This is file: lispbook.tex **LaTeX'ed: June 22, 2006** **Feb. 18, 1997**

Interpreting LISP by Gary D. Knott

Preface

I wrote this book to help teach LISP to students in a course on data structures. Consequently it contains a careful description of the data structures manipulated by LISP functions. These data structures and others, notably hash tables, are also used in constructing a LISP interpreter. I wish to acknowledge the help of my students in shaping and debugging this material.

The study of LISP, coupled with the study of a LISP interpreter intended for exhibition, is of special interest to students in the areas of programming languages and computer architecture as well as data structures. Indeed, I hope this book will be useful to students in all areas of computer science, and I also hope it will be useful to autodidacts, professional programmers, and computer enthusiasts in a wide variety of fields.

This book is intended to be accessible to a wide range of interested readers from high school students through professional programmers. I would very much like to see students use this book to help them understand LISP interpreters, and thus understand the concepts involved in building an interpreter for any language. The best way to proceed is to compile and run the C LISP interpreter, and then experiment by modifying it in various ways. Finally, I hope this book can help all who use it develop an aesthetic appreciation of this elegant programming language.

Copies of this little book in PDF format, together with the LISP interpreter source code and ancillary files, can be found at `www.civilized.com`.

Gary Knott
Civilized Software, Inc.
12109 Heritage Park Circle
Silver Spring, Maryland 20906

phone:1-(301)-962-3711
email:knott@civilized.com
URL:http://www.civilized.com

Contents

1 LISP

LISP is an interesting programming language and the ideas involved in building a LISP interpreter are equally interesting. This book contains an introduction to LISP and it also contains the data structure details and the explicit code for a working LISP interpreter.

LISP is a programming language with unique features. It is conceptually interactive. Input commands are given one by one and the associated result values are typed-out. LISP is an applicative language, meaning that it consists mainly of functional application commands. Besides function application, there are forms of assignment commands and conditional commands written in functional form. In general, iteration is replaced by recursion.

The data values on which a LISP function may operate includes real numbers. Thus, an expression like "$1.5 + 2$" is a LISP statement which means: type-out the result of applying $+$ to the arguments 1.5 and 2. In LISP, function application statements are always written in prefix form, *e.g.* $+(1.5, 2)$. Moreover, rather than writing $f(x, y)$ to indicate the result of the function f applied to the arguments x and y, we write `(`f x y`)` in LISP, so `(+ 1.5 2)` is the LISP form for "$1.5 + 2$". Finally, functions in LISP are usually specified by identifier names rather than special symbols. Thus the correct way to compute $1.5+2$ in LISP is to enter the expression `(PLUS 1.5 2)`, which will, indeed, cause "3.5" to be typed out. An expression such as `(PLUS 1.5 2)` is called a *function call expression*. LISP functions can also operate on *lists* of objects; indeed the name "LISP" is derived from the phrase "LISt Processing".

LISP is commonly implemented with an interpreter program called the LISP interpreter. This program reads LISP expressions which are entered as input and evaluates them and types out the results. Some expressions specify that state-changing side-effects also occur. We shall describe below how a particular LISP interpreter is constructed at the same time that LISP itself is described.

There are a variety of dialects of LISP, including extended forms which have enriched collections of functions and additional datatypes. We shall focus on the common core of LISP, but some definitions given here are not universal, and in a few cases they are unique to the version of LISP presented herein (GOVOL). The GOVOL dialect of LISP presented here is similar to the original LISP 1.5 [MIT62]; it is not as complex as the most frequently used varieties of LISP found these days, but it contains all the essential features of these more complex varieties, so that what you learn here will be immediately applicable for virtually every LISP dialect.

2 The Atom Table and the Number Table

Our LISP interpreter program maintains several general data structures. The first data structure is a symbol table with an entry for every named data object. These named data objects are called *ordinary atoms*, and the symbol table of ordinary atoms is called the *atom table*. (The term "atom" is historical; that is the term John McCarthy used.) The atom table is conceptually of the following form.

	name	type	value	plist	bindlist
0					
1					
⋮					
$n-1$					

The atom table has n entries where n is some reasonably large value which is unspecified for now. Each entry in the atom table has a name field, which holds the atom name, for example: "`PLUS`" or "`AXY`". Each entry also has a value field for holding the current value of the atom, and an associated type field which is an integer code that specifies the type of the value of the atom. For example, the value of the atom "`PLUS`" is a builtin function, which is classified by the integer typecode 10. Each atom table entry also has a plist field and a bindlist field to be discussed later.

The second data structure is a simple table called the *number table* in which floating-point numbers are stored. Every input number and every computed number is stored in the number table, at least for the period of time it is required to be there. The number table is conceptually of the following form:

	number
0	
1	
⋮	
$n-1$	

Since the floating-point numbers include a large complement of integers, there is no reason, except, perhaps, for speed, to have integers provided in LISP as a separate datatype.

Exercise 2.1: Precisely specify the set of integers which are expressible in floating point format on some computer with which you are familiar.

Exercise 2.2: Is there a good reason that the atom table and the number table have the same number of entries?

Solution 2.2: No, there is no good reason for this. It can be easily changed if desired.

The datatype codes are:

1	undefined
8	variable (ordinary atom)
9	number (number atom)
0	dotted-pair (non-atomic S-expression)
10	builtin function
11	builtin special form
12	user-defined function
13	user-defined special form
14	unnamed function
15	unnamed special form

These datatypes are exactly the datatypes available in the version of LISP discussed here. The reason for the seemingly peculiar typecodes will be apparent later; they are chosen so that the individual binary digits have useful meanings.

There are two kinds of *atoms* in LISP. All atoms occur in either the atom table or the number table; *ordinary atoms* are entries in the atom table, and *number atoms* are entries in the number table. An ordinary atom is like a variable in FORTRAN. It has a name and a value. The name is a character string which is kept for printing and input matching purposes. The value of an ordinary atom is a value of one of the types listed above. A number atom which represents a real constant also has a value; this value is the floating-point bitstring representing the number. The value of a number atom cannot be changed; a number atom is thus a constant.

An ordinary atom whose value is undefined is created in the atom table whenever a previously-unknown name occurs in a LISP input statement. An ordinary atom whose value is undefined has an arbitrary bit pattern in its value-field and the typecode 1 in its type-field.

An ordinary-atom-valued ordinary atom in entry i of the atom table holds an index j of an atom table entry in its value-field; the value of the entry i atom is then the entry j atom. The typecode in the type field of such an ordinary-atom-valued ordinary atom is 8.

A number-valued ordinary atom A in entry i of the atom table holds an index j of a number table entry in its value field. Entry j in the number table is where the number atom representing the number-value of the ordinary atom A is stored. The type field of entry i in the atom table holds the typecode 9.

A number atom is created in the number table whenever a number which is not already present in the number table occurs in the input, or is computed. Each distinct number in the number table occurs in one and only one entry. Similarly, an ordinary atom is created in the atom table whenever an ordinary atom occurs in the input, or is otherwise generated, which is not already present in the atom table. Each distinct ordinary atom in the atom table is stored uniquely in one and only one entry.

An atom may be its own value. In particular, this is always the case for a number atom which is always made to appear to have itself as its value in the sense that the result of evaluating the number atom named n is the corresponding floating-point bitstring which is represented lexically and typed-out as the identical name n or a synonym thereof. Since number atoms are constants,

we may consistently confuse their names and values in this fashion. Note that a number atom has many synonomous names, *e. g.* "1", "1.0", and "1." are all names of the same value.

Numeric constant names are strings of one or more digits with an optional embedded decimal point and an optional initial minus sign. Ordinary atom names are strings which do not contain an open parenthesis, a close parenthesis, a blank, an apostrophe, or a period, and which do not have an initial part consisting of a numeric constant name. As mentioned above, it is convenient to identify numbers with their corresponding number names.

Exercise 2.3: Which of the strings: `A`, `ABcd`, `++`, `A-3`, `-`, `B'C` and `3X` are ordinary atom names?

Solution 2.3: All of them are ordinary atom names, except `B'C` and `3X`.

An ordinary atom can be assigned a value by means of an assignment operator called `SETQ` which is provided in LISP. For example `(SETQ A 3)` makes the ordinary atom `A` a number-valued ordinary atom whose value is 3; any previous value of `A` is lost. The value of `A` after the assignment is the result value of the expression `(SETQ A 3)` and this value is duly typed-out when the expression `(SETQ A 3)` is entered as a command. We perhaps should say that `(SETQ A 3)` makes `A` a number-valued ordinary atom whose value is the number named by 3. However, as mentioned above, it is convenient to downplay this distinction. We are protected from abusing number names, since LISP refuses to honor input such as `(SETQ 3 4)`.

A *function* can be defined formally as a set of ordered pairs O; this is the usual means by which set-theory is made to serve as a basis for mathematics in general. The set of elements D which constitute the first-component values is called the *domain* of the function, and the set of elements R which constitute the second-component values is called the *range* of the function. The domain is the set of possible input values and the range is the set of possible output values. These domain and range elements can be whatever is desired, including k-tuple objects; however, if (a, b) is a pair of the function O with domain D and range R, so that $a \in D$ and $b \in R$, then no pair of the form (a, c), with $c \neq b$, is a member of the set of ordered pairs O representing the function. For example, the binary addition function is $\{\, ((x, y), z) \mid z = x + y \,\}$. The value of a function-valued atom is conceptually such a set of ordered pairs. Thus the ordinary atom `PLUS` is conceptually initialized to have the value $\{\, ((x, y), z) \mid z = x + y \,\}$.

Exercise 2.4: If O is a set of ordered pairs representing a function whose domain is D and whose range is R, does it follow that $O = D \times R$?

Solution 2.4: Definitely not.

Exercise 2.5: Does $((1, 2), 3)$ belong to the binary addition function? How about $((2, 1), 3)$ and $((2, 2), 3)$? Is the binary addition function an infinite set?

Solution 2.5: Yes, yes, and no. Yes, the binary addition function is an infinite set.

Exercise 2.6: The set S whose elements are the ordered pairs of numbers appearing as the first components of the ordered pairs of the binary addition function is a set of ordered pairs. Is S a function?

All LISP computations are done by defining and applying various functions. There are two forms of functions in LISP: ordinary functions and special forms. A *special form* in LISP is just a function, except that the LISP evaluation rule is different for special forms and functions. Arguments of functions are evaluated *before* the function is applied, while arguments of special forms are *not* further evaluated before the special form is applied. Thus the value of a special-form-valued atom is conceptually a set of ordered pairs just as is the value of a function-valued atom. We also need one descriptor bit to tell if such a set corresponds to a function or a special form. The typecode value incorporates this bit of information. It is vitally important for LISP to 'understand' both ordinary functions and special forms. For a hint as to why this is, note that `SETQ` is a special form.

Every input statement to LISP is an atom or an applicative expression (function call expression) which is to be evaluated, possibly with side-effects, and the resulting value is to be typed-out.

There are a number of particular ordinary atoms initially present in the atom table. One such atom is `NIL`. `NIL` is an atom-valued atom whose value is itself `NIL`. Another atom initially present is `T` which is an atom-valued atom whose value is itself `T`. One purpose of `NIL` is to stand for the Boolean value "false", and `T`, of course, is intended to stand for the Boolean value "true."

The names of all the builtin functions and special forms, such as `PLUS`, are the names of ordinary atoms which are initially established in the atom table with appropriate initialized values. The values of these ordinary atoms are conceptually ordered pairs, but in truth, the value-field of an atom whose value is a builtin function or builtin special form is either ignored or is an integer code used to determine which function or special form is at hand.

Thus if "`NIL`" is entered into LISP, "`NIL`" is the output. Similarly if "`T`" is typed-in, "`T`" is the output. If "`PLUS`" is typed-in, an infinite set of ordered pairs should be typed-out, but this is represented by typing out the name "`PLUS`" and the type of the value of "`PLUS`" in braces instead, as: {`builtin function: PLUS`}.

As discussed above, numbers are entered in the number table as number atoms. The presented name is the symbol string of the number, and the value is the associated floating-point value. Only the value is stored in the number table, however. A suitable name is reconstructed whenever this name needs to be printed-out. Thus, typing "3" into LISP, results in "3" typing out, and a number atom with the value 3 is now in the number table.

If a previously-unknown ordinary atom x is entered into LISP by typing its name, then "x is undefined" is typed-out, and the atom x, with the typecode 1 and an undefined value then exists in the atom table. If x is again entered as input, "x is undefined" is typed-out again.

Exercise 2.7: What is the result of entering "`(SETQ A 3)`" and then "`A`"?

Solution 2.7: "3" and then "3" again. Also the number atom "3" is now entered in the number table in some row j, and the ordinary atom `A` is entered in the atom table in some row k of the form ["`A`", 9, j, −, −]. Note the *value* of `A` is a data object of type 9. The atom "`A`" itself is an ordinary atom which would be described by the typecode 8 if it were to appear as the value of some other atom.

3 Evaluation

Let us denote the value of an atom or function call expression, x, by $v[x]$ from now on. The evaluation operator, v, defined here, is essentially embodied in the LISP interpreter.

When x is an ordinary atom, $v[x]$ is the data object specified in the value field of the atom table entry for x. The type of this data object is given by the typecode in the type field of the atom x. When x is a number atom, $v[x] = x$.

Exercise 3.1: What is PLUS?

Solution 3.1: PLUS is an ordinary atom.

Exercise 3.2: What is v[PLUS]?

Solution 3.2: v[PLUS] is, conceptually, a set of ordered pairs. v[PLUS] is *not* an atom.

Exercise 3.3: What does the atom table entry for "PLUS" look like?

Solution 3.3: It is a row of the atom table of the form ["PLUS", 10, x, −, −], where x is a private internal representation of the set of ordered pairs which is the value v[PLUS] described by the typecode 10.

Exercise 3.4: Why is v[NIL] = NIL?

Solution 3.4: Because the NIL entry in the atom table, say entry j, is initialized as: ["NIL", 8, j, −, −].

Exercise 3.5: What is v[3]?

Exercise 3.6: What is v[ABC], where ABC has never been assigned a value by SETQ?

Solution 3.6: v[ABC] is undefined.

A function call expression $(f\ x\ y)$ is evaluated in LISP by evaluating f, x, and y in order, *i.e.*, by computing $v[f]$, $v[x]$, and $v[y]$, and then producing the result of $v[f]$ applied to the arguments $(v[x], v[y])$. The value $v[f]$ must be a function. Note x and/or y may also be function call expressions, so this rule applies recursively.

A special-form call expression, $(s\ x\ y)$, is evaluated in LISP by evaluating s, which must result in a special form, and then producing the result value of $v[s]$ applied to the arguments (x, y). The only difference between a function application and a special form application is that the arguments of a function are evaluated before the function is applied, whereas a special form's arguments are not pre-evaluated.

The analogous definitions for the value of general k-argument function call expressions and special form call expressions, of course, hold.

Exercise 3.7: Write a careful definition of $v[(f\ x_1\ x_2\ \ldots\ x_k)]$ where $v[f]$ is a function and $k \geq 0$. Do the same when $v[f]$ is a special form.

4 Some Functions and Special Forms

We can now state a few builtin LISP functions and special forms.

- `SETQ` – **special form with a side-effect**

  $v[$`(SETQ` x y`)`$] = v[x]$, *after* $v[x]$ is made equal to $v[y]$ as an initial side-effect. x must be an ordinary atom, necessarily with a non-numeric name. The type of the value of x is changed to be the type of $v[y]$, with a special modification in the case where $v[y]$ is an unnamed function or unnamed special form which will be discussed later. Any previous value of the atom x is lost.

  Note that it is almost-always equally-correct to say that $v[$`(SETQ` x y`)`$] = v[y]$, with the side-effect of assigning $v[y]$ to be the value of the atom x.

- `QUOTE` – **special form**

  $v[$`(QUOTE` x`)`$] = x$.

- `PLUS` – **function**

  $v[$`(PLUS` n m`)`$] = v[n] + v[m]$. $v[n]$ and $v[m]$ must be numbers.

- `DIFFERENCE` – **function**

  $v[$`(DIFFERENCE` n m`)`$] = v[n] - v[m]$. $v[n]$ and $v[m]$ must be numbers.

- `MINUS` – **function**

  $v[$`(MINUS` n`)`$] = -v[n]$. $v[n]$ must be a number.

- `TIMES` – **function**

  $v[$`(TIMES` n m`)`$] = v[n] \cdot v[m]$. $v[n]$ and $v[m]$ must be numbers.

- `QUOTIENT` – **function**

  $v[$`(QUOTIENT` n m`)`$] = v[n]/v[m]$. $v[n]$ and $v[m]$ must be numbers with $v[m] \neq 0$.

- `POWER` − **function**

  $v[$`(POWER` n m`)`$] = v[n] \uparrow v[m]$. $v[n]$ and $v[m]$ must be numbers such that if $v[n] < 0$ then $v[m]$ is an integer.

- `FLOOR` − **function**

  $v[$`(FLOOR` n`)`$] = \lfloor v[n] \rfloor$, the greatest integer less than or equal to $v[n]$. $v[n]$ must be a number.

- `EVAL` − **function**

  $v[$`(EVAL` x`)`$] = v[v[x]]$.

Exercise 4.1: Since $v[$`(EVAL` x`)`$] = v[v[x]]$, why doesn't `(EVAL` x`)` $= v[x]$?

Solution 4.1: Because the inverse of the evaluation operator v does not exist.

Exercise 4.2: Suppose the atom `A` has never been assigned a value via `SETQ`. What is $v[$`(QUOTE A)`$]$? What is $v[$`A`$]$?

Solution 4.2: $v[$`(QUOTE A)`$] =$ `A`, but $v[$`A`$]$ is undefined. Note $v[v[$`(QUOTE A)`$]] = v[$`A`$]$, so $v[$`(EVAL (QUOTE A))`$] = v[$`A`$]$ whether `A` has a defined value or not.

Exercise 4.3: What is $v[$`(PLUS (QUOTE 3) 2)`$]$?

Solution 4.3: 5, since $v[x] = x$ when x is a number.

Exercise 4.4: What does `(SETQ T NIL)` do?

Solution 4.4: `NIL` is the output, and the value of `T` is now changed to be `NIL`.

Most versions of Lisp have ways of indicating that the value of an atom is constant and cannot be changed. We do not introduce this complication here, but obviously, it is perilous or worse to assign new values to important ordinary atoms like `NIL`.

Exercise 4.5: What is $v[$`(SETQ (SETQ A T) (SETQ B NIL))`$]$?

Solution 4.5: An error arises, since the first argument to the outer `SETQ` is not an ordinary atom. Remember, $v[$`SETQ`$]$ is a special form.

Exercise 4.6: What does `(SETQ P PLUS)` do?

Solution 4.6: Now $v[$`P`$] = v[$`PLUS`$]$, so now $v[$`(P 2 3)`$] = 5$. Also "{`builtin function: P`}" is typed-out.

Exercise 4.7: What does `(SETQ PLUS` -1`.)` do?

Solution 4.7: The function value of PLUS is discarded, and now $v[\texttt{PLUS}] = -1$. Also -1 is typed-out.

Note that SETQ is a special form, yet its second argument is evaluated. It is more correct to say that SETQ is passed its arguments, and then *it* performs the required computation which entails computing the value of the supplied second argument. Thus special forms may selectively evaluate some or all of their arguments, but such evaluation, if any, is done *after* the arguments are passed to the special form.

Note that not all LISP functions are defined for all possible LISP data objects occurring as input. A function which does accept any input is called a *total* function. The function TIMES, for example, only accepts numbers as input. The function EVAL may appear to be a total function, but consider $v[$(EVAL PLUS)$]$. This is $v[v[\texttt{PLUS}]]$, where $v[$PLUS$]$ is a set of ordered pairs. But the v-operator value of a set of ordered pairs is not, thus far, defined. $v[$(EVAL (QUOTE PLUS))$]$ is defined, however, and we can make LISP a little less persnickity by defining $v[x] = x$ when x is a function or special form. We shall adopt this extension henceforth. Then $v[$(EVAL PLUS)$] = v[$(EVAL (QUOTE PLUS))$]$.

Exercise 4.8: Even with the just-introduced convention, EVAL is not a total function. Give an example of illegal input to EVAL.

Solution 4.8: The input A, where A is an atom whose value is undefined, is illegal input to EVAL. Later, when dotted-pairs are introduced, we will see that input like (3 . 5) is illegal also.

Exercise 4.9: What happens if "$v[$NIL$]$" is typed-in to the LISP interpreter?

Solution 4.9: An atom whose name is "$v[$NIL$]$" is specified and its value (which is probably undefined) is printed out. The v-operator is *not* a LISP function. It *is* the LISP interpreter, and hence is more properly called a meta-operator.

A *predicate* is a function whose result values are always the boolean values *true* or *false*. In the case of LISP, the result value of a predicate is always either T or NIL. We identify functions as predicates merely for descriptive convenience. This explains the choice of the names NUMBERP and ZEROP defined below. Not every LISP predicate follows this naming convention however; for example EQ, defined below, is a predicate.

- EQ − **predicate**

  $v[$(EQ x y)$]$ = if $v[x] = v[y]$ then T else NIL where $v[x]$ and $v[y]$ are atoms. Although, strictly, $v[x]$ and $v[y]$ must be atoms, non-atoms may well work in certain circumstances.

- NUMBERP − **predicate**

  $v[$(NUMBERP x)$]$ = if $v[x]$ is a number then T else NIL

- `ZEROP` − **predicate**

  $v[$`(ZEROP` x`)`$] =$ if $v[x] = 0$ then `T` else `NIL`

  Exercise 4.10: What is $v[$`(EQ 2 (SETQ B 3))`$]$?

  Solution 4.10: `NIL`, and now $v[$`B`$] = 3$ due to the assignment side-effect.

  Exercise 4.11: What is $v[$`(EQ .33333 (QUOTIENT 1 3))`$]$?

  Solution 4.11: Probably `NIL`, since $1/3 \neq .33333$; but possibly `T` if the precision of floating-point numbers is less than 6 decimal digits. All the usual vagaries of floating-point arithmetic are present in LISP.

  Exercise 4.12: What is $v[$`(EQ (NUMBERP 0) (ZEROP 0))`$]$?

  Solution 4.12: `T`.

  Exercise 4.13: Is `EQ` a total function?

  Exercise 4.14: What is $v[$`(ZEROP (PLUS 2 (MINUS 2)))`$]$?

  Solution 4.14: `T`. Because $v[$`(PLUS 2 (MINUS 2))`$] = v[2] + v[$`(MINUS 2)`$] = 2 + (-v[2]) = 2 + (-2) = 0$.

  Exercise 4.15: What is $v[$`(EQ 0 NIL)`$]$?

  Solution 4.15: `NIL`.

  Exercise 4.16: Why are capital letters used for ordinary atom names in this book?

  Solution 4.16: Only for the sake of uniformity and tradition. Lower-case letters are perfectly acceptable within ordinary atom names, along with many special characters.

5 S-Expressions

LISP has builtin functions which deal with certain composite data objects constructed out of atoms. These data objects are called *non-atomic S-expressions*. They are binary trees whose terminal nodes are atoms. Some of these trees can be interpreted as *lists* and these are a very popular form in LISP. Indeed, as mentioned earlier, LISP derives its name from the phrase "list processing." Atoms and non-atomic S-expressions, taken together, form the class of data objects whose members are called *S-expressions*. The term "S-expression" is short for the phrase "symbolic expression." Non-atomic S-expressions play the role of arrays in other programming languages.

The class of S-expressions is defined syntactically as follows: Every atom is an S-expression, and, if a and b are S-expressions, then the *dotted-pair* `(`a `.` b`)` is an S-expression. Mathematically speaking, a dotted-pair is merely an ordered pair. Note dotted-pairs *must* be enclosed in parentheses. The terms "non-atomic S-expression" and "dotted-pair" are synonymous.

Thus, for example, all the following expressions are S-expressions, and the last four are non-atomic S-expressions.

```
T
NIL
3
(1 . T)
((0 . .1) . NIL)
(((1 . 2) . (3 . 4)) . 5)
(PLUS . A)
```

Dots are *not* operators; dots and parentheses are merely used to give a concrete form to the abstract idea of dotted-pairs in exactly the same way that digit symbols are used to provide a concrete form for the abstract idea of integers. Dots and parentheses are used within *dot notation* in LISP parlance.

Exercise 5.1: Is `(A . (B . C) . D)` an S-expression?

Solution 5.1: No. Every dot must be used to form a dotted-pair, and every dotted-pair must be enclosed in parentheses.

Exercise 5.2: How can dots be used as decimal points in numbers and also as the connectors in dotted-pairs without confusion?

Solution 5.2: Dots used as decimal points must appear immediately adjacent to one or two digit-characters; a dot used as a dotted-pair connector must have one or more blanks intervening between it and a digit.

Exercise 5.3: How many S-expressions are there?

Solution 5.3: In any LISP program only a finite number of S-expressions arise, but conceptually, the set of S-expressions has an infinite number of members. In fact, the set of ordinary atoms, by itself, has an infinite number of members.

The special form `QUOTE` is used to specify constant S-expressions. A number, like 1.5, is a constant by virtue of the convention that it is self-referentially defined, so that $v[1.5] = 1.5$. However the dotted-pair `(T . NIL)` or the atom `A` denote their values in most contexts, so if we wish to prevent such possibly-foolish evaluations, we must write `(QUOTE (T . NIL))` or `(QUOTE A)`.

Exercise 5.4: What is v[`(QUOTE 3)`]?

6 Typed-Pointers

Internally an ordinary atom in LISP is represented by an integer index into the atom table, that is, by a pointer. We use the terms *index* and *pointer* interchangeably as seems fit. By knowing a pointer to an ordinary atom, we can access both the name and the value of the atom. A number atom is

represented by an integer index into the number table where the corresponding floating-point value is stored. Similarly a non-atomic S-expression is also represented internally by a pointer. Some means are needed in order to distinguish what kind of object a pointer points to. Thus we shall carry an integer typecode with a pointer, and refer to the pair together as a *typed-pointer*.

A typecode and pointer which together form a typed-pointer will be packed into one 32-bit computer word. Thus a typed-pointer consists of two adjacent bit fields which form a 32 bit integer. The first 4 bits is the type of the pointed-to data object, and the remaining 28 bits hold the pointer or index to the pointed-to data object. A typed-pointer which forms a non-positive 32-bit integer will be a pointer to an ordinary atom in the atom table, or to a number atom in the number table, or will be, as we shall see later, a pointer to a function or special form. A typed-pointer which forms a positive integer will be a pointer to a dotted-pair.

In fact, the type-field and the value-field in each atom table entry are packed in a single 32-bit word which can be easily accessed as a typed-pointer.

We shall use the integers in $\{1, 2, \ldots, m\}$ as pointers (indices) to non-atomic S-expressions. In fact, we establish an array of structures: $P[1{:}\,m]$(integer *car*, *cdr*), called the *list area*, and each non-atomic S-expression pointer j is interpreted as an index into P. An element of P is called a *list node*. The declaration notation above is intended to indicate that each list node P_j consists of a pair of integer fields: $P_j.car$ and $P_j.cdr$.

Note that the atom table and the number table each have n entries with their indices ranging from 0 to $n-1$, and the list area has m entries with indices ranging from 1 to m.

The typecodes used in typed-pointers are:

0000:	dotted-pair (non-atomic S-expression)
0001:	undefined
1000:	variable (ordinary atom)
1001:	number (number atom)
1010:	builtin function
1011:	builtin special form
1100:	user-defined function
1101:	user-defined special form
1110:	unnamed function
1111:	unnamed special form

Thus a typed-pointer t with $t \geq 0$ points to a dotted-pair, and a typed-pointer t with $t < 0$ points to other than a dotted-pair.

Exercise 6.1: What do the individual bits of the typecode values indicate?

Solution 6.1: Numbering the bits as 1,2,3, and 4 from left-to-right, we see that bit 1 is 0 for a dotted-pair (or an undefined value), and bit 1 is 1 otherwise. Within the class of function and special form typecodes, bit 2 is 0 for a builtin function or special form, and bit 2 is 1 otherwise;

and bit 4 is 1 for a special form and bit 4 is 0 for a function. This bit-encoding is harmless, but it isn't really very important or very useful; arbitrary numbers would be nearly as convenient.

