p1> <body1>)
...
(<pN> <bodyN>))

Evaluate expression <p1> (predicate 1). When its value is not NIL, evaluate the expressions in <body1> and return its value, else proceed with the next predicate. (<p> <body>) is called a **clause** of COND. When no clause has a true (non-NIL) predicate, return NIL. When a clause has the form (<p>), return the value of the predicate, if it is not NIL. Example:

```
(LAMBDA (X)
   (COND
      ((ATOM X)
         'AN-ATOM)
      ((EQ NIL (CDR X))
         'A-SINGLETON)
      (T 'A-LIST)))
```

(LABEL
((<v1> <a1>)
...
(<vN> <aN>))
<body>)

Evaluate <a1> and bind it to the variable <v1>. With that binding in effect, evaluate the next argument <a> and bind it to the corresponding <v>, etc. When all bindings are processed, evaluate <body>, unbind all variables, and return the value of <body>. Example:

```
(LABEL
   ((X (QUOTE (2)))
    (Y (CONS (QUOTE 1) X))
    (F (LAMBDA (Z) (CONS Z NIL))))
   (F Y))
```

LEXPRS

An LEXPR is a lambda function that binds its entire list of actual arguments to a single variable. It has a single variable in the place of a list of variables.

```
(LAMBDA X X)

((LAMBDA X X) ′1 ′2 ′3)
  → (1 2 3)
```

DOTTED PAIRS

Connecting two atoms with CONS gives a **dotted pair**.

```
(CONS ′A ′B)  → (A . B)
```

A **proper list** is a special case of the dotted pair where the cdr part of the rightmost pair has a value of NIL.

```
(A . NIL)       = (A)
(A . (B . NIL)) = (A B)
```

A **dotted list** has a non-NIL atom in the cdr part of the last pair.

```
(A . (B . C)) = (A B . C)
```

INTERNAL FUNCTIONS AND VARIABLES

These functions are used internally to implement the LISP system. Do not use them in non-system programs! Instead of *RPLACA and *RPLACD use RPLACA and RPLACD. Instead of *READC and *WRITEC use READC and WRITEC or READ and PRINT.

(*CAR X)	Return car of X, regardless of type of X
(*CDR X)	Return cdr of X, regardless of type of X
(*RPLACA X Y)	Replace car part of cons/atom X with Y
(*RPLACD X Y)	Replace cdr part of cons/atom X with Y
*POOL	The address of the first cell in the pool
*LIMIT	The first address outside of cell pool
(*NEXT X)	Return address of cell after X in pool

(*SETATOM X T/NIL)	Set/clear atom tag of cell
(*SETMARK X T/NIL)	Set/clear mark tag of cell
(*SETTRAV X T/NIL)	Set/clear trav tag of cell
(*ATOMP X)	Return atom tag of X (T/NIL)
(*MARKP X)	Return mark tag of X (T/NIL)
(*TRAVP X)	Return trav tag of X (T/NIL)
(*READC)	Read char, return as atom name
(*WRITEC X)	Write first char of atom or atom name
(*DUMP X)	Write cell pool image to file
(*LOAD X)	Load cell pool image from file
(*HALT X)	Print atom X, then terminate program

An **atom name** is the name part of an atom. It can be extracted from an atom using *CAR. An atom name N can be turned into an atom using (CONS N NIL).

MORE FUNCTIONS

For definitions of derived functions of the LISP system, see the LISP system source code in the file LISP.LISP. These functions are available:

APPEND	LIST	PEEKC	RPLACA
CAAR...CDDDR	MAKESYM	PRINT	RPLACD
EQUAL	MAPCAR	READ	RREDUCE
EXPLODE	MAPCAR2	READC	SAMENAMEP
HALT	MEMBER	RECONC	SYMBOLIC
IMPLODE	NCONC	REDUCE	TERPRI
INTERN	NREVERSE	REVERSE	WRITEC

GLOSSARY

==>
See *evaluation*.

anonymous function
A *function* that has no name. See *lambda function*.

application
By applying a *function* to zero, one, or multiple *arguments*, the function is caused to map the arguments to a *value*. Function application in LISP has the general form $(f\ a_1 \cdots)$, where f is a function and each a is an argument.

argument
A *value* that is being passed to a *function*.

argument list
A list of *values* to be passed to a *function*. Sometimes also called an "actual argument list". Not to be confused with the "formal argument list", which is the list of *variables* of a function. In this text, "argument list" always denotes an actual argument list, while a formal argument list is called a "list of variables".

association list
A list of *pairs* (or two-element lists in LISP 1), where the car part of each pair serves as a key and the cdr part (or cadr part) is the associated value. The general form of an association list is $((key_1\ .\ value_1) \cdots (key_N\ .\ value_N))$

atom
A data object comprised of a sequence of individual alphanumeric (and certain special) characters. Atoms that look the same are identical (see *identity*). An atom that is bound to a *value* is a *variable* (see also: *binding*). The atom is one of the principal data types of symbolic LISP, the other one being the *list*. To obtain an atom in a program, it has to be quoted (see *quotation*).

binding
The mechanism that associates an *atom* with a *value*. When an atom is bound to a specific value, it evaluates to that value (see *evaluation*). An atom that is bound to a value can be used as a *variable*. Bindings are created by *special forms* like LAMBDA, LABEL, and SETQ.

body
Some *special forms* can evaluate multiple *expressions* in sequence (like PROG, COND, or LAMBDA). These expressions comprise the body of the form.

CAR
The *function* extracting the "car part" of a *pair*.

CDR
(Pronounced "kudder".) The *function* extracting the "cdr part" of a *pair*.

clause
An argument of the COND *special form*. Each clause of COND has the general form $(p\ x_1 \cdots)$, where p is a *predicate* and each x is an *expression*. The sequence of expressions in its entirety (but excluding the predicate) forms the *body* of the clause. The term "clause" is sometimes also used for parts of other special forms.

closure
A *function* with an *environment* attached. The environment contains the *bindings* of the *free variables* of the function. When the closure is applied, the *values* of its free variables are looked up in the attached environment. In modern LISP a closure is the same as a function.

