LISP

Programming and Proving

John McCarthy and Carolyn Talcott

Stanford University

This version printed at 0:18 on September 15, 1980.

Table of Contents

INTRODUCTION

This text is about programming in LISP and proving facts about LISP programs.

LISP is a language for programming with symbolic expressions represented in the computer by list structures. It is the programming language most used in artificial intelligence research and is also the most used language for large systems for computing with mathematical formulas. Both require extensive computing with symbolic expressions, and in both it is often just as important to realize extremely complex algorithms as to make simple algorithms run as fast as possible. The data structures and language of LISP are well suited to this and so is the fact that LISP programs are represented by LISP data structures; this makes it easy to write programs that generate new parts of themselves. This is particularly beneficial in an interactive environment, where programs that are run only once per human interaction can transform what is convenient for the user to write into complex structures of basic operations.

As a programming language, LISP is characterized by the following ideas:

•. Computing with symbolic expressions rather than numbers.

•. Representation of symbolic expressions and other information by list structure in computer memory.

•. Representation of information on paper, from keyboards and in other external media mostly by multi-level lists and sometimes by S-expressions. It has been important that any kind of data can be represented by a single general type.

•. A small set of selector and constructor operations expressed as functions, i.e. *car*, *cdr* and *cons*.

•. Composition of functions and conditional expressions as a major way of building programs from already available functions.

•. The recursive use of conditional expressions as a sufficient tool for building computable functions.

•. The use of λ-expressions for naming functions.

•. The storage of information on the property lists of atoms.

•. Representation of LISP programs as LISP data that can be manipulated by object programs. This has prevented the separation between system programmers and application programmers. Everyone can "improve" his LISP, and many of these "improvements" have developed into improvements to the language.

•. The LISP function *eval* that serves both as a formal definition of the language and as an interpreter.

•. Garbage collection as the means of memory management.

•. No requirements for declarations so that LISP statements can be executed in an on-line environment without preliminaries.

Pure LISP is the core of the language. Its programs are function expressions which are built up by composition and conditional expression recursion from basic functions. These function expressions are used for their values only and not for side effects. This makes them easy to understand and makes LISP recursive function definitions directly usable as sentences of first order mathematical logic. For this reason, proving properties of LISP functions is very straightforward.

We believe that a course in programming should include techniques for proving that the programs meet their specifications. The programmers should check their correctness proofs by computer as a replacement for conventional debugging.

This text goes part of the way to meeting this objective. It is one of the first to integrate program proving with programming itself. Students can be asked to write a program with given properties and prove that the program has these properties. The ability to prove the correctness of a program is often a better criterion for the program being "well structured" than the presence or absence of specific programming constructs. Having to prove that a program meets specifications often induces rewriting it in a more transparent style.

The major applied objective of research in mathematical theory of computation should be to make it routine to supply computer checked proofs of computer program correctness as part of the process of debugging the program. To put it strongly, money shouldn't change hands till this has been done. Many pure LISP programs have been proved correct in a course in mathematical theory of computations using Richard Weyhrauch's FOL proof-checker for first order logic, though we have not yet used it in the programming course except as an option in the term project. The experience of proving facts about programs has the same advantages as the study of any other mathematical domain. Even if the student never proves another theorem, his understanding of the subject is enhanced by the experience.

While the mathematical theory of pure LISP is sufficiently well developed to support computer-checked proofs of correctness, practical programming also involves constructs whose theories are still in a primitive state. These constructs are described in practical programming terms with some indication of how their theory is developing.

The student of LISP should finish the course with four skills. First he must understand how to represent symbolic information in list structures and to read and write LISP programs. Second he should understand how to state and prove properties of programs. Third he should be able to write programs that themselves produce programs or fragments thereof. Finally, he should understand the techniques of one or more major applications of LISP.

The text begins with a chapter on LISP data structures and on understanding pure LISP programs.

Chapter II tells how to write pure LISP programs and is organized around the different kinds of recursion appropriate for different tasks and data structures.

Chapter III shows how to represent pure LISP programs as sentences of logic and use these sentences to prove properties of the programs.

Chapter IV applies the methods of III to more difficult problems of proving correctness.

Chapter V extends pure LISP with sequential computations, arrays, macros, property lists, and operations that modify list structure.

Chapter VI describes how LISP data structures and programs are represented in a computer and how LISP interacts with a user at a terminal.

Chapter VII applies LISP to searching graphs and game trees.

Chapter VIII treats compiling pure LISP programs as a LISP programming problem. Two compilers, one short and one generating fairly efficient code, are given as examples.

Chapter IX discusses the notion of abstract syntax and uses it to prove the correctness of a compiler for a very simple language fragment.

Chapter X discusses some theoretical topics such as computability and lambda calculus in a framework made easy by the previous material.

Not all of the above material can be covered in a one quarter course. Additional chapters on pattern directed computing and on proof methods for impure LISP are will be added.

The applied material is independent of the theoretical chapters, but everything depends on the first two chapters.

Chapter I

INTRODUCTION TO LISP

LISP is a language for writing programs that do symbolic computation. Information is coded or represented as S-expressions. A LISP program can also be represented as an S-expression. This gives LISP the special ability to easily manipulate programs as well as other sorts of symbolic data. In this chapter we describe notation for lists and S-expressions, show how they are represented in a computer as list structures, and describe the basic functions and predicates on the domain of S-expressions in terms of the computer representation. We then describe the basic constructs of LISP and show how they are used to form LISP programs.

1. Lists.

We begin with some examples. The most common form of S-expression is the list. Figure 1 shows some examples of lists. The list (i) has four elements, one of which is repeated. The list (ii) has four elements one of which is itself a list. The list (iii) has one element. The list (iv) also has one element which itself is a list. The list (v) has no elements; it is also written NIL.

```
(i)     (A B A E)
(ii)    (A B (C D) E)
(iii)   (A)
(iv)    ((A B C D))
(v)     ()
```

Figure 1. Examples of lists.

Figure 2 shows how symbolic information can be represented by lists. Each example consists of a list and a "mathematical" representation of the same information. Examples (i)–(iv) represent familiar mathematical and logical expressions. The list (v) is used to represent the network or graph shown according to a scheme whereby there is a sublist for each vertex consisting of the vertex itself followed by the vertices to which it is connected.

The elements of a list are surrounded by parentheses and separated by spaces. A list may have any number of terms and any of these terms may themselves be lists. In this case, the spaces surrounding a sublist may be omitted, and extra spaces between elements of a list are allowed. Thus `(A B(C  D)  E)` and `(A B (C D) E)` are slightly different ways of writing the same list.

(i) (PLUS X Y)
$x + y$

(ii) (PLUS (TIMES X Y) X 3)
$xy + x + 3.$

(iii) (EXIST X (ALL Y (IMPLIES (P X) (P Y))))
$\exists x: \forall y: [P(x) \supset P(y)]$

(iv) (INTEGRAL 0 ∞ (TIMES (EXP (TIMES I X Y)) (F X)) X)
$\int_0^\infty e^{ixy} f(x)dx.$

(v) ((A B) (B A C D) (C B D E) (D B C E) (E C D F) (F E))

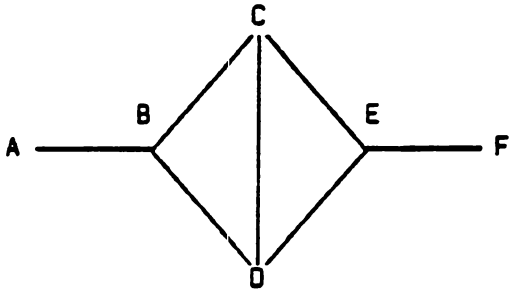


Figure 2. Information represented as lists.

2. Atoms.

The expressions A, B, X, Y, 3, PLUS, and ALL occurring in the lists in Figures 1 and 2 are called atoms. In general, an atom is written as a sequence of capital letters, digits, and special characters with certain exclusions. The exclusions are <space>, <carriage return>, and the other non-printing characters, and also the parentheses, brackets, semi-colon, and comma. Numbers are expressed as signed decimal or octal numbers, the exact convention depending on the implementation. Floating point numbers are written with decimal points and, when appropriate, an exponent notation depending on the implementation. The reader should consult the programmer's manual for the LISP implementation he intends to use. Some further examples of atoms are shown in Figure 3. From such atoms we can form lists like ((345 3.14159 -47) A387B THE-LAST-TRUMP -45.21).

```
THE-LAST-TRUMP
A387B
345
-47
-45.21
3.14159
```

Figure 3. Examples of atoms.

3. List structures.

Atoms and lists are represented in the memory of the computer by list structures. A list structure is a collection of memory units called *cons-cells*. A cons-cell generally consists of one or possibly two consecutive computer words. It is divided into two parts, the a-part and the d-part, with each part being capable of containing an address in memory. The list structure representing a list contains a cons-cell for each element of the list. The a-part of the cell contains the address of the list structure representing the element and the d-part contains the address of the cell for the next element of the list. These addresses are sometimes called pointers as they "point" to the component list structures.

A diagram shows this more clearly than words. Figure 4 shows the list structure corresponding to the list (PLUS (TIMES X Y) X 3) (which might represent the expression $xy + x + 3$).

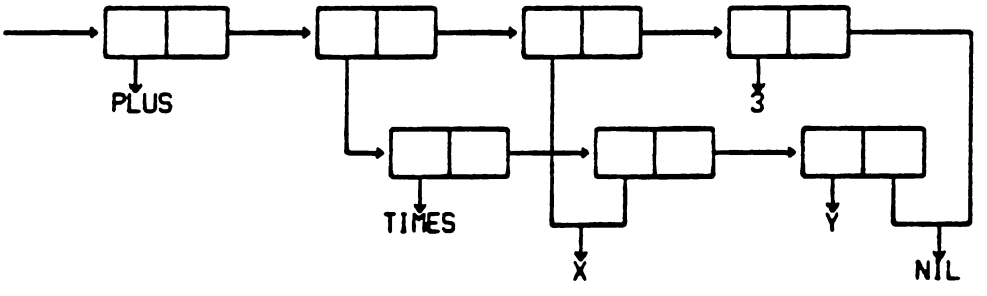


Figure 4. Representation of (PLUS (TIMES X Y) X 3) as a list structure.

A program refers to a list by the address of the cell for its first element. According to this convention, we see that the a-part of this cell is the first element of the list and the d-part is a pointer to a sublist formed by deleting the first element. Thus the first word of the list structure of Figure 4 contains a pointer to the list structure representing the atom PLUS, while its d-part points to the list ((TIMES X Y) X 3). The second word contains the list structure representing (TIMES X Y) in its a-part and the list structure representing (X 3) in its d-part. The last word

points to the atom 3 in its a-part and has a pointer to the atom NIL in its d-part. This is consistent with the convention that NIL represents the null list.