If j is a 28-bit untyped-pointer, *i.e.* a simple address or an index, then the following functions may be used to form a typed-pointer, where :: denotes bit-string concatenation. These functions are *not* LISP functions which are predefined in LISP; they are convenient "meta-functions" which allow us to describe the internal form and meaning of our LISP interpreter program.

$$\begin{aligned} se(j) &= 0000 :: j \\ oa(j) &= 1000 :: j \\ nu(j) &= 1001 :: j \\ bf(j) &= 1010 :: j \\ bs(j) &= 1011 :: j \\ uf(j) &= 1100 :: j \\ us(j) &= 1101 :: j \\ tf(j) &= 1110 :: j \\ ts(j) &= 1111 :: j \end{aligned}$$

Now we can explain how any particular non-atomic S-expression is represented. If j points to a non-atomic S-expression of the form `(B . C)`, then $P_j.car$ points to the S-expression `B` and $P_j.cdr$ points to the S-expression `C`. Note that `B` or `C` or both may be atomic S-expressions; this just means the corresponding typed-pointer may not be positive.

Exercise 6.2: Suppose `(SETQ A 3)` is typed into the LISP interpreter. Explain how the LISP interpreter computes v[`A`] in the course of executing `(SETQ A 3)`, *i.e.* in the course of computing v[`(SETQ A 3)`]. Compare this with the evaluation of `A` which is required when `A` is entered as input. Construct a fragment of a program in a conventional language like C to aid in this explanation.

7 Pictorial Notation

Let us suppose the atom table and the number table are loaded as follows:

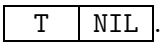
atom table:

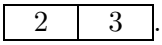
	name	type	value
0	NIL	8	0
1	T	8	1
2	PLUS	10	–
3	X	9	0
4	Y	9	0
5	Z	1	–
6	QUOTE	11	–

number table:

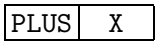
0	.5
1	1.6

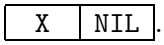
Then X is represented by the typed-pointer $oa(3)$ and NIL is represented by the typed-pointer $oa(0)$, both of which are negative integers. Remember that the type field in an atom table entry describes the type of the *value* of that ordinary atom. The S-expression (T . NIL) is represented by a positive integer j such that $P_j.car = oa(1)$ and $P_j.cdr = oa(0)$; that is (T . NIL) corresponds to j where $P_j = (oa(1), oa(0))$. Note $se(j) = j$.

We shall write the name of an atom in the *car* or *cdr* field of a pictorially-given list node to indicate that a non-positive typed-pointer to that atom is there. Thus (T . NIL) corresponds to j where $P_j =$ 

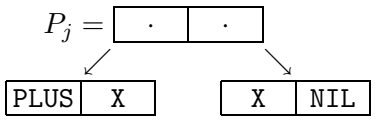
. This is intended to be a pictorial representation of the two integer fields which form the list area structure element P_j. Number atom names are used similarly. Thus (2 . 3) is represented by an integer j such that $P_j =$ 

. This means that $P_j.car$ is a negative integer x whose low 28 bits indexes the number atom 2 in the number table and $P_j.cdr$ is a negative integer y whose low 28 bits indexes the number atom 3 in the number table, so that $P_j = (x, y)$. The high four bits of x and y in this case are both 1001.

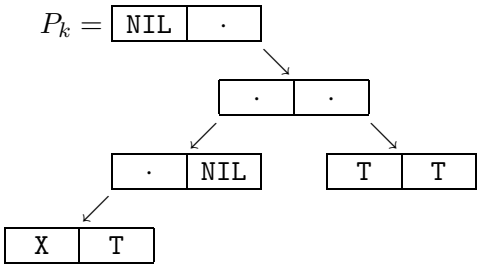
The S-expression ((PLUS . X) . (X . NIL)) is represented by a pointer j, where $P_j = (a, b)$ and $P_a =$ 

and $P_b =$ 

. This, of course, means that for the example atom table above, $P_a.car = oa(2)$, $P_a.cdr = oa(3)$, $P_b.car = oa(3)$, and $P_b.cdr = oa(0)$. Rather than introduce the intermediate pointers a and b by name, we usually shall show the same structure pictorially as:



As another example, the S-expression (NIL . (((X . T) . NIL) . (T . T))) is represented by a pointer k where



The following exercises assume the same example atom table used just above.

Exercise 7.1: What pointer represents v[`(QUOTE PLUS)`]?

Solution 7.1: $oa(2)$

Exercise 7.2: What pointer represents .5?

Solution 7.2: $nu(0)$.

Exercise 7.3: What pointer represents v[`(PLUS X X)`]?

Solution 7.3: We can't say exactly, but the result is the number 1 which is a number atom not shown above, so it must be a non-positive integer of the form $nu(j)$ for some integer $j > 1$.

Exercise 7.4: What is the S-expression represented by $se(3)$, where $P_3 = (se(1), oa(0))$ and $P_1 = (oa(1), se(2))$ and $P_2 = (oa(6), oa(6))$?

Solution 7.4: `((T . (QUOTE . QUOTE)) . NIL)`

Exercise 7.5: Is there any confusion between $oa(3)$ and $se(3)$?

Solution 7.5: No.

Exercise 7.6: What is the pictorial representation of `(((X . NIL) . NIL) . NIL)`?

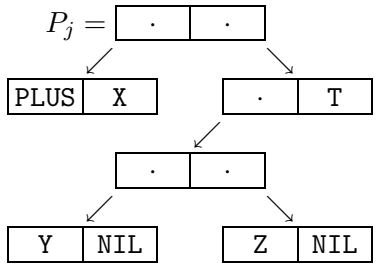
Exercise 7.7: What is the pictorial representation of `(X . (Y . (Z . NIL) . NIL) . NIL)`?

Solution 7.7: None. This is not a legal S-expression.

Exercise 7.8: What is the S-expression represented by the typed-pointer $oa(5)$?

Solution 7.8: `Z`.

Exercise 7.9: What is the S-expression represented by the positive integer j where



Exercise 7.10: Explain why non-atomic S-expressions can be described as binary trees whose terminal nodes are LISP atoms. How can the non-terminal nodes be characterized? Are there any structural constraints on the form of the binary trees which correspond to non-atomic S-expressions?

Exercise 7.11: What kinds of typed-pointers may occur in the car or cdr fields of an S-expression?

Solution 7.11: Dotted-pair(0000), ordinary atom(1000), and number atom(1001) typed-pointers comprise the only kinds of typed-pointers that appear in S-expressions.

8 More Functions

Now we may present some basic functions in LISP which operate on non-atomic, as well as atomic, S-expressions.

- `ATOM` – **predicate**

  $v[$`(ATOM` x`)`$] =$ if $v[x]$ is an ordinary atom or a number then `T` else `NIL`.

- `CONS` – **function**

  $v[$`(CONS` x y`)`$] =$ `(`$v[x]$ `.` $v[y]$`)`. x and y are arbitrary S-expressions. `CONS` is the dotted-pair construction operator.

- `CAR` – **function**

  $v[$`(CAR` x`)`$] = a$ where $v[x] =$ `(`a `.` b`)` for some S-expressions a and b. If $v[x]$ is not a non-atomic S-expression, then `(CAR` x`)` is undefined, and any result is erroneous.

- `CDR` – **function**

  $v[$`(CDR` x`)`$] = b$ where $v[x] =$ `(`a `.` b`)` for some S-expressions a and b. If $v[x]$ is not a non-atomic S-expression then `(CDR` x`)` is undefined, and any result is erroneous.

The basic relation among `CONS`, `CAR`, and `CDR` is

$$v[\texttt{(CONS (CAR } x\texttt{) (CDR } x\texttt{))}] = v[x]$$

when $v[x]$ is a non-atomic S-expression. The `CONS` function constructs a dotted-pair, the `CAR` function returns the first member of a dotted-pair and the `CDR` function returns the second member of a dotted-pair.

Exercise 8.1: Is the relation among `CONS`, `CAR` and `CDR` characterized by the statements: $v[$`(CAR (CONS` x y`))`$] = v[x]$ and $v[$`(CDR (CONS` x y`))`$] = v[y]$?

Names such as `FIRST` and `TAIL` would be more descriptive than `CAR` and `CDR`. The names `CAR` and `CDR` stand for the phrases "contents of the address register" and "contents of the decrement register" respectively. They arose because the first implementation of LISP programmed in 1958 was done

for an IBM 704 computer in which each list node was held in one 36-bit computer word. The single word instructions of the 704 had a format consisting of an op-code field, a decrement field and an address field, and in list nodes, these latter two fields were used for the second and first pointers, respectively, of a dotted-pair. The word "register" was somewhat cavalierly used instead of "field."

Exercise 8.2: Show how the names `FIRST` and `TAIL` can be used in place of the names `CAR` and `CDR`. Hint: use `SETQ`.

Exercise 8.3: Shouldn't the *car* and *cdr* fields in the list area elements P_i be called the *ar* and *dr* fields instead?

Exercise 8.4: What is v[`(CONS NIL T)`]?

Solution 8.4: `(NIL . T)`.

Exercise 8.5: What does the name "`CONS`" stand for?

Solution 8.5: It comes from the word "construction".

Exercise 8.6: What is v[`(CONS (CDR (CONS NIL T)) (CAR (CONS NIL T)))`]?

Solution 8.6: `(T . NIL)`.

Exercise 8.7: What is v[`(CAR PLUS)`]?

Solution 8.7: Undefined.

Exercise 8.8: What is v[`(CONS PLUS 3)`]?

Solution 8.8: Undefined, because v[`PLUS`] is not an S-expression! (go back and check the definition of an S-expression).

Exercise 8.9: What is v[`(CONS (QUOTE QUOTE) (QUOTE PLUS))`]?

Solution 8.9: `(QUOTE . PLUS)`.

Exercise 8.10: What is v[`(ATOM NIL)`]? What is v[`(ATOM (QUOTE NIL))`]? What is v[`(ATOM (QUOTE (QUOTE . NIL)))`]?

Solution 8.10: (1) `T`, (2) `T`, and (3) `NIL`.

Exercise 8.11: What is v[`(ATOM 12.5)`]?

Solution 8.11: `T`.

Exercise 8.12: What is v[`(ATOM PLUS)`]?

Solution 8.12: `NIL`. Because v[`PLUS`] is a function, *i.e.* a set of ordered pairs; it is *not* an atom.

9 Arguments and Results are Typed-Pointers

Internally in the LISP interpreter, callable LISP functions and special forms take typed-pointers to S-expressions as input and return a typed-pointer to an S-expression as output. The only (apparent) exceptions are those functions and special forms which accept or return functions or special forms, *i.e.* conceptually sets of ordered pairs; in fact, typed-pointers will be used in these cases as well. A final typed-pointer to the result is used to reach the pointed-to data object which is then inspected in order to print out its complete lexical representation as the final "deliverable."

We can now describe the workings of some of the functions and special forms defined before.

For `(QUOTE` x`)`, v[`QUOTE`] receives as input a typed-pointer to an S-expression x, and the same typed-pointer is returned as the result.

For `(PLUS` x y`)`, v[`PLUS`] receives two typed-pointers as input, one to $v[x]$ and one to $v[y]$ (remember, by the rules of LISP, v[`PLUS`] is a function, so x and y get evaluated before v[`PLUS`] is called). If the two typed-pointers received by v[`PLUS`] do not both point to number atoms, PLUS is being used erroneously. Otherwise the sum value is formed and a number atom with the corresponding floating-point value is formed. A typed-pointer to this number atom in the number table is the result.

For `(EQ` x y`)`, v[`EQ`] receives two typed-pointers as input, one to $v[x]$ and one to $v[y]$. If these two typed-pointers are identical, a typed-pointer to the atom `T` is returned, otherwise a typed-pointer to the atom `NIL` is returned. Thus `EQ` will correctly report whether or not two atoms are equal, since atoms are stored uniquely, but it may fail to detect that two dotted-pairs are equal (*i.e.* consist of the same atoms within the same tree shape). This failure will occur when two such equal dotted-pairs occur in different list nodes in memory. On the other hand, if we wish to test for identical, shared, dotted-pairs, `EQ` will serve this purpose.

For `(CAR` x`)`, v[`CAR`] receives a typed-pointer j to $v[x]$ as input. If $v[x]$ is not a dotted-pair, `CAR` is being used erroneously. Otherwise the typed-pointer $P_j.car$ is returned.

For `(CDR` x`)`, v[`CDR`] receives a typed-pointer k to $v[x]$ as input. If $v[x]$ is not a dotted-pair, `CDR` is being used erroneously. Otherwise the typed-pointer $P_k.cdr$ is returned.

For `(CONS` x y`)`, v[`CONS`] receives two typed-pointers as input; one typed-pointer j which points to $v[x]$ and one typed-pointer k which points to $v[y]$. An unoccupied list node P_h is obtained and $P_h.car$ is set to j and $P_h.cdr$ is set to k. Strictly, the types of the typed-pointers j and k must each be either 0 (dotted-pair), 8 (ordinary atom), or 9 (number atom). Then the typed-pointer $se(h)$ is returned. `CONS` is one of the few LISP functions which consume memory. `CONS` requires that a new list node be allocated at each invocation.

For `(SETQ` x y`)`, v[`SETQ`] receives two typed-pointers as input; one typed-pointer j which points to x and another typed-pointer k which points to y (remember `SETQ` is a special form). If j is positive or the value x pointed to by j is a number atom or a function or a special form, there is an error since x must be an ordinary atom. If j is not positive and the value x pointed to by j is an ordinary atom, then $v[y]$ is computed by applying `EVAL` to the S-expression y whose typed-pointer

is k. This results in a typed-pointer i which points to $v[y]$. Then the row determined by j in the atom table where the ordinary atom x resides has its value field set to the index corresponding to i and its typecode set to the type of $v[y]$ which is given in the typed-pointer i, except when $v[y]$ is an unnamed function or unnamed special form, in which case the ocurrence of the typecode 14 or 15 for $v[y]$ results in the newly-assigned value of x having, respectively, the typecode 12 or 13 instead.

Exercise 9.1: Specify an S-expression y such that $v[$`(SETQ A `y`)`$] = v[$`(SETQ A (QUOTE `y`))`$]$.

Solution 9.1: $y =$ `NIL` will suffice.

Exercise 9.2: What occurs when `(SETQ X PLUS)` is executed?

Solution 9.2: First recall that we interpret $v[$`PLUS`$]$ as a set of ordered pairs. Let us denote this set by {`PLUS`}. Then `SETQ` *redefines* the v-meta-operator so that $v[$`X`$]$ is now {`PLUS`}. Of course, within the LISP interpreter, {`PLUS`} is represented as a typed-pointer whose typecode is 10, and whose pointer part is some conventional value useful in identifying {`PLUS`}. After the `SETQ` application has been done, the row in the atom table where the ordinary atom `X` occurs has its type-field and value-field set to the typecode part and pointer part of this typed-pointer which represents {`PLUS`}.

Exercise 9.3: What is $v[$`(CONS (SETQ B T) (EVAL (SETQ A (QUOTE B))))`$]$?

Solution 9.3: `(T . T)`

10 List Notation

Some forms of non-atomic S-expressions arise so frequently there is a special notation for them. A dotted-pair of the form `(`S_1 `. (`S_2 `. (`$\cdots$ `. (`S_k `. NIL)`$\cdots$`))` where S_1, S_2, ..., S_k are all S-expressions is called a *list* and is written as `(`S_1 S_2 ... S_k`)`, which is the sequence of S-expressions S_1, S_2, ..., S_k written with intervening blanks and enclosed in parentheses. There is no confusion with dot notation since there are no dots between the S_i elements in the list. There may, of course, be dots within some or all of the elements S_1, ..., S_k, but then they are necessarily enclosed in parentheses and the dots occur at lower levels. Any element S_i which qualifies may itself be written in either list notation or dotted-pair notation.

The list `(`S_1`)` of the single element S_1 is written in dotted-pair notation as `(`S_1 `. NIL)`. By analogy, the atom `NIL` is used to denote the list of no elements. The symbol pattern "`()`" is also used to denote the empty list; it is to be understood as a synonym for `NIL`.

Exercise 10.1: Write the list `(A B)` in dot notation.

Solution 10.1: `(A . (B . NIL))`.

Exercise 10.2: How do we distinguish dots within ellipses from "true" dots used in dotted-pairs when writing text about S-expressions?

Exercise 10.3: Write the S-expression `(1 . (2 . (3 . NIL)))` in list notation.

Solution 10.3: `(1 2 3)`.

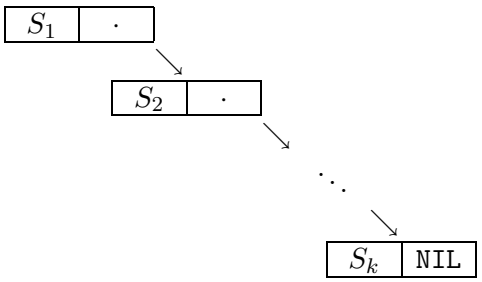
Exercise 10.4: Is every list a dotted-pair?

Solution 10.4: All lists are dotted-pairs except the empty list, `NIL`. The empty list is represented by an atom.

Exercise 10.5: Is every dotted-pair a list?

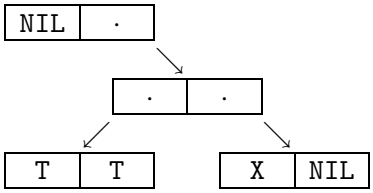
Solution 10.5: No. Only non-atomic S-expressions with a right-embedded `NIL` as shown above are lists. For example `(X . X)` is not a list, and neither is `(NIL . 3)`.

Pictorially a non-empty list $(S_1 \ \ldots \ S_k)$ is of the form



Exercise 10.6: What is the pictorial form and dot notation form of the 3 element list `(NIL (T . T) X)`?

Solution 10.6:



The dot notation form is: `(NIL . ((T . T) . (X . NIL)))`.

Exercise 10.7: How many final close-parentheses occur together at the end of a k element list written in dot notation?

Solution 10.7: Exactly k final close-parentheses occur.

Exercise 10.8: Is `NIL` a list of 0 elements?

Solution 10.8: `NIL` *represents* the empty list, but `NIL` *is* an ordinary atom. The class of lists consists of one atom, `NIL`, and an infinite number of certain non-atomic S-expressions.

Exercise 10.9: If s is a list, and x is an S-expression, what is $z = v[\texttt{(CONS } x \ s\texttt{)}]$?

Solution 10.9: z is a list. Its first element is x and its remainder is the list s. Thus CONS can be used to construct a new list formed by adding an additional element at the head of a given list. In this example, the list s shares all its list nodes with the list z, which has an additional initial node. Such sharing of list nodes is common in S-expressions and is an elegant feature of LISP.

A list whose elements consist of atoms or lists, where these lists are recursively constrained in the same manner, is called a *pure list*. A pure list can be written entirely in list notation without dots appearing at any level.

Note that the notation used for specifying function application statements as LISP interpreter input is just list notation for certain S-expressions involving certain atoms! Thus almost every legal LISP input statement is an S-expression which is either an atom or a list which itself has elements that are either atoms or lists. The only exception is that arguments to QUOTE can be arbitrary S-expressions. This is often summarized by saying that, in LISP, "programs are data". Some "data" are "programs" too, but not every S-expression is legal LISP input.

Exercise 10.10: What is v[(EVAL (CONS (QUOTE CONS) (CONS 2 (CONS 3 NIL))))]?

Solution 10.10: (2 . 3).

Exercise 10.11: What is v[(EVAL (QUOTE (CONS (QUOTE CAR) (QUOTE (A . B)))))]?

Solution 10.11: The result is the same as

```
v[(EVAL . ((QUOTE
            . ((CONS . ((QUOTE . (CAR . NIL))
                        . ((QUOTE . ((A . B) . NIL)) . NIL)))
               . NIL)) . NIL))]
  = (CAR . (A . B)).
```

Exercise 10.12: What is v[(T . NIL)]?

Solution 10.12: Undefined. v[T] is not a function or special form as is required for v to be defined on this form of input.

Remember that a function does not have to be designed to accept any member of the set of S-expressions as input. We have functions which only work on atoms, or only on dotted-pairs, and a programmer can circumscribe the legal input in almost any manner desired. A function which is designed to work on a proper subset of S-expressions is called a *partial* function (in contrast to a total function). What happens when a partial function is called with one or more illegal input arguments is implementation dependent. Ideally a partial function would be programmed to check its arguments and reject any illegal input with an accompanying error message. However, even when this is possible, it is often too inefficient, and thus many builtin LISP functions and special forms may give nonsense results, or loop, or crash when given illegal arguments. We summarize this by saying such functions and special forms are undefined on illegal input.

Now we may define the builtin function LIST. Unlike the previous functions and special forms we have seen, LIST may be invoked with differing numbers of arguments.

- `LIST` – **function with a varying number of arguments**

  v[`(LIST` x_1 x_2 ... x_k`)`] = v[`(CONS` x_1 `(CONS` x_2 `(CONS ... (CONS` x_k `NIL))...))`].

  Exercise 10.13: What is v[`(LIST NIL)`]? What is v[`(LIST)`]?

  Solution 10.13: v[`(LIST NIL)`] = `(NIL)` = `(NIL . NIL)`. v[`(LIST)`] could consistently be defined to be `NIL`, but as we have defined the function `LIST` above, v[`(LIST)`] is undefined.

11 More Special Forms

- `AND` – **special form with a varying number of arguments**

  v[`(AND` x_1 x_2 ... x_k`)`] = if $v[x_1] \neq$ `NIL` and $v[x_2] \neq$ `NIL` and ... $v[x_k] \neq$ `NIL` then `T` else `NIL`.
  The special form `AND` is evaluated using lazy evaluation; this means that the arguments are evaluated and tested against `NIL` from left to right and the first `NIL`-valued argument is the last argument evaluated.

- `OR` – **special form with a varying number of arguments**

  v[`(OR` x_1 x_2 ... x_k`)`] = if $v[x_1] \neq$ `NIL` or $v[x_2] \neq$ `NIL` or ... or $v[x_k] \neq$ `NIL` then `T` else `NIL`.
  The special form `OR` is evaluated using lazy evaluation; the arguments are evaluated and tested against `NIL` from left to right and the first non-`NIL`-valued argument is the last argument evaluated.

  Exercise 11.1: What should v[`(AND)`] and v[`(OR)`] be defined to be?

  Solution 11.1: One consistent choice is to define v[`(AND)`] = `T` and v[`(OR)`] = `NIL`. This is somewhat analogous to the definition that an empty sum is 0 and an empty product is 1.

- `COND` – **special form with a varying number of arguments**

  v[`(COND (`p_1 q_1`) (`p_2 q_2`) ... (`p_k q_k`))`] = if $v[p_1] \neq$ `NIL` then $v[q_1]$, else if $v[p_2] \neq$ `NIL` then $v[q_2]$, ..., else if $v[p_k] \neq$ `NIL` then $v[q_k]$, else `NIL`.
  `COND` stands for "conditional". It is the primary branching operator in LISP. Each argument to `COND` must be a two element list whose elements are potentially evaluatable. The special form `COND` is evaluated using lazy evaluation; the length-two list arguments are examined from left to right and the first component of each is evaluated until the resulting value is not `NIL`, then the value of the associated second component is returned. Only as many p_i's as needed are evaluated, and at most one q_i is evaluated.

The special form COND is necessary both theoretically and practically. With COND, we can in principle, dispense with AND and OR. Note

$$v[\texttt{(AND } a\ b\texttt{)}] = v[\texttt{(COND ((EQ NIL } a\texttt{) NIL) ((EQ NIL } b\texttt{) NIL) (T T))}], \quad \text{and}$$
$$v[\texttt{(OR } a\ b\texttt{)}] = v[\texttt{(COND ((EQ } b\texttt{ NIL) (COND ((EQ } a\texttt{ NIL) NIL) (T T))) (T T))}].$$

In [McC78], McCarthy writes: "I invented conditional expressions in connection with a set of chess legal move routines I wrote in FORTRAN for the IBM 704 at M.I.T. during 1957–58. This program did not use list processing. The IF statement provided in FORTRAN 1 and FORTRAN 2 was very awkward to use, and it was natural to invent a function XIF(M,N1,N2) whose value was N1 or N2 according to whether the expression M was zero or not. The function shortened many programs and made them easier to understand, but it had to be used sparingly, because all three arguments had to be evaluated before XIF was entered, since XIF was called as an ordinary FORTRAN function though written in machine language. This led to the invention of the true conditional expression which evaluates only one of N1 and N2 according to whether M is true or false and to a desire for a programming language that would allow its use."

Exercise 11.2: What is v[(OR (COND ((EQ NIL T) NIL) (T NIL)) NIL)]?

Solution 11.2: NIL.

Exercise 11.3: Do AND and OR require that arguments which evaluate to NIL or T be supplied?

Solution 11.3: No.

Exercise 11.4: What is v[(AND (COND ((SETQ A NIL) T) (T T) ((SETQ A T) T)) A (SETQ A 0))]?

Solution 11.4: NIL, and v[A] = NIL afterwards as a result of the assignment side-effect.

Exercise 11.5: Suppose A is an ordinary atom which has been assigned a number value with an application of SETQ. Write a LISP functional application command which causes the ceiling of the value of A to be printed-out.

Solution 11.5: (COND ((EQ (FLOOR A) A) A) (T (PLUS 1 (FLOOR A))))

Exercise 11.6: What would be the effect if T were to be redefined so as to have the value 3 rather than itself?

Solution 11.6: The effect would be minor. But leaving T undefined would not be so benign. And redefining NIL to be 3 would seriously damage LISP.

You may have noticed that we have lapsed into the usual sloppiness found in discussions of programming languages (for good reason). We say "the special form COND is necessary ..." when we should say "the special form v[COND], whose name is COND, is necessary ...". It is convenient to agree to tolerate such ambiguity.

12 Defining Functions: λ-Expressions

We can use the LISP interpreter to compute the value of combinations of builtin functions and special forms applied to arguments, but to use LISP as a programming language rather than as a curious kind of calculator, we must have a way to *define* functions of our own choosing and use them, rather than just use unnamed compositions of pre-existing functions.

The special form `LAMBDA` is used in LISP to *create* a user-defined function. `LAMBDA` takes two arguments which are both S-expressions. The first argument is a list of ordinary atoms denoting the *formal arguments* of the function being defined, and the second argument is an S-expression expressing the definition of the function being defined. This second argument is required to be legal LISP input, so it is either an atom or a functional application expression. This second argument is called the *body* of the function being defined.

A list expressing an application of the special form `LAMBDA` is called a LISP λ-expression. For example, the λ-expression `(LAMBDA (X Y) (CONS Y X))` denotes a function. That means, conceptually, its value is a set of ordered pairs. It is the function which, given two S-expressions, a and b, as input, returns the dotted-pair $(b\ .\ a)$ as output. Note this function has no name. In order to use this function in LISP we can enter `((LAMBDA (X Y) (CONS Y X)) 2 3)` to the LISP interpreter, and `(3 . 2)` will be typed-out. In the λ-expression `(LAMBDA (X Y) (CONS Y X))`, the S-expression `(CONS Y X)` is the *body* and `(X Y)` is the *list of formal arguments*.

Continuing this example, the evaluation of `((LAMBDA (X Y) (CONS Y X)) 2 3)` proceeds by *binding* the value of the first actual argument 2 to the first formal argument `X`, and *binding* the value of the second actual argument 3 to the second formal argument `Y`, and then evaluating the body `(CONS Y X)` by computing $v[$`(CONS Y X)`$]$ with the understanding that each occurrence of `X` in the body is evaluated to yield its associated bound value 2 and each occurrence of `Y` in the body is evaluated to yield its associated bound value 3. This means that $v[$`X`$] = 2$ and $v[$`Y`$] = 3$ in the body expression `(CONS Y X)`, regardless of the global values of `X` and `Y` in the atom table.

Let a be a list of $k \geq 0$ ordinary atoms and let b be an evaluatable S-expression. In general, the value of the λ-expression `(LAMBDA` a b`)` is defined so that $v[$`(LAMBDA` a b`)`$] =$ the function which is computed on a list of k actual arguments r by computing $e[b, a, r]$, where $e[b, a, r] = v[b]$ in the *context* such that $v[a_i] = r_i$ for $1 \leq i \leq k$. The symbol e stands for "environmental evaluation". It is a form of the v-operator which depends upon a *context* specified by a binding of actual argument values to formal arguments.

It is awkward to write a λ-expression to specify a function for each use, so we adopt a device to give names to functions. The value of a λ-expression can be assigned a name using `SETQ`. Thus for example, evaluating `(SETQ G (LAMBDA (X Y) (CONS Y X)))` results in the ordinary atom `G` having the value $v[$`(LAMBDA (X Y) (CONS Y X))`$]$. The result which is typed-out is conventionally understood to be the set of ordered pairs conceptually assigned to `G` which is $v[$`(LAMBDA (X Y) (CONS Y X))`$]$ and this is denoted by "{`user function: G`}". Now we can write `(G 2 3)` to denote `(3 . 2)`.