COMMON LISP
A huge dialect of LISP that was developed by a committee whose members represented the largest LISP vendors of the 1980's. It was first described in 1984 and subsequently standardized by the American National Standards Institute (ANSI) in 1994. It is one of

the two major dialects of LISP today, the other one being *SCHEME*.

compilation
The process of translating a program from human-readable form to a form that is suitable for execution by a computer.

cons cell
The data object implementing the *pair*.

consequent
The *body* of a *clause* of the COND *special form*.

deep binding
A strategy for implementing *bindings* where the names of *variables* are looked up in an *association list*. Deep binding is very easy to implement, but also very inefficient. (Cf. *shallow binding*.)

dotted list
A *list* whose last (rightmost) element is not *NIL*. E.g.: (A B C . D) or (A . B). Called so because of the dot that separates the last element from the rest of the list.

dotted pair
A *pair*. More specifically a *pair* whose right (cdr) element is not *NIL*. E.g.: (A . B). The dotted pair is a special case of the *dotted list*. A pair whose cdr part is NIL is a singleton list, e.g.: (A).

dynamic scoping
In a dynamically scoped system, each *atom* has only one *binding* that changes over time. Therefore, the value of a *variable* can be altered (see *mutation*) anywhere in a program. Given the function (LAMBDA () X), the form (SETQ X Y) will change the binding of the *free variable* X in the function to Y. (Cf. *lexical scoping*.)

effect
A persistent change that is caused by applying a *function* to *values* (see *application*). A pure function (a function without any effects) only maps values to values. An impure function may have an effect on the state of the system, like changing the values of

atoms (see *mutation*) or performing input or output operations. For instance, SETQ, READ, and PRINT have effects.

environment
A map from *variable* names (*atoms*) to their corresponding *values*. The mechanism that implements *bindings*.

eta expansion
(From *lambda calculus*.) The process of wrapping a *function* up in an extra *lambda function*. E.g.: $f \rightarrow_\eta (\lambda x(f\ x))$. Note that the transformation does not change the meaning of the function, i.e. $((\lambda x(f\ x))\ Y)$ is the same as $(f\ Y)$. Eta expansion has many applications, like changing the order of evaluation in *expressions*, but in this text its only purpose is to make a function f a first-class value when the compiler (see *compilation*) would otherwise inline it.

EVAL
A *function* that maps LISP *expressions* to their *normal forms*. Also called the "universal function". Basically an interpreter of LISP.

evaluation
The process of converting an *expression* to its *normal form* (or *value*). *Atoms* evaluate to the values bound to them (see *binding*) and the *application* of *functions* evaluates to the value returned by them. Evaluation of *special forms* depends on their semantics. Evaluation of an expression A to a value B is usually indicated by the notation A ==> B.

EXPR
In some LISP texts (and by some early LISP systems) used to denote a *lambda function*. (In this text EXPR denotes a *register*.)

expression
An *S-expression* that can be evaluated (see *evaluation*), giving a *value*. Basically the same as a LISP program. For example, the expression (CONS ′A ′(B)) will evaluate to the *list* (A B), but the

S-expression (NIL ′ A ′ (B)) is not an expression, because NIL is not a *function* and hence cannot be applied (see *application*).

FEXPR
"Form EXPRession". A *lambda function* that does not evaluate its arguments and whose result replaces its *application* before evaluating it. Basically the same as a *macro*.

form
See *expression*.

free variable
A *variable* that is not bound (see *binding*) in a *lambda function* (or another binding construct, like LABEL). A variable is bound in a function, if it is a *variable* of that function. For instance, X is bound in (LAMBDA (X) (F X)), but F is free in the same function.

freelist
A *list* of unused *cons cells* (and other cells) that is maintained by the LISP system. Data objects are allocated by removing cells from the freelist and the list is refilled by *garbage collection*.

function
A program that maps *values* to values, just like a mathematical function. For instance, the identity function $f(x) = x$ can be written as the *anonymous function* (LAMBDA (X) X).

function call
See *application*.

garbage collection
The process that recycles unused memory by adding it to the *freelist*. The garbage collector reclaims an object only when it can prove that the object can no longer be accessed by any program in memory.

higher-order function
A *function* expecting a function as one of its arguments (like MAPCAR or REDUCE). In theory (and in lexically scoped LISP) this also includes functions that return functions as their *values*.

homoiconicity
A property of programming languages that use the same representation for code and data. Every LISP program is an *S-expression*, so the usual *list* manipulation *functions* can be used to transform programs.

identity
The "sameness" of *atoms*. Every textual representation of the same interned atom (see *interning*) actually refers to the same—identical—atom. Identity is what makes atoms equal in the sense of EQ.

image file
A file containing the entire state of a LISP system, including all data objects, *functions*, the *freelist*, etc. By dumping (writing) an image file and loading it later, the previous state of the system is restored.

in-situ application
The *application* of a *lambda function* "on the spot", without binding it to a *variable* first. The expression ((LAMBDA (X) X) NIL) applies a lambda function in-situ to the value NIL.

interning
The process of storing an *atom* in an internal structure in order to make it unique, thereby implementing the *identity* of atoms.

lambda
A name used to denote *anonymous functions* (see *lambda functions*). In *M-expressions* and *lambda calculus* written as λ.

lambda calculus
A formal term rewriting system invented by Alonzo Church and his students in the 1930's. LISP is loosely based on lambda calculus, but also differs from it in substantial ways. (Lambda calculus uses *lexical scoping*, allows partial *application*, and has no type except for the *function*).

lambda function
A LISP function. Every function is denoted by the name “lambda” and hence has no individual name. For instance, (LAMBDA (X) X) is the identity function that maps every value to itself, and (LAMBDA (X) (LIST X)) is a function that puts its argument in a singleton list. Note that both functions have the “name” LAMBDA. A lambda function can be given an individual name by *binding* it to an *atom*.

lexical scoping
In a lexically scoped system, each *atom* can have any number of *bindings* that are created when a *closure* is formed. Each binding is local to a closure and therefore the value of each *variable* can only be changed (see *mutation*) in the lexical context (function) in which it was created. Given the function (LAMBDA () X), the form (SETQ X Y) will **not** change the binding of the *free variable* X in the function. (Cf. *dynamic scoping*.)

LEXPR
“List EXPRession”. A *lambda function* that has a single *variable* that will be bound to a *list* of all actual *arguments* passed to the function. Hence LEXPRs accept a variable number of arguments. For instance, (LAMBDA X X) implements the LIST function.

LISP
A family of programming languages. Short for "LISt Processor".

LISP 1
The original LISP language as described in [McCarthy1960].

LISP 1.5
The first version of the LISP language that was used outside of the M.I.T., described in [McCarthy1962].

LISP 1.6
A lesser-known LISP language that forms a link between LISP 1.5 and MACLISP. Described in [MIT1967]. A.k.a. PDP-6 LISP.

LISP-1
A LISP system with a single *namespace* that contains *bindings* for all kinds of objects and in particular for both *variables* and *functions*. Hence the following program would only work in a LISP-1: ((LAMBDA (F) (F ′ (A))) CAR). In a *LISP-N* system, the wrong binding of both CAR and F would be looked up in the example. *SCHEME* is a LISP-1.

LISP-2
A *LISP-N* with two *namespaces*, usually one for *functions* and one for *variables*. The following program works only in such a LISP-2: ((LAMBDA (LIST) (LIST LIST)) ′ FOO). In a *LISP-1* system, both occurrences of LIST in (LIST LIST) would refer to the variable, thereby *shadowing* the function.

