LISP FROM NOTHING

NILS M HOLM

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!

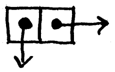
As usual, all code shown in the book (except for the most trivial examples) can be found on my homepage, **`t3x.org`**.

Enjoy the tour!

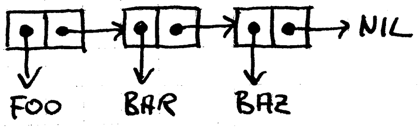
Nils M Holm, October 2020

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

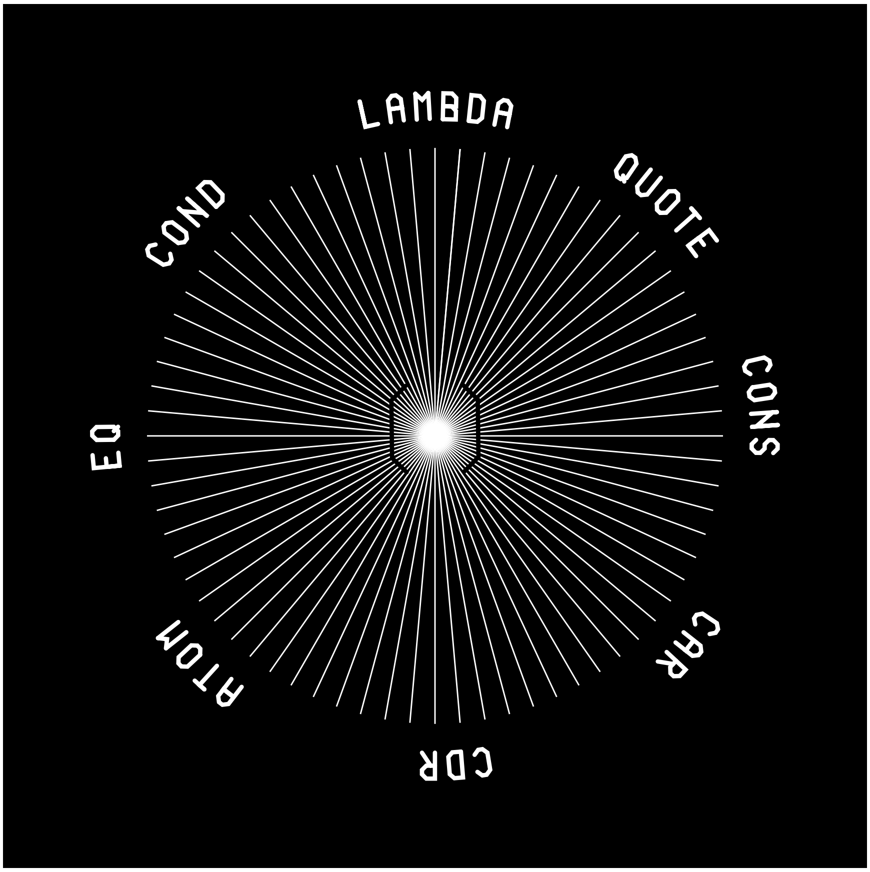


means, and how



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.



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.

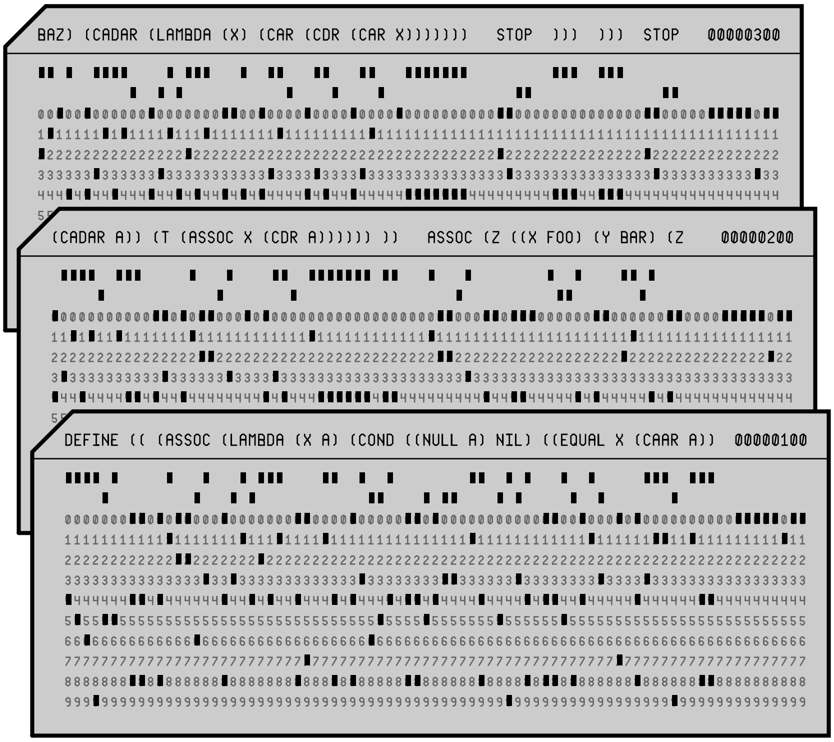
They silently assume that there already is

- 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


BAZ) (CADAR (LAMBDA (X) (CAR (CDR (CAR X))))))) STOP)))))) STOP 00000300
(CADAR A)) (T (ASSOC X (CDR A)))))))) ASSOC (Z ((X FOO) (Y BAR) (Z 00000200
DEFINE (((ASSOC (LAMBDA (X A) (COND ((NULL A) NIL) ((EQUAL X (CAAR A)) 00000100

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.

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*.

LIST OF FIGURES

INDEX