If you look at various LISP interpreters, you will find that a special form variously called `DEFINE` or `DEFUN` (for "define-function") is commonly used to assign a function to be the value of an ordinary

atom, but there is no reason SETQ can't serve this purpose, so we eschew DEFUN in this book. The use of DEFUN is often coupled with a mechanism for allowing an ordinary atom to have an S-expression value *and* a function value simultaneously. This option is not possible as we have defined the atom table and doesn't seem very felicitous in any event.

Note that the definition of SETQ is that the result is the value of its first argument *after* assignment. This means that the first argument already has a user-defined *named* function as its value which appears as the result when SETQ is used to assign a function to an ordinary atom.

The most interesting case of defining and naming a function arises when recursion is utilized. For example,

```
(SETQ FACT (LAMBDA (X)
                   (COND ((EQ X 0) 1)
                         (T (TIMES X (FACT (DIFFERENCE X 1)))))))
```

defines the factorial function $fact(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$. This definition is an *impredicative* definition; that means the name FACT, which is being assigned a value, is itself used in defining that value. This is an uncomfortable state of affairs, but completely understandable pragmatically.

What the special form LAMBDA really does is to CONS its two arguments together and return an unnamed-function-typed-pointer to the resulting dotted-pair; this is a typed-pointer whose typecode is 14. The arguments themselves are not examined until the function is actually applied. If this unnamed function which is represented as a dotted-pair is assigned as the value of an ordinary atom b, the value of b is then just the pointer to the dotted-pair, and the type of this value is set to 12 to indicate a user-defined function. Thus, for example, when FACT as assigned above is used, the recursive reference to FACT within the body is correctly interpreted because the value of FACT *at the time of evaluation of the body* is consistently defined.

Remember that $v[$(SETQ x y)$]$ has been defined to be $v[x]$ after $v[x]$ is made equal to $v[y]$; but when y is a λ-expression, this does not hold. In this case, $v[y]$ is a typed-pointer to the CONSed dotted-pair representing the value of the λ-expression y. The typecode of this typed-pointer is 14. However, after the assignment to redefine $v[x]$, $v[x]$ is a typed-pointer to the same dotted-pair, but the typecode of $v[x]$ is 12 rather than 14. Thus in the case of assigning the value of a λ-expression y to an ordinary atom x, the typecode of $v[x]$ after the assignment is *not* the typecode of $v[y]$. This awkward exception arises because functions need not have names. If names were always required, this distinction encoded in typecode values would be unnecessary.

Exercise 12.1: What is $v[$((LAMBDA (X) X) y)$]$?

Solution 12.1: $v[y]$. So typing ((LAMBDA (X) X) y) produces the same result as typing y.

Exercise 12.2: What does (SETQ CONS (LAMBDA (X) (CONS 1 X))) do?

Solution 12.2: The function CONS is redefined to be a non-terminating recursive function. The CONS in the body refers to the same atom as the atom being reassigned a value, and in the future, as at all other times, that atom has just one value. Most versions of LISP do not permit atoms whose values are builtin functions or special forms to be redefined in this manner.

Exercise 12.3: (J. McCarthy) Specify an input non-atomic S-expression, a, which has itself as its value.

Solution 12.3: $a =$ `((LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))` .
`(QUOTE (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))))`

Exercise 12.4: Is `(LAMBDA NIL 3.1415926)` a legal λ-expression?

Solution 12.4: Yes.

The name `LAMBDA` and the term λ-expression are borrowed from the so-called λ-calculus of Alonzo Church [Kle52]. This is a mathematical invention used to explore the syntactic and semantic nature and power of substitution. It is, in some sense, a theory of macro languages.

13 More Functions

There are many builtin functions in LISP which are not logically required to be builtin. They are there for convenience, and in some cases because they are faster that way. The three functions presented below are builtin functions which have definitions in terms of other more basic functions and special forms.

- `APPEND` − **function**

  Defined by:

```
(SETQ APPEND
      (LAMBDA (X Y)
              (COND ((EQ X NIL) Y)
                    ((ATOM X) (CONS X Y))
                    (T (CONS (CAR X) (APPEND (CDR X) Y))))))
```

  This version of `APPEND` is slightly more general than the commonly-found definition which omits the `((ATOM X) (CONS X Y))` pair. Given two *lists* $(a_1\ a_2\ \ldots\ a_h)$ and $(b_1\ b_2\ \ldots\ b_k)$ as input, the list $(a_1\ a_2\ \ldots\ a_h\ b_1\ b_2\ \ldots\ b_k)$ is produced as the output. The input is not damaged in any way. In a sense, this function embodies the essence of LISP. When you understand in detail at the machine level what happens during the application of this function to arguments, then you understand the essence of LISP.

Exercise 13.1: What is v[`(APPEND (QUOTE (A . NIL)) T)`]? What is v[`(APPEND (QUOTE (A B)) NIL)`]? What is v[`(APPEND T T)`]?

Exercise 13.2: Suppose the ordinary atom `X` has the value `(NIL . (T . NIL))` and that the ordinary atom `Y` has the value `(X . (Y . NIL))`, where these S-expressions are stored in the list area as follows.

atom table				
	name	type	value	
1	NIL	8	1	–
2	T	8	2	–
3	X	0	1	–
4	Y	0	2	–

list area	
1	$(-1, 3)$
2	$(-3, 4)$
3	$(-2, -1)$
4	$(-4, -1)$
5	first free

Assume new list area nodes are allocated and used in the order 5, 6, ..., etc. Tabulate the changes which occur in the list area when `(APPEND X Y)` is executed, and present the resulting contents of the list area. (Actually, the typed-pointer value denoted by -1 is $oa(1) = 1000 :: 1$. What exactly is the typed-pointer denoted by -3?)

- `REVERSE` – **function**

  Defined by:

```
(SETQ REVERSE
      (LAMBDA (X)
              (COND ((ATOM X) X)
                    (T (APPEND (REVERSE (CDR X)) (CONS (CAR X) NIL))))))
```

Exercise 13.3: Suppose the functions `A` and `B` are defined by

```
(SETQ A (LAMBDA (X Y)
                (COND ((EQ X NIL) Y)
                      (T (B (REVERSE X) Y)))))
```

and

```
(SETQ B (LAMBDA (X Y)
                (COND ((EQ X NIL) Y)
                      (T (B (CDR X) (CONS (CAR X) Y))))))
```

What does `A` do?

Solution 13.3: The function `A` is another version of `APPEND`. It behaves the same way `APPEND` does on list arguments.

- `EQUAL` – **function**

  Defined by:

```
(SETQ EQUAL (LAMBDA (X Y)
                    (COND ((OR (ATOM X) (ATOM Y)) (EQ X Y))
                          ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y)))
                          (T NIL))))
```

Exercise 13.4: Is the last `(T NIL)` pair appearing in the definition of `EQUAL` necessary?

Solution 13.4: No, as we have defined COND herein it is not necessary, but it is harmless, and it is required in some dialects of LISP where a COND does not have a NIL final value by default.

Exercise 13.5: Does the EQUAL function consume list area nodes during its application?

Solution 13.5: No. The CONS function is not applied directly or indirectly.

Exercise 13.6: Define a LISP function LENGTH which takes a list as input and returns the number of elements in the list as output.

Exercise 13.7: Define a LISP predicate LISTP which returns T if its assignment is a list and which returns NIL otherwise.

Solution 13.7: Define LISTP by

```
(SETQ LISTP (LAMBDA (S) (COND ((ATOM S) (EQ S NIL)) (T (LISTP (CDR S))))))
```

Exercise 13.8: Define a LISP predicate MEMBER which returns T if the input S-expression, a, is an element of the input list, s, and which returns NIL otherwise.

Solution 13.8: Define MEMBER by:

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

Exercise 13.9: Define a LISP function PLACE, which is like MEMBER in that an input list s is searched for an S-expression a, but the remainder of the list s after the point at which a is found is returned as the result, or the atom NULL is returned as the result if a is not an element of s. Why is NULL specified as the atom which is returned to indicate a failure?

Exercise 13.10: Define a LISP predicate DEEPMEM which returns T if the input S-expression a occurs as an element of the input list s, or as an element of any sublist of s at any level, and returns NIL otherwise. You may assume that s is a pure list, all of whose elements are atoms or pure lists.

Solution 13.10: Define DEEPMEM by:

```
(SETQ DEEPMEM (LAMBDA (A S)
                      (COND ((ATOM S) NIL)
                            ((OR (EQUAL A (CAR S)) (DEEPMEM A (CAR S))) T)
                            (T (DEEPMEM A (CDR S))))))
```

Exercise 13.11: Define a LISP function TRUNC which takes a non-empty list s as input and returns a list identical to s, but with the last element removed.

Exercise 13.12: Define a LISP function DEPTH which takes an S-expression a as input and returns 0 if a is an atom and returns the number of levels in the binary tree picture of a otherwise. Thus the depth of the S-expression a is the maximum number of parentheses pairs which enclose an atom in a.

Exercise 13.13: What happens when `((SETQ G (LAMBDA (X) (CONS X (G X)))) 3)` is typed-in to the LISP interpreter?

Solution 13.13: First the value of the ordinary atom `G` will become the specified function, and then the LISP interpreter will run out of stack space because of the infinite recursion specified in `G` applied to 3. Note that the list area will not be exhausted, since no `CONS` application is ever actually consumated.

Exercise 13.14: Define a *flat* list to be a list whose elements are atoms. Write a LISP function `FLATP` to test whether an S-expression is flat. Write another LISP function `FLAT` which returns a flat list containing all the atoms found within the input S-expression x.

Exercise 13.15: Define a LISP predicate `PURE` which takes an S-expression x as input and returns `T` if x is a pure list and returns `NIL` otherwise.

Exercise 13.16: Can there be two list area entries P_i and P_j with $i \neq j$ such that $P_i = P_j$? That is, does list-node sharing preclude duplicate list-nodes existing?

Solution 13.16: Yes, duplicate list-nodes can exist. But it is interesting to contemplate ways in which duplicate list-nodes can be avoided.

14 Defining Special Forms

User-defined special forms are created by applying the special form `SPECIAL`, which is completely analogous to the special form `LAMBDA`, except that an unnamed-special-form-typed-pointer to the dotted-pair of the input arguments is returned, so that the result will, upon later use, be interpreted as a special form rather than a function. An unnamed-special-form-typed-pointer has the typecode 15.

Thus the body and formal argument list of a special form created by `SPECIAL` are represented by `CONS`ing the input arguments together to form a dotted-pair and an unnamed-special-form-typed-pointer whose pointer part indexes this dotted-pair is returned. Just as with λ-expressions, the typecode of the value of an ordinary atom which is assigned an unnamed-special-form is forced to be 13 rather than 15. The value 13 is the typecode denoting a named special form. The newly-reset value field contains a pointer to the dotted-pair of the argument list and body of the special form.

Let a be a list of $k \geq 0$ ordinary atoms and let b be an evaluatable S-expression. In general, the value of the special form-expression `(SPECIAL` a b`)` is defined as v[`(SPECIAL` a b`)`] = the function which is computed on a list of k actual arguments r by computing $e[b, a, r]$, where $e[b, a, r] = v[b]$ in the context such that $v[a_i] = r_i$ for $1 \leq i \leq k$. Unlike a λ-expression, the actual arguments r are taken as they are given, and are not computed by evaluating the given actual arguments.

It is important to note that the definition of the value, *e.g.* the meaning, of a LISP special form or function in terms of a context is *not* the same as the meaning which is induced using the often-used *substitution* rule used in Algol, where we would say that v[`(SPECIAL` a b`)`] is that function which is

computed on a list of k actual arguments r by computing the value of b with r_i substituted for each occurrence of a_i in b for $1 \leq i \leq k$. Binding temporarily supercedes meanings of symbols in the atom table, as seen by every ordinary atom evaluation which is done while the binding is in force, whereas substitution does not cover up such meanings everywhere.

For example, `(SETQ EVALQUOTE (SPECIAL (X) (EVAL X)))` defines the special form `EVALQUOTE` which is used in the following exercise.

Exercise 14.1: Suppose the atom `A` has the undefined value. What output results from the following commands given in sequence?

```
(SETQ B (QUOTE A))
(EVALQUOTE B)
(EVAL (QUOTE B))
(EVAL B)
```

Solution 14.1: The four successive outputs which result are: (1) `A`, (2) `A`, (3) `A`, and (4) undefined.

Exercise 14.2: Suppose the atom `X` has the undefined value. Keep in mind that `EVALQUOTE` is defined with a formal argument `X`. What output results from the following commands given in sequence?

```
(EVALQUOTE X)
(SETQ X 3)
(EVAL (QUOTE X))
(EVALQUOTE X)
```

Solution 14.2: The four successive outputs which result are: (1) `X`, (2) 3, (3) 3, and (4) `X`.

Exercise 14.3: Define the special form `SETQQ` which assigns the value of its unevaluated second argument to its first atom-valued unevaluated argument so as to become the value of the first atom.

Solution 14.3:

```
(SETQ SETQQ
      (SPECIAL (X Y)
               (EVAL (CONS (QUOTE SETQ)
                           (CONS X (CONS (CONS (QUOTE QUOTE)
                                               (CONS Y NIL))
                                         NIL))))))
```

This special form operates so that the name bound to `X` is used independently of any context. Thus for example, `(SETQQ P Q)` sets the current value of the atom `P` equal to the atom `Q` in any context, in or out of a function body or special form body. However, this current value is lost when the context in which `P` has been assigned a bound value terminates.

Exercise 14.4: Define the function `SET` which assigns its evaluated second argument to become the value of its atom-valued evaluated first argument.

Solution 14.4:

```
(SETQ SET (LAMBDA (X Y)
                  (EVAL (CONS (QUOTE SETQ)
                              (CONS X (CONS (QUOTE Y) NIL))))))
```

Another solution is:

```
(SETQ SET (SPECIAL (X Y)
                   (EVAL (CONS (QUOTE SETQ)
                               (CONS (EVAL X) (CONS Y NIL))))))
```

Exercise 14.5: "SETQ" stands for "set-quote"; explain how the special form QUOTE is related to SETQ.

Because of the possibly counter-intuitive behavior of some functions or special forms when applied to arguments whose names are also used as formal argument names, where the substitution rule and the binding rule give differing results, such functions and special forms should be used sparingly. It is a good idea to have useful functions and special forms like SET and EVALQUOTE *builtin* to the LISP interpreter so that no such formal argument name conflicts can arise.

User-defined and unnamed functions and special forms are stored as simple dotted-pair S-expressions formed from the S-expressions given to define them. It is sometimes useful to explicitly look at the defining dotted-pair of a user-defined or unnamed function or special form. This can be done with the function BODY, where $v[$(BODY x)$] =$ the dotted-pair which encodes the definition of the user-defined or unnamed function or special form $v[x]$.

Exercise 14.6: What output will appear as the result of the following input?

```
(SETQ E (LAMBDA (X) (MINUS X)))
(BODY E)
```

Solution 14.6: {user-defined function: E} and ((X) MINUS X).

Exercise 14.7: Define the special form IF such that:
$v[$(IF a b c)$]$ = if $v[a]$ = NIL then $v[c]$ else $v[b]$.

Solution 14.7:

```
(SETQ IF (SPECIAL (A B C)
                  (COND ((EVAL A) (EVAL B))
                        (T (EVAL C)))))
```

15 The Label Special Form

Recursive functions or special forms can be named and defined in LISP using SETQ. This is convenient, and we almost always define functions and special forms this way in practice. But this leaves a theoretical difficulty about the LAMBDA operator, namely, we *must* use SETQ and assign a name in order to define a recursive function. Thus not every desired function can, in theory, be written in place as a λ-expression. In order to dispose of this difficulty, the special form LABEL is introduced.

- `LABEL` – **special form**

  $v[$`(LABEL` $g\ h\ a_1\ \ldots\ a_k$`)`$] = e[$`(`$h\ v[a_1]\ \ldots\ v[a_k]$`)`$, g, h]$, where g is an ordinary atom and h is a λ-expression and $a_1, \ldots, a_k$ are S-expressions. The idea is that `(LABEL` $g\ h\ a_1\ \ldots\ a_k$`)` $= v[$`(`$h\ a_1\ \ldots\ a_k$`)`$]$ in the context where the atom g is evaluated as the λ-expression h at every ocurrence of g in the body of the λ-expression h. Thus g is, in effect, the name or label of h in the body of h.

The extension of the definition of `LABEL` to apply to special forms, is left as an exercise, since `LABEL` is primarily needed to provide a pleasant theoretical closure and is not often used in practice, and also since, except for `QUOTE`, special forms are, in theory, avoidable by using `QUOTE` to protect the arguments of corresponding λ-expressions.

Exercise 15.1: What is

$v[$
```
(LABEL FACTL
       (LAMBDA (X) (COND ((EQ X 0) 1)
                         (T (TIMES X (FACTL (DIFFERENCE X 1))))))
       3)
```
$]$?

Exercise 15.2: Carefully state the appropriate definition of `(LABEL` $g\ h\ a_1\ \ldots\ a_k$`)` where g is an ordinary atom and h is a special-expression.

16 The Quote Macro

The special form `QUOTE` is used so frequently that a special notation is provided. The single quote or apostrophe symbol is used to denote a unary prefix operator, defined so that $'e =$ `(QUOTE` e`)`. Thus for example, the `SET` function given above can be defined by the input:

```
(SETQ SET (LAMBDA (X Y)
                  (EVAL (CONS 'SETQ (CONS X (CONS 'Y NIL))))))
```

The single quote symbol acts as a macro; it is expanded upon input. Thus ' cannot be manipulated as an atom within LISP functions. Remember ' does not mean "`QUOTE`" standing alone; it must take an argument.

Exercise 16.1: Can the expression `'(QUOTE` e`)` be written as $''e$?

Solution 16.1: Yes.

Exercise 16.2: Describe the objects x which satisfy $v['x] = v[x]$.

Solution 16.2: $\{\, x \mid v['x] = v[x] \,\} = \{\, y \mid v[y] = y \,\}$, which is the set of all number atoms, LISP-functions, LISP-special-forms, and certain non-atomic S-expressions that evaluate to themselves.

17 More Functions

- `NOT` – **predicate**

  Defined by: `(SETQ NOT (LAMBDA (X) (EQ X NIL)))`.

- `NULL` – **predicate**

  Defined by: `(SETQ NULL (LAMBDA (X) (NOT X)))`.

  Exercise 17.1: Show that $v[$`(NOT (NULL` y`))`$] = v[$`(AND` y`)`$]$.

- `GREATERP` – **predicate**

  $v[$`(GREATERP` n m`)`$]$ = if $v[n] > v[m]$ then `T` else `NIL`. $v[n]$ and $v[m]$ must be numbers.

- `LESSP` – **predicate**

  $v[$`(LESSP` n m`)`$]$ = if $v[n] < v[m]$ then `T` else `NIL`. $v[n]$ and $v[m]$ must be numbers.

  Exercise 17.2: Show that when x and y are integers,

  $$v[\texttt{(GREATERP } x\ y\texttt{)}] = v[\texttt{((LAMBDA (X Y) (GCHECK (DIFFERENCE (DIFFERENCE X Y) 1),}\\ \texttt{(DIFFERENCE Y X))) } x\ y\texttt{)}]$$

  where `GCHECK` is defined by

```
(SETQ GCHECK (LAMBDA (A B)
                     (COND ((EQ A 0) T)
                           ((EQ B 0) NIL)
                           (T (GCHECK (PLUS A -1) (PLUS B -1))))))
```

  Exercise 17.3: Define the function `GCD` in LISP, where $v[$`(GCD` a b`)`$]$ is the greatest common positive integer divisor of the two positive integers a and b. Hint: investigate the Euclidean Algorithm.

18 More About Typed-Pointers

In order to handle number atoms efficiently as values within the LISP interpreter, we have used pointers to numbers in sketching the working of the v-operator. This use of pointers is also required to allow functions to be treated as values. The guiding idea is that the LISP interpreter needs to know, at least potentially, the name or lexical expression associated with any value being manipulated, and every such value is represented by a typed-pointer to some complete representation of that value.

In particular, then, a number must be represented in a way which permits the proper result when printing it, using it as an argument in a computation, or as a CONS argument, and when creating a new number as an intermediate result.

Consider:

v[3.1],
v[A], where v[A] has been made equal to 3 via (SETQ A 3),
v[(PLUS 2 3)],
v[(PLUS 2 A)],
v[(CONS (PLUS 2 A) 1)].

Whenever a number is computationally created, by evaluating (PLUS 2 3) for example, that number is entered in an available row in the number table, if and only if it is not already present. For such computationally-created numbers, some time can be saved by *not* constructing the number-name text-string; in fact, for the sake of uniformity, we discard the text-string name of input numbers. Such nameless number atoms are called *lazy* number atoms. If a lazy number atom must be printed out, we only then construct its text-string name. Also, the computationally-created number atoms which are no longer needed should, from time to time, be removed from the number table.

Similarly, a function or special form must be represented in a way which permits the proper result when applying it, printing it, or using it as an argument in a computation.

Consider:

v[PLUS],
v[(PLUS 2 3)],
v[(EVAL (APPEND 'PLUS '(2 3)))],
v[(EVAL (APPEND PLUS '(2 3)))].

An appropriate internal representation of a function such as v[PLUS] is as a typed-pointer to the atom PLUS, where this typed-pointer has the typecode 10, as opposed to 8, indicating that it refers to a builtin function which is the value of the pointed-to atom. In this way, we know both the function and its name.

Thus, a builtin function which is the value of an ordinary atom x stored at an entry j in the atom table is represented by the typed-pointer $bf(j)$. This atom table entry appears as $[x, 10, t, -, -]$. The value field entry t is an integer which indicates the particular builtin function which is the value of x. In contrast, the ordinary atom x is represented by the typed-pointer $oa(j)$.

Also, a builtin special-form which is the value of an ordinary atom x stored at an entry j in the atom table is represented by the typed-pointer $bs(j)$. This atom table entry appears as $[x, 11, t, -, -]$. The value field entry t is an integer which indicates the particular builtin special-form which is the value of x.

A user-defined function is constructed by the special form LAMBDA by CONSing together the argument list and body S-expressions given as input and returning a typed-pointer to this constructed S-expression in the list area. This typed-pointer has the typecode 14, indicating an *unnamed function*. Similarly, the special form SPECIAL constructs a dotted-pair and returns a typed-pointer to this dotted-pair with the typecode 15, indicating an *unnamed special form*.

When SETQ assigns an unnamed function or special-form to an ordinary atom, the value of that atom is henceforth represented by a typed-pointer to that atom whose typecode is 12 for a user-defined function and 13 for a user-defined special-form. Suppose the ordinary atom F occupies row i in the atom table and suppose we evaluate (SETQ F (LAMBDA (X) 1)). Then the ordinary atom F has $i : [\text{F}, 12, j, -, -]$ as its row in the atom table, where $P_j =$ | k | 1 | and $P_k =$ | X | NIL |. The typed-pointer representing $v[\text{F}]$ then has 12 as its typecode and i as its pointer part. This allows us to know the name as well as the defining S-expression of the function which is the value of F.

Exercise 18.1: Can the builtin function EQ be used to test whether two functions are equal?

Solution 18.1: Two typed-pointers can be checked for equality with EQ, and thus EQ can be used to determine when exactly the same named identical builtin or user-defined function or special forms are presented. Identically defined user-defined functions or special forms which have different names (or different defining dotted-pairs if they are unnamed) cannot be successfully tested for equality with EQ. Of course, the general logical problem of deciding when two functions specified by formulas are the same is undecidable, although many special cases can be handled.

Exercise 18.2: How could two unnamed functions with the same shared defining dotted-pair be presented to EQ for comparison?

Solution 18.2: The rather pointless function (LAMBDA (X) (EQ X X)) allows the same defining dotted-pair, or indeed, the same typed-pointer to any value to be presented to EQ for comparison.

We may describe the typed-pointers used to represent builtin or user-defined named functions and special forms as *doubly-indirect* typed-pointers. Of course the LISP interpreter must convert such a doubly-indirect typed-pointer to the desired singly-indirect index value whenever this is required.

19 Binding Actual Values to Formal Arguments

In order to implement the e operator for evaluating λ-expression bodies within the LISP interpreter, we must devise a mechanistic way to bind actual arguments to formal arguments and to honor the contexts thus established during the time that a related λ-expression is being evaluated. There are several ways to do this. The earliest approach, which was employed in the original LISP interpreter for the IBM 704 and 709, is to maintain a so-called *association list*. It is convenient to describe this approach and then use it as a model to explain how a LISP interpreter works in effect, if not in fact.

The association list is a LISP list which is the value of the builtin ordinary atom ALIST. The elements of the association list are dotted-pairs $(n_i \ . \ v_i)$, where n_i is an ordinary atom used as

a formal argument, and v_i is an S-expression or function which is an actual argument bound to n_i. Such dotted-pairs are put on the association list as follows when a function call expression (*g* a_1 ... a_k) is to be evaluated. The formal argument list of $v[g]$, $(f_1, \ldots, f_k)$ is retrieved and the formal argument—actual argument dotted-pairs (*f*$_i$. $v[a_i]$) (when $v[g]$ is a function), or (*f*$_i$. a_i) (when $v[g]$ is a special form) are formed and CONSed onto v[ALIST] in order from the first to the last argument. An exception is made within the LISP interpreter so that functions and special forms like v[PLUS] are allowed to be elements in association list dotted-pairs. This is required since LISP functions and special forms may take functions and/or special forms as arguments.

Now the body of $v[g]$ is obtained and the LISP interpreter is *recursively* invoked to evaluate it. During this evaluation, whenever an ordinary atom d is encountered, it is evaluated in the current context by first searching the association list from front to back for the first dotted-pair of the form (*d* . *w*); and if such a pair is found, the value of d is taken to be w. Otherwise, if no such pair is found on v[ALIST], then the value of d is sought in the atom table entry for d. When the body of $v[g]$ has been evaluated within the context given by the current association list, the initial k dotted-pairs corresponding to the formal argument bindings of $v[g]$ are removed from the association list v[ALIST]. This model of resolving the value bound to an ordinary atom is called *dynamic scoping*, as opposed to the *lexical* or *static* scoping found in languages like Algol.

In Algol, function definitions may be lexically nested in the defining text. The value of a variable x in the body of a function p is determined as the value bound to x in the closest function body in which the function p is nested, including the body of p itself in the case where x is a locally-defined variable or an argument of p, or in the atom table if p is not nested within any other functions. This lexical scoping search to determine the value of a variable is equivalent to lexically substituting the actual value to be bound to x at the time of calling a function p which has an formal argument named x for the name x within the function body of p, excepting those function bodies nested within p which have a formal argument called x.

Suppose we have two functions, $f(a, b)$ and $g(b, c)$, at the same lexical level, where g is called from within f. With Algol lexical scoping, the value of a within g is the global atom table value of a, whether g is running called from f or not. With LISP dynamic scoping, the value of a within g is the global atom table value if g is running from a top-level call to g, and the value of a is the actual argument value passed to f for binding to a if g is running called from f.

Notice that if the same ordinary atom x is used several times in differing λ-expressions as a formal argument, and if the corresponding functions are called, one within another, then the same atom x will occur several times in dotted-pairs on the association list during a certain time period. Whenever x must be evaluated, the latest existing binding will be used. This is exactly the right thing to do, but apparent mistakes can occur when this fact is forgotten.

For example, if we type-in

```
(SETQ G 3)                          followed by
(SETQ F (LAMBDA (X) (CONS X G)))    and
(SETQ B (LAMBDA (G X) (CONS G (F (PLUS X 1)))))
```

then $v[$`(B 2 0)`$] =$ `(2 . (1 . 2))`, but $v[$`(F 1)`$] =$ `(1 . 3)`. The ordinary atom `G` occurring in the body of the function $v[$`F`$]$ is said to be a *free variable* of the function $v[$`F`$]$. In general, *any* ordinary atom occurring in a function or special form body which is not listed as a formal argument of that function or special form is a free variable of that function or special form.

When the body of a function or special form is evaluated in a context provided by the bound pairs of the association list, the values of the free variables encountered are determined by the current association list if possible. Thus you cannot write a function like $v[$`F`$]$ above and expect its free variables to have the values of the atom table entries of those names, unless you are careful to not use those names elsewhere as formal arguments, or at least arrange to never have any of them bound on the association list when the function is called.

This situation is not so bad. We only have to use sufficient discipline to avoid picking formal argument names and global atom table names from among the same candidates whenever confusion might result.