LISP-N
A LISP system with multiple *namespaces*, for instance for variables, functions, macros, and other forms. *COMMON LISP* is a LISP-N. See also: *LISP-2*.

list
One of the principal data types of LISP, the other one being the *atom*. A list is any sequence of *S-expressions* enclosed in parentheses, like (A (B) C). The last cdr part (see *pair*) of a (proper) list is *NIL*. When that cdr part is something else, the list is a *dotted list*. A list has to be quoted (see *quotation*) to distinguish it from an *expression*.

LSUBR
An *LEXPR* that is written in (or compiled to) machine code. Considered archaic.

M-expression
“Meta expression”. A notation developed for writing programs before LISP was first implemented. The notation was never used to actually program computers, but remained popular in textbooks for some time.

MACLISP
The LISP language developed as part of Project MAC at the M.I.T. One of the most influential and wide-spread LISP systems before *COMMON LISP*. Described in [Moon1974].

macro
A function that transforms (rewrites) *special forms*. See *macro expansion*. Macros allow the programmer to add own syntax constructs to the LISP language.

macro expansion
The process of rewriting a "derived" *special form* (a *macro*) to an expression that does not contain any macro *applications*. Macro expansion takes place after reading an *expression* (see *reader*), but before *evaluation*.

meta expression
See *M-expression*.

metacircular evaluator
(1) An *EVAL* function that delegates evaluation of *primitive functions* of the language which it implements to its own primitive functions. For instance, a metacircular evaluator may use its own CONS function to implement CONS.
(2) An EVAL function that is written in the language that it implements, so it can evaluate itself.
The term is often used to denote either of the above, or both.

mutation
The modification of the *binding* of an *atom*, so that the atom is subsequently bound to a different *value*.

namespace
The separation of *bindings* for different entities within an *environment* results in different namespaces, where different kinds of *objects* can have the same names without interfering with each other. For instance, in a *LISP-2*, *functions* and *variables* can have the same names, because functions are only looked up in the

function part of an *application*, and variables are only looked up in the *argument* part.

NIL
A special *atom* that indicates both the "false" truth value (*T* being "true") and the end of a (proper) *list*. NIL is *self-quoting*.

normal form
(From *lambda calculus*.) Sometimes used to indicate the "result" (or *value*) of evaluating an *expression*. In lambda calculus any form that does not contain any *function application* is in normal form.

OBARRAY
See *object list*.

object
"Data object". Anything that is stored in the cell pool of the LISP system, like *S-expressions* and *functions*. Not related to the term "object" as used in the context of "object-oriented programming".

object list
A list (or other, more efficient data structure) containing all data objects (like *lists* and *atoms*) known to a LISP system. *Interning* an atom adds it to the object list. (Note: LISCMP uses a separate list for atoms.)

OBLIST
See *object list*.

outer value
A *value* that is currently not accessible, because the *atom* bound to it is temporarily bound to a different value. (See *binding*.) For example, given X=FOO, the *expression* ((LAMBDA (X) X) T) will temporarily bind the atom X to the value T. While the *lambda function* evaluates (see *evaluation*), FOO is the outer value of X.

pair
A data structure consisting of two components called its "car part'

and “cdr part”. Each component can be any *S-expression*. Pairs are formed by CONS and decomposed by *CAR* and *CDR*.

parser
A *function* or program that accepts sentences of a formal language (like a programming language) as its input and returns a tree that reflects the logical structure of that sentence. The LISP *reader* (READ) is a parser accepting *S-expressions*.

PDP-6 LISP
See *LISP 1.6*.

predicate
A *function* that always returns *T* (truth) or *NIL* (falsity). Also: the first element of a COND *clause*.

primitive function
A *function* that is implemented in the implementation language of a LISP system rather than in LISP itself. Sometimes also used to indicate an essential function of LISP. Considered archaic.

printer
An umbrella term for *functions* that generate and/or output readable representations of data objects (e.g. PRIN1 and PRINT).

property list
A *list* that is attached to an *atom* and contains pairs of “properties” (keys) and corresponding values. Unlike an *association list* the property list is a flat list with keys in odd and values in even positions: $(key_1\ value_1 \cdots key_N\ value_N)$. In early LISP, *EXPR*, *FEXPR*, and *LEXPR* were properties.

pseudo function
A *function* or *special form* with an effect, like input, output, or *mutation*.

pseudostring
An *atom* that serves as a simplicistic substitute for the “string” data object. Pseudostrings typically require some mechanism for

including special characters or characters reserved for LISP. See *slashification*.

quotation
A quoted *S-expression* is a data object while an S-expression that is not quoted is an *expression*. In symbolic LISP, a quoted atom is an atom object, an unquoted atom is a *variable*, a quoted *list* is a list object, and an unquoted list is a *function application* or a *special form*.

read eval print loop
The top-level loop of a LISP system that reads *expressions*, evaluates them (see *evaluation*), and prints their *values*.

reader
The *function* (e.g.: READ) that reads and analyzes the textual representations of *S-expressions* and converts them into internal structures, like *lists* and *atoms*. Also: the human being reading this text.

recursion
Self-*application*. A *function* that applies itself at some point is a recursive function. Recursion can be used to implement iteration (loops). It can be optimized by *tail call optimization*.

register
A small but fast portion of a computer's memory that is used to store *values* while performing computations with them.

REPL
See *read eval print loop*.

S-expression
"Symbolic expression". In symbolic LISP each S-expressions is either an *atom* or a *list*. In more complex LISP systems S-expressions may also include data objects like numbers, strings, arrays, etc.

SCHEME
The LISP language that first introduced *lexical scoping*. One of

the two major dialects of LISP that is still in use today, the other one being *COMMON LISP*. Described in [Scheme1975] and, more recently, [Scheme1991].

scope
The temporal extent or spatial region in which a *binding* is visible or accessible. When the visibility of bindings is limited to a textual (spatial) region, *lexical scoping* is used, and when and bindings are accessible during a certain period of time (their "dynamic extent"), *dynamical scoping* is used.

self-quotation
Some objects quote (see *quotation*) themselves, so they do not have to be quoted explicitly in *expressions*. In purely symbolic LISP, the *atoms T* and *NIL* are self-quoting. In more complex LISP systems, other data objects, like numbers and strings, are typically also self-quoting.

shadowing
When an *atom* is temporarily bound (see *binding*) to a different *value*, its original value is "shadowed" by the new value and (temporarily) becomes its *outer value*.

shallow binding
A strategy for implementing *bindings* where *atoms* link directly to *values*. Shallow binding is very efficient, but needs a mechanism for saving *outer values* during *function application*. (Cf. *deep binding*.)

slashification
A mechanism for including certain special characters (such as those reserved by the LISP language) in *atom* names. Often used to implement *pseudostrings*. The name comes from the slash ("/") character that was used in LISP 1.6 and MACLISP to prefix special characters in atoms.

special form
A form (*expression*) that looks like a *function application*, but has

some special meaning. For example, the COND, LAMBDA, and SETQ forms are special forms.

special operator
The *atom* that appears in the place of a *function* in a *special form* (q.v.).