Atoms are represented by the addresses of their property lists which are list structures of a special kind depending on the implementation. (In some implementations, the first word of a property list is in a special area of memory, in others the first word is distinguished by sign, in still others it has a special a-part. For basic LISP programming, it is enough to know that atoms are distinguishable from other list structures by a predicate called at.) Each atom is represented uniquely, .e.g there is only one pointer that prints as A. In the diagram, we don't give the structure of property lists but just label them with their *print names.*

A partial dump of the memory of a computer containing the list structure of Figure 4 might look as shown in Figure 5. Here X denotes the address of the property list of the atom named X and /3 denotes that of the atom named 3. Notice that the addresses of consecutive list elements need not be consecutive. Also the last word of a list cannot have the address of a next word in its d-part since there isn't any next word, so it has the address of a special atom called NIL.

address	a-part	d-part	address	a-part	d-part
86	TIMES	87	1000	PLUS	1001
87	X	88	1001	86	1002
88	Y	NIL	1002	X	2653
			2653	/3	NIL

Figure 5. A memory map for the list structure of Figure 4..

4. S-expressions.

When we examine the way list structures represent lists we see a curious asymmetry. Namely, the a-part of a list word can contain an atom or a list, but the d-part can contain only a list or the special atom NIL. This restriction is quite unnatural from the computing point of view, and we shall allow arbitrary atoms to inhabit the d-parts of words, but then we must generalize the way list structures are expressed as character strings. To do this, we introduce the notion of S-expression.

An S-expression is either an atom or a pair of S-expressions. To write a non-atomic S-expression we write the pair of subexpression separated by " . " and surrounded by parentheses. In BNF this rule is given by:

<S-expression> ::= <atom> | (<S-expression> . <S-expression>).

Figure 6 shows some examples of S-expressions.

```
A
(A . B)
(A . (B . A))
(PLUS . (X . (Y . NIL)))
(3 . 3.4)
```

Figure 6. Examples of S-expressions.

In writing an S-expression, the spaces around the " . " may be omitted when this will not cause confusion. The only possible confusion is of the dot separator with a decimal point in numbers. Thus, in the above cases, we may write (A.B), (A.(B.A)), and (PLUS.(X.(Y.NIL))), but if we wrote (3.3.4) confusion would result.

In the memory of a computer, an S-expression is represented by the address of a cons-cell whose a-part points to the first element of the pair and whose d-part points to the second element of the pair. Thus, the S-expressions (A.B), (A.(B.A)), and (PLUS.(X.(Y.NIL))) are represented by the list structures of Figure 7.

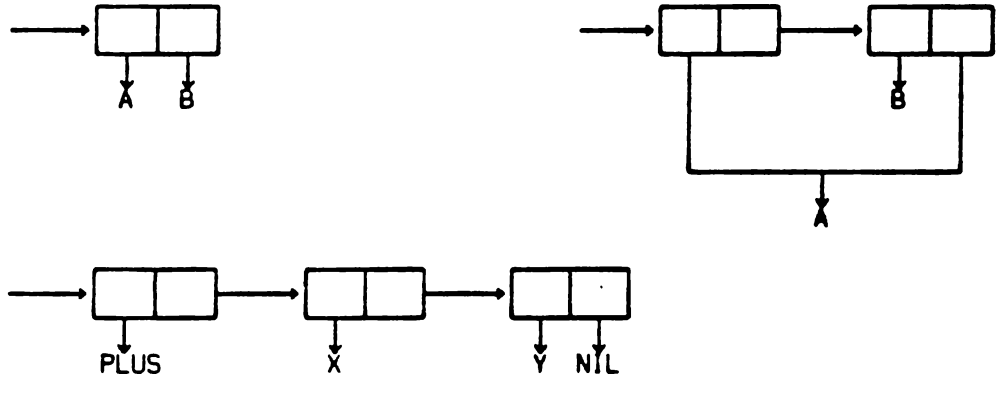


Figure 7. Representation of S-expressions as list structures.

Note that the list (PLUS X Y) and the S-expression (PLUS . (X . (Y . NIL))) are represented in memory by the same list structure. The simplest way to treat this is to regard S-expressions as primary and lists as special kinds of S-expressions, namely those that never have any atom but NIL as the second part of a pair. The list notation can then be considered as an abbreviated way of writing these S-expressions, Thus, the list notation (A B . . . Z) may be regarded as an abbreviation of the S-expression notation (A . (B . (C . (... (Z . NIL) ...)))). Note that the a-part of a list can be any S-expression, but the d-part of a list must be a list and and the only atomic list is NIL. In giving input to LISP, either the list form or the S-expression form may be used for lists. On output, LISP will print a list structure as a list as far as it can, otherwise as an S-expression. Thus, some list structures will be printed in a mixed notation, e.g. ((A . B) (C . D) (3)). Some LISPs use

an "extended" list notation for printing S-expressions which minimizes the number of parentheses and dots printed. In this notation (A . (B . (C . D))) would print as (A B C . D). (Programs for reading and printing S-expressions in the two notations are given in Chapter VI.)

Exercises

1. If we represent sums and products as indicated above and use (MINUS X), (QUOTIENT X Y), and (POWER X Y) as representations of $-x$, x/y, and x^y respectively, then

a. What do the following lists represent?

```
(QUOTIENT 2 (POWER (PLUS X (MINUS Y)) 3))
(PLUS -2 (MINUS 2) (TIMES X (POWER Y 3.3)))
```

b. How are the expressions $xyz+3(u+v)^{-3}$ and $(xy-yx)/(xy+yx)$ to be represented?

2. In the above mentioned graph notation, what graph is represented by the list

((A D E F) (B D E F) (C D E F) (D A B C) (E A B C) (F A B C))?

3. Write the list (PLUS (TIMES X Y) X 3) as an S-expression. This is sometimes referred to as "dot-notation".

4. Write the following S-expressions in list notation to whatever extent is possible:

a. (A . NIL)
b. (A . B)
c. ((A . NIL) . B)
d. ((A . B) . ((C . D) . NIL)).

5. How would you represent polynomials in one variable as lists? Can you think of more that one way? How would you represent polynomials in several variables? Can you tell if two lists represent the same polynomial?

(The next two mathematical exercises are unconnected with subsequent material in this book.)

6. Prove that the number of "shapes" of S-expressions with n atoms is $(2n-2)!/(n!(n-1)!)$. The two shapes of S-expressions with three atoms are (A.(B.C)) and ((A.B).C). These numbers are called Catalan numbers.

7. The above result can also be obtained by writing $S = A + S \times S$ as an "equation" satisfied by the set of S-expressions with the interpretation that an S-expression is either an atom or a pair of S-expressions. The next step is to regard the equation as the quadratic $S^2 - S + A = 0$, solve it by the quadratic formula choosing the minus sign for the square root. Then the radical is regarded as the 1/2 power and expanded by the binomial theorem. Manipulating the binomial coefficients yields the above formula as the coefficient of A^n in the expansion. Why does this somewhat ill-motivated and irregular procedure give the right answer?

5. The basic functions and predicates on S-expressions.

There are three basic functions on S-expressions: *cons* for constructing an S-expression from two given S-expressions, and *car* and *cdr* for selecting the first and second components of a non-atomic S-expression. There are two predicates, *atom* for testing whether or not an S-expression is atomic and *eq* for testing equality of atoms. In this section we will describe the basic LISP programs that compute these functions and predicates from list structure representations of the arguments. All LISP programs for computing functions and predicates on S-expressions are built from these basic programs as will be explained in the sections following this. This is analogous to numeric programs which are based on operations computing the arithmetic functions of addition subtraction, multiplication, and division, and the arithmetic predicates of equality and comparison.

First we introduce the notation to be used for writing LISP terms. This notation will be extended in later sections to express LISP programs as well. A LISP term is an expression intended to denote an S-expression (the value of the term). We will use two languages for writing LISP terms and programs - *internal language* and *external language*. Internal language represents LISP terms and programs as S-expressions. Internal language programs are somewhat harder for a person to read and write, but it is easier for a person to write a program to manipulate *object programs* when the object programs are in internal language (e.g. represented as S-expressions). External language is a notation that is compact and easier for people to read and write. However, external language programs are not S-expressions, and therefore it is not easy to write LISP programs to generate, interpret, compile or otherwise manipulate programs written in external language. (An additional advantage of the external notation is that it is very similar to the language used to express facts about the abstract domain of S-expressions given in Chapter III).

[Remark: most LISP implementations accept only internal language programs.]

The simplest LISP terms are variables and constants. In the external language the notation for S-expression constants is that described in the preceeding sections. (S-expression constants will always appear in the special font used in the previous sections.) In the internal language there is a possible ambiguity as to whether an S-expression is to represent itself, or the value of the LISP term that it represents (if any). LISP takes the latter point of view. An S-expression constant is represented by a list whose first element is the atom QOUTE and whose second element is the S-expression. Thus the S-expression constant (A B) is represented by the S-expression (QUOTE (A B)).

Variables are written in external notation as lower case italic identifiers. Frequently just single letters such as x and u are used, but sometimes we use names suggestive of the kind of object the variable represents. The corresponding internal form will be the atom whose name is the same identifier but in upper case (and appearing in the S-expression font). Thus X corresponds to x and U to u. We often use x, y, z to range over arbitrary S-expressions and u, v, w to range over lists.

Additional terms (called application terms) can be formed denoting the result of one of the basic LISP programs. These are expressed in external language using ordinary mathematical notation for the application of functions and predicates to a suitable number of arguments. Program names are used for function symbols and variables or S-expression constants as

arguments. Argument lists will be surrounded by "[]" thus reserving parentheses for S-expressions. The brackets will often be omitted for single arguments. The internal notation for application terms is more uniform. Namely such an application term is represented by a list, where the first element is the atom corresponding to the program name and the remaining elements of the list represent the arguments.

Table I. gives some examples of LISP terms. Each term is shown in external and internal notation along with the S-expression value that it denotes. Note that no variables appear in these examples because the value denoted by a LISP term containing a variable depends on the value assigned to the variable.