A countervailing benefit of the simple LISP argument binding rule is that we can use the fact that the latest binding on the association list is the current value of an ordinary atom to skip passing arguments to functions explicitly when we know that the desired binding will already be in force. For example, consider:

```
(SETQ MAXF (LAMBDA (L F)
                   (COND ((NULL (CDR L)) (CAR L))
                   (T (MH (CAR L) (MAXF (CDR L) F))))))
```

and

```
(SETQ MH (LAMBDA (A B)
                 (COND ((GREATERP (F A) (F B)) A)
                       (T B)))).
```

Here `F` is a free variable in `MH` which will be properly interpreted as the function bound to `F` when `MAXF` is invoked, since `MH` is only intended to be used from within `MAXF`. This device is called *skip-binding*.

There are situations, however, in which the evaluation of an apparently-free variable done by following the simple association list model for context-evaluation is clearly counter-intuitive. Consider the example:

```
(SETQ F (LAMBDA (A)
                (G (PLUS A 1)
                   (LAMBDA (X) (CONS X A))))).
```

Here we have a function body consisting of a call to some function `G` which is passed a function created by `LAMBDA` as one of its actual arguments, namely $v[$`(LAMBDA (X) (CONS X A))`$]$. The ordinary atom `A` is a free variable of this argument function, but `A` is a formal argument, not a free variable, of the containing function `F` being defined.

Now suppose we define G with (SETQ G (LAMBDA (A H) (H A))). Then v[(F 1)] = (2 . 2). But, if we define G with (SETQ G (LAMBDA (O H) (H O))), then v[(F 1)] = (2 . 1). Thus, in order to avoid such surprises, we have to not only avoid conflicts with our choice of global names and formal argument names, but we also must beware of conflicts if we use the same names for formal arguments in different functions whenever functional arguments are used. This difficulty is called the *functional argument* problem or "funarg" problem.

Exercise 19.1: Show that the functional argument difficulty can occur in the form of a free-variable conflict, without the use of a λ-expression or special-expression appearing as an actual argument.

Solution 19.1: Consider:

```
(SETQ Z (LAMBDA (X) (CONS X A)))
(SETQ F (LAMBDA (A) (G (PLUS A 1) Z)))
(SETQ G (LAMBDA (A H) (H A))).
```

Then v[(F 1)] = (2 . 2).

The functional argument problem is really just a matter of potentially misunderstanding *binding times*. A variable may be a global variable at some times and a formal argument at other times during the evaluation of various S-expressions which are being evaluated in order to evaluate a root S-expression. The notion of 'free variable' is syntactic; a variable is free with respect to a function or special form according to its appearence in the associated body; it is not a temporal notion. However, such function body S-expressions are evaluated in a temporal sequence, and particular variables may be assigned values, *i.e.* be *bound*, differently at different times. Every ordinary atom must have something bound to it whenever it is evaluated (or else we have an evaluation error), but at some times this value is found in the association list, and at other times it is found in the atom table. In any event, the value of an ordinary atom at a given point in time is that value to which it has been most recently bound. Thus the value of an ordinary atom may change with time. The time at which each temporal act of the binding of values to ordinary atom variables occurs is *as late as possible* in LISP.

Returning to our example above: consider v[(F 1)], based on F defined by

```
(SETQ F (LAMBDA (A)
                (G (PLUS A 1)
                   (LAMBDA (X) (CONS X A))))),
```

and G defined by (SETQ G (LAMBDA (A H) (H A))). We see that the value of A in (PLUS A 1) is 1 when (PLUS A 1) is evaluated, and this binding is found in the association list. But the value of A in (CONS X A) is 2 *when it is evaluated* and this later binding of 2 to A supercedes the earlier binding of A in the association list.

Late-as-possible binding is effected if binding occurs when functions are called; thus we will refer to this kind of binding as *call-time* binding, since the context that is used when a formal argument of a function or special form is evaluated is created when the function or special form is entered.

A direct solution to the functional argument problem which also prevents any global-local name conflicts, is to program the builtin LAMBDA and SPECIAL special forms to scan through the argument list and body of each function and special form being defined and substitute a unique new name for each formal argument name at each occurrence. These unique names could be formed from an increasing counter and a non-printing character joined with the user-specified name. Of course, the original user-specified name should be kept for printing-out purposes. The effect of this would be to change the LISP temporal binding rule so that each ordinary atom is bound as early as possible, while still maintaining a call-time binding regime. This earliest binding time cannot be at the time of definition, but it can be just before a function or special form is applied to arguments at the top-level of the LISP interpreter.

Exercise 19.2: How does this device of using unique formal argument names compare with using the Algol substitution rule for function calls instead of the LISP association list binding rule?

In practice the functional argument problem is not serious. Discipline and due care in choosing names is all that is needed to avoid trouble.

Exercise 19.3: Explain how the function M defined here works: `(SETQ M (LAMBDA (M X) (M X)))`. Describe the contexts in which this function fails to work.

Some versions of LISP use a rule for evaluating selected atoms which relegates the association list to a lower priority. Such a high-priority atom is evaluated by looking at its global atom table value first, and then, only if this is undefined, is the association list searched. This exception is annoyingly nonuniform, so we use strictly temporal binding here. Note, however, that this approach of relegating association list bindings to be of lower priority than global atom table bindings for selected atoms insures that atoms like NIL and PLUS can be forced to always have their familiar values (except if explicit SETQ operations are done, and even this can be restricted.) A similar variation is commonly found where an atom's global atom table value is used whenever that value is a function or special form and the atom is being used as such. Another kind of binding rule, instead of call-time binding, is often used in LISP, where the free-variables in a function body are bound, with respect to that function body only, at the time the function is computed. This requires that a tree of association lists or other equivalent structures be maintained. Indeed the elaborate binding rules which have been introduced into current LISP dialects to "cure" perceived anomalies constitute one of the major areas of extension of LISP.

Exercise 19.4: Explain how the following function B, defined below, works on number arguments.

```
(SETQ B (LAMBDA (NIL)
                (COND ((GREATERP NIL 0) T)
                (T 'NIL)))).
```

Solution 19.4: It works just fine.

Searching for a formal argument atom in a linear association list can be time-consuming. Another strategy for binding values to formal argument atoms, which is called *shallow binding*, is a preferable way to handle argument binding. With shallow binding, whenever a value is to be bound to

an ordinary atom, we arrange to save the present value of the ordinary atom at the top of a corresponding private push-down list associated with that atom. After having introduced such private push-down lists, argument binding may then be done by first pushing the current value of each ordinary atom to be bound, together with its typecode on its corresponding push-down list, and then reassigning the values of each such atom in order to effect the bindings. When a user-defined function or special form has been evaluated, the unbinding which restores the previous context is done by popping the push-down list of each formal argument ordinary atom. Shallow call-time binding is used in the LISP interpreter program given below.

The operation of binding a value, represented by a typed-pointer p, as the current value of an ordinary atom is similar to the effect of the `SETQ` function, but not identical. In particular, if p is a type 14 or 15 typed-pointer, it is *not* transmuted into a type 12 or 13 type-pointer. Moreover, a doubly indirect type 12 or type 13 typed-pointer *is* transmuted by one level of dereferencing into the index of the list area node where the associated (argument list, body) dotted pair resides, and this index, together with the associated typecode, 12 or 13, becomes the current value of the ordinary atom being bound. These binding transmutation rules are appropriate because binding a function value to a formal argument atom is the act of associating a set of ordered pairs with the formal argument atom. The name, if any, of a function is not preserved by binding, however. Thus, the act of binding a named function to a formal argument atom causes that function to temporarily have a new name. For example, typing in `((LAMBDA (X) X) PLUS)` results in {`builtin function: X`} typing out.

Exercise 19.5: How does a recursive function, like `FACT`, continue to work when it is bound to another symbol, say `G`, and then invoked as `(G 3)`?

Note that the following identities are a consequence of the simple association-list binding rule discussed above:

$$\begin{aligned} v[\texttt{((SPECIAL (X) (EVAL X)) Y)}] &= v[\texttt{Y}], \quad \text{and} \\ v[\texttt{((LAMBDA (X) X) Y)}] &= v[\texttt{Y}], \quad \text{but} \\ v[\texttt{((SPECIAL (X) (EVAL X)) X)}] &= \texttt{X}, \quad \text{and} \\ v[\texttt{((LAMBDA (X) X) X)}] &= v[\texttt{X}]. \end{aligned}$$

The identity $v[\texttt{((SPECIAL (X) (EVAL X)) X)}] = \texttt{X}$ demonstrates the distinction between the LISP context-dependent association-list binding rule and the Algol substitution rule in assigning a meaning to a special form. There is no association list with the substitution rule. Substitution of `X` for `X` would produce `EVAL` applied to `X`, the value of which would then be $v[\texttt{X}]$ as found in the ordinary atom table. The LISP context-dependent binding rule leads us to compute `EVAL` applied to `X`, *after* `X` is bound to `X`, so that the resulting value is again `X`. It is not the case that one of these rules is "better" than the other; however, our expectations may be violated when we replace one rule with the other in our minds.

Exercise 19.6: What is $v[\texttt{((LAMBDA (X) (CONS (SETQ X 2) X)) 3)}]$?

Solution 19.6: `(2 . 2)`. When `SETQ` is used to assign a value to an ordinary atom which is a currently-active formal argument, the actual argument value which was bound to it is lost. This

is, in essence, a dynamic rebinding operation. In the form of LISP given in this book, it is not possible to change the global atom-table value of an atom which is active as a formal argument without introducing a new builtin function for this purpose.

Exercise 19.7: Define a LISP special form FREEVAR which takes a λ-expression L as input and which returns a list of all the atoms in L which are free variables within L.

Exercise 19.8: Explain the difficulty hidden in the following LISP input.

```
(SETQ F1 (LAMBDA (G L) (F2 (CAR L) (CDR L))))
(SETQ F2 (LAMBDA (H L) (CONS (G H) L)))
(SETQ H (LAMBDA (A B) (COND ((NULL A) B) (T (H (CDR A) (PLUS B 1))))))
(F1 H (QUOTE ((1))))
```

Most LISP dialects provide an additional class of functions called *macros*. We could introduce macro-functions by defining a builtin special form called MACRO which behaves like LAMBDA and SPECIAL, and builds an argument, body dotted-pair in the same manner. Macro functions obey the following evaluation rule. If m is a macro function which has k arguments then

$$v[(m\ a_1\ a_2\ \ldots\ a_k)] = v[((\texttt{EVAL (LIST 'SPECIAL (CAR (BODY } m\texttt{))}\\ \texttt{(LIST 'EVAL (CDR (BODY } m\texttt{)))))}\ a_1\ a_2\ \ldots\ a_k)].$$

Exercise 19.9: Define a LISP function called MAC which takes as input a user-defined LISP function f and an argument list w and returns the value of f on the arguments w computed as though f were a macro.

20 Minimal LISP

Let the set of basic S-expressions be the set of ordinary atoms (with non-numeric names) and non-atomic S-expressions formed from these. The following nine functions and special forms constitute a set of functions and special forms which are universal in the sense that, with these, any computable function of basic S-expression arguments can be expressed.

QUOTE, ATOM, EQ, CONS, CAR, CDR, COND, LAMBDA, LABEL.

Remember that LABEL is really a notational device to allow the statement of recursive λ-expressions, so we might (weakly) say that there are just eight functional components of minimal LISP.

The computable functions of basic S-expression arguments correspond to the computable functions of non-negative integers since we can prescribe an effective enumerating mapping which assigns a non-negative integer to every basic S-expression. It is somewhat easier to show just that the computable functions of non-negative integers are subsumed by the computable functions of basic S-expression arguments. To do this associate the integer k with the atom NIL if k is 0, and with the list (T ... T) consisting of k T's if $k > 0$. Thus non-negative integers, and by further extension, rational numbers are, in effect, included in the domains of various minimal LISP functions.

Exercise 20.1: Write the minimal LISP function which corresponds to addition using the number-to-list correspondence stated just above. Then do the same for subtraction of a lesser number list from a greater number list.

Exercise 20.2: Discuss the pros and cons of extending the definitions of `CAR` and `CDR` so that $v[\texttt{(CAR NIL)}] = v[\texttt{(CDR NIL)}] = \texttt{NIL}$.

21 More Functions

- `SUM` – **function with a varying number of arguments**

  $v[\texttt{(SUM } n_1\ n_2\ \ldots\ n_k\texttt{)}] = v[\texttt{(PLUS } n_1\ \texttt{(PLUS } n_2\ \texttt{(}\ldots\ \texttt{(PLUS } n_k\ 0\texttt{))}\ldots\texttt{))}]$.

- `PRODUCT` – **function with a varying number of arguments**

  $v[\texttt{(PRODUCT } n_1\ n_2\ \ldots\ n_k\texttt{)}] = v[\texttt{(TIMES } n_1\ \texttt{(TIMES } n_2\ \texttt{(}\ldots\ \texttt{(TIMES } n_k\ 1\texttt{))}\ldots\texttt{))}]$.

- `DO` – **function with a varying number of arguments**

  $v[\texttt{(DO } x_1\ x_2\ \ldots\ x_k\texttt{)}] = v[x_k]$.

  Since `DO` is a function, its arguments are all evaluated from left to right, and the last argument value is then returned. This function is useful when its arguments have side-effects which occur during their evaluation.

- `INTO` – **function**

  Defined by:

```
(SETQ INTO (LAMBDA (G L)
                   (COND ((NULL L) L)
                         (T (CONS (G (CAR L))
                                  (INTO G (CDR L)))))))
```

- `ONTO` – **function**

  Defined by:

```
(SETQ ONTO (LAMBDA (G L)
                   (COND ((NULL L) L)
                         (T (CONS (G L)
                                  (ONTO G (CDR L)))))))
```

- `APPLY` – **special form**

  Defined by:

```
(SETQ APPLY (SPECIAL (G X) (EVAL (CONS G X)))).
```

Exercise 21.1: Note v[`(APPLY CAR ('(1 . 2)))`] = 1 and v[`(APPLY CONS ('A 'B))`] = `(A . B)`. What is v[`(APPLY CAR (1 . 2))`]? What is v[`(APPLY CAR ((1 . 2)))`]? What is v[`(APPLY CAR '(1 . 2))`]? What is v[`(DO (SETQ A '(1 . 2)) (APPLY CAR (A)))`]?

Exercise 21.2: v[`(APPLY G (`x_1 ...x_n`))`] is intended to be the same as v[`(G `x_1 ...x_n`)`]. But we may sometimes forget to enclose the arguments x_1, ..., x_n in parentheses. What is wrong with defining `APPLY` with `(SETQ APPLY (SPECIAL (G X) (EVAL (CONS G (LIST X)))))`? Can you think of a better "solution"?

Exercise 21.3: Can you think of a reason why the name `G` should be replaced by an unlikely name like `$G` in the definition of `APPLY`? Hint: consider the situation where the value of the ordinary atom `G` is a user-defined function and `G` is used in a call to `APPLY`. Why isn't this a problem in `INTO` and `ONTO`? Shouldn't `X` be replaced by unlikely name also?

Exercise 21.4: The version of `INTO` given above only works for functions of one argument. Give a modified version which works for functions of k arguments with a list of k-tuples provided as the other input. Hint: use `APPLY`.

Exercise 21.5: Define a LISP function `COPYL` which takes a single list `L` as input and returns a copy of `L` which shares no nodes with the input list `L`.

Exercise 21.6: Define a LISP function `NAINX` which takes an atom a and a list x as input and returns the number of times the atom a occurs at any level in the list x.

Exercise 21.7: Define a LISP function `NAINS` which takes an atom a and a non-atomic S-expression x as input and returns the number of times the atom a occurs in x at any level.

Exercise 21.8: Define a LISP function `NEINS1` which takes an S-expression e and a list s as input and returns the number of times e occurs as an element of the list s.

Exercise 21.9: Define a LISP function `NEINSX` which takes an S-expression e and a list s as input and returns the number of times e occurs as an element of the list s or as an element of any list occuring in the list s at any level, including s itself.

Exercise 21.10: Define the LISP function `UNION` which takes two lists x and y as input and returns the list z whose elements are the elements of the set union of x and y.

Exercise 21.11: Define the LISP function `SORT` which takes a list of numbers as input and returns the corresponding sorted list of numbers as the result.

Solution 21.11:

```
(SETQ SORT
      (LAMBDA (X)
              (COND ((NULL X) X)
                    (T ((LABEL MERGE
                               (LAMBDA (V L)
                                       (COND ((OR (NULL L) (LESSP V (CAR L)))
                                              (CONS V L))
                                             (T (CONS (CAR L)
                                                      (MERGE V (CDR L))))))))
                       (CAR X) (SORT (CDR X))))))
```

Exercise 21.12: Define the LISP function SIGMA which takes a number-valued function g of an integer argument as input, together with two integers a and b as additional input, and which returns the value $\sum_{a \leq i \leq b} g(i)$.

Solution 21.12:

```
(SETQ SIGMA (LAMBDA (G A B)
                    (COND ((LESSP B A) 0)
                          (T (PLUS (G A) (SIGMA G (PLUS A 1) B))))))
```

Exercise 21.13: Define the LISP function FACTORS which takes an integer n as input and returns a list consisting of the prime factors of n.

22 Input and Output

In order to be able to specify a LISP function which behaves like an interactive program, we need a mechanism for printing messages to a user and for reading user-supplied input. The functions READ and PRINT satisfy these requirements.

- READ – **pseudo-function**

  $v[$(READ)$] =$ the S-expression which is typed-in in response to the prompt-symbol "!" typed to the user.

- PRINT – **function with a varying number of arguments and innocuous side-effect**

  $v[$(PRINT x_1 x_2 ... x_k)$] =$ NIL, with $v[x_1]$, $v[x_2]$, ..., $v[x_k]$ each printed-out at the terminal as a side-effect. If k is 0, a single blank is printed-out.

- PRINTCR – **function with a varying number of arguments and innocuous side-effect**

  $v[$(PRINTCR x_1 x_2 ... x_k)$] =$ NIL, with $v[x_1]$, $v[x_2]$, ..., $v[x_k]$ each followed by a carriage-return and printed-out at the terminal as a side-effect. If k is 0, a single carriage-return is printed-out.

23 Property Lists

Ordinary atoms have values, and it is often convenient to imagine that an ordinary atom has several values. This can be accomplished by making a list of these values and using this list as *the* value whose elements are the desired several values. Frequently, however, these multiple values are used to encode *properties*. This is so common that LISP provides special features to accomodate properties.

Abstractly, a property is a function. LISP only provides for properties which apply to ordinary atoms and whose values are S-expressions, and it establishes and uses a special set of lists to tabulate input-output pairs of such properties. Suppose `A` is an ordinary atom and g is an abstract property function. Then the value of g on `A` is called the *g-property-value* of `A`. Rather than computing a particular property-value of an atom when it is needed, LISP provides a means to record such property-values along with the corresponding ordinary atom, so that a property-value may be retrieved rather than computed.

Each ordinary atom `A` has an associated list provided called the *property list* of `A`. The typed-pointer to this list is stored in the plist field of the ordinary atom `A`. This list may be thought of as an "alternate value" of `A`. Generally the property list of an ordinary atom `A` is either `NIL`, or it is a list of dotted-pairs. Each such dotted-pair, `(`p `.` w`)`, in the property list for `A` is interpreted as meaning that the value of property p on `A` is w. Thus properties and property-values are represented by S-expressions. In fact, usually properties are represented by ordinary atoms.

The following functions are provided in order to aid in handling property lists.

- `PUTPROP` – **function with a side-effect**

  v[`(PUTPROP` a p w`)`] $= v[a]$, and the property, property-value dotted-pair `(`$v[p]$ `.` $v[w]$`)` is inserted in the property list for the atom $v[a]$. If another duplicate property, property-value dotted-pair for the property $v[p]$ and the property-value $v[w]$ is present, it is removed. $v[a]$ must be an ordinary atom, and $v[p]$ and $v[w]$ are arbitrary S-expressions.

- `GETPROP` – **function**

  v[`(GETPROP` a p`)`] $=$ the current property-value which is dotted with the property $v[p]$ on the property list for the ordinary atom $v[a]$. If a dotted-pair for the property $v[p]$ does not exist on the property list of $v[a]$, then `NIL` is returned.

- `REMPROP` – **function with a side-effect**

  v[`(REMPROP` a p w`)`] $= v[a]$, and the property, property-value dotted-pair `(`$v[p]$ `.` $v[w]$`)` for the property $v[p]$ is removed from the property list of the ordinary atom $v[a]$ if such a dotted-pair exists on the property list for $v[a]$.

Exercise 23.1: Define a special form called SPUTPROP analogous to PUTPROP which does not have its arguments evaluated.

Solution 23.1: `(SETQ SPUTPROP (SPECIAL (A P W) (PUTPROP A P W))).`

The functions PUTPROP, GETPROP, and REMPROP can all be defined in terms of the basic builtin functions GETPLIST and PUTPLIST, where $v[$(GETPLIST a)$]$ = the property list for the ordinary atom $v[a]$, and where $v[$(PUTPLIST a s)$] = v[a]$, with the side-effect that the property list of the ordinary atom $v[a]$ is replaced with the list $v[s]$. Now PUTPROP, GETPROP, and REMPROP are definable as follows.

```
(SETQ GETPROP (LAMBDA (A P)
                      (ASSOC (GETPLIST A) P)))

(SETQ ASSOC (LAMBDA (L P)
                    (COND ((NULL L) NIL)
                          (T (COND ((EQUAL P (CAR (CAR L))) (CDR (CAR L)))
                                   (T (ASSOC (CDR L) P)))))))

(SETQ REMPROP (LAMBDA (A P W)
                      (PUTPLIST A (NPROP NIL (GETPLIST A) (CONS P W)))))

(SETQ NPROP (LAMBDA (H L P)
                    (COND ((NULL L) (REVERSE H))
                          ((EQUAL P (CAR L)) (APPEND (REVERSE H) (CDR L)))
                          (T (NPROP (CONS (CAR L) H) (CDR L) P)))))

(SETQ PUTPROP (LAMBDA (A P W)
                      (PUTPLIST A (CONS (CONS P W)
                                        (GETPLIST (REMPROP A P W))))))
```

Exercise 23.2: Why don't we define PUTPROP as

```
(LAMBDA (A P W)
        (COND ((EQ (GETPROP A P) W) A)
              (T (PUTPLIST (CONS (CONS P W)
                                 (GETPLIST A))))))?
```

Exercise 23.3: Can the same property,value pair occur on the same property list several times?

Solution 23.3: No, not if only PUTPROP is used to build property lists. And, in fact, REMPROP assumes duplicates are not present.

Exercise 23.4: Why is REVERSE used in NPROP? Is it necessary?

Exercise 23.5: Give definitions of PUTPROP, GETPROP, and REMPROP, so that properties can be relations rather than just functions. Thus multiple $(p\ .\ w)$ dotted-pairs occuring on the same property list with the same p-value and differing w-values are to be handled properly.

Solution 23.5: Only GETPROP needs redefinition, so that it returns a list of property-values. It may be a beneficial practice to keep all properties functional however. This is no real hardship, since, for example, (COLORS . RED), (COLORS . BLUE), and (COLORS . GREEN) can be recast as (COLORS . (RED BLUE GREEN)).

Exercise 23.6: One useful device that is commonly used is to specify class subset relations with T-valued properties. For example, we might execute:

```
(PUTPROP 'MEN 'MORTAL T),
(PUTPROP 'MAN 'MEN T),    and
(PUTPROP 'SOCRATES 'MAN T).
```

But note that v[(GETPROP 'SOCRATES 'MORTAL)] = NIL.

Write a LISP function LINKGETPROP which will trace all property lists of atom properties with the associated value T reachable from the property list of an input atom A in order to find the specified property-value pair (P,T) and return T when this pair is found. The value NIL is to be returned when no chain leads to the sought-for property P with a T value.

Solution 23.6:

```
(SETQ LINKGETPROP (LAMBDA (A P)
                          (COND ((EQ (GETPROP A P) T) T)
                                (T (SEARCH (GETPLIST A) P)))))

(SETQ SEARCH (LAMBDA (L P)
                     (COND ((NULL L) L)
                           ((AND (EQ (CDR (CAR L)) T)
                                 (ATOM (CAR (CAR L)))
                                 (EQ (LINKGETPROP (CAR (CAR L)) P) T)) T)
                           (T (SEARCH (CDR L) P)))))
```

Exercise 23.7: Write a LISP function CKPROP such that (CKPROP a p w) returns T if ($v[p]$. $v[w]$) is on the property list for the atom $v[a]$, and NIL otherwise. Now write a generalized form of CKPROP called TCPROP which is defined such that (TCPROP a p w) returns T if there exists a sequence of atoms m_0, m_1, ..., m_k, with $k \geq 1$, such that $v[a] = m_0$, $v[w] = m_k$, and ($v[p]$. m_{i+1}) is on the property list of m_i for $i = 0, 1, \ldots, k-1$. "TC" stands for *transitive closure*.

The property list functions defined above, especially REMPROP, and hence PUTPROP, are slow, and moreover they can create lots of garbage unreachable list-nodes. It would be more efficient if we could just *change* property lists as required, rather than rebuild them with CONSing each time a REMPROP is done.

There are two low-level functions with side-effects defined in LISP which provide the ability to *change* existing list-nodes which have been created with CONS. These functions are RPLACA and RPLACD.

- RPLACA – **function with a side-effect**

  v[(RPLACA x y)] = $v[x]$, *after* the *car* field of the list node representing the dotted-pair $v[x]$ is *changed* to the typed-pointer $v[y]$.

- RPLACD – **function with a side-effect**

$v[$`(RPLACD` x y`)`$] = v[x]$, *after* the cdr field of the list node representing the dotted-pair $v[x]$ is *changed* to the typed-pointer $v[y]$.

Now `REMPROP` can be programmed as follows.

```
(SETQ REMPROP (LAMBDA (A P W)
                      (PUTPLIST A (NAX (GETPLIST A) (CONS P W)))))

(SETQ NAX (LAMBDA (L P)
                  (COND ((NULL L) NIL)
                        ((EQUAL (CAR L) P) (CDR L))
                        (T (DO (NX L P) L)))))

(SETQ NX (LAMBDA (L P)
                 (COND ((NULL (CDR L)) NIL)
                       ((EQUAL P (CAR (CDR L))) (RPLACD L (CDR L)))
                       (T (NX (CDR L) P)))))
```

Exercise 23.8: Could `NX` be defined with an empty formal argument list `()` rather than `(L P)`, and be called as `(NX)` in `NAX`?

Note this version of `REMPROP` can only be safely used when we can guarantee that the backbone nodes of property lists are not shared within other list structures.

Exercise 23.9: Program a version of `APPEND`, called `NCONC`, which uses `RPLACD` to append one list onto another by physical relinking.

24 What is LISP Good For?

The quick answer to the question "what is LISP good for?" is (1) ideas, and (2) experimental programs.

The algorithmic ideas which LISP inspires are often powerful and elegant. Even if LISP is not the target programming language, thinking about how to tackle a programming job using LISP can pay worthwhile dividends in ideas for data structures, for the use of recursion, and for functional programming approaches. The use of Algol-like languages or FORTRAN tend to limit a programmers' imagination, and both applications programmmers and systems programmers can benefit by remembering LISP.

LISP is useful for building and trying-out programs to solve predominantly non-numeric problems such as natural language parsing or dialog processing, symbolic formula manipulation, retrieval in a LISP-encoded database, backtrack searching (*e.g.* game playing) and pattern recognition programs.

Most versions of LISP have been extended with arrays, strings, a FORTRAN-like statement-based programming facility via the so-called `PROG` special form, and many other features. Indeed, the enthusiasm for extending LISP is perennially high. Many extensions take the form of control structures and data structures for managing abstract search. As LISP is extended, however, it

seems to lose its sparse pure elegance and uniform view of data. The added features are sometimes baroquely complex, and the programmers' mental burden is correspondingly increased. Moreover at some point it is legitimate to ask why not extend FORTRAN or C to contain LISP features, rather than conversely? And indeed this has also been done.

Exercise 24.1: Define an extension to LISP to handle the class of strings of characters as a datatype. In particular, let strings of characters be a class of legal LISP values, just as numbers, functions, and S-expressions are. What about a typecode for strings? Explain how strings might be stored. (Hint: use self-referential ordinary atoms.) How are constant strings written? Define the functions which provide for string manipulation. Include `(CAT` *a b*`)`, `(STRLEN` *a*`)`, and `(SUBSTR` *a i j*`)`. Would `STR`, where v[`(STR` x`)`] = the string of $v[x]$, be useful? What about introducing an inverse to `STR`? Define `(READSTR)` in a useful way. How will `PRINT` and `PRINTCR` handle strings? Can `READ` be redefined to optionally apply to a string? Are there other potentially-useful new functions and extensions of old functions which are of interest? Strike a synthetic balance between utility and complexity.