SUBR
"Subroutine". A *function* that is implemented in machine code. Considered archaic.

symbol
See *atom*. In modern LISP there are multiple types of atoms (like numbers, strings, etc) and not just symbols. In purely symbolic LISP, as discussed in this book, the terms "atom" and "symbol" are synonyms.

symbolic expression
See *S-expression*.

T
A *self-quoting atom* that denotes logical truth ("falsity" is denoted by NIL). In fact all *values* except for NIL are "true", T is just the canonical true value.

tail application
A *function application* that is the last operation that takes place in a function. For example the application of the function F in the *expression* (LAMBDA (X) (F (G X))) is a tail application, but the application of G is not, because its *value* will be passed to F when it returns.

tail call optimization
An optimization that makes *tail applications* evaluate in constant space by restoring the *outer values* of *variables* **before** passing control to the applied function (but **after** evaluating the arguments of the function). Tail call optimization effectively turns *tail recursion* into loops.

tail recursion
When all recursive (see *recursion*) *applications* in a *function* are *tail applications*, the function is called “tail-recursive”. Tail-recursive functions evaluate (see *evaluation*) in constant space.

unbinding
The process of restoring the *outer values* of *variables*.

value
The result of the process of *evaluation*. Every LISP *expression* (as long as it terminates) has a value that is determined by evaluating that expression. The value of a *variable* is the *S-expression* bound to it and the value of a *function application* is the S-expression returned by it. Note that a function both expects values (as *arguments*) **and** returns a value.

variable
An *atom* that is bound in a *function* (or in a different *binding* construct, such as LABEL) or assigned a value by the SETQ *special form*. A variable has the appearance of an unquoted *atom* (see *quotation*) and evaluates to the value bound to it (see *evaluation*). For example, X and Y are variables of the function (LAMBDA (X Y) (F X Y)). F may also be a variable, but it is not a variable **of** (or bound **in**) the function. See also: *free variable*.

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>

int  infd = 0;
char inbuf[256];
int  inptr = 0,
     inlim = 0;

int rreadc(void) {
    if (inptr >= inlim) {
        inlim = read(infd, inbuf, 256);
        if (inlim < 1) return 0;
        inptr = 0;
    }
    inptr++;
    return toupper(inbuf[inptr-1]);
}

int  outfd = 1;
char outbuf[256];
int  outptr = 0;

void flush(void) {
    write(outfd, outbuf, outptr);
    outptr = 0;
}

void rwritec(int c) {
    if (outptr >= 256) flush();
    outbuf[outptr++] = c;
    if ('\n' == c) flush();
}

void pr(char *s) {
    while (*s)
        rwritec(*s++);
```

```
}

void prnum2(int n) {
    if (n != 0) {
        prnum2(n / 10);
        rwritec('0' + n%10);
    }
}

void prnum(int n) {
    if (0 == n)
        rwritec('0');
    else
        prnum2(n);
}

#define limit   65535

#define cell    unsigned short

void print2(cell n);

void halt(char *s, cell n) {
    flush();
    write(2, "*** ", 4);
    write(2, s, strlen(s));
    if (n != limit) {
        write(2, ": ", 2);
        outfd = 2;
        print2(n);
        flush();
    }
    write(2, "\n", 1);
    exit(1);
}

#define nil     0
#define true    1
#define symlis  2
#define oblist  3
#define stack   4
#define expr    5
#define expr2   6
```

```
#define tmp     7
#define funtag  8
#define tname   9
#define frelis  10

#define atomtag 1
#define marktag 2
#define travtag 4

cell car[limit],
     cdr[limit];
char tag[limit];

cell peak = limit;

void mark(cell n) {
    cell p, x;

    p = nil;
    for (;;) {
        if (tag[n] & marktag) {
            if (nil == p) break;
            if (tag[p] & travtag) {
                x = cdr[p];
                cdr[p] = car[p];
                car[p] = n;
                tag[p] &= ~travtag;
                n = x;
            }
            else {
                x = p;
                p = cdr[x];
                cdr[x] = n;
                n = x;
            }
        }
        else if (tag[n] & atomtag) {
            x = cdr[n];
            cdr[n] = p;
            p = n;
            n = x;
            tag[p] |= marktag;
        }
```

```
        else {
            x = car[n];
            car[n] = p;
            tag[n] |= marktag;
            p = n;
            n = x;
            tag[p] |= travtag;
        }
    }
}

int verbose = 0;

#define ref(x) car[x]
#define val(x) cdr[x]

int gc(void) {
    int i, k;

    k = 0;
    ref(frelis) = nil;
    for (i=0; i<=frelis; i++) mark(i);
    for (i=0; i<limit; i++) {
        if (0 == (tag[i] & marktag)) {
            cdr[i] = ref(frelis);
            ref(frelis) = i;
            k++;
        }
        else {
            tag[i] &= ~marktag;
        }
    }
    if (verbose) {
        if (k < peak) peak = k;
        prnum(k);
        pr(" CELLS RECLAIMED\n");
    }
    return k;
}

cell cons3(cell a, cell d, cell t) {
    int n;
```

```
    if (nil == ref(frelis)) {
        cdr[tmp] = d;
        if (0 == t) car[tmp] = a;
        gc();
        car[tmp] = cdr[tmp] = nil;
        if (nil == ref(frelis))
            halt("OUT OF CELLS", limit);
    }
    n = ref(frelis);
    ref(frelis) = cdr[ref(frelis)];
    car[n] = a;
    cdr[n] = d;
    tag[n] = t;
    return n;
}

cell mkfun(int k) {
    return cons3(cons3(funtag,
                       cons3(k, nil, atomtag),
                       atomtag),
                 nil, 0);
}

char *atomname(cell n) {
    static char s[128];
    int     i;

    i = 0;
    for (n = car[n]; n != nil; n = cdr[n]) {
        if (i > 127) break;
        s[i++] = car[n];
    }
    s[i] = 0;
    return s;
}

void dump(cell n) {
    char *s;
    int  fd;
    char *m;

    m = "IMAGE DUMP FAILED\n";
    s = atomname(n);
```

```
    fd = creat(s, 0644);
    if (fd < 0) {
        pr(m); return;
    }
    if (write(fd, car, limit*sizeof(cell))
        != limit*sizeof(cell))
    {
        pr(m);
    }
    else if (write(fd, cdr, limit*sizeof(cell))
        != limit*sizeof(cell))
    {
        pr(m);
    }
    else if (write(fd, tag, limit) != limit) {
        pr(m);
    }
    close(fd);
}

void load(cell n) {
    char    *s;
    int fd;
    char    *m;

    m = "IMAGE LOAD FAILED, ABORTING";
    s = atomname(n);
    fd = open(s, O_RDONLY);
    if (fd < 0) {
        pr("IMAGE NOT LOADED\n");
        return;
    }
    if (read(fd, car, limit*sizeof(cell))
       != limit*sizeof(cell))
    {
        halt(m, limit);
    }
    if (read(fd, cdr, limit*sizeof(cell))
       != limit*sizeof(cell))
    {
        halt(m, limit);
    }
    if (read(fd, tag, limit) != limit)
```