	External form	Internal form	Value denoted
(i)	(A . B)	(QUOTE (A . B))	(A . B)
	(CAR X)	(QUOTE (CAR X))	(CAR X)
(iia)	a(A . B)	(CAR (QUOTE (A . B)))	A
(iid)	d(A . B)	(CDR (QUOTE (A . B)))	B
(iic)	*cons*[A,B]	(CONS (QUOTE A) (QUOTE B))	(A . B)
(iiia)	a((A . B) . A)	(CAR (QUOTE ((A . B) . A))	(A . B)
(iiid)	d((A . B) . A)	(CDR (QUOTE ((A . B) . A))	A
(iiic)	*cons*[(A . B),A]	(CONS (QUOTE (A . B)) (QUOTE A))	((A . B) . A)
(iva)	a(A B C D)	(CAR (QUOTE (A B C D)))	A
(ivd)	d(A B C D)	(CDR (QUOTE (A B C D)))	(B C D)
(ivc)	*cons*[A,(B C D)]	(CONS (QUOTE A) (QUOTE (B C D)))	(A B C D)
(v)	at A	(ATOM (QUOTE A))	T
	at(A . B)	(ATOM (QUOTE (A . B))	NIL
(vi)	A eq A	(EQ (QUOTE A) (QUOTE A))	T
	A eq B	(EQ (QUOTE A) (QUOTE B))	NIL
	(A . B) eq (A . B)	(EQ (QUOTE (A . B)) (QUOTE (A . B)))	?

Table I. Examples of LISP terms in external and internal notation and their values.

We now describe the basic LISP programs and give their external and internal names. The external names are generally abbreviations for the internal names and will appear in bold face type.

We begin with the programs for the selector functions. *car* is represented externally by **a** and internally by **CAR**. When applied to a non-atomic list structure the result is the a-part. See Table I (iia) and (iiia) for examples. *cdr* is represented externally by **d** and internally by **CDR**. When applied to a non-atomic list structure the result is the d-part. See Table I (iid) and (iiid). The action of the selector operations on atoms is not defined in basic LISP. However, most implementations assign a some "meaning" to such terms, possibly an error message.

Since lists are a particular kind of S-expression, the action of the selector operations on non-empty lists is also determined by the above definitions. In particular a selects the first element of the list and d selects the rest of the list (e.g. the list formed by removing that first element). See Table I (iva) and (ivd). Recall that the a-part of a non-empty list can be any S-expression, while the d-part is always a list. Thus the selectors behave symmetrically on S-expressions but unsymmetrically on lists reflecting the symmetry properties of the list structure representation of S-expressions and lists.

[Historical note: the names "CAR" and "CDR" stood for "contents of the address part of register" and "contents of the decrement part of register" in a 1957 precursor of LISP projected for the IBM 704 computer. The names have persisted for lack of a clearly preferable alternative.]

cons is represented externally by cons or (more often) by an infixed " . " and internally by CONS. The program for *cons* must have access to a supply of cons-cells called the free storage list. Given pointers to two existing list structures, the program removes a cons-cell from the free storage list and puts the first pointer (address) into the a-part and the second into the d-part of this cell. The result is the S-expression represented by the pointer to the new cons-cell. See Table I (iic) and (iiic).

We see that for lists, *cons* is a prefixing operation. Namely, if the second argument is a list then the result of applying *cons* is the list obtained by putting the first argument onto the front of that list. See Table I (ivc).

[Note that giving the same pair of pointers to the *cons* program a second time will result in the construction of a list structure distinct from the first, although both represent the same S-expression.]

The predicate *atom* is represented externally by at and internally by ATOM. The program for *atom* when given a pointer to a list structure determines whether that list structure represents and atom or not. The result is T if the list structure is atomic and NIL otherwise. See Table I (v). We see that the truthvalues true and false are represented in LISP by the atoms T and NIL respectively.

The predicate *eq* is represented externally by the infix eq and internally by EQ. As mentioned earlier, it is only required to test for equality of atoms. The program for *eq* actually does a bit more. Namely, given pointers to two list structures it compares these addresses. If they are equal the result is T, otherwise it is NIL. Since each atom is represented by a unique list structure, the comparison is indeed a test of equality if at least one of the arguments is atomic. More generally if two list structures have the same address, they represent the same S-expression. The converse is false because there is no guarantee that the same S-expression is not represented by two different list structures. (Two applications of *cons* to a pair of list structures in most LISPs will result in two list structures representing the same S-expression, but with different addresses.) Equality of S-expressions is tested by a program based on *eq* which we will describe in a later section. If this program were used instead of *eq* as a basic function, LISP would be logically simpler, but many programs would be slower.

This concludes the description of the basic LISP programs. To conclude the section we give some further notational extensions, conventions and abbreviations.

Besides the basic functions of LISP, there will be user-defined functions. We haven't given

the mechanism for function definition yet, but suppose a function *subst* taking three arguments has been defined. It may be used in terms like *subst*[*x*, *y*, *z*] having internal form (SUBST X Y Z).

As in other programming languages and in mathematics generally, application terms can have arbitrary terms as arguments, not just variables and constants. Thus we have terms like [*subst*[*x*, *y*, a *z*] . *subst*[*x*, *y*, d *z*]] involving a user defined function *subst*. Its internal form is (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))).

n *x* is an abbreviation for *x* eq NIL. It rates a special notation because NIL plays the same role among lists that zero plays among numbers, and list variables often have to be tested to see if their value is NIL. Its internal form is (NULL X).

The notation *list*[*x*1, *x*2, . . . , *xn*] is used to denote the composition of *cons*'s that forms a list from its elements. Thus *list*[*x*, *y*, *z*] denotes *cons*[*x*, *cons*[*y*, *cons*[*z*, NIL]]] and forms a list of three elements out of the quantities *x*, *y*, and *z*. We often write <*x*, *y*, . . ., *z*> instead of *list*[*x*, *y*, . . . , *z*] when this will not cause confusion. The experienced implementer of programming languages will expect that since *list* has a variable number of arguments, its implementation will pose problems. That is indeed true. The internal form of <*x*, *y*, . . ., *z*> is (LIST X Y . . . Z).

Compositions of a and d are written by concatenating the letters a and d. Thus, it is easily seen that ad*u* denotes the second element of the list *u*, and add*u* denotes the third element of that list. The internal names of these functions are CADR and CADDR respectively.

In external language, we often omit brackets for functions of one argument. Thus *alt x* stands for *alt*[*x*], and *alt alt x* stands for *alt*[*alt*[*x*]]. This convention was also used in the descriptions of a, d, at, and n.

Exercise

1. What is the value of < A . a(B . C), D . d(B . C)>?

2. What combinations of a and d select (X Y) and (CONS X Y) respectively from

```
((LAMBDA (X Y) (CONS X Y)) (QUOTE A) (QUOTE B))
```

6. Conditional terms.

When the value of a function depends on whether a certain predicate is true of the arguments, conditional terms are used to describe the function. In external language a simple conditional term has the form

6.1) **if** *p* **then** *a* **else** *b*

where p is a propositional term and a and b can be any terms. By propositional term we mean a term whose value represents a truthvalue (true or false). A (program computing a) predicate applied to a list of arguments is a propositional term. We will explain how to form such terms in general in the next section.

The conditional term (6.1) is evaluated as follows: first p is evaluated and determined to be true or false. If p is true, then a is evaluated and its value is the value of the expression. If p is false, then b is evaluated and its value is the value of the expression. Note that if p is true, b is never evaluated, and if p is false, then a is never evaluated. The importance of this will be explained later.

A familiar function that can be written conveniently using conditional expressions is the absolute value of a real number. We have

6.2) $$|x| = \text{if } x<0 \text{ then } -x \text{ else } x.$$

[Remark: The common mathematical notation for such definitions is

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise,} \end{cases}$$

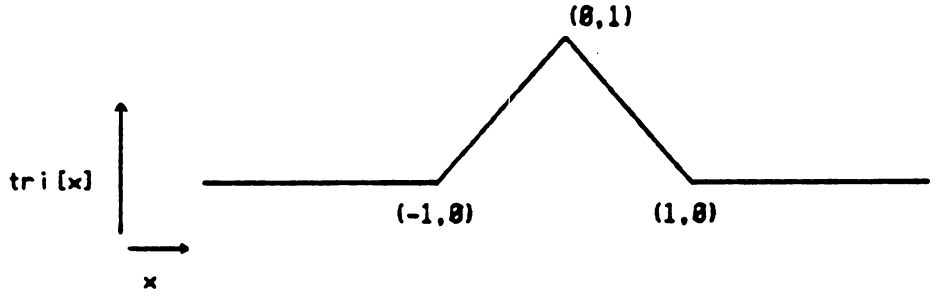
but the right hand side is not allowed to be part of another expression.]

More generally a conditional term has the form

6.3) $$\text{if } p \text{ then } a \text{ else if } q \text{ then } b \ldots \text{ else if } s \text{ then } d \text{ else } e.$$

There can be any number of clauses. The value is determined by evaluating the propositional terms p, q, etc. until a true term is found, the value is then that of the term following the next then. If none of the propositional terms is true, then the value is that of the term following the last else.

Figure 8 shows an example of a (numeric) function defined using a conditional expression.



tri[x] = if x<-1 then 0 else if x<0 then 1+x else if x<1 then 1-x else 0.

Figure 8. The triangle function.

The conditional term in internal language has the form

6.4) (COND (*p*1 *e*1) (*p*2 *e*2) ... (*pn en*))

with any number of *p*–*e* pairs. Its value is determined by evaluating the *p*'s successively until one is found with a value corresponding to true. The value of the corresponding *e* is then taken as the value of the conditional term. If none of the *p*'s is true, then the value of the conditional term is undefined. (In some implementations the conditional term is given a default value such as NIL if none of the *p*'s are true.)

[Remark: In the internal form, all the *e*'s are treated the same which makes programs for interpreting or compiling conditional expressions easier to write. This is another way in which expressions in internal language are more regular than in external language.]

Conditional expressions may be compounded with functions to get terms like

6.5) if n*u* then *v* else a*u* . *append*[d*u*, *v*]

involving a yet to be defined function *append*. The internal form of this is

6.6) (COND ((NULL U) V) (T (CONS (CAR U) (APPEND (CDR U) V)))).

One would normally have expected to write (QUOTE T), but there is a special convention that T may be written without QUOTE. This convention applies to NIL also. This means that these symbols cannot be used as variables. As mentioned in section § 5, T represents the propositional constant true, i.e. it is always true so that it is impossible to "fall off the end" of a conditional expression with T as its last *p*. It is the translation into internal form of the final else of a conditional expression in external form.

7. Propositional terms.

As mentioned in the previous section, propositional terms are a terms whose value represents a truth value. The simplest propositional terms are the constants T and NIL. If ϕ is a predicate of n-arguments then $\phi[x1, \ldots, xn]$ is a propositional term. Additional propositional terms can be formed by combining terms already formed using the logical operations of conjunction(and), disjunction(or) and negation(not). Both and and or are associative, so we can write expressions like $p1$ and $p2$ and $p3$ without worrying about the grouping.

The value of a propositional term can be determined using the truth table given in Table 2.