Often features are added to LISP to increase its speed. The standard accessing strategy in LISP is, in essence, linear searching, and many attempts have been made to circumvent this. LISP with extensions, then, is likely to be a curious amalgam of considerable complexity, sometimes with the possibility of constructing faster programs as a compensating factor. (Although, if a program is perceived to be fast enough by its users, then it *is* fast enough, no matter what language it is written in.)

Not only does LISP perform slowly in comparison to conventional loop-based programs; it is designed as a self-contained programming system. Thus, like APL, it may be difficult to employ and/or control computer system resources, access files, and handle interrrupts in a general and convenient manner. In short, without suitable extensions, LISP is not a systems programming tool, and insofar as a program must deal with such environmental issues, LISP is generally inadequate to the challenge. Since most programs with a long half-life have systems programming aspects, LISP is generally not the tool of choice for building a robust, efficient system for long-term use. Note, however, system programming extensions can be added to LISP by introducing suitable "hooks" to the operating system; the well-known *emacs* text-editor is written in an extended version of LISP.

Minor details also mitigate against LISP. Format control for terminal input and output is often lacking for example, and this can be frustrating in many applications. Moreover, although LISP list notation is adequate for short functions (indeed, it encourages them), it is cumbersome compared to Algol notation, and the lack of traditional mathematical notation is a severe handicap in many cases. Notational extensions have been proposed, but again, simplicity is sacrificed.

On the other hand, LISP is an excellent tool for experimentation. A pattern-matching idea may be able to be programmed and tested in a preliminary way more easily and quickly in LISP than in Pascal for example. The lack of a variety of constrained data types, and the absence of declarations and multiple statement forms often gives an advantage to LISP as long as we banish efficiency considerations from our minds. Moreover there are certain applications, notably formula manipulation tasks, where S-expressions and recursion are so well suited for the job that LISP matches the utility of any other language.

Unfortunately many versions of LISP do not gracefully cohabit with programs written in other programming languages, so a system can't generally be easily constructed which employs LISP just for selected subtasks. However, a special embeddable LISP interpreter could be relatively-easily constructed as a C or FORTRAN callable subroutine package which would allow the use of LISP for specialized purposes within a larger non-LISP system.

We shall consider several classical applications of LISP below.

25 Symbolic Differentiation

The rules for differentiation are explicitly recursive, so it is easy to use LISP to compute the symbolic derivative of a real-valued function of real number arguments. The biggest difficulty is the problem of input and output notation. If we agree to accept the LISP prefix notation for algebraic expressions, so that we enter $1+sin(x+y)/x$ as `(PLUS 1 (QUOTIENT (APPLY SIN ((PLUS` x y`))) `x`))` for example, then the task of differentiation is truly simple. We shall instead demand that the input and output be in traditional infix form, however, so the job is more complicated.

We shall present a collection of LISP functions and special forms below which provide the ability to define, differentiate, and print-out elementary real-valued functions of real arguments. These LISP functions and special forms include `FSET`, `DIFF`, and `FPRINT`.

We will use the special form `FSET` to define a real-valued function of real arguments by typing `(FSET G (X Y ...) (`f`))`, where `G` is an ordinary atom, `(X Y ...)` is a list of ordinary atoms denoting the formal arguments of the function `G` being defined, and f is an *infix* form which is the body that defines `G`, *e.g.*, `(FSET G1 (X Y) (X + Y * EXP(X)))`.

Note that an infix form here is a kind of pun; it denotes the intended algebraic expression, and also it is a particular LISP S-expression. The only caveat that must be remembered is that blanks must delimit operators in infix forms to insure that they will be interpreted as atoms within the S-expression.

Such a function can be differentiated with respect to a variable which may be either a free variable or a formal argument. The derivative is itself a function which has the same formal arguments. The derivative of such an `FSET`-defined function `G` with respect to `X` is denoted `G#X`. The LISP function `DIFF` will be defined to compute the derivative of such a real-valued function, `G`, with respect to a specified symbol `X`, and store the resulting function as the value of the created atom `G#X`. This is done for a function `G` by typing `(DIFF G X)`.

In order to construct atoms with names like `G#X` which depend upon the names of other atoms, we will use the builtin LISP function `MKATOM` which makes atoms whose names are composed from the names of other atoms. `MKATOM` is defined such that $v[$`(MKATOM` x y`)`$] =$ the ordinary atom whose name is the string formed by concatenating the names of the ordinary atoms $v[x]$ and $v[y]$.

Functions defined by `FSET` or by `DIFF` may be printed-out for examination by using the LISP function `FPRINT` defined below.

A function h defined by FSET or DIFF may be evaluated at the point $(x_1, x_2, \dots)$, where x_1, x_2, ... are numbers, by using APPLY. Thus typing (APPLY h (x_1 x_2 ...)) causes the number $\mathtt{h}(x_1, x_2, \dots)$ to be typed-out. Of course, all the subsidiary functions called in h must be defined as well.

The differentiation rules used by DIFF are as follows. The symbol D denotes differentiation with respect to a specific understood ordinary atom variable name x, called the *symbol of* D.

$$
\begin{array}{lcl}
\mathtt{D}(\mathtt{A}+\mathtt{B}) & = & \mathtt{D}(\mathtt{A})+\mathtt{D}(\mathtt{B}) \\
\mathtt{D}(\mathtt{A}-\mathtt{B}) & = & \mathtt{D}(\mathtt{A})-\mathtt{D}(\mathtt{B}) \\
\mathtt{D}(-\mathtt{A}) & = & -\mathtt{D}(\mathtt{A}) \\
\mathtt{D}(\mathtt{A}*\mathtt{B}) & = & \mathtt{A}*\mathtt{D}(\mathtt{B})+\mathtt{B}*\mathtt{D}(\mathtt{A}) \\
\mathtt{D}(\mathtt{A}/\mathtt{B}) & = & \mathtt{D}(\mathtt{A})/\mathtt{B}+\mathtt{A}*\mathtt{D}(\mathtt{B}\uparrow(-1)) \\
\mathtt{D}(\mathtt{A}\uparrow\mathtt{B}) & = & \mathtt{B}*\mathtt{A}\uparrow(\mathtt{B}-1)*\mathtt{D}(\mathtt{A})+\mathtt{LOG}(\mathtt{A})*\mathtt{A}\uparrow\mathtt{B}*\mathtt{D}(\mathtt{B}) \text{ (}\uparrow\text{ denotes exponentiation)} \\
\mathtt{D}(\mathtt{LOG}(\mathtt{A})) & = & \mathtt{D}(\mathtt{A})/\mathtt{A}
\end{array}
$$

If $\mathtt{F}(\mathtt{X}\ \mathtt{Y}\ \dots)$ has been defined via FSET, then $\mathtt{D}(\mathtt{F}(\mathtt{A}\ \mathtt{B}\ \dots)) = \mathtt{F\#X}(\mathtt{A}\ \mathtt{B}\ \dots)*\mathtt{D}(\mathtt{A})+\mathtt{F\#Y}(\mathtt{A}\ \mathtt{B}\ \dots)*\mathtt{D}(\mathtt{B})+\dots$.

If A is an ordinary atom variable then $\mathtt{D}(\mathtt{A})=1$ if A is the symbol of D, and $\mathtt{D}(\mathtt{A})=0$ otherwise.

If A is a number, then $\mathtt{D}(\mathtt{A})=0$.

The infix forms used to define functions with FSET are translated into corresponding S-expressions which will be called E-expressions in order to distinguish them. E-expressions are made up of evaluatable prefix S-expressions involving numbers and ordinary atoms and other E-expressions.

The basic LISP functional forms used in E-expressions are:

(MINUS a),
(PLUS a b),
(DIFFERENCE a b),
(TIMES a b),
(QUOTIENT a b), (POWER a b), where a and b are E-expressions, and
(APPLY h (x_1 x_2 ...)), where x_1, x_2, ... are E-expressions, and h is an FSET-defined or DIFF-defined function.

The definitions of FSET, DIFF, and FPRINT follow.

- FSET – **special form**

  v[(FSET G (X Y ...) (e))] = v[G] where G is an ordinary atom whose name does not begin with "$", and (X Y ...) is a list of atoms which are the formal arguments to the function F being defined. e is an infix functional form which is used to define $\mathtt{G}(\mathtt{X}, \mathtt{Y}, \dots)$.

FSET is defined by

```
(SETQ FSET (SPECIAL ($G $L $Q)
                    (SET $G (EVAL (LIST 'LAMBDA $L (ELIST $Q))))))
```

The function SET has been defined above. Here we redefine SET by:

```
(SETQ SET (LAMBDA ($X $Y)
                  (EVAL (CONS 'SETQ (CONS $X (CONS '$Y NIL))))))
```

Exercise 25.1: Why are such strange formal argument names used in defining FSET and SET? Hint: think about using SETQ to assign values to ordinary atoms which are active formal arguments. In particular, suppose the $-signs were dropped, and we executed (FSET Y (X) X)?

Exercise 25.2: Can you think of a way to allow the user to write (FSET Y (X) = e)? Note the third argument here is "=".

ELIST constructs the E-expression corresponding to its argument by using the Bauer-Samelson-Ershov postfix Polish translation algorithm. ELIST is also used to process the argument of a unary minus operation.

Exercise 25.3: Use a library to find out about *Jan Łukasiewicz* and about "Polish notation".

```
(SETQ ELIST (LAMBDA (L)
                    (COND ((ATOM L) L)
                          ((EQ (CDR L) NIL) (ELIST (CAR L)))
                          (T (EP NIL NIL (DLIST L))))))

(SETQ DLIST (LAMBDA (L)
                    (COND ((ATOM L) L)
                          ((NULL (CDR L)) (DLIST (CAR L)))
                          (T L))))
```

(EP OP P L) is the result of completing the polish translation of L using the stacks OP and P.

```
(SETQ EP (LAMBDA (OP P L)
                 (COND ((NULL L) (COND ((NULL OP) (DLIST P))
                                       (T (BUILD OP P L))))
                       (T (POLISH OP P (CAR L) (CDR L))))))

(SETQ CADR (LAMBDA (L) (CAR (CDR L))))
```

(BUILD OP P L) puts the phrase $\langle$(CAR OP) P_2 $P_1\rangle$ on the P stack in place of P_1 and P_2, and goes back to EP to continue the translation.

```
(SETQ BUILD (LAMBDA (OP P L)
                    (EP (CDR OP) (CONS (LIST (NAME (CAR OP))
                                             (CADR P)
                                             (CAR P))
                                       (CDR (CDR P))) L)))
```

In (POLISH OP P B L), B is the current input symbol, and L is the remaining infix input. Either B is to be pushed on OP, or else B is used to make a phrase reduction and a corresponding entry is placed on the P stack.

```
(SETQ POLISH (LAMBDA (OP P B L)
                     (COND ((EQ B '-)
                            (H (LIST 'MINUS (ELIST (CAR L))) (CDR L)))
                           ((NOT (ATOM B)) (H (ELIST B) L))
                           ((NOT (MEMBER B '(+ : * / **)))
                            (COND ((OR (NULL L) (ATOM (CAR L))) (H B L))
                                  (T (H (LIST 'APPLY B (INTO ELIST (CAR L)))
                                        (CDR L)))))
                           ((OR (NULL OP) (CKLESS (CAR OP) B))
                            (POLISH (CONS B OP) P (CAR L) (CDR L)))
                           (T (BUILD OP P (CONS B L))))))
```

(H V L) pushes V onto the P stack and goes back to EP. It also changes a binary minus-sign at the front of L to the unambiguous internal code ":".

```
(SETQ H (LAMBDA (V L)
                (EP OP (CONS (COND ((AND (NOT (ATOM V))
                                         (NULL (CDR V))) (CAR V))
                                   (T V))
                             P)
                    (COND ((AND (NOT (NULL L))
                                (EQ '- (CAR L))) (CONS ': (CDR L)))
                          (T L)))))

(SETQ CKLESS (LAMBDA (A B)
                     (COND ((EQ '** A) NIL)
                           ((OR (EQ '* A) (EQ '/ A))
                            (COND ((EQ '** B) T)
                                  (T NIL)))
                           ((OR (EQ '+ B) (EQ ': B)) NIL)
                           (T T))))

(SETQ NAME (LAMBDA (A)
                   (ASSO A '((** . POWER) (* . TIMES) (/ . QUOTIENT)
                             (: . DIFFERENCE) (+ . PLUS)))))

(SETQ ASSO (LAMBDA (A L)
                   (COND ((NULL L) L)
                         ((EQ A (CAR (CAR L))) (CDR (CAR L)))
                         (T (ASSO A (CDR L))))))
```

- FPRINT − **special form**

  v[(FPRINT G)] = NIL, and the E-expression value of the function-valued atom G is printed-out in infix form.

  FPRINT is defined by:

```
(SETQ FPRINT (SPECIAL ($G)
                      (DO (PRINT $G (CAR (BODY (EVAL $G)))
                                 '= (HP (CDR (BODY (EVAL $G)))))
                          (PRINTCR))))

(SETQ HP (LAMBDA (G)
                 (COND ((ATOM G) G)
                       ((NULL (CDR G)) (HP (CAR G)))
                       ((EQ (CAR G) 'MINUS) (LIST '- (HP (CADR G))))
                       ((EQ (CAR G) 'APPLY)  (LIST (CADR G)
                                                   (INTO HP (CADR (CDR G)))))
                       (T (LIST (HP (CADR G)) (OPSYMBOL (CAR G))
                                (HP (CADR (CDR G))))))))

(SETQ OPSYMBOL (LAMBDA (X)
                       (ASSO X '((POWER . **) (TIMES . *) (QUOTIENT . /)
                                 (DIFFERENCE . -) (PLUS . +)))))
```

- DIFF – **special form**

  v[(DIFF G X)] = v[G#X], after the function G#X is defined as a side-effect, where G is a known function and X is an ordinary atom.

  DIFF is defined by:

```
(SETQ DIFF (SPECIAL ($G X)
                    (SET (MKATOM $G (MKATOM '# X))
                         (EVAL (LIST 'LAMBDA (CAR (BODY (EVAL $G)))
                                     (DI (CDR (BODY (EVAL $G))) X))))))
```

(DI G X) is the E-expression form derivative of the E-expression G with respect to the atom X.

```
(SETQ DI (LAMBDA (E X)
                 (COND ((ATOM E) (COND ((EQ E X) 1) (T 0)))
                       ((NULL (CDR E)) (DI (CAR E) X))
                       (T (DF (CAR E) (CDR E) X)))))

(SETQ DF (LAMBDA (OP L X)
                 (COND ((EQ OP 'MINUS) (LIST OP (DI (CAR L) X)))
                       ((EQ OP 'PLUS) (LIST OP (DI (CAR L) X)
                                               (DI (CADR L) X)))
                       ((EQ OP 'DIFFERENCE) (LIST OP (DI (CAR L) X)
                                                     (DI (CADR L) X)))
                       ((EQ OP 'TIMES) (LIST 'PLUS
                                             (LIST OP (DI (CAR L) X) (CADR L))
                                             (LIST OP (CAR L)
                                                   (DI (CADR L) X))))
                       ((EQ OP 'QUOTIENT) (LIST 'PLUS
                                                (LIST OP (DI (CAR L) X)
                                                      (CADR L))
                                                (LIST 'TIMES (CAR L)
                                                      (DI (LIST 'POWER
                                                                (CADR L) -1)
                                                          X))))
                       ((EQ OP 'POWER) (LIST 'PLUS
```

```
                                        (LIST 'TIMES (CADR L)
                                              (LIST 'TIMES
                                                    (DI (CAR L) X)
                                                    (LIST 'POWER (CAR L)
                                                          (LIST 'DIFFERENCE
                                                                (CADR L) 1))))
                                        (LIST 'TIMES
                                              (LIST 'APPLY 'LOG
                                                    (LIST (CAR L)))
                                              (LIST 'TIMES
                                                    (DI (CADR L) X)
                                                    (LIST 'POWER
                                                          (CAR L)
                                                          (CADR L))))))
                    ((EQ OP 'APPLY)
                     (COND ((EQ (CAR L) 'LOG)
                            (LIST 'QUOTIENT (DI (CADR L) X) (CADR L)))
                           (T (CHAIN (CAR L) (CAR (BODY (EVAL (CAR L))))
                                     (CADR L) (CADR L) X)))))))

(SETQ CHAIN (LAMBDA (F A B R X)
                    (COND ((NULL (CDR A)) (TERM F A B R X))
                          (T (LIST 'PLUS (TERM F A B R X)
                                   (CHAIN F (CDR A) B (CDR B) X))))))

(SETQ TERM (LAMBDA (F A B R X)
                   (LIST 'TIMES (DI (CAR R) X)
                         (LIST 'APPLY (MKATOM F (MKATOM '# (CAR A))) B))))
```

Exercise 25.4: Would it be okay to define TERM as

```
(LAMBDA (A R) (LIST 'TIMES (DI (CAR R) X)
                    (LIST 'APPLY (MKATOM F (MKATOM '# (CAR A))) B))),
```

and appropriately modify the associated calls?

Exercise 25.5: Is it necessary to use APPLY to compute functions at particular points? Is it necessary to use APPLY embedded in the bodies of functions created by FSET and DIFF at all?

Solution 25.5: The functions built by FSET and DIFF are full-fledged LISP functions. The use of APPLY could be dispensed with, except that we then need to provide the function LOG which is used in DIFF. Also, this would avoid the need to watch out for the formal argument names used in the special form APPLY in order to avoid the special form functional binding problem. Using APPLY is convenient, however, since we then do not have to distinguish user function names from operator names such as PLUS and TIMES.

Exercise 25.6: The function DF allows products with 1 and sums and differences with 0 to be created. Write a modified version of DF which uses some auxilliary functions to simplify the E-expressions being formed so as to avoid such needless operations.

Exercise 25.7: Introduce the elementary functions EXP, SIN, and COS, and implement the explicit differentiation rules which apply to them.

26 Game-Playing

In this section we shall discuss writing a LISP program to play a *zero-sum, perfect-information* game. A perfect-information game is a game such as chess where the entire state of the game (*i.e.* the current position and its history) is known to all players. Poker is not a perfect information game. A zero-sum game is just a game such that if the game were played for money, then the sum of the winnings and losses is necessarily zero. Poker is a zero-sum game.

A *fair* game is a game where, for perfect players, the expected return to each player in points or money is 0. No one knows whether or not chess is a fair game. We need not assume that we are playing a fair game.

We shall also assume that:

1. The game we wish to program is played by two players making alternate moves.
2. Each position p is either *terminal*, in which case it represents a win, loss, or tie for each player as defined by the rules of the game; or, *non-terminal*, in which case it has a finite, non-zero number of successor positions, among which the player who is to move from position p must select.
3. Every game must eventually reach a terminal position.

These hypotheses are satisfied by many games (although in chess, the rules are such that the position must be considered to include not only the current locations of the pieces and the specification of which player is to move, but also the past history of the game; this is required in the rules for castling, capturing *en passant*, and draw by repetition).

By tradition, one player is called *box*, and a position from which *box* is to move is denoted by a box symbol: □. Such a position is called a □ position. The other player is called *circle*, and a position from which *circle* is to move is denoted by a circle symbol: ◯. Such a position is called a ◯ position.

Terminal positions will be assigned *values* so that the *value* of a □ or ◯ terminal position is 1 if the player *box* has won in that position, and −1 if the player *circle* has won in that position. If tie terminal positions exist, we assign them the value 0. This means that *box* plays to *maximize* the value of the final terminal position, and *circle* plays to *minimize* the terminal position to be achieved. *box* is called the *maximizing* player, and *circle* is called the *minimizing* player. Thus if we are playing chess, and the maximizing player *box* is white, then a terminal position where white is checkmated is a □ position and has the value −1, while a terminal position where black is checkmated is a ◯ position and has the value 1. If Nim is being played, where the player who takes the last stone and leaves an empty position consisting of an empty arrangement loses, then the empty position with *box* to move is a □ terminal position, since there are no moves possible, and its value is 1, since *circle* lost by allowing this position to be reached. The empty position with *circle* to move is a ◯ terminal position with the value −1.

Because of the assumption that our game is a zero-sum game, we see that we are assigning values to terminal positions entirely from *box's* point of view. Thus if the value of a terminal position is 1, then its value to *box* is 1, and hence by the zero-sum property, its value to *circle* is -1. If, on the other hand, the value of a terminal position is -1, its value to *box* is -1 and hence its value to *circle* is 1. This form of assigning values to positions so that positions with positive values are good for *box* and bad for *circle*, while positions with negative values are good for *circle* and bad for *box*, is known as *asymmetric* valuation.

We should like to assign the values -1, 0, or 1 to all positions which may arise, not just terminal positions; and moreover we should like to assign them consistently in such a way that the value of a position is the value of that position to *box*, so that we continue to use asymmetric valuation. We shall achieve this objective as follows.

Let b represent a $\Box$ position or a $\bigcirc$ position of the game, and let $n(b)$ be the number of immediate successor positions, which may be achieved by moving from b. Let $a_1, \ldots, a_{n(b)}$ be the $n(b)$ successor positions in some particular order. If b is a terminal position, $n(b) = 0$. Finally, when b is a terminal position, let $u(b)$ be the value of b.

Now let us define the value of a position b, not necessarily terminal, as $w(b)$, where $w(b)$ is defined recursively as:

$$w(b) := \begin{cases} u(b) & \text{if } n(b) = 0, \\ \max\limits_{1 \le i \le n(b)} w(a_i) & \text{if } b \text{ is a } \Box \text{ position}, \\ \min\limits_{1 \le i \le n(b)} w(a_i) & \text{otherwise.} \end{cases}$$

Hypotheses (1), (2), and (3) listed above ensure that w is well-defined.

Note the asymmetry in the definition of w. Note also that the use of w to compute the value of a position b requires that all possible successor positions be valuated first and so on recursively. Thus we must generate the full *game tree* with root b whose leaf nodes are terminal positions which are valued by means of u. Then for each node whose sons all have values, we can back-up the minimum or the maximum of these values as may be appropriate to the father node. By iteratively backing-up, we may eventually compute the value of b. This process is known as *minimaxing*.

Von Neumann and Morgenstern[NM44] introduced the notion of minimaxing in a game tree in their 1944 treatise on game theory. Their purpose was to show that even for multi-stage games with alternative choices at each stage, any line of play could be considered to be the result of fixed strategies, *i.e.* predetermined plans which could be adopted at the outset of a game by each player and followed mechanically thereafter. Such plans are multi-stage contingency plans.

Thus every possible sequence of moves as represented by a path in the game tree is the joint result of the two strategies defined by that path.

Von Neumann's and Morgenstern's argument was a short inductive statement. They observed that games of 0 moves whose initial position is terminal have an obvious strategy, the null strategy; and then they pointed out that the first step of the strategy for a game of n moves is just to make that

move which yields the highest value for the resulting game of $n-1$ moves (as established by the use of the known optimal strategy which exists by the induction hypothesis.)

They also proved that the value of a position always exists, and that minimaxing is the explicit method of computing it.

The basic principle of dynamic programming as formulated by Richard Bellman[Bel57] is another statement of the idea underlying minimaxing. Given a multi-stage decision process to be undertaken where one wishes to know a sequence of decisions which optimizes some desired quantity such as profit, the principle of dynamic programming is: select that decision which results in the state such that the subsequent use of an optimal policy from that state yields the fully-optimal result. Thus to implement dynamic programming, we back-up scores in the tree of decisions to find that sequence of decisions which optimizes the initial state. Often there are mathematical short-cut approaches to solving dynamic programming problems as opposed to brute force back-up, but the objective is always to discover the optimizing path in the tree of decisions

It is a remarkable fact that in a terminating two player perfect-information game where ties are not allowed, there must be a forced win for either the first or second player. That is: there exists a strategy for one player which guarantees that player will win regardless of the opponent's actions. The proof of this fact is based upon the valuation function scheme given above. The value $w(b)$ is 1 if and only if *box* has a forced win and -1 if and only if *circle* has a forced win, where b is the initial $\square$ or $\bigcirc$ position.

We are interested not only in computing the maximum value to *box* of a position b, but also in discovering the particular move from position b which eventually results in a terminal position from which that value is inherited.

The following LISP function `PICK` takes a game position as an argument and computes the value of that position to *box* and the corresponding best successor position for the player to move as the two elements of the output dotted-pair. If there is no successor position to be recommended, then `NIL` is returned as the second element.

To use LISP, we must encode game positions as S-expressions. Such a game position S-expression includes, among other things, whether the position is a $\square$ or a $\bigcirc$ position.

`(PICK B)` returns a dotted-pair `(`x `.` m`)` where m is an optimal successor position for *box* of position `B`, and x is its score to *box*, where x and m are computed by minimaxing.

```
(SETQ PICK (LAMBDA (B)
                   (COND ((DEEPENOUGH B) (CONS (U B) NIL))
                         (T (OPT (GENMOVES B)
                                 (COND ((BOXPOS B) GREATERP)
                                       (T LESSP)))))))
```

`(DEEPENOUGH B)` returns `T` if v[`B`] is a terminal position, and `NIL` otherwise.

`(BOXPOS B)` returns `T` if v[`B`] is a $\square$ position and `NIL` if v[`B`] is a $\bigcirc$ position.

`(U B)` returns the value -1, 0, or 1 which indicates the value to *box* of the terminal position v[`B`].

`(GENMOVES B)` returns a list of all the successor positions of v[`B`].

`(W B)` returns the minimaxed value to *box* of the position v[`B`].

```
(SETQ W (LAMBDA (M) (CAR (PICK M))))
```

`(OPT L P)` returns a dotted-pair $(x \ . \ m)$ where x is a greatest score value among the scores of the positions in the list v[`L`] if `P` = `GREATERP`, and x is a least score value among the scores of the positions in the list v[`L`] if `P` = `LESSP`. In either case, m is a position which is an element of the list v[`L`] whose score is x.

```
(SETQ OPT (LAMBDA (L P)
                  (COND ((EQ (CDR L) NIL) (CONS (W (CAR L)) (CAR L)))
                        (T (MB (CAR L) (W (CAR L)) (OPT (CDR L) P))))))

(SETQ MB (LAMBDA (A S Q)
                 (COND ((P S (CAR Q)) (CONS S A))
                       (T Q))))
```

Exercise 26.1: Why isn't `P` an argument of `MB`?

Now we have spelled out the machinery needed to program a great many games; namely the functions `PICK`, `W`, `OPT`, and `MB`. The function `PICK` is the game-independent minimaxing program. The remaining required functions `DEEPENOUGH`, `GENMOVES`, `U`, and `BOXPOS` depend upon the game to be programmed and must be especially written for each game.

A major problem in using LISP to minimax a game tree is the requirement that an S-expression representation of game positions be devised and used. Such representations are generally large and complex and require complex functions to generate and valuate them. Often machine language or C permits a more compact and simple data structure to be used.

Another more fundamental problem is that simple minimaxing is too slow for all but very small game trees. This forces us to abandon searching most game trees completely. Instead we shall search to a certain depth, and then *estimate* the values of the positions at which we stop. These estimates will then be backed-up according to the minimaxing process. The estimated values will not be only -1, 0, and 1, but may be any real number in $[-1, 1]$ which serves as a score value of a position for *box*.

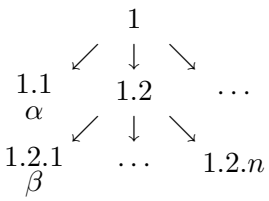
In order to evaluate a position when we cannot search ahead to terminal positions, we shall introduce a valuation function s which can compute an estimate of the score of *any* position, including non-terminal positions. By convention, this score lies in $[-1, 1]$ and is always interpreted as being from the point of view of *box*. The function s is often called a *static valuation function* to emphasize its use in valuating a position without minimaxing.

The quality of the static valuation function can be critical, since valuation errors can occasionally be magnified rather than damped by minimaxing. Generally, a valuation function is defined as

a real-valued function based on a collection of measurements or feature-values which describe the position in question. Often a weighted linear combination of feature values is used. A more elaborate function which may take into account various dependencies between features can also be devised. Features can be concrete, such as "the relative piece advantage", or abstract, such as "center control". Abstract features must finally be defined in terms of concrete features. The selection of features is often difficult. The problem of having a program automatically discover good features which "span" the description of a position is completely unsolved, although much interest in this and other related issues exists. Machine learning, as pioneered by Arthur Samuel, has generally meant adaptively modifying coefficients in the evaluation function so as to change its value to better describe positions which are known to be good or bad by other means. A program which truly learns a game, however, must learn features as well as coefficients.