```
        halt(m, limit);
    close(fd);
}

cell pop(void) {
    cell    x;

    x = car[ref(stack)];
    ref(stack) = cdr[ref(stack)];
    return x;
}

cell readc(void) {
    int c;

    c = rreadc();
    if (0 == c) return nil;
    return cons3(c, nil, atomtag);
}

void writec(cell x) {
    if (tag[x] & atomtag)
        rwritec(car[x]);
    else
        rwritec(car[car[x]]);
}

void print2(cell n) {
    if (n == nil) {
        pr("NIL");
    }
    else if (n == true) {
        pr("T");
    }
    else if (tag[n] & atomtag) {
        pr("<");
        rwritec(car[n]);
        pr(">");
    }
    else if (tag[car[n]] & atomtag) {
        if (funtag == car[car[n]]) {
            pr("<FUNCTION-");
            prnum(car[cdr[car[n]]]);
```

```
            pr(">");
        }
        else {
            for (n = car[n]; n != nil; n = cdr[n])
                rwritec(car[n]);
        }
    }
    else {
        rwritec('(');
        while (n != nil) {
            if (n <= 1 || tag[n] & atomtag ||
                tag[car[n]] & atomtag)
            {
                pr(". ");
                print2(n);
                break;
            }
            print2(car[n]);
            if (cdr[n] != nil) rwritec(' ');
            n = cdr[n];
        }
        rwritec(')');
    }
}

void print(cell n) {
    print2(n);
    rwritec('\n');
}

void init(void) {
    memset(car, 0, limit*sizeof(cell));
    memset(cdr, 0, limit*sizeof(cell));
    memset(tag, 0, limit);
    car[true] = tname;
    cdr[true] = true;
    car[tname] = 84; /* T */
    cdr[tname] = nil;
    tag[tname] = atomtag;
}

#define toframe() \
    car[ref(stack)] = \
```

```
      cons3(ref(expr), car[ref(stack)], 0)

#define newframe() \
    ref(stack) = cons3(nil, ref(stack), 0)

#define push() \
    ref(stack) = cons3(ref(expr), ref(stack), 0)

#define label(x) case x:

#define goto(x) k = x; break

#define symbolic() (tag[car[ref(expr)]] & atomtag)

#define atomic() (ref(expr) <= 1 || symbolic())

#define unbind() restore(car[ref(stack)])

#define u(c) toupper(c)

#define car_err() \
    halt("CAR: EXPECTED LIST", ref(expr))

#define cdr_err() \
    halt("CDR: EXPECTED LIST", ref(expr))

void bind(cell v) {
    cell    a, p, n;

    a = cdr[car[ref(stack)]];
    if (tag[car[v]] & atomtag) {
        cdr[car[ref(stack)]] =
            cons3(nil,
                  cons3(v,
                        cons3(val(v), a, 0),
                        0),
                  0);
        val(v) = a;
    }
    else {
        for (p = a; p != nil; p = cdr[p]) {
            if (nil == v) break;
            n = car[p];
```

```
            car[p] = cons3(car[v], val(car[v]), 0);
            val(car[v]) = n;
            v = cdr[v];
        }
        if (p != nil || v != nil)
            halt("WRONG NUMBER OF ARGUMENTS", limit);
    }
}

int restore(cell p) {
    int k;

    k = car[p];
    if (nil == car[cdr[p]]) {
        p = cdr[cdr[p]];
        val(car[p]) = car[cdr[p]];
    }
    else {
        for (p = cdr[p]; p != nil; p = cdr[p])
            val(car[car[p]]) = cdr[car[p]];
    }
    return k;
}

int apply(int n) {
    if (!symbolic() || car[car[ref(expr)]] != funtag)
        halt("APPLICATION OF NON-FUNCTION",
             ref(expr));
    if (n < 0) {
        car[ref(stack)] =
            cons3(restore(car[cdr[ref(stack)]]),
                    car[ref(stack)], 0);
        cdr[ref(stack)] = cdr[cdr[ref(stack)]];
    }
    else {
        car[ref(stack)] = cons3(n, car[ref(stack)],
                                atomtag);
    }
    return car[cdr[car[ref(expr)]]];
}

int retn(void) {
    int k;
```

```
    k = car[car[ref(stack)]];
    ref(stack) = cdr[ref(stack)];
    return k;
}

void run(void);

int main(int argc, char **argv) {
    if (argc > 1) verbose = 1;
    init();
    run();
    print(ref(expr));
    if (verbose) {
        gc();
        prnum(limit-peak);
        pr(" CELLS USED MAX\n");
    }
    flush();
}

void setup(void);
```

COMPILER DRIVER SCRIPT

```
#!/bin/sh

### WARNING:
### This script WILL OVERWRITE the files
### "run.lisp", "run.c", and "run"!

SCM=s9

rm -f run
cp lisrun.c run.c
cp lisp.lisp run.lisp
if [ "$1" = "-" ]; then
    cat >>run.lisp
else
    cat $1 >>run.lisp
fi
echo "*STOP" >>run.lisp
echo COMPILE
if [ -f liscmp ]; then
    ./liscmp <run.lisp \
        | dd conv=lcase 2>/dev/null >>run.c
elif [ -f stage2 ]; then
    ./stage2 <run.lisp \
        | dd conv=lcase 2>/dev/null >>run.c
else
    ${SCM} liscmp.scm <run.lisp >>run.c
fi
cc -O1 -o run run.c
if [ $? = 0 ]; then
    echo RUN
    ./run $2
else
    tail -3 run.c
fi
```

```
(define addr '(1 0 0))

(define lbl '(0))

(define nil-addr '(0))
(define t-addr '(1))
(define limit '(6 5 5 3 5))

(define succ
  (lambda (x)
    (cdr (assv x '((0 . 1) (1 . 2) (2 . 3)
                   (3 . 4) (4 . 5) (5 . 6)
                   (6 . 7) (7 . 8) (8 . 9)
                   (9 . 0))))))