T and T = T	T or T = T	
T and NIL = NIL	T or NIL = T	not T = NIL
NIL and T = NIL	NIL or T = T	not NIL = T
NIL and NIL = NIL	NIL or NIL = NIL	

Table 2. Truth table for propositional terms.

The evaluation of propositional terms is defined using conditional terms as follows:

7.1) p and q = if p then q else NIL

7.2) p or q = if p then p else q

7.3) not p = if p then NIL else T

The internal forms of these definitions are

```
(AND P Q) = (COND (P Q) (T NIL))
 (OR P Q) = (COND (P P) (T Q))
  (NOT P) = (COND (P NIL) (T T))
```

Note that if p is false then p and q = NIL independent of the value of q, and likewise if p is true, then p or q = T independent of q. If p has been evaluated and found to be false, then q does not have to be evaluated at all to find the value of p and q, and, in fact, LISP does not evaluate q in this case. Similarly, q is not evaluated in evaluating p or q if p is true. This procedure is in accordance with the above conditional expression descriptions of these operators. The importance of this convention, which contrasts with that of ALGOL 60, will be apparent later when we discuss recursive definitions of functions and predicates.

Propositional expressions can be combined with functions and conditional expressions to get terms like

7.4) if [n u or n d u] then u else a u . *alt* dd u

whose internal form is

7.5) (COND ((OR (NULL U) (NULL (CDR U))) U) (T (CONS (CAR U) (ALT (CDDR U))))).

We conclude this section with two remarks. First, the only S-expressions that we have said represent truthvalues are T and NIL. However, most LISP implementations regard any non-NIL S-expression as representing true. These LISPs do not distinguish between between propositional terms and more general terms. Thus any term can appear in the p part of a conditional term. Second, most LISP implementations allow and and or to have arbitrary numbers of arguments. Since these operations are associative, this can be viewed as simply omitting parentheses as it won't matter how the arguments are grouped.

8. Recursive function definitions.

In such languages as FORTRAN and ALGOL, computer programs are expressed in the main as sequences of assignment statements and conditional go to's. In LISP, programs are mainly expressed in the form of recursively defined functions. We begin with an example.

We want a function *alt* such that *alt* u gives a list whose elements are alternate elements of the list u beginning with the first. For example we want

alt (A B C D E) = (A C E),
alt ((A B) (C D)) = ((A B)),
alt (A) = (A),
alt NIL = NIL.

and in LISP we can define *alt* by the recursion equation

8.1) *alt* $u \leftarrow$ if nu or nd$u$ then u else au . *alt* dd$u$.

In case you need a review of our precedence conventions, fully bracketed it looks like

alt[u] $\leftarrow$ [if [n[u] or n[d[u]]] then u else [a[u] . *alt*[dd[u]]]].

The terms appearing in the recursion equation are formed by the methods previously discussed. Notice that the function *alt* occurs in the right hand side of its own defining equation and that we use "$\leftarrow$" instead of "=". The recursion equation tells us that *alt* is a function of one variable u, and that the value of *alt* u for some particular list is computed by evaluating the right hand side of the definition, substituting that list for u whenever u is encountered and re-evaluating the right hand side with a new u whenever a term *alt* e is encountered. This process is best illustrated by evaluating some expressions.

Consider evaluating *alt* NIL. We do it by evaluating the expression to the right of the "$\leftarrow$" in (8.1) remembering that the variable u has the value NIL. A conditional expression is evaluated by first evaluating the first propositional term -- in this case nu or nd$u$. This expression is a disjunction so we first evaluate the first disjunct, namely nu. (Recall the convention given in

section § 7). Since u = NIL, nu is true, the disjunction is true, and the value of the conditional expression is the value of the term after the first true propositional term. This term is u, and its value is NIL, so the value of *alt*[NIL] is NIL. Obeying the rules about the order of evaluation of terms in conditional and propositional expressions is important in this case since if we didn't obey these rules, we might find ourselves trying to evaluate n du or *alt*[dd$u$], and since u is NIL, neither of these has a value. (An attempt to evaluate them would result in an error return in most LISPs.)

As a second example, consider *alt* (A B). Since neither nu nor n du is true in this case, the disjunction nu or n du is false and the value of the expression is the value of au . *alt* dd$u$. au has the value A, and dd$u$ has the value NIL, so we must now evaluate *alt* NIL, and we already know that this is NIL. Therefore, the value of *alt* (A B) is that of A . NIL which is (A).

This evaluation is described by the following sequence of equations:

alt (A B) = if [n (A B) or n d (A B)] then (A B) else a (A B) . *alt*[dd (A B)]
= if [false or n d (A B)] then (A B) else a (A B) . *alt*[dd (A B)]
= if false then (A B) else a (A B) . *alt*[dd (A B)]
= a (A B) . *alt*[dd (A B)]
= A . *alt*[NIL]
= A . if [n NIL or n d NIL] then NIL else a NIL . *alt*[dd NIL]
= A . if [true or n d NIL] then NIL else a NIL . *alt*[dd NIL]
= A . if true then NIL else a NIL . *alt*[dd NIL]
= A . NIL
= (A).

This is still very long-winded, and now that the reader understands the order of evaluation of conditional and propositional expressions, we can proceed more briefly to evaluate

alt[(A B C D E)] = A . *alt*[(C D E)]
= A . [C . *alt*[(E)]]
= A . [(C E)]
= (A C E).

Here are three more examples of recursive functions and their application. We define *last* by

8.2) *last* u ← if n du then au else *last* du,

and we compute

last (A B C) = if n d (A B C) then a (A B C) else *last* d (A B C)
= *last* (B C)
= *last* (C)
= C.

Clearly *last* computes the last element of a list. *last* NIL is undefined in the LISP language; the result of trying to compute it may be an error message or may be some random result depending on the implementation. (A heavy duty version of *last* would explicitly call a function called *error* with a string expressing the complaint that the program had tried to compute *last* NIL.)

The function *subst* is defined by

8.3) $subst[x, y, z] \leftarrow$ if atom z then [if z eq y then x else z] else $subst[x, y, a z] \,.\, subst[x, y, d z]$.

We have

$$\begin{aligned} subst[(A.B), X, ((X.A).X)] &= subst[(A.B), X, (X.A)] \,.\, subst[(A.B), X, X] \\ &= [subst[(A.B), X, X] \,.\, subst[(A.B), X, A]] \,.\, (A.B) \\ &= [[(A.B)].A].(A.B)] \\ &= (((A.B).A).(A.B)). \end{aligned}$$

subst computes the result of substituting the S-expression x for the atom y in the S-expression z. This operation is important in all kinds of symbolic computation.

The function $append[u, v]$ which gives the concatenation of the lists u and v is also important. It is also denoted by the infixed expression $u * v$. For example we have

$$(A\ B\ C) * (D\ E\ F) = (A\ B\ C\ D\ E\ F),$$

$$NIL * (A\ B) = (A\ B) = (A\ B) * NIL,$$

and the formal definition

8.4) $$u * v \leftarrow \text{if } n u \text{ then } v \text{ else } a u \,.\, [d u] * v.$$

The *append* function is machine coded in most Lisp systems in a way that allows an arbitrary number of arguments, e.g. (APPEND (QUOTE (A B)) (QUOTE (C D)) (QUOTE (E F))) is (A B C D E F). It is convenient to use an infix for *append* because it is associative, i.e. $u * [v * w] = [u * v] * w$ so we can just write $u * v * w$. We often use infix notation for recursively defined functions when it is mathematically conventional or convenient.

The propositional operations can also be used in making recursive definitions since we take advantage of the order of evaluation of constituents. Thus, we define a predicate *equal* by

8.5) $equal[x, y] \leftarrow x$ eq y or [not at x and not at y and $equal[a x, a y]$ and $equal[d x, d y]$].

$equal[x, y]$ is true if and only if x and y are the same S-expression, and the use of this predicate makes up for the fact that the basic predicate eq is guaranteed to test equality only when one of the operands is known to be an atom. We shall also use the infixes $=$ and $\neq$.

Membership of an S-expression x in a list u is tested by

8.6) $$member[x, u] \leftarrow \text{not } n u \text{ and } [[x = a u] \text{ or } member[x, d u]].$$

This relation is also denoted by $x \in u$. Here are some computations:

member[B, (A B)] = not n(A B) and [[B = a(A B)] or *member*[B, d(A B)]]
= *member*[B, (B)]
= true.

Sometimes a function is defined with the help of auxiliary functions. Thus, the reverse of a list u, (e.g. *reverse*[(A B C D)] = (D C B A)), is given by the pair of equations

8.7)
$$reverse[u] \leftarrow rev[u, NIL]$$
$$rev[u, v] \leftarrow \text{if } n\,u \text{ then } v \text{ else } rev[d\,u, [a\,u].v]$$

A computation is:

reverse[(A B C)] = *rev*[(A B C), NIL]
= *rev*[(B C), (A)]
= *rev*[(C), (B A)]
= *rev*[NIL, (C B A)]
= (C B A).

A more elaborate example of recursive definition is given by

8.8)
$$flatten[x] \leftarrow flat[x, NIL]$$
$$flat[x, u] \leftarrow \text{if } at\,x \text{ then } x.u \text{ else } flat[a\,x, flat[d\,x, u]]$$

We have

flatten[((A.B).C)] = *flat*[((A.B).C), NIL]
= *flat*[(A.B), *flat*[C, NIL]]
= *flat*[(A.B), (C)]
= *flat*[A, *flat*[B, (C)]]
= *flat*[A, (B C)]
= (A B C).

The reader will see that the value of *flatten*[x] is a list of the atoms of the S-expression x from left to right. Thus *flatten*[((A B) A)] = (A B NIL A NIL).

Many functions can be conveniently written in more than one way. For example, the function *reverse* mentioned above can be defined without an auxiliary function as follows:

8.9) $$reverse1\ u \leftarrow \text{if } n\,u \text{ then NIL else } reverse1\ d\,u * \langle a\,u \rangle,$$

but the earlier definition involves less computation, because * takes time proportional to the length of its first argument. Similarly the function computed by *flatten* can also be computed by the simpler but less efficient program *fringe*

8.10) $$fringe\ x \leftarrow \text{if } at\,x \text{ then } \langle x \rangle \text{ else } fringe\ a\,x * fringe\ d\,x,$$

The use of conditional expressions for recursive function definition is not limited to functions of S-expressions. For example, the factorial function and the Euclidean algorithm for the greatest common divisor on the non-negative integers may be written as follows:

8.11) $$n! \leftarrow \text{if } n=0 \text{ then } 1 \text{ else } n\cdot[n-1]!$$

and

8.12) $$gcd[m,\ n] \leftarrow \text{if } m>n \text{ then } gcd[n,\ m] \text{ else if } m=0 \text{ then } n \text{ else } gcd[n \bmod m,\ m]$$

where $n \bmod m$ denotes the remainder when n is divided by m and may itself be expressed recursively by

8.13) $$n \bmod m \leftarrow \text{if } n<m \text{ then } n \text{ else } [n-m] \bmod m.$$

(Note that these definitions must be modified if negative integers are to be included in the domain.)