We shall now consider an improved form of the minimax procedure known as the $\alpha\beta$ algorithm. A complete exposition of the $\alpha\beta$ algorithm is given by D.E. Knuth and R.W. Moore in their paper titled "An Analysis of Alpha-Beta Pruning", pp. 293–326, Vol. 6, Artificial Intellegence, 1975.

The $\alpha\beta$ algorithm is based on the observation that if we have the situation



where α is the value of position 1.1 and β is the value of position 1.2.1 with $\beta < \alpha$, and where position 1 is a $\Box$ position, then the score of position 1.2 will be at most β, which cannot compete with the value α to become the value of position 1, and hence we need not compute the scores of the nodes 1.2.2 through 1.2.n. A similar observation holds when $\beta > \alpha$ and node 1 is a $\bigcirc$ position. If these cut-off conditions are systematically tested for and taken advantage of in an appropriate modification of `PICK`, we have the $\alpha\beta$ algorithm.

Exercise 26.2: Explain why the value of position 1.2 will be at most β.

LISP functions for the $\alpha\beta$ algorithm are given below. Note that skip-binding is extensively used.

`(ABPICK B)` returns a dotted-pair $(x \ . \ m)$ where m is an optimal successor position for *box* of position `B`, and x is its score to *box*, where x and m are computed with the $\alpha\beta$ algorithm.

```
(SETQ ABPICK (LAMBDA (B)
                     (HPICK B (COND ((BOXPOS B) -1.1) (T 1.1)
                                    (COND ((BOXPOS B) 1.1)
                                          (T -1.1))
                                    0)))
```

`(BOXPOS B)` returns `T` if v[`B`] is a $\Box$ position and `NIL` if v[`B`] is a $\bigcirc$ position. (Note v denotes the Lisp evaluation function as usual.)

(HPICK B α β N) returns an optimal score-move dotted-pair (x . m) where m is an optimal successor position for *box* of position v[B]. Position v[B] is at depth v[N] in the game tree being minimaxed. The arguments α and β are the □ and ◯ $\alpha\beta$-cutoff values used to reduce the amount of work required.

```
(SETQ HPICK (LAMBDA (B ALPHA BETA N)
                    (COND ((DEEPENOUGH B N) (CONS (SCORE B) NIL))
                          (T (HOPT (GENMOVES B)
                                   (COND ((BOXPOS B) GREATERP) (T LESSP))
                                   BETA ALPHA (PLUS N 1))))))
```

(DEEPENOUGH B N) returns T if v[B] is a terminal position, or is deemed to be deep enough in the game tree. Otherwise NIL is returned. v[N] is the depth of the position v[B] in the game tree.

(GENMOVES B) returns a list of all the successor positions of v[B].

(SCORE B) returns the estimated value to *box* of the position v[B]. It is the static valuation function.

(HOPT G P α β N) returns an optimal score-move dotted-pair (x . m) where m is an optimal successor position in the list v[G] of a superior position which is a □ position if v[P] = GREATERP and a ◯ position if v[P] = LESSP. The list of successor positions v[G] is at depth v[N] in the game tree being heuristically minimaxed. The arguments α and β are the □ and ◯ $\alpha\beta$-cutoff values used to reduce the amount of work required.

```
(SETQ HOPT (LAMBDA (G P ALPHA BETA N)
                   (OJ (CDR G) (HPICK (CAR G) ALPHA BETA N))))

(SETQ OJ (LAMBDA (G W)
                 (COND ((OR (P (CAR W) ALPHA) (NULL G)) W)
                       (T (OPTC (HOPT G P ALPHA (CAR W) N))))))
```

(OPTC Z) returns a score-move dotted-pair (x . m) which is either the score-move pair v[W] or v[Z], or is such that x is a score value of one of the positions in the list v[D] and m is the corresponding successor position which results in attaining that score. The score value x is a greatest score value if v[P] = GREATERP and x is a least score value of the positions in the list v[D] if v[P] = LESSP. In either case, m is a position which is an element of the list v[L] whose score is x.

```
(SETQ OPTC (LAMBDA (Z) (COND ((P (CAR W) (CAR Z)) W) (T Z))))
```

The $\alpha\beta$ algorithm may be elaborated in several ways to cope more effectively with complex games. However, as this is done, it becomes more difficult to express effectively in LISP.

The $\alpha\beta$ algorithm normally declares a position to be "good" if it is possible to follow a line of play from that position which results in a good position in the future. Because of uncertainty about the quality of the static valuation function, it may be safer to consider a position to be good if one can reach a number of good positions in the future. This may be approximated by giving a

position a bonus if many of its successors are deemed to be bad for the opposing player. This device is discussed by Slagle and Dixon in more general form in "Experiments with the M&N Tree-Searching Program", CACM, Vol. 13, No. 3, pp. 147–155, March 1970. Slagle and Dixon give the rather complicated logic needed to adapt the $\alpha\beta$ algorithm to handle this notion in "Game Trees, M&N Minimaxing, and the M&N Alpha-Beta Procedure", A.I. Group Report No. 3, Lawrence Radiation Laboratory, Livermore, California. In order to avoid such complexity, one can compute the bonus for a position based upon the static scores of the successor positions as estimated by the static valuation function. This is relatively easy to incorporate into the usual $\alpha\beta$ algorithm.

It is also interesting to note that a move which leads to a position which has a poor score according to the static valuation function, but which has a dramatically improved score as a backed-up score from later resulting moves is a machine-discovered "trap" move. There may be some advantage to occasionally following such moves, depending, of course, on the reliability of the static valuation function. Active moves such as sacrifices are often trap moves.

We note that, generally, `ABPICK` will use less computation when we require that the list of successor positions produced by the `GENMOVES` function be ordered in sequence from best to worst positions with respect to the player-to-move. In this case the very first move will have the score which is to be eventually backed-up. This yields the maximum number of cut-offs in the course of considering such an ordered list of successor positions. Of course in practice we can only roughly approximate a perfect ordering and, as Knuth and Moore point out, a full ordering isn't really very useful anyway.

The `GENMOVES` function may also selectively omit some of the worst successor positions in order to avoid their consideration by the $\alpha\beta$ algorithm. Such *forward pruning* is not hazardless, but it is a virtual necessity in games such as chess in order to keep the growth of the game tree managable. If detectable, similar positions to those already generated should be omitted. Similar positions may arise due to symmetry or other special circumstances.

It is possible to use the $\alpha\beta$ algorithm to search just a few levels in order to improve the scores used for ordering. This preliminary exploration is a special case of a more general idea called dynamic ordering where we continually re-order successor position lists as the values of the positions are refined, based on more information. In this case an entire game tree must be maintained whose subtrees can be rearranged as required. Even in the fixed-ordering case, it might be convenient to maintain an entire tree so as to reutilize the positions generated in the minimax searching. In fact, in a real game-playing situation, it is possible to use the subtree selected out of the current game tree by the opponent's move as the beginning game tree for the program's response. These elaborate full-tree schemes, particularly dynamic ordering, are of questionable use. Much depends upon the quality of the static valuation function being used.

Another difficult problem which requires a global consideration of the game tree is that of avoiding equivalent sequences of successive positions whose inclusion merely wastes time. Such equivalent sequences are often due to permutations of independent moves which produce the same final position. The $\alpha\beta$ algorithm will cut-off some equivalent sequences, but not those which must be followed to establish initial backed-up values.

An observation of interest is that the `HPICK` and `GENMOVES` functions could be reprogrammed in an appropriate language other than LISP to operate as coroutines, rather than the latter being a

subroutine of the former. In order to do this, we would define genmoves(b) to yield successively, in a common location, the various moves which produce the successors of b. Now when a next son of position b is needed in the course of minimaxing the game tree, the genmoves coroutine is restarted to compute the next move from where it left off within the current incarnation, or else a first move from b if no previous moves have been computed. The genmoves coroutine will signal when no more moves remain to be considered. Note that when a cut-off occurs, we save the time and space of generating all the remaining moves which do not need to be examined. Of course, if genmoves secretly generates a complete list of moves, then this saving is illusory.

The question of when the $\alpha\beta$ algorithm has probed deeply enough in the game tree often depends upon the state of the game at the node we are at when the question is asked. In particular, if that position can be seen to be difficult to valuate statically, then it may pay to go deeper to resolve the difficulty. This is often the case when the position is one in a sequence of exchanges such as jump situations in checkers or piece exchanges in chess. Samuel calls such active positions *pitch* positions. The deepenough test may be that the level at which a position being considered is greater than certain minimum and that the position in question is not a pitch position. The deepenough test may also depend upon the estimated quality of the positions on the path which we are following. Moreover, time and space constraints should preferably be used in place of a simple depth criterion.

The problem of determining the savings gained by using the $\alpha\beta$ algorithm in place of simple minimaxing reduces to counting the number of nodes in a game tree of $d+1$ levels which are visited during a typical $\alpha\beta$ search. For simplicity, we shall assume that every node has a constant branching factor f. Moreover we shall assume perfect ordering of successor positions by their scores, so that the maximum number of $\alpha\beta$ cut-offs occur. Under these assumptions, the number of positions processed is

$$(f+3)(1-f)^{\lceil d/2\rceil}/(1-f) - (d+1) + 2(1-d \bmod 2)f^{\lceil d/2\rceil}.$$

Compared with the $(1-f)^{d+1}/(1-f)$ nodes visited with minimaxing, we see that using the $\alpha\beta$ algorithm is roughly equivalent to reducing the branching factor of our game tree from f to $f^{1/2}$.

Detailed arguments which lead to the formula above are given by J.R. Slagle and J.K. Dixon in "Experiments with some Programs that search Game Trees", JACM, Vol. 16, No. 2, pp. 189–207, April 1969. A more realistic analysis is presented in the paper by Knuth and Moore.

27 The LISP Interpreter Program

The LISP interpreter presented below consists of three major subroutines: *sread*, *seval*, and *swrite*. The *sread* procedure reads input text, decomposes it into atom names and punctuation characters, enters all the new ordinary atoms and new number atoms which are encountered into the atom table and the number table respectively, and constructs an S-expression representing the input. A typed-pointer to this S-expression is returned. If the input S-expression is not an atom, a corresponding list structure will be created in the list area. New ordinary atoms are entered in the atom table with the value: undefined.

The typed-pointer produced as output by *sread* is then passed as input to the *seval* subroutine. The procedure *seval* is the procedure which implements the LISP function `EVAL`. *seval* interpretively evaluates the S-expression pointed to by its input, calling itself recursively to evaluate sub-expressions. The final result is an S-expression which is constructed in the list area if necessary, and a typed-pointer to this result S-expression is returned as output. If the input S-expression is illegal, *seval* prints an error message and goes to the point where *sread* is called to obtain more input. If the original input is evaluatable, the typed-pointer output from *seval* is provided as input to the *swrite* procedure. The *swrite* procedure assembles the names and punctuation characters needed to form a text string representing the given S-expression and prints this string at the terminal.

The entire LISP interpreter is thus of the following form.

1. initialization steps
2. $i \leftarrow sread()$
3. $j \leftarrow seval(i)$
4. $swrite(j)$
5. goto step 2.

We shall present explicit data structures and procedures which comprise much of a working LISP interpreter below. It is common to present an interpreter function for LISP written in LISP as was done in the original LISP 1.5 manual [MIT62]. However, except for its pedagogical value, this is a sterile undertaking. In this book, the C programming language will be used.

Actually we will first use the K programming language rather than C in order to provide a gentle introduction to this closely-related new programming language and to avoid spoiling the fun of those who wish to try their hand at building a C version of this program.

K code is easy to read, so a full definition of K is not given, however a few remarks will be of use. Comments begin with a vertical bar (`|`), and are terminated with another vertical bar or with a carriage-return or line-feed, whichever comes first. Thus a single vertical bar appearing in a line denotes that the remainder of the line is a comment. In publication format, which is used here, any line which has a non-blank in column 1 is a comment line. Also any string constant appearing as a statement is ignored. The C notation for embedded comments may also be used, with the nesting of such comments allowed, so that the `/*`'s must be properly matched by `*/`'s.

Predefined K library subroutines which are used below are:

print(s) The string s is typed on the terminal. Nothing is typed if s is the empty string.

readline(j) A string from the stream file j containing the characters appearing until the next carriage-return is returned. Any accompanying linefeed is discarded. When reading from stream file 0, which is the terminal keyboard, readline will wait until a carriage-return is typed.

numstring(x) The string of characters corresponding to the floating-point number x is returned.

lop(s) The first character of the string s is returned as an integer, and this character is removed from the front of s as a side-effect. If s is the empty string, -1 is returned.

sopen(s) The file named in the string s is opened as a stream file and an *integer*(32) file handle for this file is returned. If the file cannot be opened, a negative integer specifying the reason is returned.

sclose(j) The stream file j is closed.

length(s) The length of the string s is returned.

size(A) The number of entries in the array A is returned.

abs(x) The absolute value of the number x is returned.

In K, the coercion rule for strings appearing where a number is required is that the integer code of the first character of the string is used in place of the string; -1 is used if the string is empty. Moreover all integers appearing in expressions directly or as the result of string coercion are represented by 32-bit integers, and a 32-bit integer may be further coerced to a floating-point value when this latter coercion is required. A *real* is assumed to be a 32-bit floating-point value herein. The coercion rule for a number appearing where a string is required is that the number is coerced to a 32-bit integer by use of the floor function, and then the low 8 bits of this integer are taken as a string of length one containing the corresponding 8-bit byte.

The identifier *loc* is the *address-of* operator, and the symbol `@` is the *contents-of* operator. Conversions via cast operators similar to those introduced in C are used below. The symbol `&` denotes string concatenation.

Relational operators return 0 or 1. The boolean operators *not, xor, and* and *or* denote the corresponding bitwise logical operations. Expressions employed in tests, such as within if-statements or while-statements, are treated as false if the *integer*(32) result is 0, and treated as true otherwise. Moreover, expressions employed in tests are computed using left-to-right lazy-evaluation, so that not all arguments of boolean operators need always be computed within tests.

A *single case* statement has implied breaks inserted after the labeled statement for each case, unlike an ordinary *case* statement in which drop-through occurs.

The following general definitions will be used.
define $n = 5000$;
define $m = 50000$;
define $CR =$ `'fH`;

Now we need to define the following global data structures.

The Atom Table:
structure array *atomtable*$[0 : n-1]$(string *name*; integer(32) L, *bindlist*, *plist*) $= \{(nullstr, 0, 0, 0)\}$;

The Number Hash Index Table:

integer(16) array $nx[0 : n - 1] = -1$;

The Number Table:
union array $numtable[0 : n - 1]$(real num; integer(16) $nlink$);

The Number Table Free Space List Pointer:
integer(32) $nf = -1$;

The Number Table Mark Array: Each 1-bit element of the array is initialized to 0.
integer(1) array $nmark[0 : n - 1] = 0$;

The List Area:
structure array $P[1 : m]$(integer(32) car, cdr);
define $A[j] = P_j.car$;
define $B[j] = P_j.cdr$;

The List-area Free-Space List Pointer:
integer(32) $fp = -1$;

The "put-back" variable:
integer(32) $pb = 0$;

The input string:
string $g = null$;

The input-prompt character string:
string $prompt$;

The input stream file control block pointer:
integer(32) $instream = 0$;

Global ordinary atom typed-pointers:
integer(32) $nilptr$, $tptr$, $currentin$, eaL, $quoteptr$;

define $type(f) = (((f) \gg 28)$ and $15)$;
define $ptrv(f) = ((f)$ and `'ffffffffH`$)$;
define $fctform(t) = ((t) > 9)$;
define $builtin(t) = ((t) = 10$ or $(t) = 11)$;
define $dottedpair(t) = ((t) = 0)$;
define $fct(t) = ((t) = 10$ or $(t) = 12$ or $(t) = 14)$;
define $namedfsf(t) = ((t) > 9$ and $(t) < 14)$;
define $unnamedfsf(t) = ((t) > 13)$;

define $tp(t, j) = ((t \ll 28)$ or $j)$;
define $ud(j) = tp($`'0001B`$, j)$;

```
define se(j) = tp('0000B, j);
define oa(j) = tp('1000B, j);
define nu(j) = tp('1001B, j);
define bf(j) = tp('1010B, j);
define bs(j) = tp('1011B, j);
define uf(j) = tp('1100B, j);
define us(j) = tp('1101B, j);
define tf(j) = tp('1110B, j);
define ts(j) = tp('1111B, j);
```

Now we begin by presenting the *sread* procedure. The procedure *newloc*(u, v), which will be presented later in section 28, is used in *sread*. It returns the address of a new list node whose car and cdr fields are loaded with the values u and v.

```
integer(32) procedure sread():
```

This procedure scans an input string, g, using a lexical token scanning routine E, where E returns 1 if the next token is "(", 2 if the next token is "'", 3 if the next token is ".", 4 if the next token is ")", and a typed-pointer d if the next token is an ordinary atom or number atom stored in row *ptr*(d) of the atom table or number table respectively. Due to the typecode 8 or 9 of d, d is a negative 32-bit integer. The token found by E is stripped from the front of g.

sread constructs an S-expression corresponding to the scanned input string and returns a typed-pointer to it as its result.

There is occasion to "put back" open parentheses and single quote symbols onto g. This is done by loading the global putback variable, *pb*, which is interrogated and reset as appropriate within E.

```
{ integer(32) j, k, t, c;

if (c ← E()) ≤ 0 then return(c);
k ← newloc(nilptr, nilptr); j ← k;
/* we will return k, but we fill node j first. */
if c = 1 then {scan:A_j ← sread();
          next:if (c ← E()) ≤ 2 then
                  {t ← newloc(nilptr, nilptr); B_j ← t; j ← t; if c ≤ 0 then {A_j ← c; goto next};
                   pb ← c; goto scan};

                  if c ≠ 4 then {B_j ← sread(); if E() ≠ 4 then error("bad syntax")};
                  return(k)};
if c = 2 then {A_j ← quoteptr; t ← newloc(nilptr, nilptr); B_j ← t; A_t ← sread(); return(k)};
error("bad syntax")
};
```

Exercise 27.1: The *sread* procedure does not handle the NIL-macro "()". Propose a modification which does handle "()".

Solution 27.1: Just before "$k \leftarrow newloc(nilptr, nilptr)$" insert "if $c = 1$ then if $(k \leftarrow E()) = 4$ then return(*nilptr*) else $pb \leftarrow k$". Another solution, which is adopted here, is to have the E procedure recognize "()" and return *nilptr* when "()" is seen.

The lexical token scanning routine E is presented below.

```
 integer(32) procedure E():
 { integer(32) t, c, sign; string n; real v, f, k;
   define openp = "(";
   define closep = ")";
   define blank = " ";
   define singleq = """";
   define dot = ".";
   define plus = "+";
   define minus = "-";
   define chval(c) = (c-"0");

 procedure getgchar():
{fillg(); return(lop(g))};

 procedure lookgchar():
{fillg(); return(g1)};

 procedure fillg():
{while length(g) = 0 do {print(prompt); g ← readline(instream); prompt ← "*"}};
 integer(32) procedure digit(integer(8) v):
{return("0"≤ v and v ≤ "9")}

 if pb ≠ 0 then {t ← pb; pb ← 0; return(t)};

 do c ← getgchar() until c ≠ blank;
 if c = openp then {while lookgchar() = blank do getgchar();
                    if lookgchar() = closep then {getgchar(); return(nilptr)} else return(1)};
 if c = singleq then return(2);
 if c = closep then return(4);
 if c = dot then if not digit(lookgchar()) then return(3) else {sign ← 1; v ← 0; goto fraction};
 if not(digit(c) or ((c = plus or c = minus) and  (digit(lookgchar()) or lookgchar() = dot))) then
 {n ← c;
  repeat{c ← lookgchar();
       if (c = blank or c = dot or c = openp or c = closep) then break;
```

```
      n ← n&getgchar()};
 return(ordatom(n))};
 if c = minus then {v ← 0; sign ← −1} else {v ← chval(c); sign ← 1};
 while digit(lookgchar()) do v ← 10*v + chval(getgchar());
 if lookgchar() = dot then {getgchar();
 if digit(lookgchar()) then
 fraction: {k ← 1; f ← 0; do {k ← 10*k; f ← 10*f + chval(getgchar())}
  until not digit(lookgchar()); v ← v + f/k}};
 return(numatom(sign*v));
 };
```

Exercise 27.2: The procedure E does not handle LISP input precisely as defined in this book. Describe the sins of omission and commission occuring in E.

Now the routines *numatom* and *ordatom* are presented. The procedure *gc* which is called within *numatom* is the garbage collector routine. It is discussed in section 28.

```
 integer(32) procedure numatom(real r):
/* The number r is looked-up in the number table, and stored there as a lazy number atom if it is
    not already present. The typed-pointer to this number atom is returned. */
  { integer(32) j;
  define hashnum(r) = abs(@((integer(32)pointer)loc(r))) mod n;
  j ← hashnum(r);
  while nx_j ≠ −1 do
    if num[nx_j] = r then {j ← nx_j; goto ret} else if (j ← j + 1) = n then j ← 0;
   if nf < 0 then {gc(); if nf < 0 then error("The number table is full.")};
  nx_j ← nf; j ← nf; nf ← nlink_nf; num_j ← r;
 ret: return(nu(j))
  };
```

Exercise 27.3: Introduce a trick so that *gc* will be called when the number table is about 90 percent full, rather than allowing the last 10 percent to be used before *gc* is called.

```
 integer(32) procedure ordatom(string s):
/* The ordinary atom whose name is given as the argument string s is looked-up in the atom
    table and stored there as an atom with the value undefined if it is not already present. The
    typed-pointer to this ordinary atom is then returned. */
 { integer(32) j, c;
  define hashname(s) = abs(s[1] ≪ 16 + s[length(s)] ≪ 8 + length(s)) mod n;
  j ← hashname(s); c ← 0;
  while length(name_j) ≠ 0 do
    if name_j = s then goto ret else
    if (j ← j + 1) = n then {j ← 0; if (c ← c + 1) > 1 then error("atom table is full")};
  name_j ← s; L_j ← ud(j);
```

ret: return($oa(j)$)
};

The *initlisp* procedure initializes all the needed global variables. It must be called before *sread* is used.

procedure *initlisp*():
/* *The atom table is initialized. The atoms currentin,* NIL, T, *eaL, and those given in the array BI, and in the initialization file are defined.* */
{string array $BI[\,] =$ {"CAR", "CDR", "CONS", "LAMBDA", "SPECIAL", "SETQ", "ATOM", "NUMBERP", "QUOTE", "LIST", "DO", "COND", "PLUS", "TIMES", "DIFFERENCE", "QUOTIENT", "POWER", "FLOOR", "MINUS", "LESSP", "GREATERP", "EVAL", "EQ", "AND", "OR", "SUM", "PRODUCT", "PUTPLIST", "GETPLIST", "READ", "PRINT", "PRINTCR", "MKATOM"};

integer(8) array $BItype[\,] =$
{10,10,10,11,11, 11,10,10,11,10, 10,11,10,10,10,
10,10,10,10,10, 10,10,10,11,11, 10,10,10,10,10, 10,10,10};

integer(32) i, j;
/* *Define the atomtable entries for the builtin functions and special-forms.* */
for $i \leftarrow 1$:size(BI) do $\{j \leftarrow ptrv(ordatom(BI_i))$; $L_j \leftarrow tp(BItype_i, i)$ };

$nilptr \leftarrow ordatom($"NIL"$)$; $L[ptrv(nilptr)] \leftarrow nilptr$;
$tptr \leftarrow ordatom($"T"$)$; $L[ptrv(tptr)] \leftarrow tptr$;
$L[currentin \leftarrow ptrv(ordatom($"$currentin$"$))] \leftarrow nilptr$;
$L[eaL \leftarrow ptrv(ordatom($"$eaL$"$))] \leftarrow nilptr$;
$quoteptr \leftarrow ptrv(ordatom($"QUOTE"$))$;
/* *Setup the list-area free-space list.* */
for $i \leftarrow 1 : m$ do $\{B_i \leftarrow fp$; $fp \leftarrow i\}$
/* *Set-up the number table free-space list and initialize the Atom table.* */
for $i \leftarrow 0 : n-1$ do $\{nlink_i \leftarrow nf$; $nf \leftarrow i$; $bindlist_i, plist_i \leftarrow nilptr\}$;
/* *Read the file of functions and special forms to be pre-defined. These are:* APPEND, REVERSE, EQUAL, APPLY, INTO, ONTO, NOT, NULL, ASSOC, NPROP, PUTPROP, GETPROP, *and* REMPROP. */
fileinit();
};

Exercise 27.4: Program the *fileinit* procedure. Hint: temporarily modify *instream*, and use *sread* and *seval* without *swrite*.

Now the *swrite* routine used for output is presented.

procedure *swrite*(integer(32) j):
/* *The S-expression pointed-to by the typed-pointer j is typed-out.* */

```
{ integer(32) listsw, i;

i ← ptrv(j);
single case type(j) {
0:{ "check for a list"
    j ← i; while type(B_j) = 0 do j ← B_j;
    if B_j = nilptr then listsw ← 1 else listsw ← 0;
    print("(");
    if listsw then
    repeat {swrite(A_i); if (i ← B_i) = nilptr then goto fin else print(" ")}
    else {swrite(A_i); print("."); swrite(B_i)};
    fin:print(")")};

8:{print(name_i)};
9:{print(numstring(num_i))};
10:{print("{builtin function:"); print(name_i); print("}")};
11:{print("{builtin special-form:"); print(name_i); print("}")};
12:{print("{user-defined function:"); print(name_i); print("}")};
13:{print("{user-defined special-form:"); print(name_i); print("}")};
14:{print("{unnamed function}")};
15:{print("{unnamed special-form}")}
}};

/* Now the central interpreter procedure seval is presented. */

integer(32) procedure seval(copy integer(32) p):
/* Evaluate the S-expression pointed to by the typed-pointer p, construct the result value as necessary, and return a typed-pointer to the result. */
{integer(32) ft, t, v, j, f, fa, na;  pointer endeaL; static real s;

if type(p) ≠ 0 then {
 /* p does not point to a non-atomic S-expression.

    If p is a type-8 typed-pointer to an ordinary atom whose value is a builtin or user-defined function or special form, then a typed-pointer to that atom-table entry with the typecode 10, 11, 12, or 13, depending upon the value of the atom, is returned. Note this permits us to know the names of functions and special forms.

    If p is a type-8 typed-pointer to an ordinary atom whose value is not a builtin or user-defined function or special form, and thus has the typecode 8, 9, 14, or 15, then a typed-pointer corresponding to the value of this atom is returned.
```

If p is a non-type-8 typed-pointer to a number atom or to a function or special form (named or unnamed) then the same pointer p is returned. */*

if $(t \leftarrow type(p)) \neq 8$ then return(p) else $j \leftarrow ptrv(p)$;

/* *The association list is implemented with shallow call-time binding in the atom-table, so the current values of all atoms are found in the atom table.* */

if $(t \leftarrow type(L_j)) = 1$ then $error(name_j \& \text{" is undefined."})$;

if $namedfsf(t)$ then return($tp(t, j)$);

return(L_j)};

Exercise 27.5: What are the pros and cons of replacing the statement "return(L_j)" in the compound statement just above with the statement:
"if $L_j = p$ and $bindlist_j \neq nilptr$ then return($A[bindlist_j]$) else return(L_j)"?
Hint: think about evaluating `(F G)` where `F` and `G` are defined by
`(SETQ F (SPECIAL (G) ((EVAL G) 1)))` and `(SETQ G (LAMBDA (F) (PLUS F F))).`

define $ciLp = L[currentin]$;

define $eaLp = L[eaL]$;

Save the list consisting of the current function and the supplied arguments as the top value of the *currentin* list to protect it from garbage collection. The *currentin* list is a list of lists.

$ciLp \leftarrow newloc(p, ciLp)$;

Compute the function (or special form) to be applied.

$f \leftarrow seval(A_p)$; $ft \leftarrow type(f)$; if not $fctform(ft)$ then $error$("invalid function");

$f \leftarrow ptrv(f)$; if not $unnamedfsf(ft)$ then $f \leftarrow ptrv(L_f)$;

Exercise 27.6: Would it be useful to insert the code: "if $ft = 8$ then $\{f \leftarrow bindlist[ptrv(f)]$; if $f \neq nilptr$ then $f \leftarrow A_f$; $ft \leftarrow type(f)\}$;" just after the statement "$ft \leftarrow type(f)$" above?

Now let go of the supplied input function.