(define incr
  (lambda (x)
    (let loop ((x (reverse x)) (c #t) (y '()))
      (cond ((null? x)
             (if c (cons 1 y) y))
            (c
              (let ((d (succ (car x))))
                (loop (cdr x) (eqv? d 0) (cons d y))))
            (else
              (loop (cdr x) #f (cons (car x) y)))))))

(define number
  (lambda (n)
    (if (equal? n '(0))
        0
        (let loop ((n n) (m 0))
          (cond ((null? n) m)
                (else (loop (cdr n)
                            (+ (* m 10)
                               (car n)))))))))

(define cells '())

; (define symlis '((t 1)))
```

```
(define symlis (list (list 't '(t 1))))

(define oblist '())

(define make-label
  (lambda ()
    (set! lbl (incr lbl))
    lbl))

(define make-cell
  (lambda (a d t)
    (set! cells (cons (list addr t a d) cells))
    (let ((n addr))
      (set! addr (incr addr))
      n)))

(define make-atom
  (lambda (x)
    (let loop ((x (reverse (string->list
                            (symbol->string x))))
               (p nil-addr))
      (cond ((null? x)
             (make-cell p nil-addr #f))
            (else
              (loop (cdr x)
                    (make-cell (char-upcase (car x))
                               p #t)))))))

(define first
  (lambda (x)
    (string->symbol
      (substring (symbol->string x) 0 1))))

(define add-atom
  (lambda (x)
    (let ((fc (first x)))
      (cond ((assq fc symlis)
             => (lambda (b)
                  (cond
                    ((assq x (cdr b)) => cdr)
                    (else (let ((a (make-atom x)))
                            (set-cdr! b
```

```
                                (cons (cons x a)
                                      (cdr b)))
                              a)))))
            (else (let ((a (make-atom x)))
                    (set! symlis
                      (cons (cons fc
                                  (list (cons x a)))
                            symlis))
                    a))))))

(define number->symbol
  (lambda (x)
    (string->symbol
      (number->string x))))

(define locase
  (lambda (x)
    (list->string
      (map char-downcase
           (string->list x)))))

(define make-object
  (lambda (x)
    (cond ((null? x)
           nil-addr)
          ((eq? 'nil x)
           nil-addr)
          ((eq? 't x)
           t-addr)
          ((symbol? x)
           (add-atom x))
          ((string? x)
           (add-atom (string->symbol (locase x))))
          ((integer? x)
           (add-atom (number->symbol x)))
          (else
           (make-cell (make-object (car x))
                      (make-object (cdr x))
                      #f)))))

(define add-object
  (lambda (x)
    (cond ((eq? x 'nil) nil-addr)
```

```
          ((eq? x 't) t-addr)
          ((symbol? x)
            (add-atom x))
          (else
            (let ((n (make-object x)))
              (set! oblist (cons n oblist))
              n)))))

(define emit
  (lambda x
    (let loop ((x x))
      (cond ((null? x)
             (newline))
            (else
              (display (car x))
              (loop (cdr x)))))))

(define blockcom
  (lambda (x ta)
     (cond ((null? x))
           ((null? (cdr x))
             (exprcom (car x) ta))
           (else
             (exprcom (car x) #f)
             (blockcom (cdr x) ta)))))

(define condcom
  (lambda (x ta)
    (cond ((null? x)
           (emit "ref(expr) = nil;"))
          ((eq? 't (caar x))
            (cond ((null? (cdar x))
                     (blockcom (car x) ta))
                  (else (blockcom (cdar x) ta))))
          (else
            (exprcom (caar x) #f)
            (emit "if (ref(expr) != nil) {")
            (blockcom (cdar x) ta)
            (emit "} else {")
            (condcom (cdr x) ta)
            (emit "}")))))

(define varsymp
```

```
  (lambda (x)
    (cond ((eq? x 'nil) #f)
          ((eq? x 't) #f)
          (else (symbol? x)))))

(define labcom
  (lambda (x)
    (emit "newframe();")
    (let loop ((x (cadr x)))
      (cond
        ((null? x))
        (else (if (not (varsymp (caar x)))
                  (halt "LABEL: EXPECTED VARIABLE"))
              (emit "ref(expr) = cons3("
                    (number (add-atom (caar x)))
                    ", val("
                    (number (add-atom (caar x)))
                    "), 0);")
              (emit "toframe();")
              (exprcom (cadar x) #f)
              (emit "val(" (number (add-atom (caar x)))
                    ") = ref(expr);")
              (loop (cdr x)))))
    (emit "ref(expr) = nil;")
    (emit "toframe();")
    (blockcom (cddr x) #f)
    (emit "unbind();")
    (emit "pop();")))

(define funcom
  (lambda (x ta)
    (let* ((skip (make-label))
           (fun  (make-label)))
      (emit "goto(" (number skip) ");")
      (emit "label(" (number fun) ");")
      (emit "bind(" (number (add-object (cadr x)))
            ");")
      (blockcom (cddr x) #t)
      (emit "unbind();")
      (emit "k = retn(); break;")
      (emit "label(" (number skip) ");")
      (emit "ref(expr) = mkfun(" (number fun) ");"))))
```

```
(define atomcom
  (lambda (x)
    (exprcom (cadr x) #f)
    (emit "ref(expr) = atomic()? true: nil;")))

(define twoargs
  (lambda (x)
    (exprcom (caddr x) #f)
    (emit "push();")
    (exprcom (cadr x) #f)
    (emit "ref(expr2) = pop();")))

(define eqcom
  (lambda (x)
    (twoargs x)
    (emit "ref(expr) = ref(expr) == ref(expr2)? "
          "true: nil;")))

(define funapp
  (lambda (x ta)
    (emit "newframe();")
    (let loop ((a (reverse (cdr x))))
      (cond ((null? a))
            (else
              (exprcom (car a) #f)
              (emit "toframe();")
              (loop (cdr a)))))
    (exprcom (car x) #f)
    (if ta
        (emit "k = apply(-1); break;")
        (let ((ret (make-label)))
          (emit "k = apply(" (number ret) "); break;")
          (emit "label(" (number ret) ");")))))

(define checktag
  (lambda (x)
    (exprcom (cadr x) #f)
    (let ((f (cadr (assq (car x)
                         '((*atomp "atomtag")
                           (*markp "marktag")
                           (*travp "travtag"))))))
      (emit "ref(expr) = (tag[ref(expr)] & "
            f ")? true: nil;"))))
```

```
(define settag
  (lambda (x)
    (exprcom (cadr x) #f)
    (let ((f (cadr (assq (car x)
                         '((*setatom "atomtag")
                           (*setmark "marktag")
                           (*settrav "travtag")))))
          (m (not (eq? (caddr x) 'nil))))
      (emit "tag[ref(expr)]" (if m " |= " " &= ~")
            f ";"))))