The internal form of function definitions depends on the implementation. MACLISP uses the form

(DEFUN <function name> <list of variables> <right hand side>).

Stanford LISP and UCI LISP for the PDP-10 computer use the same form with DE in place of DEFUN. Thus in MACLISP the definition of *subst* is

```
(DEFUN SUBST (X Y Z)
  (COND ((ATOM Z) (COND ((EQ Z Y) X) (T Z)))
        (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))))
```

and the definition of *alt* is

```
(DEFUN ALT (U)
  (COND ((OR (NULL U) (NULL (CDR U))) U)
        (T (CONS (CAR U) (ALT (CDDR U))))))
```

Yet another notation for function definition called the DEFPROP notation will be explained after λ-expressions have been introduced.

Exercises

1. Consider the function *drop* defined by

$$drop[u] \leftarrow \text{if } nu \text{ then NIL else } \langle au \rangle \ .\ drop[du]$$

Compute (by hand) *drop*[(A B C)]. What does *drop* do to lists in general? Write *drop* in internal notation using DEFUN.

2. What does the function

$$r2[u] \leftarrow \text{if } n\,u \text{ then NIL else } reverse[a\,u]\ .\ r2[d\,u]$$

do to lists of lists? How about

$$r3[x] \leftarrow \text{if } at\,x \text{ then } x \text{ else } reverse[r4[x]]$$

$$r4[u] \leftarrow \text{if } n\,u \text{ then NIL else } r3[a\,u]\ .\ r4[d\,u]?$$

3. Compare

$$r3'[x] \leftarrow \text{if } at\,x \text{ then } x \text{ else } \langle r3'[d\,x]\rangle * \langle r3'[a\,x]\rangle$$

with the function $r3$ of the preceding example.

4. Consider $r5$ defined by

$$r5[u] \leftarrow \text{if } n\,u \text{ or } n\,d\,u \text{ then } u \text{ else } [a\,r5[d\,u]]\ .\ r5[a\,u\ .\ r5[d\,r5[d\,u]]].$$

Compute $r5$[(A B C D)]. What does $r5$ do? Needless to say, this is not a good way of computing this function even though it involves no auxiliary functions. [This ingenious program was discovered by S. Ness]

9. Numerical computation.

Numerical calculation and symbolic calculation must often be combined, so LISP provides for numerical computation also. (Numbers could be represented as lists of digits, but then it would be difficult to take proper advantage of machine instructions that work with numbers directly). Since we need to include numbers as parts of symbolic expressions, LISP has both integer and floating point numbers which are regarded as atoms that may be included as atoms in writing S-expressions. Thus we have the lists:

```
(1 3 5)
(3.5 6.1 -7.2E9)
(PLUS X 1.3).
```

The first is a list of integers, the second a list of floating point numbers, and the third a symbolic list containing both numerical and non-numerical atoms. Integers are written without decimal points which are used to signal floating point numbers. As in FORTRAN, the letter E is used to signal the exponent of a floating point number which is a signed integer. The sizes of numbers admitted depends on the implementation. When a dotted pair, say (1 . 2) is wanted, the spaces around the dot distinguish it from the list (1.2) whose sole element is the floating point number 1.2.

In external language we will use ordinary mathematical notation for numerical functions. As an example of a function combining numeric and symbolic calculation we have the function giving the length of a list defined by

9.1) *length u* ← **if** n*u* **then** 0 **else** 1 + *length* d*u*.

The internal notation for numerical functions is shown in the following examples:

(PLUS X Y ... Z) for $x+y+\ldots+z$,
(TIMES X ... Z) for $xy\ldots z$,
(MINUS X) for $-x$,
(DIFFERENCE X Y) for $x-y$,
(QUOTIENT X Y) for x/y,
(POWER X Y) for x^y.

Besides functions of numbers we need predicates on numbers and the usual =, <, >, ≤, and ≥ are used with the internal names EQUAL, LESSP, GREATERP, LESSEQP, and GREATEREQP, respectively. Not all are implemented in all LISP systems, but of course the remaining ones can be defined. An additional predicate, *numberp*, is used to distinguish numbers from other atoms.

Since numbers that form part of list structures must be represented by pointers anyway, there is room for a flag distinguishing floating point numbers and integers. Therefore, the arithmetic operations are programmed to treat types dynamically, i.e. a variable may take an integer value at one step of computation and a real value at another. The subroutines realizing the arithmetic functions make the appropriate tests and create results of appropriate types.

It is worth remarking that including type flags in numbers would benefit many programming languages besides LISP and would not cost much in either storage or hardware.

Dynamic typing of variables is slow compared to direct use of the machine's arithmetic instructions, so that LISP can be efficiently used interpretively only when the numerical calculations are small or at least small compared to the symbolic calculations in a problem. The MACLISP compiler NCOMPLR is able to generate efficient arithmetic code. In particular, variables can be declared to be of a fixed numerical type and the correct machine instructions are then generated so that runtime testing is not necessary. Also, it uses separate stacks to store numerical results so that unecessary conversion from the LISP representation of a number to the machine representation can be avoided. This saves both time and space as each conversion from a machine number to a LISP number requires a *cons* operation.

LISP can also deal with integers too large to be represented as a single machine word. Such numbers are called "bignums" and are repsented in LISP by a pointer to a list structure which contains the sign, a flag saying that this a "bignum", and a list of the numbers corresponding to a base B representation of the number for some suitable B (depending on the machine and implementation).

As another example of a combined numeric and symbolic computation, we give an evaluator for expressions with sums and products. The expressions are built up from atoms denoting variables and integer constants according to the syntax

9.2)
<expression> ::= <variable> | <integer> | (PLUS <explist>) | (TIMES <explist>)
<explist> ::= <expression> | <expression><explist>

Thus a list of expressions beginning with the atom PLUS represents the sum of these expressions and a list of expressions beginning with the atom TIMES represents the product of these expressions. In order to evaluate an expression we need a way of giving values to the variables occuring in the expression. We will use an *association list* for this purpose. An association list (a-list) is a list of pairs, in our case the first element of each pair is a variable and the second element is the value associated with it. For example ((X . 5) (Y . 9.3) (Z . 2.1)) is an a-list. To look up the value of a variable in an a-list we use the function *assoc*, which returns the first pair in the list such that the variable matches the variable argument. It returns NIL if no value is found. Thus

$$assoc[Y, ((X . 5) (Y . 9.3) (Z . 2.1))] = (Y . 9.3)$$

and the definition of *assoc* is given by

9.3) $$assoc[x, a] \leftarrow \text{if } n a \text{ then NIL else if } a a a \text{ eq } x \text{ then } a a \text{ else } assoc[x, d a].$$

Association lists are generally useful for associating "values" with "symbols" and they will appear in many later examples. We note here two features which are due to the way *assoc* is defined. Since *assoc* returns the pair rather than the value when the desired symbol is found or NIL if no value is found, the case of no value found can be detected. If just the value were returned there would be no distinction between no value and a value of NIL. The calling program must check to see if a value was returned (not $n assoc[x, a]$) and then extract that value ($d assoc[x, a]$). Also, *assoc* returns the first pair it finds, thus a symbol can be associated with several values in the a-list, but always the first occurence determines the result that is returned. Thus

$$assoc[X, ((U . 1.41) (X . 5) (Y . 9.3) (X . 1.2) (Z . 2.1))] = (X . 5).$$

assoc is built into most LISP systems.

The interpreter, *numval*, can now be defined.

9.4)
$$\begin{aligned} numval[e, a] \leftarrow\ & \text{if } numberp\ e \text{ then } e \\ & \text{else if } a t e \text{ then } d assoc[e, a] \\ & \text{else if } a e \text{ eq PLUS then } evplus[d e, a] \\ & \text{else if } a e \text{ eq TIMES then } evtimes[d e, a] \end{aligned}$$

where

9.5) $$evplus[u, a] \leftarrow \text{if } n u \text{ then } 0 \text{ else } numval[a u, a] + evplus[d u, a],$$

9.6) $$evtimes[u, a] \leftarrow \text{if } n u \text{ then } 1 \text{ else } numval[a u, a] \cdot evtimes[d u, a],$$

10. Bitwise Boolean Operations

Besides the usual arithmetic operations on numbers, LISP allows bitwise boolean operations

on words in memory. Letting m, and n be strings of 1s and 0s of some fixed length depending on the machine word size, then m bool-op n denotes the word whose binary repsentation is given by performing the boolean operation on each pair of corresponding bits. Thus we have

```
4 bool-and 6 = 4
 4 bool-or 6 = 6
4 bool-xor 6 = 2
```

Here bool-xor is the exclusive or operation. In general we will use terminology obtained by prefixing bool- to the usual name of the corresponding logical operation for a boolean operation.

The internal form for boolean operations on words has not been standardized, but the form used by MACLISP is

(BOOLE <code> <n1> <n2>)

where <code> ranges from 0 to 15 (decimal) and each value of <code> corresponds to a different boolean operation. We have the following simple rule for determining what operation corresponds to a particular value of <code>. If the binary representation of <code> is "abcd" then result of the operation on a pair of bits x and y is given in table 3.

x	y	(BOOLE "abcd" x y)
0	0	a
0	1	c
1	0	b
1	1	d

Table 3. Boolean operation encoding.

In particular if <code> is 0 we have the identically 0 operation, if <code> is 15 we have the identically 1 operation, if <code> is 1 we have bool-and, and if <code> is 7 we have bool-or. Note that all 16 boolean functions of two variables are realized by the LISP BOOLE operation.

In addition to the above operations LISP has a shift operation $lsh[n, d]$ which shifts the bits of the binary representation of n, d positions to the left, filling the vacated spaces with zeroes. If d is negative the shifting is to the right instead. The internal form of this operation is (LSH <n> <d>).

Boolean operations are particularly useful when you have a problem involving data that can easily be represented by bit strings or vectors. Using the Boolean operations is very much more efficient than representing the same data as lists of 0s and 1s, both in terms of space required and in terms of computation time. It does not necessarily make the problem or the programs easier to understand, however. We will see an example of a problem solved using both list and bit represtations in Chapter VII.

Exercises

1. What boolean operation does BOOLE perform when <code>=11 (decimal)?

2. What <code> corresponds to the operation x=y ?

11. Lambda expressions and functions with functions as arguments.

It is common to use phrases like "the function $2x+y$". This is not a precise notation because we cannot say $[2x+y][3, 4]$ and know whether the desired result is $2\cdot3+4$ or $2\cdot4+3$ regarding the expression as a function of two variables. Worse yet, we might have meant a one-variable function of x wherein y is regarded as a parameter.