$A[ciLp], p \leftarrow B_p$;

/* *If f is a function (not a special form), build a new list of its evaluated arguments, and then let go of the list of supplied arguments, replacing it with the new list of evaluated arguments.* */

if $fct(ft)$ then { /* *Compute the actual arguments.* */

$eaLp \leftarrow newloc(nilptr, eaLp)$;

/* *Evaluate the actual arguments and build a list by tail-cons-ing!.* */

$endeaL \leftarrow loc(A[eaLp])$;

while $p \neq nilptr$ do

{@$endeaL \leftarrow newloc(seval(A_p), nilptr)$; $endeaL \leftarrow loc(B[$@$endeaL])$; $p \leftarrow B_p$};

/* *Set p to be the first node in the evaluated arguments list.* */

$p \leftarrow A[eaLp]$;

/* *Throw away the current supplied arguments list by popping the currentin list.* */
$ciLp \leftarrow B[ciLp]$ };

/* *At this point, p points to the first node of the actual argument list. If* $p = nilptr$, *we have a function or special-form with no arguments.* */

if not *builtin*(*ft*) then {
/* *f is a non-builtin function or non-builtin special form. do shallow binding of the arguments and evaluate the body of* f. */
$fa \leftarrow A_f$; /* *fa points to the first node of the formal argument list.* */
$na \leftarrow 0$; /* *na counts the number of arguments.* */
/* *Run through the arguments and place them as the top values of the formal argument atoms in the atom-table. Push the old value of each formal-argument atom on its binding list.* */
if $type(fa) = 8$ then $\{t \leftarrow ptrv(fa)$; $bindlist_t \leftarrow newloc(L_t, bindlist)$; $L_t \leftarrow p\}$ else
while $p \neq nilptr$ do
$\{t \leftarrow ptrv(A_{fa})$; $fa \leftarrow B_{fa}$; $bindlist_t \leftarrow newloc(L_t, bindlist_t)$;
$v \leftarrow A_p$; if $namedfsf(type(v))$ then $v \leftarrow L_{ptrv(v)}$; $L_t \leftarrow v$; $na \leftarrow na + 1$; $p \leftarrow B_p\}$;

Exercise 27.7: Why did we need to build the *eaL* list, rather than just directly place each *seval* result as the new value of the corresponding formal argument?

Solution 27.7: To make sure that the successive formal argument bindings don't effect the evaluation of succeeding actual arguments which happen, directly or indirectly, to involve free variables whose names are also used within the current list of formal arguments, the rebinding of all formal arguments is delayed until all the actual arguments have been evaluated.

Now apply the non-builtin function or special form f.
$v \leftarrow seval(B_f)$;

/* *Unbind the actual arguments.* */
$fa \leftarrow A_f$;
if $type(fa) = 8$ then $\{t \leftarrow ptrv(fa)$; $L_t \leftarrow A[bindlist_t]$; $bindlist_t \leftarrow B[bindlist_t]\}$ else
while $na > 0$ do
$\{t \leftarrow ptrv(A_{fa})$; $fa \leftarrow B_{fa}$; $L_t \leftarrow A[bindlist_t]$; $bindlist_t \leftarrow B[bindlist_t]$; $na \leftarrow na - 1\}$;
}; /* *end not builtin* */

Exercise 27.8: Explain what this program does about an excess or deficit of actual arguments for a user-defined function or special form. Can such a mismatching have a useful meaning? Suggest a change which will report such a mismatching to the user.

Exercise 27.9: Explain the meaning of the code which is executed when $type(fa) = 8$ is true in the "not builtin" block above. Hint: think how an atom in the position of the formal argument

list in a λ-expression could be interpreted to provide a device for defining functions and special forms, like SUM, which can be called with varying numbers of actual arguments.

```
else { /* begin builtin */
define U1 = A[p]; define U2 = A[B[p]];  /* U stands for unevaluated */
define E1 = A[p]; define E2 = A[B[p]];  /* E stands for evaluated */
define Sexp(t) = (type(t) = 0 or type(t) = 8 or type(t) = 9);

v ← nilptr;

single case f of {
1:{"car" if not dottedpair(type(E1)) then error("illegal car argument"); v ← A_E1};
2:{"cdr" if not dottedpair(type(E1)) then error("illegal cdr argument"); v ← B_E1};
3:{"cons" if Sexp(E1) and Sexp(E2) then v ← newloc(E1, E2)
else error("Illegal CONS argument")};
/* For LAMBDA or SPECIAL, we could check that U1 is a list of ordinary atoms.*/
4:{"lambda"  v ← tp(14, newloc(U1, U2))};
5:{"special"  v ← tp(15, newloc(U1, U2))};

6:{"setq" if type(U1) ≠ 8 then error("illegal assignment");
        t ← seval(U2); v ← ptrv(U1);
        case type(t) of
         {0:"dotted-pair"
          8:"ordinary atom"
          9:"number atom"
            {L_v ← t; break};
          10:"builtin function"
          11:"builtin special form"
          12:"user-defined function"
          13:"user-defined special form"
            {L_v ← L_ptrv(t); break};
          14:"unnamed function"
            {L_v ← uf(ptrv(t)); break}
          15:"unnamed special form"
            {L_v ← us(ptrv(t)); break}
          };
```

Exercise 27.10: Explain why a case above for *type*(t)=1 is not necessary.

```
    v ← eval(U1) };

7:{"atom" if type(E1) = 8 or type(E1) = 9 then v ← tptr};
```

8:{"numberp" if $type(E1) = 9$ then $v \leftarrow tptr$};
9:{"quote" $v \leftarrow U1$};
10:{"list" $v \leftarrow p$};
11:{"do" while $p \neq nilptr$ do $\{v \leftarrow A_p\ \}$; $p \leftarrow B_p$};
12:{"cond" while $p \neq nilptr$ do
if $seval(A[t \leftarrow A_p]) \neq nilptr$ then $\{v \leftarrow seval(A[B_t])$; break$\}$ else $p \leftarrow B_p$};
13:{"plus" $v \leftarrow numatom(numtab[ptrv(E1)] + numtab[ptrv(E2)]))\ \}$;
14:{"times" $v \leftarrow numatom(numtab[ptrv(E1)] * numtab[ptrv(E2)]))\ \}$;
15:{"difference" $v \leftarrow numatom(numtab[ptrv(E1)] - numtab[ptrv(E2)]))\ \}$;
16:{"quotient" $v \leftarrow numatom(numtab[ptrv(E1)]/numtab[ptrv(E2)]))\ \}$;
17:{"power" $v \leftarrow numatom(numtab[ptrv(E1)] \uparrow numtab[ptrv(E2)]))\ \}$;
18:{"floor" $v \leftarrow numatom(floor(numtab[ptrv(E1)]))\ \}$;
19:{"minus" $v \leftarrow numatom(-numtab[ptrv(E1)])\ \}$;
20:{"lessp" if $numtab[ptrv(E1)] < numtab[ptrv(E2)]$ then $v \leftarrow tptr$};
21:{"greaterp" if $numtab[ptrv(E1)] > numtab[ptrv(E2)]$ then $v \leftarrow tptr$};
22:{"eval" $v \leftarrow seval(E1)$};
23:{"eq" $v \leftarrow$ if $E1 = E2$ then $tptr$ else $nilptr$};
24:{"and" while $p \neq nilptr$ and $seval(A_p) \neq nilptr$ do $p \leftarrow B_p$;
if $p = nilptr$ then $v \leftarrow tptr$};
25:{"or" while $p \neq nilptr$ and $seval(A_p) = nilptr$ do $p \leftarrow B_p$;
if $p \neq nilptr$ then $v \leftarrow tptr$};
26:{"sum" $s \leftarrow 0$; while $p \neq nilptr$ do $\{s \leftarrow s + numtab[ptrv(A_p)]$; $p \leftarrow B_p\}$; $v \leftarrow numatom(s)$};
27:{"product" $s \leftarrow 1$; while $p \neq nilptr$ do $\{s \leftarrow s * numtab[ptrv(A_p)]$; $p \leftarrow B_p\}$; $v \leftarrow numatom(s)$};
28:{"putplist" $plist[ptrv(v \leftarrow E1)] \leftarrow E2$};
29:{"getplist" $v \leftarrow plist[ptrv(E1)]$};
30:{"read" print(CR); print("!"); $prompt \leftarrow NULL$; $v \leftarrow sread$};
31:{"print" if $p = nilptr$ then print(" ") else
while $p \neq nilptr$ do $\{swrite(A_p)$; print(" "); $p \leftarrow B_p\}$};
32:{"printcr" if $p = nilptr$ then print(CR) else
while $p \neq nilptr$ do $\{swrite(A_p)$; print(CR); $p \leftarrow B_p\}$};
33:{"mkatom" $v \leftarrow ordatom(name[ptrv(E1)]\&name[ptrv(E1)]$}

} } /* *end builtin* */;

Exercise 27.11: Would it be a good idea if the "cond" case were rewritten as follows?
12:{"cond" while $p \neq nilptr$ do if $(v \leftarrow seval(A[t \leftarrow A_p])) \neq nilptr$
then {if $B_t \neq nilptr$ then $v \leftarrow seval(A[B_t])$; break} else $p \leftarrow B_p$};

Exercise 27.12: Modify the "cond" case to make the final "else" option not require a T predicate; i.e, `(COND` $(P_1\ R_1) \cdots (P_k\ R_k)\ R_{k+1}$`)` is to be interpreted as `(COND` $(P_1\ R_1) \cdots (P_k\ R_k)$ `(T` R_{k+1}`))`.

Exercise 27.13: If there are unexpectedly few actual arguments provided for a builtin function

or special form, this program may fail catastrophically. Suggest an approach to detect this type of error and prevent any failure. What happens if too many arguments are given?

Exercise 27.14: Modify the builtin function READ to take an optional filename argument, and to read its input from this file when specified.

/* *Pop the eaL list or pop the currentin list, whichever is active.* */
if *fct*(*ft*) then $eaLp \leftarrow B[eaLp]$ else $ciLp \leftarrow B[ciLp]$;

return(v)
} *seval*;

Exercise 27.15: Functions and special forms are not included in the class of atoms, and thus cannot appear in dotted-pairs. There is an exception to this statement. What is it? Could the class of atoms be enlarged to include functions? Should it be?

Exercise 27.16: The builtin special form LABEL is not handled. Provide code to implement LABEL. Hint: use the binding list of the label variable.

Exercise 27.17: The builtin functions RPLACA and RPLACD are not provided above. Write the code needed to implement them.

Exercise 27.18: Often lists are used as one-dimensional arrays. In order to access the i-th element, a function called ACCESS may be defined. For example, with:

```
(SETQ ACCESS (LAMBDA (L I)
                     (COND ((EQ I 1) (CAR L))
                           (T (ACCESS (CDR L) (PLUS I -1)))))).
```

This function is often painfully slow. Add iterative code to make ACCESS a builtin function.

Exercise 27.19: Most versions of LISP provide the potentially-infinite class of functions CAAR, CADR, CDAR, CDDR, CAAAR, ..., where the interior A and D letters in the name of such a function specifies that a corresponding sequence of CAR and CDR applications is to be done to the argument S-expression. Modify the program given here to provide for this class of functions.

Exercise 27.20: The builtin function BODY is not given above. Write the code needed to provide this function.

Exercise 27.21: Many of the sections of code for builtin functions and special forms given above do not adequately check their arguments. In each such case, propose suitable error-checking code so that the LISP interpreter cannot directly or indirectly crash due to such incorrect input.

Exercise 27.22: Define a builtin function named TYPECODE which returns the number typecode of its argument. Explain how this can be used to write LISP functions which check their arguments and issue error reports when necessary.

Exercise 27.23: Why isn't it a good idea to make REVERSE explicitly builtin with the following code?

{integer(32) i, j, t; $i \leftarrow E1$; $j \leftarrow B_i$; $B_i \leftarrow nilptr$;
while $j \neq nilptr$ do $\{t \leftarrow B_j$; $B_j \leftarrow i$; $i \leftarrow j$; $j \leftarrow t\}$; $v \leftarrow i\}$.

Exercise 27.24: Define a builtin special form called TSETQ which takes an ordinary atom A and an S-expression s as input and sets the global top-level value of A to be $v[s]$, and returns $v[s]$ as its result.

Exercise 27.25: Define a builtin function called PEVAL which thakes an ordinary atom A as input and returns the previous pushed-down value of A at the head of the *bindlist* of A.

Now the main LISP interpreter procedure and the error procedure are given.

main procedure *LISP*():
{*initlisp*(); free(*initlisp*);
entry loop: *prompt* $\leftarrow$ "$*$"; *swrite*(*seval*(*sread*())); print(CR); goto loop};

procedure *error*(string s):
{*ciLp*, *eaLp* $\leftarrow$ *nilptr*; print(s); print(CR); returnto loop};

Exercise 27.26: The K-language contains the ability to free the space used by a subroutine in a non-shared program. This is used above applied to *initlisp*, so that after *initlisp* has been executed, the space it occupies can be reused. Actually it is unnecessary to have or execute *initlisp* at all in the final executable file containing the LISP interpreter program. This executable file could contain the atomtable, number table, and list area already pre-set with the desired contents! How can such an executable file be conveniently created? Is this a feature which requires the aid of the operating system?

Exercise 27.27: Explain the returnto statement used in *error*. How is returnto different from goto? How might returnto be implemented within an K runtime subroutine?

Exercise 27.28: Repair the code given above so that for the functions PRINT and PRINTCR, the result "NIL" will not appear when these functions are executed at the top level.

Exercise 27.29: Modify the LISP interpreter program so that it will print the name of every invoked function or special form each time they are called when an ordinary atom named TRACE is not NIL.

Exercise 27.30: Modify the LISP interpreter program so that, when @x is typed-in, the interpreter will take its subsequent input from the file named x, until EOF is encountered.

28 Garbage Collection

As input S-expressions are provided to the LISP interpreter to evaluate, various list structures in the list area are constructed, both from our input and during the process of evaluation. Many of these list structures have no use after they have been initially used. Such useless data objects are

called *garbage* data objects. In the case of list area nodes, a node is garbage if it cannot be reached. This means that it is not a node in any list structure which is the value of an ordinary atom, including user-defined function and special form values, nor is it reachable from any pushed-down values held in a *bindlist* list, nor does it occur within any property lists.

We need to find such useless list nodes and put them on the list-area free-space list so they may be reused; otherwise we will run out of list area memory on all but modest computations. This process is called *garbage collection*. A procedure for performing garbage collection called *gc* is given below.

Useless garbage entries also accumulate in the number table as input numbers and computed numbers are created and become useless. The *gc* procedure also collects the garbage entries in the number table and makes them available for reuse. The *gc* procedure is only invoked when the list area is exhausted or when the number table cannot hold more numbers. In particular the *gc* procedure is called from within the *newloc* procedure given below whenever it is discovered that no list nodes are available.

Exercise 28.1: Enumerate all the places in the LISP interpreter where the *gc* procedure is called.

```
integer(32) procedure newloc(integer(32) ca, cd):
{integer(32) t;
 if fp < 0 then {gc(); if fp < 0 then error("out of space")};
 t ← fp; fp ← B_fp; A_t ← ca; B_t ← cd;
 return(t)};

procedure gc():
{integer(32) i, t;

 define marked(p) = ((A_p and '08000000H) ≠ 0);
 define marknode(p) = A_p ← A_p or '08000000H;
 define unmark(p) = A_p ← A_p and 'f7ffffffH;

 procedure gcmark(copy integer(32) p):

 {t ← type(p);
 if t = 0 or t > 11 then
{p ← ptrv(p); if marked(p) then return; marknode(p); gcmark(A_p); gcmark(B_p)} else
 if t = 9 then nmark[ptrv(p)] ← 1
};
/* Run through the atom table and mark all accessible list nodes, including all list nodes within
   or reachable from pushed-down values on bindlists and property lists. */
 for i ← 0 : n − 1 do {gcmark(L_i); gcmark(bindlist_i); gcmark(plist_i)};
```

```
/* Delete all the unneeded numbers. Note we have pinned items. */
 nf ← −1; for i ← 0 : n − 1 do nx_i ← −1;
 for i ← 0 : n − 1 do if nmark_i = 0 then {nlink_i ← nf; nf ← i} else
  {/* Restore num_i */
   t ← hashnum(num_i);
   while nx_t ≠ −1 do if (t ← t + 1) = n then t ← 0; nx_t ← i; nmark_i ← 0};

/* Build the new list-node free-space list. */
 fp ← −1;
 for i ← 1 : m do if not marked(i) then {B_i ← fp; fp ← i} else unmark(i)
 };
```

Exercise 28.2: What purpose do the assignments of *nilptr* within the *error* procedure serve?

Exercise 28.3: Some list-nodes allocated during the execution of *sread* are in peril of being erroneously-reclaimed during garbage-collection, if it should happen to occur while *sread* is executing. Propose code to repair this bug.

Solution 28.3: Define the global integer *sk*, and introduce the statement:
"$L[sk \leftarrow ptrv(ordatom(\texttt{"sreadlist"}))] \leftarrow nilptr$" in *initlisp*.
Then macro-define $skp = L[sk]$, and replace the statement "$k \leftarrow newloc(nilptr, nilptr)$" in *sread* with "$skp \leftarrow newloc(nilptr, skp)$; $A_{skp}, k \leftarrow newloc(nilptr, nilptr)$", and non-recursively replace each occurrence of "return(k)" in *sread* with "$skp \leftarrow B_{skp}$; return(k)".

Exercise 28.4: Can you devise a way to discard unneeded ordinary atoms and thus reclaim space in the atom table for reuse? Hint: first try to handle those unreferenced ordinary atoms whose values are undefined.

Exercise 28.5: Can you cause *gc* to loop forever by building a circular list using `RPLACA` and/or `RPLACD`?

Exercise 28.6: The K language passes procedure arguments by value, that is: a copy of each actual argument is passed, unless reference-type arguments are explicitly specified. Also, arguments can be assigned values within a procedure; in the case of a copy argument, this merely changes the copy. Use this device and add code to the *gc* procedure which looks ahead from a node p, and avoids recursivly calling *gc* when at most one of the nodes A_p or B_p need to be marked. What benefits can be expected from using this code?

Exercise 28.7: Program the entire LISP interpreter in LISP to run on an available LISP system. Some trickery will be needed to make your interpreter conform to the GOVOL dialect of the LISP language given in this book.

29 LISP in C

The following C program runs on MS-DOS and on Unix/Linux from the command-line, although some minor fiddling may be needed to get it compiled in a specific environment..

```
/* Filename:  c:\lisp\lisp.c          Revision Date: Sept. 3, 1999 */
/* ================================================== */
/* --------------------------------------------------
   LISP INTERPRETER

   This progam is a GOVOL LISP interpreter. This interpreter consists of three major functions:
   SREAD, SEVAL, and SWRITE. SREAD scans the input string for input S-expressions (atoms
   and dotted pairs) and returns a corresponding typed- pointer. The SEVAL function takes as
   input a typed-pointer p to an input S-expression and evaluates it and returns a typed pointer
   to its result. SWRITE takes as input the typed pointer returned from SEVAL and prints out
   the result.

   LISP input lines beginning with a "/" are comment lines. Indirect input text is taken from a
   file Z to replace the directive of the form "@Z". SEVAL tracing can be turned on by using the
   directive "!trace", and turned off with the directive "!notrace".
   -------------------------------------------------- */
/* ================================================== */

#define int16 int
#define int32 long
#define forward

#include "cenv.h"
/* The header file cenv.h declares strlen(), strcpy(), strcmp(), calloc(), fflush(), fopen(),
   fclose(), fprintf(), sprintf(), fgetc(), labs(), floor(), and pow().
   Also the type FILE is defined, and the longjump register-save structure
   template: jmp_buf is defined. This include will need to be modified
   for any particular system. */

#define NULL 0L
#define EOF (-1)
#define EOS (0)

#define EQ ==
#define OR ||
#define AND &&
#define NOT !

#define n 1000          /* n = size of atom table and number table */
#define m 6000          /* m = size of list-area */
```

```
jmp_buf env;          /* struct to hold environment for longjump */
char *sout;           /* general output buffer pointer */

/* The atom table */
struct Atomtable {char name[16]; int32 L; int32 bl; int32 plist;} Atab[n];

/* The number table is used for storing floating point numbers.  The field
   nlink is used for linking number table nodes on the number table free space
   list. */
union Numbertable {double num; int16 nlink;} Ntab[n];

/* the number hash index table */
int16 nx[n];

/* the number table free space list head pointer */
int16 nf=-1;

/* the number table mark array is used in garbage collection to mark words
   not to be returned to the free space list */
char nmark[n]; /* an array of 1-bit entries would suffice */

/* The list area */
struct Listarea {int32 car; int32 cdr;} *P;

/* the list area free space list head pointer */
int16 fp=-1;

/* the put-back variable */
int32 pb=0;

/* The input string and related pointers */
char *g,*pg,*pge;

/* the input stream stack structure and head pointer */
struct Insave {struct Insave *link; char *pg, *pge; char g[202]; FILE *filep;};
struct Insave *topInsave;

/* the input prompt character */
char prompt;

/* seval depth count and trace switch */
int16 ct=0, tracesw=0;

/* Global ordinary atom typed-pointers */
```

```
int32 nilptr,tptr,currentin,eaL,quoteptr,sk,traceptr;

/* define global macros */

#define A(j)           P[j].car
#define B(j)           P[j].cdr
#define AL(j)          Atab[j].L
#define Abl(j)         Atab[j].bl

#define type(f)        (((f)≫28) & 0xf)
#define ptrv(f)        (0x0fffffff & (f))
#define sexp(t)        ((t) EQ 0 OR (t) EQ 8 OR (t) EQ 9)
#define fctform(t)     ((t)>9)
#define builtin(t)     ((t) EQ 10 OR (t) EQ 11)
#define userdefd(t)    ((t) EQ 12 OR (t) EQ 13)
#define dottedpair(t)  ((t) EQ 0)
#define fct(t)         ((t) EQ 10 OR (t) EQ 12 OR (t) EQ 14)
#define unnamedfsf(t)  ((t)>13)
#define namedfsf(t)    ((t)>9 AND (t)<14)
#define tp(t,j)        ((t) | (j))
#define ud(j)          (0x10000000 | (j))
#define se(j)          (0x00000000 | (j))
#define oa(j)          (0x80000000 | (j))
#define nu(j)          (0x90000000 | (j))
#define bf(j)          (0xa0000000 | (j))
#define bs(j)          (0xb0000000 | (j))
#define uf(j)          (0xc0000000 | (j))
#define us(j)          (0xd0000000 | (j))
#define tf(j)          (0xe0000000 | (j))
#define ts(j)          (0xf0000000 | (j))

/* variables used in file operations */
FILE *filep;
FILE *logfilep;

/* forward references */
forward int32 seval(int32 i);
forward void initlisp(void);
forward int32 sread(void);
forward void swrite(int32 i);
forward int32 newloc(int32 x, int32 y);
forward int32 numatom (double r);
forward int32 ordatom (char *s);
forward void gc(void);
forward void gcmark(int32 p);
```

```
forward char getgchar(void);
forward char lookgchar(void);
forward void fillg(void);
forward int32 e(void);
forward void error(char *s);
forward int16 fgetline(char *s, int16 lim, FILE *stream);
forward void ourprint(char *s);

/* ========================================================== */
void main(void)
/* ----------------------------------------------------------
   This is the main read/eval/print loop.
   ---------------------------------------------------------- */
{initlisp();
 setjmp(env); /* calling error() returns to here by longjmp() */

 for (;;) {prompt='*'; swrite(seval(sread())); ourprint("\n");}
}

/* ========================================================== */
void error(char *msg)
/* char *msg;   message to type out */
/* ----------------------------------------------------------
   Type out the message in msg and do a longjmp() to top level.
   ---------------------------------------------------------- */
{int32 i,t;

 /* discard all input S-expression and argument list stacks */
 Atab[currentin].L=nilptr; Atab[eaL].L=nilptr; Atab[sk].L=nilptr;
 /* reset all atoms to their top-level values */
 for (i=0; i<n; i++) if ((t=Atab[i].bl)≠nilptr)
    {while (t≠nilptr) t=B(t); Atab[i].L=A(t); Atab[i].bl=nilptr;}
 ct=0; ourprint("::"); ourprint(msg); ourprint("\n");
 longjmp(env,-1);
}

/* ========================================================== */
void ourprint(char *s)
/* char *s; message to be typed out and logged */
/* ----------------------------------------------------------
   Print the string s in the log-file and on the terminal.
   ---------------------------------------------------------- */
{printf("%s",s); fprintf(logfilep,"%s",s); fflush(logfilep);}

/* ========================================================== */
```

```
void initlisp(void)
/* ----------------------------------------------------------
   This procedure installs all builtin functions and special forms into the atom table.  It also
   initializes the number table and listarea.
   ---------------------------------------------------------- */
{int32 i;

 static char *BI[] =
    {"CAR","CDR","CONS","LAMBDA","SPECIAL","SETQ","ATOM","NUMBERP","QUOTE",
     "LIST","DO","COND","PLUS","TIMES","DIFFERENCE","QUOTIENT","POWER",
     "FLOOR","MINUS","LESSP","GREATERP","EVAL","EQ","AND","OR","SUM","PRODUCT",
     "PUTPLIST","GETPLIST","READ","PRINT","PRINTCR","MKATOM","BODY","RPLACA",
     "RPLACD","TSETQ", "NULL", "SET"
    };

 static char BItype[] =
    {10,10,10,11,11,11,10,10,11,10,
     10,11,10,10,10,10,10,10,10,10,
     10,10,10,11,11,10,10,10,10,10,
     10,10,10,10,10,10,11,10,11
    };

 /* number of builtin's in BI[~] and BItype[~] above */
#define NBI 39

 /* allocate a global character array for messages */
 sout=(char *)calloc(80,sizeof(char));

 /* allocate the input string */
 g=(char *)calloc(202,sizeof(char));

 /* allocate the list area */
 P=(struct Listarea *)calloc(m,sizeof(struct Listarea));

/* initialize atom table names and the number table */
 for (i=0; i<n; i++)
    {Atab[i].name[0]='\0'; nmark[i]=0; nx[i]=-1; Ntab[i].nlink=nf; nf=i;}

 /* install typed-case numbers for builtin functions and and special forms into
   the atom table */
 for (i=0; i<NBI; i++)
    {Atab[ptrv(ordatom(BI[i]))].L=tp((((int32)BItype[i])≪28),(i+1));}

 nilptr=ordatom("NIL"); Atab[ptrv(nilptr)].L=nilptr;
 tptr=ordatom("T");     Atab[ptrv(tptr)].L=tptr;
```

```
 quoteptr=ordatom("QUOTE");

 currentin=ptrv(ordatom("CURRENTIN")); Atab[currentin].L=nilptr;
 eaL=ptrv(ordatom("eaL")); Atab[eaL].L=nilptr;
 sk=ptrv(ordatom("sreadlist")); Atab[sk].L=nilptr;

#define cilp Atab[currentin].L
#define eaLp Atab[eaL].L
#define skp Atab[sk].L

 /* initialize the bindlist (bl) and plist fields */
 for (i=0; i<n; i++) Atab[i].bl=Atab[i].plist=nilptr;

 /* set up the list area free space list */
 for (i=1; i<m; i++) {B(i)=fp; fp=i;}

 /* read in predefined functions and special forms from the file lispinit:
    these are APPEND, REVERSE, EQUAL, APPLY, INTO, ONTO, NOT, NULL, ASSOC,
    NPROP, PUTPROP, GETPROP, and REMPROP */

 /* open the logfile */
 logfilep=fopen("lisp.log","w");
 ourprint("ENTERING THE GOVOL LISP INTERPRETER\n");

 /* establish the input buffer and the input stream stack */
 topInsave=NULL;
 strcpy(g,"@lispinit ");
 pg=g; pge=g+strlen(g);/* initialize start & end pointers to string g */
 filep=stdin;
}

/* ================================================== */
int32 sread(void)
/* --------------------------------------------------
   This procedure scans an input string g using a lexical token scanning routine, e(), where e()
   returns


   1 if the token is '('
   2 if the token is "'
   3 if the token is '.'
   4 if the token is ')'
```

or a typed pointer d to an atom or number stored in row ptrv(d) in the atom or number tables. Due to the typecode (8 or 9) of d, d is a negative 32-bit integer. The token found by e() is stripped from the front of g.