(define exprcom
  (lambda (x ta)
    (cond ((eq? 'nil x)
           (emit "ref(expr) = nil;"))
          ((eq? 't x)
           (emit "ref(expr) = true;"))
          ((eq? '*pool x)
           (emit "ref(expr) = 0;"))
          ((eq? '*limit x)
           (emit "ref(expr) = " (number limit) ";"))
          ((symbol? x)
           (emit "ref(expr) = val("
                 (number (add-atom x)) ");"))
          ((string? x)
           (emit "ref(expr) = "
                 (number (add-atom (string->symbol
                                     (locase x))))
                 ";"))
          ((integer? x)
           (emit "ref(expr) = "
                 (number (add-atom
                           (number->symbol x)))
                 ";"))
          ((eq? 'cond (car x))
           (condcom (cdr x) ta))
          ((eq? 'label (car x))
           (labcom x))
          ((eq? 'lambda (car x))
           (funcom x ta))
          ((eq? 'progn (car x))
           (blockcom (cdr x) ta))
```

```
((eq? 'quote (car x))
  (emit "ref(expr) = "
        (number (add-object (cadr x)))
        ";"))
((eq? 'setq (car x))
  (if (not (varsymp (cadr x)))
      (halt "SETQ: EXPECTED VARIABLE"))
  (exprcom (caddr x) #f)
  (emit "val(" (number (add-atom (cadr x)))
        ") = ref(expr);"))
((eq? 'atom (car x))
  (atomcom x))
((eq? 'car (car x))
  (exprcom (cadr x) #f)
  (emit "if (symbolic()) car_err();")
  (emit "ref(expr) = car[ref(expr)];"))
((eq? 'cdr (car x))
  (exprcom (cadr x) #f)
  (emit "if (symbolic()) car_err();")
  (emit "ref(expr) = cdr[ref(expr)];"))
((eq? 'cons (car x))
  (twoargs x)
  (emit "ref(expr) = cons3(ref(expr), "
        "ref(expr2), 0);"))
((eq? 'eq (car x))
  (eqcom x))
((eq? '*halt (car x))
  (exprcom (cadr x) #f)
  (emit "halt(atomname(ref(expr)), "
        (number limit) ");"))
((memq (car x) '(*atomp *markp *travp))
  (checktag x))
((eq? '*car (car x))
  (exprcom (cadr x) #f)
  (emit "ref(expr) = car[ref(expr)];"))
((eq? '*cdr (car x))
  (exprcom (cadr x) #f)
  (emit "ref(expr) = cdr[ref(expr)];"))
((eq? '*dump (car x))
  (exprcom (cadr x) #f)
  (emit "dump(ref(expr));"))
((eq? '*load (car x))
  (exprcom (cadr x) #f)
```

```
            (emit "load(ref(expr));"))
          ((eq? '*next (car x))
            (exprcom (cadr x) #f)
            (emit "ref(expr)++;"))
          ((memq (car x)
                 '(*setatom *setmark *settrav))
            (settag x))
          ((eq? '*readc (car x))
            (emit "ref(expr) = readc();"))
          ((eq? '*rplaca (car x))
            (twoargs x)
            (emit "car[ref(expr)] = ref(expr2);"))
          ((eq? '*rplacd (car x))
            (twoargs x)
            (emit "cdr[ref(expr)] = ref(expr2);"))
          ((eq? '*writec (car x))
            (exprcom (cadr x) #f)
            (emit "writec(ref(expr));"))
          ((pair? (car x))
            (funapp x #f))
          (else
            (funapp x ta)))))

(define prolog
  (lambda ()
    (emit "")
    (emit "/******** LISCMP OUTPUT FOLLOWS ********/")
    (emit "")
    (emit "void run(void) {")
    (emit "int k;")
    (emit "setup();")
    (emit "for (k=0;;) switch (k) {")
    (emit "case 0:")))

(define emit-oblist
  (lambda ()
    (let loop ((lp oblist) (lplist nil-addr))
      (cond ((null? lp)
             (emit "ref(oblist) = "
                   (number lplist) ";"))
            (else
              (loop (cdr lp)
                    (make-cell (car lp)
```

```
                                lplist #f)))))))

(define init-cells
  (lambda ()
    (let loop ((n cells))
      (cond ((null? n))
            (else
              (let ((x (number (caar n)))
                    (a (if (pair? (caddar n))
                           (number (caddar n))
                           (char->integer
                             (caddar n))))
                    (d (number (cadddr (car n))))
                    (f (if (cadar n) "atomtag" "0")))
                (emit "car[" x "] = " a "; "
                      "cdr[" x "] = " d "; "
                      "tag[" x "] = " f ";")
                (loop (cdr n))))))))

(define epilog
  (lambda ()
    (emit "return;")
    (emit "}}")
    (emit "")
    (emit "void setup(void) {")
    (let ((y (map (lambda (x)
                    (cons (car x) (map car (cdr x))))
                  symlis)))
      (emit "ref(symlis) = "
            (number (make-object y)) ";"))
    (emit-oblist)
    (init-cells)
    (emit "}")))

(define dump-pool
  (lambda ()
    (emit "/******** POOL DUMP FOLLOWS ********")
    (for-each (lambda (x)
                (display (number (car x)))
                (display #\space)
                (display (if (cadr x) "A" "-"))
                (display #\space)
                (display (if (pair? (caddr x))
```

```
                        (number (caddr x))
                        (caddr x)))
                (display #\space)
                (display (number (cadddr x)))
                (newline))
              cells)
    (emit "*********************************/")))

(define liscmp
  (lambda ()
    (prolog)
    (let loop ((x (read)))
      (cond ((eof-object? x))
            ((eq? '*stop x))
            (else
              (exprcom x #f)
              (loop (read)))))
    (epilog)
    (dump-pool)))