Church's λ-notation provides a convenient way of giving names to functions. In the above example, we would write $\lambda x\ y\colon 2x+y$ to denote the function of two variables with first argument x and second argument y whose value is given by the expression $2x+y$. Thus, $[\lambda x\ y\colon 2x+y][3, 4] = 10$ and $[\lambda y\ x\colon 2x+y][3, 4] = 11$.

Like variables of integration and the bound variables of quantifiers in logic, variables following the λ are bound or dummy and may be replaced by any others provided the replacement is done consistently throughout the expression and does not make any variable bound by λ the same as a free variable in the expression. Thus $\lambda x\ y\colon 2x+y$ represents the same function as $\lambda y\ x\colon 2y+x$ or $\lambda u\ v\colon 2u+v$, but in the function of one argument $\lambda x\colon 2x+y$, we cannot replace the variable x by y, though we could replace it by u.

λ-notation plays two important roles in LISP. First, it allows us to rewrite an expression containing two or more occurrences of the same sub-expression in such a way that the expression occurs only once. Thus $(2x+1)^4+3(2x+1)^3$ can be written $[\lambda w\colon w^4+3w^3][2x+1]$. This can save considerable computation, and corresponds to the practice in sequential programming of assigning to a variable the value of a sub-expression that occurs more than once in an expression and then writing the expression in terms of the variable. It is sometimes called λ-binding.

The second use of λ-expressions is in using functions that take functions as arguments. Suppose we want to form a new list from an old one by applying a function f to each element of the list. This can be done using the function *mapcar* defined by

11.1) $$mapcar[f,\ u] \leftarrow \text{if } n\,u \text{ then NIL else } f[a\,u]\ .\ mapcar[f,\ d\,u]$$

Such a function is called a *functional* since one (or more) of its arguments is a function. *mapcar* maps a function and a list into a list.

Suppose the operation we want to perform is squaring, and we want to apply it to the list (1 2 3 4 5 6 7). We have

$$mapcar[\lambda x\colon x^2,\ (1\ 2\ 3\ 4\ 5\ 6\ 7)] = (1\ 4\ 9\ 16\ 25\ 36\ 49).$$

[Some implementations of LISP allow mapping functions to take an arbitrary number of

lists as arguments. The number of lists is the number of arguments expected by the functional argument and the mapping terminates when the shortest list is exhausted. Some implementations of LISP require the arguments in reverse order - functional argument second.]

A more general operation than *mapcar* is *maplist* in which the function is applied to the successive sublists of the list rather than to the elements. *maplist* is defined by

11.2) $$maplist[f, u] \leftarrow \text{if } nu \text{ then NIL else } f[u] \,.\, maplist[f, du]$$

As an application of *maplist* and *mapcar* with functional arguments, we shall define a function for differentiating algebraic expressions involving sums and products. The syntax for such expressions was given in (9.2). Recall, PLUS followed by a list of arguments denotes the sum of these arguments and TIMES followed by a list of arguments denotes their product. The function *diff*[*e*, *v*] gives the partial derivative of the expression *e* with respect to the variable *v*. We have

```
                diff[e, v] ←
                    if at e then [if e = v then 1 else 0]
                    else if ae = PLUS then PLUS . mapcar[[λx: diff[x, v]], de]
                    else if ae = TIMES then
11.3)                 PLUS
                      . maplist[
                        [λx: TIMES
                              . maplist[
                                  [λy: if x = y then diff[ay, v] else ay], de]],
                        de]
```

The term that describes the rule for differentiating products corresponds to the rule

$$\partial/\partial v[\Pi_i\, e_i] = \Sigma_i \Pi_j\, [\text{if } i=j \text{ then } \partial e_j/\partial v \text{ else } e_j] \,.$$

and *maplist* has to be used rather than *mapcar* since whether to differentiate in forming the product is determined by equality of the indices i and j rather than equality of the terms e_i and e_j.

The internal form for a λ-expression is

(LAMBDA <list of variables> <expression to be evaluated>).

Thus λx: *diff*[*x*, *v*] is written (LAMBDA (X) (DIFF X V)). The internal form of of *diff* is given below. Notice that the function arguments to *maplist* and *mapcar* have the form

(FUNCTION <λ-expression>).

This is necessary if the definition is to be compiled as the compiler must recognize the fact that the following code must be compiled as a function. If the definition is only to be interpreted then FUNCTION has the same effect as QUOTE and in fact you may use QUOTE instead of FUNCTION in definitions that will only be interpreted by MACLISP.

```
(DEFUN DIFF (E V)
  (COND ((ATOM E) (COND ((EQ E V) 1) (T 0)))
        ((EQ (CAR E) 'PLUS)
         (CONS 'PLUS
               (MAPCAR (FUNCTION (LAMBDA (X) (DIFF X V))) (CDR E))))
        ((EQ (CAR E) 'TIMES)
         (CONS 'PLUS
               (MAPLIST (FUNCTION
                          (LAMBDA (X)
                            (CONS 'TIMES
                                  (MAPLIST (FUNCTION
                                             (LAMBDA (Y)
                                               (COND ((EQ X Y) (DIFF (CAR Y) V))
                                                     (T (CAR Y)))))
                                           (CDR E)))))
                        (CDR E))))))
```

The above paragraphing (known as "pretty printing") makes function definitions easier to read because items beginning in the same column are at the same parenthetical level.

Two additional useful functions with functions as arguments are the predicates *andlis* and *orlis* defined by the equations

11.4) $$andlis[p,\ u] \leftarrow n\,u \text{ or } [p[a\,u] \text{ and } andlis[p,\ d\,u]]$$

11.5) $$orlis[p,\ u] \leftarrow \text{not } n\,u \text{ and } [p[a\,u] \text{ or } orlis[p,\ d\,u]].$$

Another way of writing function definitions in internal notation uses LAMBDA to make a function of the right side of a definition. It is like writing *subst* and *alt* as

$$subst \leftarrow \lambda x\ y\ z.[\text{if } at\,z \text{ then } [\text{if } z \text{ eq } y \text{ then } x \text{ else } z] \text{ else } subst[x, y, a\,z]\ .\ subst[x, y, d\,z]]$$

$$alt \leftarrow \lambda u.[\text{if } n\,u \text{ or } n\,d\,u \text{ then } u \text{ else } a\,u\ .\ alt\ dd\,u].$$

Internally these definitions of *subst* and *alt* take the forms

```
(DEFPROP SUBST
  (LAMBDA (X Y Z)
    (COND ((ATOM Z) (COND ((EQ Z X) Y) (T Z)))
          (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))))
 EXPR)


(DEFPROP ALT
  (LAMBDA (U) (COND ((OR (NULL U) (NULL (CDR U))) U)
                    (T (CONS (CAR U) (ALT (CDDR U)))))))
 EXPR).
```

The general form for this manner of writing functon definitions is

(DEFPROP <function name> <defining λ-expression> EXPR)

and it is often used by programs that output LISP. It is a special case of an operation on property lists. It puts the λ-expression on the EXPR property of the function name (which is an atom). EXPR says that the item is a LISP function defined by an S-expression. (Rather than by a machine language subroutine, for instance). The DEFUN form of function definition has the same effect in that it forms a λ-expression out of the list of variables and the right hand side and puts it on the EXPR property of the function name.

Exercises

1. Compute *diff*[(TIMES X (PLUS Y 1) 3), X] using the above definition of *diff*. Now do you see why algebraic simplification is important?

2. Compute *orlis*[at, ((A B) (C D) E)]

3. Evaluate the S-expression

```
((LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X))) (QUOTE (LAMBDA (X) (LIST X (LIST
(QUOTE QUOTE) X)))))
```

and say what its interesting property is. Variants of this expression have important theoretical properties.

12. Label.

The λ mechanism is not adequate for providing names for recursive functions, because in this case there has to be a way of referring to the function name within the function. Therefore, we use the notation *label*[*f*, *e*] to denote the expression *e* but where occurrences of *f* within *e* refer to the whole expression. For example, suppose we wished to define a function that takes alternate elements of each element of a list and makes a list of these. Thus, we want

altlis[((A B C) (A B C D) (X Y Z))] = ((A C) (A C) (X Z)).

We can make the definition

12.1) *altlis*[*u*] ← *mapcar*[*label*[*alt*, λ*u*: **if** n*u* **or** n d*u* **then** *u* **else** a*u* . *alt*[dd*u*]], *u*]

in internal form this would be written

```
(DEFUN ALTLIS (X)
  (MAPCAR (QUOTE (LABEL ALT
                        (LAMBDA (X)
                          (COND (OR (NULL X) (NULL (CDR X))) X)
                                (T (CONS (CAR X) (ALT (CDDR X))))))))
          X)).
```

The general internal form of the label construct is

(LABEL <name> <function expression>),

The identifier *alt* in the above example is bound by *label* and is local to that expression, and this is the general rule. The label construct is not often used in LISP since it is more usual to give functions global definitions.

13. The function *eval*.

eval plays both a theoretical and a practical role in LISP. Historically, the list notation for LISP functions and *eval* were first devised in order to show how easy it is to define a universal function in LISP - the idea was to advocate LISP as an alternative to Turing machines for doing the elementary theory of computability. This role will be discussed in a later chapter. S. R. Russell noted that *eval* could serve as an interpreter for LISP and promptly programmed it in machine language with minor modifications to make it more practical. An interpreter based on *eval* has remained a feature of most LISP systems. Thus when you talking to LISP the system is in a loop that *read*s what you type, *eval*s it and *print*s the result. [Of course a real LISP system does many other things too, such a storage management, error handling, etc.]

eval for LISP expressions is analogous to the interpreter *numval* for arithmetic expressions given in (9.4). The first argument to *eval* is a LISP expression in internal notation. The second argument is an association list that tells *eval* what value each variable has, and what function definition is to be associated with each function name. Thus the association list is a list of pairs where each pair consists either of a variable and the S-expression corresponding to its value, or a function name and the S-expression representing the function expression defining the function. (Here a function expression is either a function name, or a lambda expression.) The result of applying *eval* is the value of the term represented by the S-expression in an environment where the free variables are assigned the values given by the association list and where the function names occuring free (i.e. not bound in a label expression) denote functions defined by the associated expressions in the association list.

Since any computation can be described as evaluating an expression without free variables or function names, the second argument theoretically plays a role mainly in the recursive definition of *eval*. Thus we usually start our computations with the second argument NIL.