SREAD constructs an S-expression and returns a typed pointer to it as its result.
−− */

```
{int32 j,k,t,c;

 if ((c=e())≤0) return(c);
 if (c EQ 1) if ((k=e()) EQ 4) return(nilptr); else pb=k;
 skp=newloc(nilptr,skp);
 A(skp)=j=k=newloc(nilptr,nilptr);

 /* we will return k, but we will fill node j first */
 if (c EQ 1)
    {scan:  A(j)=sread();
     next:  if ((c=e())≤2) {t=newloc(nilptr,nilptr); B(j)=t; j=t;
                             if (c≤0) {A(j)=c; goto next;}
                             pb=c; goto scan;
                            }
     if (c≠4) {B(j)=sread(); if (e()≠4) error("syntax error");}
     skp=B(skp); return(k);
    }
 if (c EQ 2)
    {A(j)=quoteptr; B(j)=t=newloc(nilptr,nilptr); A(t)=sread();
     skp=B(skp); return(k);
    }
 error("bad syntax");
}

/* ================================================== */
int32 e(void)
/* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

E is a lexical token scanning routine which scans the chars in the input stream to extract the next token and returns

1 if the token is '('
2 if the token is '''
3 if the token is '.'
4 if the token is ')'

or a negative typed-pointer to an entry in the atom table or the the number table.
−− */

```
{double v,f,k,sign;
 int32 i,t,c;
```

```
 char nc[15], *np;
 struct Insave *tb;

#define OPENP '('
#define CLOSEP ')'
#define BLANK ' '
#define SINGLEQ ''
#define DOT '.'
#define PLUS '+'
#define MINUS '-'
#define CHVAL(c) (c-'0')
#define DIGIT(c) ('0'≤(c) AND (c)≤'9')
#define TOUPPER(c) ((c) + 'A'-'a')
#define ISLOWER(c) ((c)≥'a' AND (c)≤'z')

 if (pb≠0) {t=pb; pb=0; return(t);}

start:while ((c=getgchar()) EQ BLANK);  /* remove blanks */

 if (c EQ OPENP)
    {while (lookgchar() EQ BLANK) getgchar(); /* remove blanks */
     if (lookgchar() EQ CLOSEP) {getgchar(); return(nilptr);} else return(1);
    }
 if (c EQ EOS)
    {if (topInsave EQ NULL) {fclose(logfilep); exit(0);}
     /* restore the previous input stream */
     fclose(filep);
     strcpy(g,topInsave→g); pg=topInsave→pg; pge=topInsave→pge;
     filep=topInsave→filep; topInsave=topInsave→link;
     if (prompt EQ '@') prompt='>';
     goto start;
    }
 if (c EQ SINGLEQ) return(2);
 if (c EQ CLOSEP) return(4);
 if (c EQ DOT)
    {if (DIGIT(lookgchar())) {sign=1.0; v=0.0; goto fraction;} return(3);}
 if (NOT (DIGIT(c) OR ((c EQ PLUS OR c EQ MINUS) AND
     (DIGIT(lookgchar()) OR lookgchar() EQ DOT))))
    {np=nc; *np++=c;     /* put c in nc[0] */
     for (c=lookgchar();
          c≠BLANK AND c≠DOT AND c≠OPENP AND c≠CLOSEP;
          c=lookgchar())
        *(np++)=getgchar(); /* add a character */
     *np=EOS; /* nc is now a string */
     if (*nc EQ '@')
```

```
        {/* switch input streams */
         /* save the current input stream */
         tb=(struct Insave *)calloc(1,sizeof(struct Insave));
         tb→link=topInsave; topInsave=tb;
         strcpy(tb→g,g); tb→pg=pg; tb→pge=pge; tb→filep=filep;

         /* set up the new input stream */
         *g=EOS; pg=pge=g; prompt='@';
         filep=fopen(nc+1,"r"); /* skip over the @ */
         goto start;
        }
     /* convert the string nc to upper case */
     for (np=nc; *np≠EOS; np++)
        if (ISLOWER((int16)*np)) *np=(char)TOUPPER((int16)*np);
     return(ordatom(nc));
    }
 if (c EQ MINUS) {v=0.0; sign=-1.0;} else {v=CHVAL(c); sign=1.0;}
 while (DIGIT(lookgchar())) v=10.0*v+CHVAL(getgchar());
 if (lookgchar() EQ DOT)
    {getgchar();
     if (DIGIT(lookgchar()))
        {fraction:
         k=1.0; f=0.0;
         do {k=10.*k;f=10.*f+CHVAL(getgchar());} while (DIGIT(lookgchar()));
         v=v+f/k;
        }
    }
 return(numatom(sign*v));
}

/* ========================================================== */
char getgchar(void)
/* ----------------------------------------------------------
   Get a character from g.
   ---------------------------------------------------------- */
{fillg(); return(*pg++);}

/* ========================================================== */
char lookgchar(void)
/* ----------------------------------------------------------
   Look at the next character in g, but do not advance.
   ---------------------------------------------------------- */
{fillg(); return(*pg);}

/* ========================================================== */
```

```
void fillg(void)
/* ------------------------------------------------------------
   Read a line into g[ ]. A line starting with a "/" is a comment line to be discarded.
   ------------------------------------------------------------ */
{while (pg≥pge)
   {sprompt:  if (filep EQ stdin) {sprintf(sout,"%c",prompt); ourprint(sout);}
    if (fgetline(g,200,filep)<0) return;
    if (filep EQ stdin) {fprintf(logfilep,"%s\n",g); fflush(logfilep);}
    if (*g EQ '/') goto sprompt;
    pg=g; pge=g+strlen(g); *pge++=' '; *pge='\0'; prompt='>';
   }
}

/* ============================================================ */
int16 fgetline(char *s, int16 lim, FILE *stream)
/* char s[~];
 * int16 lim;
 * FILE *stream;
 */
/* ------------------------------------------------------------
   fgetline() gets a line (LFCR or just LF delimited) from stream and puts it into s (up to lim
   chars). The function returns the length of this string. If there are no characters but just EOF,
   it returns -1 (EOF) as the length. There is no deblanking except to drop the LF, and if present,
   the CR ('\n').
   ------------------------------------------------------------ */
{int16 c,i;
#define TAB 9
 for (i=0; i<lim AND (c=fgetc(stream))≠EOF AND c≠'\n'; ++i)
    {if (c EQ TAB) c=BLANK; s[i]=c;}
 s[i]='\0';
 if (c EQ EOF AND i EQ 0) return(-1); else return(i);
}

/* ============================================================ */
int32 numatom(double r)
/* double r; */
/* ------------------------------------------------------------
   The number r is looked-up in the number table and stored there as a lazy number atom if it is
   not already present. The typed-pointer to this number atom is returned.
   ------------------------------------------------------------ */
{int32 j;
#define hashnum(r) ((*(1+(int32 *)(&r)) & 0x7fffffff) % n)

 j=hashnum(r);
 while (nx[j]≠-1)
```

```
    if (Ntab[nx[j]].num EQ r) {j=nx[j]; goto ret;} else if (++j EQ n) j=0;

 if (nf<0) {gc(); if (nf<0) error("The number table is full");}
 nx[j]=nf; j=nf; nf=Ntab[nf].nlink; Ntab[j].num=r;
ret:  return(nu(j));
}

/* ========================================================== */
int32 ordatom (char *s)
/* char *s; */
/* ----------------------------------------------------------
   The ordinary atom whose name is given as the argument string s is looked-up in the atom
   table and stored there as an atom with the value undefined if it is not already present. The
   typed-pointer to this ordinary atom is then returned.
   ---------------------------------------------------------- */
{int32 j,c;
#define hashname(s) (labs((s[0]≪16)+(s[(j=strlen(s))-1]≪8)+j) % n)

 j=hashname(s); c=0;
 while (Atab[j].name[0]≠EOS)
    {if (strcmp(Atab[j].name,s) EQ 0) goto ret;
     else if (++j EQ n) {j=0; if (++c>1) error("atom table is full");}
    }

 strcpy(Atab[j].name,s); Atab[j].L=ud(j);
ret:  return(oa(j));
}

/* ========================================================== */
void swrite(int32 j)
/* int32 j; */
/* ----------------------------------------------------------
   The S-expression pointed to by the typed-pointer j is typed-out.
   ---------------------------------------------------------- */
{int32 i;
 int16 listsw;

 i=ptrv(j);
 switch (type(j))
    {case 0:  /* check for a list */
        j=i;
        while (type(B(j)) EQ 0) j=B(j);
        listsw=(B(j) EQ nilptr);
        ourprint("(");
        while (listsw)
```

```
          {swrite(A(i)); if ((i=B(i)) EQ nilptr) goto close; else ourprint(" ");}
        swrite(A(i)); ourprint(" . "); swrite(B(i));
close:    ourprint(")");
        break;

     case  8:  ourprint(Atab[i].name); break;
     case  9:  sprintf(sout,"%-g",Ntab[i].num); ourprint(sout); break;
     case 10:  sprintf(sout,"{builtin function: %s}",Atab[i].name);
               ourprint(sout); break;
     case 11:  sprintf(sout,"{builtin special form: %s}",Atab[i].name);
               ourprint(sout); break;
     case 12:  sprintf(sout,"{user defined function: %s}",Atab[i].name);
               ourprint(sout); break;
     case 13:  sprintf(sout,"{user defined special form: %s}",Atab[i].name);
               ourprint(sout); break;
     case 14:  ourprint("{unnamed function}"); break;
     case 15:  ourprint("{unnamed special form}"); break;
    }
}

/* ========================================================== */
void traceprint(int32 v, int16 osw)
/* int32 v; the object to be printed
 * int16 osw; 1 for seval() output, 0 for seval() input
 */
/* ----------------------------------------------------------
   This function prints out the input and the result for each successive invocation of seval() when
   tracing is requested.
   ---------------------------------------------------------- */
{if (tracesw>0)
    {if (osw EQ 1) sprintf(sout,"%d result:",ct--);
     else sprintf(sout,"%d seval:",++ct);
     ourprint(sout); swrite(v); ourprint("\n");
    }
}

/* ========================================================== */
int32 seval(int32 p)
/* int32 p; */
/* ----------------------------------------------------------
   Evaluate the S-expression pointed to by the typed-pointer p; construct the result value as
   necessary; return a typed-pointer to the result.
   ---------------------------------------------------------- */
{int32 ty,t,v,j,f,fa,na;
 int32 *endeaL;
```

```
 static double s;

#define U1 A(p)
#define U2 A(B(p))
#define E1 A(p)
#define E2 A(B(p))
#define Return(v) {traceprint(v,1); return(v);}

 traceprint(p,0);

 if(type(p)≠0)
   {
  /* p does not point to a non-atomic S-expression.

     If p is a type-8 typed pointer to an ordinary atom whose value is a builtin or user-defined
     function or special form, then a typed-pointer to that atom-table entry with typecode 10, 11,
     12, or 13, depending upon the value of the atom, is returned. Note that this permits us to
     know the names of functions and special forms.

     if p is a type-8 typed pointer to an ordinary atom whose value is not a builtin or user defined
     function or special form, and thus has the type- code 8, 9, 14, or 15, then a typed-pointer
     corresponding to the value of this atom is returned.

     if p is a non-type-8 typed-pointer to a number atom or to a function or special form (named
     or unnamed), then the same pointer p is returned. */
    if ((t=type(p))≠8) Return(p); j=ptrv(p);

   /* The association list is implemented with shallow binding in the atom-
    table, so the current values of all atoms are found in the atom table. */

    if (Atab[j].name[0] EQ '!')
       {tracesw=(strcmp(Atab[j].name,"!TRACE") EQ 0)?1:0; longjmp(env,-1);}

    if ((t=type(Atab[j].L)) EQ 1)
       {sprintf(sout,"%s is undefined\n",Atab[j].name); error(sout);}

    if (namedfsf(t)) Return(tp(t≪28,j));
    Return(Atab[j].L);
  } /* end of if (type(p)≠0) */

 /* Save the list p consisting of the current function and the supplied arguments
   as the top value of the currentin list to protect it from garbage
   collection. The currentin list is a list of lists. */

 cilp=newloc(p,cilp);
```

```
/* compute the function or special form to be applied */
tracesw-- ; f=seval(A(p)); tracesw++; ty=type(f);
if (NOT fctform(ty)) error(" invalid function or special form");
f=ptrv(f); if (NOT unnamedfsf(ty)) f=ptrv(Atab[f].L);

/* now let go of the supplied input function */
A(cilp)=p=B(p);

/* If f is a function (not a special form), build a new list of its evaluated
  arguments and then let go of the list of supplied arguments, replacing it
  with the new list of evaluated arguments */
if (fct(ty))
   {/* compute the actual arguments */
    eaLp=newloc(nilptr,eaLp);
    /* evaluate the actual arguments and build a list by tail-cons-ing! */
    endeaL=&A(eaLp);
    while (p≠nilptr)
       {*endeaL=newloc(seval(A(p)),nilptr); endeaL=&B(*endeaL); p=B(p);}
    /* Set p to be the first node in the evaluated arguments list. */
    p=A(eaLp);

    /* Throw away the current supplied arguments list by popping the curentin
    list */
    cilp=B(cilp);
   }

/* At this point p points to the first node of the actual argument list.  if
  p EQ nilptr, we have a function or special form with no arguments */
if (NOT builtin(ty))
   {/* f is a non-builtin function or non-builtin special form.  do shallow
    binding of the arguments and evaluate the body of f by calling seval */
    fa=A(f); /* fa points to the first node of the formal argument list */
    na=0;    /* na counts the number of arguments */
    /* run through the arguments and place them as the top values of the
    formal argument atoms in the atom-table.  Push the old value of each
    formal argument on its binding list. */
    if (type(fa) EQ 8)
       {t=ptrv(fa); Atab[t].bl=newloc(Atab[t].L,Atab[t].bl); Atab[t].L=p;}
    else
       while (p≠nilptr AND dottedpair(type(fa)))
          {t=ptrv(A(fa)); fa=B(fa);
           Atab[t].bl=newloc(Atab[t].L,Atab[t].bl);
           v=A(p); if (namedfsf(type(v))) v=Atab[ptrv(v)].L;
           Atab[t].L=v; ++na; p=B(p);
          }
```

```
    if (p≠nilptr) error("too many actuals");
    if (fa≠nilptr) error("too many formals");

    /* now apply the non-builtin special form or function */
    v=seval(B(f));

    /* now unbind the actual arguments */
    fa=A(f);
    if (type(fa) EQ 8)
       {t=ptrv(fa); Atab[t].L=A(Atab[t].bl); Atab[t].bl=B(Atab[t].bl);}
    else
       while (na-->0)
          {t=ptrv(A(fa)); fa=B(fa);
           Atab[t].L=A(Atab[t].bl); Atab[t].bl=B(Atab[t].bl);
          }
   } /* end non-builtins */
else
   {/* at this point we have a builtin function or special form.  f is the
    pointer value of the atom in the atom table for the called function
    or special form */

    v=nilptr;
    switch (f) /* begin builtins */
    {case 1:  /* CAR */
        if (NOT dottedpair(type(E1))) error("illegal CAR argument");
        v=A(E1); break;
     case 2:  /* CDR */
        if (NOT dottedpair(type(E1))) error("illegal CDR argument");
        v=B(E1); break;
     case 3:  /* CONS */
        if (sexp(type(E1)) AND sexp(type(E2))) v=newloc(E1,E2);
        else error("Illegal CONS arguments");
        break;

        /* for LAMBDA and SPECIAL, we could check that U1 is either an
       ordinary atom or a list of ordinary atoms */
     case 4:/* LAMBDA */ v=tf(newloc(U1,U2)); break;
     case 5:/* SPECIAL */ v=ts(newloc(U1,U2)); break;
     case 6:/* SETQ */
        f=U1; if (type(f)≠8) error("illegal assignment");
        assign:  v=ptrv(f); endeaL=&AL(v);
        doit:  t=seval(U2);
        switch (type(t))
           {case 0:  /* dotted pair */
```

```
        case 8:  /* ordinary atom */
        case 9:  /* number atom */
           *endeaL=t; break;
        case 10:  /* builtin function */
        case 11:  /* builtin special form */
        case 12:  /* user-defined function */
        case 13:  /* user-defined special form */
           *endeaL=Atab[ptrv(t)].L; break;
        case 14:  /* unnamed function */
           *endeaL=uf(ptrv(t)); break;
        case 15:  /* unamed special form */
           *endeaL=us(ptrv(t)); break;
       } /* end of type(t) switch cases */

   tracesw--; v=seval(f); tracesw++; break;

case 7:  /* ATOM */
   if ((type(E1)) EQ 8 OR (type(E1)) EQ 9) v=tptr; break;

case 8:  /* NUMBERP */
   if (type(E1) EQ 9) v=tptr; break;

case 9:  /* QUOTE */ v=U1; break;
case 10:  /* LIST */ v=p; break;
case 11:  /* DO */ while (p≠nilptr) {v=A(p); p=B(p);} break;

case 12:  /* COND */
   while (p≠nilptr)
      {t=A(p);
       if (seval(A(t))≠nilptr) {v=seval(A(B(t))); break;} else p=B(p);
      }
   break;

case 13:  /* PLUS */
   v=numatom(Ntab[ptrv(E1)].num+Ntab[ptrv(E2)].num); break;

case 14:  /* TIMES */
   v=numatom(Ntab[ptrv(E1)].num*Ntab[ptrv(E2)].num); break;

case 15:  /* DIFFERENCE */
   v=numatom(Ntab[ptrv(E1)].num-Ntab[ptrv(E2)].num); break;

case 16:  /* QUOTIENT */
   v=numatom(Ntab[ptrv(E1)].num/Ntab[ptrv(E2)].num); break;
```

```
case 17:  /* POWER */
   v=numatom(pow(Ntab[ptrv(E1)].num,Ntab[ptrv(E2)].num));
   break;

case 18:  /* FLOOR */ v=numatom(floor(Ntab[ptrv(E1)].num)); break;
case 19:  /* MINUS */ v=numatom(-Ntab[ptrv(E1)].num); break;
case 20:  /* LESSP */
   if(Ntab[ptrv(E1)].num<Ntab[ptrv(E2)].num) v=tptr; break;

case 21:  /* GREATERP */
   if (Ntab[ptrv(E1)].num>Ntab[ptrv(E2)].num); v=tptr; break;

case 22:  /* EVAL */ v=seval(E1); break;
case 23:  /* EQ */ v=(E1 EQ E2)?tptr:nilptr; break;

case 24:  /* AND */
   while (p≠nilptr AND seval(A(p))≠nilptr) p=B(p);
   if (p EQ nilptr) v=tptr;  /* else v remains nilptr */
   break;

case 25:  /* OR */
   while (p≠nilptr AND seval(A(p)) EQ nilptr) p=B(p);
   if (p≠nilptr) v=tptr;  /* else v remains nilptr */
   break;

case 26:  /* SUM */
   for (s=0.0; p≠nilptr; s=s+Ntab[ptrv(A(p))].num, p=B(p));
   v=numatom(s); break;

case 27:  /* PRODUCT */
   for (s=1.0; p≠nilptr; s=s*Ntab[ptrv(A(p))].num, p=B(p));
   v=numatom(s); break;

case 28:  /* PUTPLIST */ v=E1; Atab[ptrv(v)].plist=E2; break;
case 29:  /* GETPLIST */ v=Atab[ptrv(E1)].plist; break;
case 30:  /* READ */ ourprint("\n!"); prompt=EOS; v=sread(); break;
case 31:  /* PRINT */
   if (p EQ nilptr) ourprint(" ");
   else while (p≠nilptr) {swrite(A(p)); ourprint(" "); p=B(p);}
   break;

case 32:  /* PRINTCR */
   if (p EQ nilptr) ourprint("\n");
   else while (p≠nilptr) {swrite(A(p)); ourprint("\n"); p=B(p);}
   break;
```

```
      case 33:  /* MKATOM */
         strcpy(sout,Atab[ptrv(E1)].name); strcat(sout,Atab[ptrv(E2)].name);
         v=ordatom(sout); break;

      case 34:  /* BODY */
         if (unnamedfsf(type(E1))) v=ptrv(E1);
         else if (userdefd(type(E1))) v=ptrv(Atab[ptrv(E1)].L);
         else error("illegal BODY argument");
         break;

      case 35:  /* RPLACA */
         v=E1;
         if (NOT dottedpair(type(v))) error("illegal RPLACA argument");
         A(v)=E2; break;

      case 36:  /* RPLACD */
         v=E1;
         if (NOT dottedpair(type(v))) error("illegal RPLACD argument");
         B(v)=E2; break;

      case 37:  /* TSETQ */
         if (Abl(f=ptrv(U1)) EQ nilptr) goto assign;
         v=Abl(f); while (B(v)≠nilptr) v=B(v);
         endeaL=&A(v); goto doit;

      case 38:  /* NULL */
         if (E1 EQ nilptr) v=tptr; break;

      case 39:    /* SET */
         f=seval(U1); goto assign;

      default:  error("dryrot: bad builtin case number");
     } /* end of switch cases */

    } /* end builtins */

 /* pop the eaL list or pop the currentin list, whichever is active */
 if (fct(ty)) eaLp=B(eaLp); else cilp=B(cilp);

 Return(v);
}

/* ============================================================ */
int32 newloc(int32 x, int32 y)
```

```
/* int32 x,y; */
/* ------------------------------------------------------------
   Allocates and loads the fields of a new location in the list area. The car field is set to X and the
   cdr field is set to Y. The index of the new location is returned.
   ------------------------------------------------------------ */
{int32 j;
 if (fp<0) {gc(); if (fp<0) error("out of space");}
 j=fp; fp=B(j); A(j)=x; B(j)=y; return(j);
}

/* ============================================================ */
void gc(void)
/* ------------------------------------------------------------
   Garbage collector for the number table and the listarea.
   ------------------------------------------------------------ */
{int32 i,t;
#define marked(p)    ((A(p) & 0x08000000)≠0)
#define unmark(p)    (A(p) &= 0xf7ffffff)

 for (i=0; i<n; i++)
    {gcmark(Atab[i].L); gcmark(Atab[i].bl); gcmark(Atab[i].plist);}
 for (i=0; i<n; i++) nx[i]=-1;

 for (nf=-1,i=0; i<n; i++)
    if (nmark[i] EQ 0) {Ntab[i].nlink=nf; nf=i;}
    else  /* restore num[i] */
       {t=hashnum(Ntab[i].num);
        while (nx[t]≠-1) if ((++t) EQ n) t=0;
        nx[t]=i; nmark[i]=0;
       }

 /* build the new list-node free-space list */
 fp=-1;
 for (i=1; i≤m; i++) if (NOT marked(i)) {B(i)=fp; fp=i;} else unmark(i);
}

/* ============================================================ */
void gcmark(int32 p)
/* int32 p; */
/* ------------------------------------------------------------
   Mark the object, possibly a sub-tree, specified by p.
   ------------------------------------------------------------ */
{int32 t;
#define marknode(p)  (A(p) |= 0x08000000)
#define marknum(t,p)   if ((t) EQ 9) nmark[ptrv(p)]=1
```

```
#define listp(t)       ((t) EQ 0 OR (t)>11)

start:
 t=type(p);
 if (listp(t))
    {p=ptrv(p); if (marked(p)) return; marknode(p);
     t=A(p); if (NOT listp(type(t))) {marknum(type(t),t); p=B(p); goto start;}
     t=B(p); if (NOT listp(type(t))) {marknum(type(t),t); p=A(p); goto start;}
     gcmark(A(p));        /* recursive call */
     p=B(p); goto start; /* equivalent to gcmark(B(p)) */
    }
 else marknum(t,p);
}

/* end of file lisp.c */
```

Exercise 29.1: How does one 'exit' from the Lisp intepreter given above?

Exercise 29.2: The LISP interpreter program given above can crash in many circumstances where illegal input is supplied, because such illegal input is not checked for. Fix this. Try to maintain whatever readability is currently present, since readability is a virtue equally as important as efficiency.

Exercise 29.3: Rerogram the LISP interpreter given here to solve the problem that strings are not automatically allocated or automatically garbage-collected in C by building a collection of string-handling toolkit procedures. Hint: arrange for all strings to live in a special garbage-collected area established especially for strings.

Exercise 29.4: Rerogram the LISP interpreter given here to solve the problem that the list-area, the atom table, and the number table have fixed sizes. Devise methods for allocating and using more space for these data-structures when they become full rather than failing as the program above does now.

Of course, for modern paging systems, we might be able to specify that these areas be very large, but only use part of the space for such a data area initially, secure in the knowledge that pages that are never accessed, never exist until they are accessed. This still leaves the problem of rehashing for the atom table.

One can imagine that a suitable cooperative paging system might make it possible to request that certain data areas be started on a new page in the logical address space and occupy a block of contiguous pages at the end of the allocated memory, and that such a data area is accessed only through a pointer, and that we can ask for such aress to be extended in size by some integral number of pages, and that all the address variables for these extensible areas be appropriately reset.

In this excercise, however, you should not assume any help, and only use the basic C malloc routines.

30 Bibliography

References

[Abr89] Bruce Abramson. Control strategies for two-player games. *ACM Computing Surveys*, 21(2):137–161, June 1989.

[ACR87] John R. Anderson, Albert T. Corbett, and Brian J. Reiser. *Essential Lisp*. Addison Wesley, Reading, Mass., 1987.

[All78] John Allen. *Anatomy of LISP*. McGraw-Hill, NY, 1978.

[BB66] E. C. Berkeley and Daniel G. Bobrow, editors. *The Programming Language LISP: Its Operation and Applications*. The MIT Press, Cambridge, Mass., 1966.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[Fod79] John Foderaro. *The Franz LISP Manual*. Univ. of Calif., Berkeley, Calif., 1979.

[Fri74] Daniel P. Friedman. *The Little LISPer*. SRA Inc., Chicago, IL, 1974.

[Fri86] Daniel P. Friedman. *The Little LISPer*. SRA Inc., Chicago, IL, second edition, 1986.

[Hof85] Douglas R. Hofstadter. *Metamagical Themas*. Basic Books, NY, 1985.

[Jr84] Guy L. Steele Jr. *Common Lisp*. Digital Press, DEC, Billerica, Mass., 1984.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand Co. Inc., Princeton, NJ, 1952.

[Knu68] Donald Ervin Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.

[Kur81] Toshiaki Kurokawa. A new fast and safe marking algorithm. *Software Practice and Experience*, 11:671–682, 1981.

[Mag79] Byte Magazine, August 1979. Issue devoted to LISP.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *CACM*, 3(4):184–195, April 1960.

[McC61] John McCarthy. A basis for a mathematical theory of computation. *Proc. WJCC*, 19(225-238), 1961.

[McC78] John McCarthy. History of lisp. *ACM SIGPLAN Notices*, 13(8), Aug. 1978.

[Mee79] J.R. Meehan. *The New UCI LISP Manual*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1979.

[MIT62] MIT Press, Cambridge, Mass. *LISP 1.5 Programmer's Manual*, 1962.

[Moo74] David Moon. *MACLISP Reference Manual, Version 0*. Laboratory for Computer Science, MIT, Cambridge, Mass., April 1974.

[MSW83] David Moon, Richard Stallman, and Daniel Weinreb. *LISP Machine Manual*. MIT Artificial Intellegence Laboratory, Cambridge, Mass., fifth edition edition, 1983.

[NM44] John Von Neumann and Oskar Morgenstern. *The Theory of Games and Economic Behavior*. Princeton Univ. Press, 1980, 1944.

[PT87] Andrew R. Pleszkun and Matthew J. Thazhuthaveetil. The architecture of lisp machines. *IEEE Computer*, March 1987.

[SA83] G. Sussman and H. Abelson. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1983.

[Sam79] Hanan Samet. Deep and shallow binding: the assignment operation. *Computer Languages*, 4:187–198, 1979.

[Sik76] Laurent Siklossy. *Let's Talk LISP*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Sta89] Richard W. Stark. *LISP, Lore. and Logic*. Springer-Verlag, NY, 1989.

[Tha86] Matthew J. Thazhuthaveetil. *A Structured Memory Access Architecture for LISP*. PhD thesis, Univ. of Wisconson, Aug. 1986. Ph.D. thesis, CS report 658.

[Tou84] David S. Touretzky. *LISP - A Gentle Introduction to Symbolic Computation*. Harper & Row, NY, 1984.

[Wan84] Mitchell Wand. What is lisp? *American Mathematical Monthly*, 91(1):32–42, Jan. 1984.

[Wei67] Clark Weissman. *LISP 1.5 Primer*. Dickenson Pub. Co., Belmont, Calif., 1967.

[WH89] Patrick Henry Winston and Berthold K.P. Horn. *LISP*. Addison-Wesley, Reading Mass., first edition 1981, second edition 1984, third edition 1989. edition, 1981, 84, 89.

[Whi79] Jon L. White. Macsyma symbolic manipulation program. In *Proc. of the 1979 MACSYMA Users' Conference*, Cambridge, Mass., 1979. MIT Laboratory for Computer Science.

[Wil84] Robert Wilenski. *LISPcraft*. W.W. Norton & Co., NY, 1984.

Index