(liscmp)
```

DOWNCASE PROGRAM

```
(let loop ((c (read-char)))
  (cond ((not (eof-object? c))
         (write-char (char-downcase c))
         (loop (read-char)))))
```

REFERENCES

[Backus1956] J.W. Backus, et. al.; “The FORTRAN Automatic Coding System for the IBM 704 EDPM”; Applied Science Division and Programming Research Dept., IBM, 1956

[Baker1992] Henry G. Baker; “Metacircular Semantics for Common Lisp Special Forms”; ACM Lisp Pointers V, 4 (Oct/Dec 1992), 11-20

[Barnett2020]; Jeff Barnett; USENET message <rjdt1j$q6c$1@dont-email.me> in the newsgroup comp.lang.lisp.; 2020-Sep-10

[Church1941] Alonzo Church; “The Calculi of Lambda Conversion”; Princeton University Press, 1941

[CLtL1984] Guy L. Steele, Jr, et. al.; “COMMON LISP The Language”; Digital Press, 1984

[DEC1968] “PDP-10 System Reference Manual”; Digital Equipment Corporation, 1968, Part No. DEC-10-HGAA-D

[DEC1987] Timothy E. Leonard (editor); “The VAX Architecture Reference Manual”; Digital Equipment Corporation, 1987

[DR1976] (No author given); “CP/M Operating System Manual”; Digital Research, 1976

[Feeley2004] Marc Feeley; “The 90-minute Scheme Compiler”; Talk, Montreal Scheme/LISP User Group, October 20, 2004

[Friden1959] (No author given); “Friden Flexowriter Technical Manual”; Friden, Inc, 1959

[Gabriel1985] Richard P. Gabriel; “Performance and Evaluation of Lisp Systems”; MIT Press, 1985

[Golden1970] Jeffrey P. Golden; “A User’s Guide to the A.I. Group LISCOM Lisp Compiler: Interim Report”; AI Memo 210, M.I.T., Project MAC, 1970

[Holm2016] Nils M Holm; “Compiling Lambda Calculus”; Lulu Press, 2016

[Holm2019] Nils M Holm; “LISP System Implementation”; Lulu Press, 2019

[HOPL1993] Richard P. Gabriel, Guy L. Steele, Jr.; “The Evolution of Lisp” in Proceedings, HOPL-II, The second ACM SIGPLAN conference on history of programming languages; ACM Press, 1993

[IBM1954] (No author given); “704 Electronic Data-Processing Machine, Manual of Operation”; Form 24-6661-2, International Business Machines Corporation, 1954

[IBM1959] (No author given); “IBM 7090 Data-Processing Machine Reference Manual”; Form A22-6528-4, IBM Corp., 1959, 1962 (revised)

[IBM1965] (No author given); “IBM 24 Card Punch / IBM 26 Printing Card Punch Reference Manual” (revised); IBM Corp., 1965

[Intel1986] (No author given); “Intel 80386 Programmer’s Reference Manual”; Intel Corporation Literature Distribution, 1986

[JL1996] Richard Jones, Rafael D. Lins; “Garbage Collection: Algorithms for Automatic Dynamic Memory Management”; Wiley & Sons, 1996

[K&R1988] Brian W. Kernighan, Dennis M. Ritchie; “The C Programming Language, Second Edition”; Prentice Hall, 1988

[Landin1964] Peter J. Landin; “The Mechanical Evaluation of Expressions”; The Computer Journal 6 (4): 308-320, 1964

[Maurer1972] W.D. Maurer; “The Programmer’s Introduction to LISP”; Macdonald, 1972

[McCarthy1958] John McCarthy; “An Algebraic Language for the Manipulation of Symbolic Expressions”; MIT AI Lab., AI Memo No. 1, Sept. 1958

[McCarthy1960] John McCarthy, et. al.; “LISP I Programmer’s Manual”; Computation Center and Research Laboratory of Electronics, M.I.T., 1960

[McCarthy1962] John McCarthy, et. al.; “LISP 1.5 Programmer’s Manual”; MIT Press, 1962

[McCarthy1979] John McCarthy; “An Interesting LISP Function”; ACM Lisp Bulletin, Vol. 3, Dec 1979

[McCarthy1981] John McCarthy; “History of LISP” in Wexelblad, Richard L. (ed); “History of Programming Languages” (ACM Monograph); Academic Press, 1981

[MIT1967] (No author given); “PDP-6 LISP (LISP 1.6)”; Massachusetts Institute of Technology, A.I. Memo 116A, 1967

[Moon1974] David A. Moon; “MACLISP Reference Manual”; Massachusetts Institute of Technology, 1974

[Moses2012] Joel Moses; “Macsyma: A personal history”; Journal of Symbolic Computation, Volume 47, Issue 2, February 2012, pages 123-130

[Nordstrom1970] Mads Nordström; “LISP F1 - A FORTRAN Implementation of LISP 1.5”; Institut för Informationsbehandling, Uppsala Universitet, 1970

[Phillips1990] Eve M. Phillips; “If It Works, It’s Not AI – A Commercial Look at Artificial Intelligence Startups”; Bachelor Thesis, Department of Electrical Engineering and Computer Science, M.I.T., May 7 1999

[Scheme1975] Gerald Jay Sussman, Guy Lewis Steele, Jr.; “SCHEME - an Interpreter for Extended Lambda Calculus”; AI Memo No. 349; Massachusetts Institute of Technology, AI Laboratory, 1975

[Scheme1991] William Clinger, Jonathan Rees (editors); “The Revised(4) Report on the Algorithmic Language Scheme”; ACM SIGPLAN Lisp Pointers Volume IV Issue 3, July, 1991, Pages 1-55

[Steele1977] Guy Lewis Steele, Jr.; “Data Representations in PDP-10 MACLISP”; Massachusetts Institute of Technology, AI Memo 420, 1977

[SW1967] Herbert Schorr, William M. Waite; “An efficient machine-independent procedure for garbage collection in various list structures”; Communications of the ACM (CACM), Volume 10 Issue 8, Aug. 1967, 501-506

[Weissman1967] Clark Weissman; “LISP 1.5 Primer”; Dickinson Publishing, 1967

[Weizenbaum1968] Joseph Weizenbaum; “The FUNARG Problem Explained”; unpublished memorandum, MIT, 1968

[WR1990] Benjamin W. Wah, C.V. Ramamoorthy (editors); “Computers for Artificial Intelligence Processing”; Wiley & Sons, 1990

BIBLIOGRAPHY

[Allen1978] John Allen; “Anatomy of LISP”; McGraw-Hill, 1978

[Barendregt1981] Henk P. Barendregt; “The Lambda Calculus, its Syntax and Semantics”; North-Holland Publications, 1981; American Elsevier, 1981

[DEC1964] “PDP-6 Arithmetic Processor 166 Instruction Manual, Vol. 1”; Digital Equipment Corporation, 1964

[DEC1968b] “PDP-10 KA10 Central Processor Maintenance Manual, Vol 1”; Digital Equipment Corporation, 1968

[DEC1976] “MH10 Maintenance Manual” Digital Equipment Corporation, 1976, Part No. EK-MH10-MM-003

[McCarthy1960b] John McCarthy; “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”; Communications of the ACM, Vol. 3, No. 4, 1960, pp. 184-195.

[Pitman2007] Kent M. Pitman; “The Revised Maclisp Manual, Saturday Morning Edition”; http://maclisp.info/pitmanual/index.html, 1983, 2007

[Wooldrige1964] Dean E. Wooldridge; “The New LISP System (LISP 1.55)”; Stanford Artitficial Intelligence Laboratory (SAIL) Memo 13, 1964

LIST OF FIGURES

INDEX