To illustrate this, suppose we want to apply the function *alt* to the list (A B C D E), i.e. we wish to evaluate *alt*[(A B C D E)]. This can be obtained by computing

eval[((LABEL ALT

```
    (LAMBDA (X) (COND ((OR (NULL X) (NULL (CDR X))) X)
                      (T (CONS (CAR X) (ALT (CDDR X)))))))
  (QUOTE (A B C D E))),
NIL].
```

which gives the expected result (A C E).

This manner of evaluating requires that auxiliary functions must be defined within any expression where they are used (by using the label construct) and worse yet, they must be defined separately for separate occurrences. This can become very cumbersome and also makes expressions fairly difficult to understand. Another approach is to use an association list which associates each function name appearing in the expression with the appropriate defining expression. Thus we could evaluate *alt*[(A B C D E)] by computing

```
eval[(ALT (QUOTE (A B C D E))),
    ((ALT LAMBDA (X) (COND ((OR (NULL X) (NULL (CDR X))) X)
                           (T (CONS (CAR X) (ALT (CDDR X)))))))].
```

A simplified version of the usual LISP *eval* is the following:

```
eval[e, a] ←
    if at e then [if numberp e or e = NIL or e = T then e else dassoc[e, a]]
    else if at ae then
        [if ae = QUOTE then ade
        else if ae = COND then evcond[de, a]
        else if ae = LIST then evlist[de, a]
        else if ae = CAR then aeval[ade, a]
13.1)   else if ae = CDR then deval[ade, a]
        else if ae = CONS then eval[ade, a] . eval[adde, a]
        else if ae = ATOM then at eval[ade, a]
        else if ae = EQ then eval[ade, a] eq eval[adde, a]
        else eval[dassoc[ae, a] . de, a]]
    else if aae = LAMBDA then eval[adde, prup[ade, evlist[de, a]] * a]
    else if aae = LABEL then eval[adde . de, [ade . adde] . a].
```

where the auxiliary functions *evcond* and *evlist* are defined by

13.2) $evcond[u, a] \leftarrow$ if $eval[aau, a]$ then $eval[adau, a]$ else $evcond[du, a]$.

13.3) $evlist[u, a] \leftarrow$ if nu then NIL else $eval[au, a] . evlist[du, a]$.

and the auxiliary function *prup*, used for pairing up the elements of two lists of equal length, is defined by

13.4) $prup[u, v] \leftarrow$ if nu then NIL else $[au . av] . prup[du, dv]$.

Recall that *assoc* is defined by

13.5) $assoc[x, a] \leftarrow$ if na then NIL else if aaa eq x then aa else $assoc[x, da]$.

This simple *eval* expects that an expression is either a constant (*number*, T, NIL, or a QUOTEd S-expression), a variable whose value can be found on the association list, a conditional expression, a list making expression, or an application of a function, lambda expression or label expression to a list of arguments. Thus *eval* checks to see which of the above classes the expression to be evaluated falls into and proceeds accordingly. If the expression, *e*, is atomic then it is either a non QUOTEd constant or a variable. If the former then *eval* just returns the expression, if the latter it looks up the value on the association list, *a*.

If *e* is non-atomic but a*e* is atomic then *e* is either a QUOTEd constant, a conditional, a list maker, or an application of a function to a list of arguments. In the constant case the expression being quoted (ad*e*) is returned. If *e* is a conditional then the list of pairs is processed by the auxiliary evaluator *evcond*, which *evals* the "if" parts (aa*u*) in order until a true one is found then returns the result of *eval*ing the corresponding "then" part (ada*u*). In the list case the list of expressions to be evaluated is given to *evlist* which returns a list of the values. In the function application case there are two possibilities. If the function to be applied is one of the elementary functions, the indicated operation is performed on the result of evaluating the arguments. Otherwise the function must be defined in *a*, so *eval* looks up the definition, replaces the function name by the function definition in the expression and restarts the evaluation.

If neither *e* nor a*e* are atomic then it must be a lambda or label application. In the lambda case the argument list is given to *evlist* to be evaluated, the values are then paired with the list of variables to be bound by the lambda (ada*e*) using *prup* and put on the front of *a*. The body of the lambda (adda*e*) is then evaluated using this new association list. In the label case a new association list is formed by pairing the function name (ada*e*) with the defining expression (adda*e*) and adding the result to the front of *a*. Then the label expression is replaced in *e* by the defining expression and this is evaluated using the new association list.

If *e* is not an expression of the sort expected by *eval*, then the result is not defined. It would not be difficult to add additional clauses to *eval* so that it would return reasonable error messages rather than just being undefined (or dying in some strange way as would be likely in an actual computer). It would be necessary to be make the error messages distinguishable from legitimate values of the expression so that errors at inner levels of evaluation could be passed up the chain. This could be done by returning legitimate values as pairs whose first element is one atom and error messages as pairs prefixed by another. Terms containing *eval* would have to distinguish the cases.

Notice that COND and LIST considered as pseudo-functions behave differently than ordinary functions in that both can take an arbitrary number of arguments while functions defined by a lambda expression have a fixed number of arguments determined by the variable list occuring in the lambda expression. Also, the usual manner of evaluation an application term is LISP is to evaluate all of the arguments then apply the function. This will not work for COND as the main reason for a conditional is to be able to select a term to evaluate depending on some set of conditions and not to evaluate other terms under those conditions.

The above version of *eval* does not handle the propositional constructs and, or, and not. The effect of these constructs can be obtained by appropriate use of the condtional, but simply defining functions for and and or will not work as the evaluation of a function application requires that all of the arguments of of the function be evaluated before it is applied while the specification of and and or require that only as many of the arguments be evaluated as are required to determine the answer. Another problem is that in most implementations they are

allowed to take an arbitrary number of arguments. Thus we need to build them into *eval* for things to work properly. We will see later that LISP systems provide alternate solutions to this problem by providing a variety of modes of function application. (These are known as EXPR, FEXPR and LEXPRs.) Macros are also used for this purpose and will be explained later. Arithmetic is also missing from our *eval*. Adding these constructs is essentially like combining *eval* with the earlier evaluator *numval* and adding any additional primitive operations that are desired.

We note that *eval* can evaluate itself if it is given an association list containing the pairs (EVAL . λeval), (EVCOND . λevcond), (EVLIST . λevlist), (ASSOC . λassoc), (PRUP . λprup), and (APPEND . λappend) where λ<function name> stands for the internal form of the expression defining the named function. Thus

eval[(EVAL '(CAR '(A.B)) NIL), *alist*] = A

where *alist* is some association list containing the above mentioned pairs (and no other definitions of those functions).

When you talk to LISP you do not explicitly tell the interpreter what association list to use generally. This is because the LISP associates with each atom a list of properties, among these are the value, and/or the function definition associated with that atom. The LISP interpreter typically looks up variable values and function definitions on the corresponding property lists. Thus, instead of making up an association list with the appropriate variables and functions defined, the property lists must first be primed by doing SETQs for giving variables initial values and DEFUNs (or DEFPROPs) for any functions that need to be defined. These features will be discussed in more detail in Chapter V.

Exercises.

1. What is the value of

```
eval[(LEFT (QUOTE (A . B))),
    ((LEFT LAMBDA (X) (COND ((ATOM X) X) (T (LEFT (CAR X))))))]?
```

2. Translate the definition of *eval* given above into internal notation.

3. Go to your nearest LISP system, type in your answer to the second exercise and use it to check your answer to the first exercise.

Chapter II

WRITING RECURSIVE FUNCTION DEFINITIONS

In Chapter I we discussed the basic constructs of LISP and explained how LISP evaluates terms built up from them. The notion of recursively defined function was introduced and the rules for computing the value of a recursively defined function were given. In addition we showed how LISP programs are represented as S-expressions and how these programs are interpretered by the function *eval*. By now you should be able to read and understand simple LISP programs. The next step is learning to write LISP programs. In principle you already know all that is necessary, however there are some basic ideas and techniques which are useful in solving LISP programming problems. The purpose of this chapter is to help you learn to think recursively and to familiarize you with some of the basic techniques and standard forms of recursive programs. The final section contains a collection of problems of varying degrees difficulty for you practice on.

1. Static and dynamic ways of programming.

In order to write recursive function definitions, one must think about programming differently than is customary when writing programs in languages like FORTRAN or ALGOL or in machine language. In these languages, one has in mind the state of the computation as represented by the values of certain variables or locations in the memory of the machine, and then one writes statements or machine instructions in order to make the state change in an appropriate way. When writing recursive function definitions one takes a different approach. Namely, one thinks about the value of the function, asks for what values of the arguments the value of the function is immediate, and, given arbitrary values of the arguments, for what simpler arguments must the function be known in order to give the value of the function for the given arguments.

Let us consider a numerical example; namely, suppose we want to compute the function $n!$. For what argument is the value of the function immediate. Clearly, for $n = 0$ or $n = 1$, the value is immediately seen to be 1. Moreover, we can get the value for an arbitrary n if we know the value for $n-1$. Also, we see that knowing the value for $n = 1$ is redundant, since it can be obtained from the $n = 0$ case by the same rule as gets it for a general n from the value for $n-1$. All this talk leads to the simple recursive definition:

1.1) $$n! \leftarrow \text{if } n=0 \text{ then } 1 \text{ else } n\cdot[n-1]!.$$

We may regard this as a static way of looking at programming. We ask what simpler cases the general case of our function depends on rather than how we build up the desired state of the computation.

An example of the dynamic approach to programming is the following obvious ALGOL 60 program for computing $n!$:

```
        integer procedure factorial(n); integer s, i;
begin
        s := 1;
        i := n;
loop:   if i = 0 then go to done;
1.2)    s := i*s;
        i := i-1;
        go to loop;
done:   factorial := s;
end;
```

One often hears that static ways are bad and dynamic are good, but in many cases the the LISP style program is shorter and clearer. This style also provides a built in mechanism of problem solving which is rather like what is usually called *subgoaling*. We shall also see later how it also leads to rather natural methods of proving statements about programs.

Actually, when we discuss the mechanism of recursion, it will turn out that the above LISP program for factorial is inefficient because it uses the pushdown mechanism unnecessarily and should be replaced by the following somewhat longer program that corresponds to the above ALGOL program rather precisely:

$$n! \leftarrow fact[n, 1],$$

1.3)

$$fact[i, s] \leftarrow \text{if } i=0 \text{ then } s \text{ else } fact[i-1, i*s].$$

Perhaps the distinction between the two styles is equivalent to what some people call the distinction between *top–down* and *bottom–up* programming with *static* corresponding to *top–down*. LISP offers both, but the static style is better developed in LISP than in other languages, and we will emphasize it.

2. Recursive definition of functions on natural numbers.

In the next several sections we examine various forms of recursive function definition and programs having such forms. We begin by considering recursive definitions of numerical functions. We have already seen one example, namely the factorial function. The basic idea is to give a rule for computing the value of the function for a particular value of the argument (input) in terms of some collection of given (defined or built in) functions and values of the function for smaller arguments. Notice that we will have to specify the value for 0 directly as there are no smaller numbers. The method of function definition is parallel to the description of the domain of natural numbers in terms of the number 0 and the operation of successor, namely a number is either 0 or obtained by applying successor to a previously constructed number.

Recursive functions of natural numbers have the the subject of much study by logicians and mathematicians. One fairly nice result is that any such function can be computed starting with the constant 0, the basic functions *add*1 (more commonly known as *successor*) *sub*1 (also

known as *predecessor*) and the tools for recursive definition described in Chapter I. (Actually one can get by with less, but that is not the point of this discussion.) For example the sum and difference of two numbers are given by

2.1) $$plus[n, m] \leftarrow \text{if } n \text{ eq } 0 \text{ then } m \text{ else } plus[n-1, m]+1$$

2.2) $$differ[n, m] \leftarrow \text{if } n \text{ eq } 0 \text{ then } 0 \text{ else if } m \text{ eq } 0 \text{ then } n \text{ else } differ[n-1, m-1]$$

while the predicate *greaterp* which is true if the first argument is greater that the second can be computed by

2.3) $$greaterp[n, m] \leftarrow \text{if } n \text{ eq } 0 \text{ then } 0 \text{ else if } m \text{ eq } 0 \text{ then } 1 \text{ else } greaterp[n-1, m-1]$$

Here we use 0 to represent false and 1 to represent true in order to keep within the domain of numbers. We also write $n+1$ instead of *add*1 n and $n-1$ instead of *sub*1 n.

We could continue along these lines using *plus* to define *times*, *times* to define *exp*, *differ* to define *quot* and *rem*, and so forth building up a collection of useful functions. We will leave this as an exercise for the reader. The point is that at each stage the recursive definition is expressed in terms of given functions in a very simple form. Perhaps the simplest such form is the following schema:

2.4) $$f[n] \leftarrow \text{if } n \text{ eq } 0 \text{ then } a \text{ else } h[n-1, f[n-1]].$$

Here f is defined in terms of a fixed constant a and a given function h. This corresponds to what logicians call "primitive recursion without parameters". If we take $h[k, m]=[k+1] \cdot m$ and $a = 1$ we have the definition of *fact* given above (1.1).

If we allow parameters to be carried along we get the usual form of primitive recursion (shown with only one parameter for simplicity)

2.5) $$f[n, m] \leftarrow \text{if } n \text{ eq } 0 \text{ then } g[m] \text{ else } h[n-1, m, f[n-1, m]]$$

The definition of *plus* given above has this form with $g[m]=m$ and $h[n, m, k]=k+1$. As a further generalization we allow the parametric argument of f (in the recursive call) to be a given function of the parameter and the first argument giving

2.6) $$f[n, m] \leftarrow \text{if } n \text{ eq } 0 \text{ then } g[m] \text{ else } h[n-1, m, f[n-1, j[n-1, m]]].$$

The definitions of *differ* and *greaterp* given above has this form. Also the alternate recursive definition of the factorial function (1.3) is of this form. We may wish to express the computation directly in terms of several preceding values instead of just $f[n-1]$. One form of this "course of values" type recursion is given by

2.7)
$$\begin{aligned} f[n] \leftarrow\ & \text{if } n \text{ eq } 0 \text{ then } a_0 \text{ else} \\ & \text{if } n \text{ eq } 1 \text{ then } a_1 \text{ else} \\ & \cdots\cdots \\ & \text{if } n \text{ eq } b \text{ then } a_b \text{ else} \\ & h[n, f[p_1[n]], \ldots f[p_c[n]]] \end{aligned}$$

where $0 \le p_i[n] < n$ for $1 \le i \le c$ when $b < n$. As an example we have the definition of the function giving the nth Fibonacci number:

2.8) $$fib[n] \leftarrow \text{if } n \text{ eq } 0 \text{ then } 0 \text{ else if } n \text{ eq } 1 \text{ then } 1 \text{ else } fib[n-1] + fib[n-2]$$

We note that this is a particularly unfavorable case for recursively defined functions as the evaluation from this definition takes order of $fib\ n$ amount of work for input of n. A more efficient definition is:

2.9) $$fiba[n] \leftarrow fibb[n, 0, 1]$$
$$fibb[n, k, m] \leftarrow \text{if } n \text{ eq } 0 \text{ then } k \text{ else } fibb[n-1, m, m+k]$$

Evaluation of the nth Fibonacci number using this definition requires only order of n amount of work.

The above forms of recursive function definition all have a very nice property. Namely, assuming that the given functions appearing in the schemas are total, the function defined by the schema is total. That is the rules given for evaluation of such functions will always produce an answer in a finite number of steps. The general form of recursive definition

2.10) $$f[n1, \ldots, nk] \leftarrow \tau[f, n1, \ldots, nk]$$

where τ is some term involving f, the arguments ni and perhaps additional given functions. Definitions of this general form are not guaranteed to define total functions. For example

2.11) $$loop\ n \leftarrow loop\ n$$

is a perfectly good definition, however the rules for computation will not produce an answer for any value of n. There are of course many cases where the function defined by a recursive definition is total, but where the definition does not fit any of the above patterns.

The standard example of a total function on natural numbers that is not definable by primitive recursion is known as Ackermann's function and is defined by

2.12) $ack[m, n] \leftarrow \text{if } m \text{ eq } 0 \text{ then } n+1 \text{ else if } n \text{ eq } 0 \text{ then } ack[m-1, 1] \text{ else } ack[m-1, ack[m, n-1]]$.

The proof is based on showing that $ack[m, n]$ grows faster than any primitive recursive function.

3. Simple list recursion.

About the simplest form of recursion in LISP occurs when one of the arguments is a list, the result is immediate when the argument is null, and otherwise we need only know the result for the d-part of that argument. Several of the functions defined in Chapter I are of this form. Consider, for example, $u*v$, the result of *appending* the list u to the list v. The result is immediate for the case $n\,u$ and otherwise depends on the result for $d\,u$. Thus, we have

3.1) $$u*v \leftarrow \text{if } n\,u \text{ then } v \text{ else } a\,u\ .\ [d\,u * v]$$

Notice that if we had tried to recur on v rather than on u we would not have been successful. The result would be immediate for $\mathbf{n}\,v$, but $u*v$ cannot be constructed in any direct way from $u*\mathbf{d}\,v$. Often a function is easily defined by a recursion on one of its arguments and not by recursions on others.

Another example is the function *member* that tests for membership in a list. If the list is empty then the answer is NIL otherwise either the element we are searching for is the first thing on the list, or it is in the rest of the list. This reasoning gives rise to the recursive definition

3.2) $$\mathit{member}[x, u] \leftarrow \text{if } \mathbf{n}\,u \text{ then NIL else } x \text{ eq } \mathbf{a}\,u \text{ or } \mathit{member}[x, \mathbf{d}\,u].$$

The reader should observe that this definition is equivalent to that given by (I.8.6) using only propositional operations.

The function *reverse* is another example of simple list recursion.

3.3) $$\mathit{reverse}[u] \leftarrow \text{if } \mathbf{n}\,u \text{ then NIL else } [\mathit{reverse}\ \mathbf{d}\,u] * [\mathbf{a}\,u\ .\ \text{NIL}]$$

This straight forward definition of *reverse* involves *appending* the *reverse* of the rest of the list to the list containing only the first element of the list.

As with recursive definition of numerical functions we see that simple list recursion parallels the description of the domain of lists which says that a list is either NIL or it is the result of *cons*ing an S-expression onto a "smaller" list. We can give recursion schemas for list recursion analagous to those for numerical functions. In general, the recursive definition of a function on lists must take into account the fact that we have not one "successor" of a given list, but one for each possible S-expression. For example "primitive list recursion" without parameters has the form

3.4) $$f[u] \leftarrow \text{if } \mathbf{n}\,u \text{ then } g0 \text{ else } h[\mathbf{a}\,u,\ \mathbf{d}\,u,\ f[\mathbf{d}\,u]]$$

Taking g0 = NIL and $h[x, v, w] = w * [x\ .\ \text{NIL}]$ we get the definition of *reverse* given above (3.3). If we mix numerical and list computation taking $g0 = 0$ and $h[x, u, n] = n+1$ we get

3.5) $$\mathit{length}[u] \leftarrow \text{if } \mathbf{n}\,u \text{ then } 0 \text{ else } \mathit{length}[\mathbf{d}\,u]+1$$

The function *last* which computes the last element of a list is another example.

3.6) $$\mathit{last}\ u \leftarrow \text{if } \mathbf{n}\,\mathbf{d}\,u \text{ then } \mathbf{a}\,u \text{ else } \mathit{last}\ \mathbf{d}\,u,$$

In order to fit the schema exactly the above definition needs to be patched somewhat. See if you can figure out what must be done and what the "given" function h should be.

The schema for primitive list recursion with parameters (only one parameter shown) is

3.7) $$f[u, x] \leftarrow \text{if } \mathbf{n}\,u \text{ then } g[x] \text{ else } h[\mathbf{a}\,u,\ \mathbf{d}\,u,\ x,\ f[\mathbf{d}\,u,\ x]].$$

append (3.1), *member* (3.2) and *assoc* (I.9.3) are examples of this form of definition. Allowing a given function of the arguments to occur in the parameter position in the recursive call on f we obtain the schema

3.8) $f[u, x] \leftarrow \mathbf{if}\ \mathbf{n}u\ \mathbf{then}\ g[x]\ \mathbf{else}\ h[\mathbf{a}u, \mathbf{d}u, x, f[\mathbf{d}u, f[\mathbf{a}u, \mathbf{d}u, x]]]$.

The definition of *reverse* (I.8.7) in terms of the basic *cons* operation, itself and an auxiliary variable has this form.

The simple recursion on lists without parameters simply proceeds through a list gathering results to be used in constructing the answer, but provides no access to information previously obtained. When we allow parameters to be carried along the information can be passed down the list as well as results being passed back up. The appropriate mixing of control structure and information passing is often the key to solving a programming problem. We shall see in a later sections how one can sometimes solve the two aspects of the problem separately and then piece together the results.

The function *alt* (I.8.1) which returns a list of alternate elements of a list is also based on list recursion, however the form of the recursion is different. Here the base cases are the empty list and the list containing one element. In the general case we use the result for $\mathbf{dd}u$ to compute the result for u. This is analogous to the "course of values" recursion on numbers (2.7).

The above schemas, like their numerical counterparts yield definitions of total functions (assuming that the given functions are total). The most general form of list recursion is the same as that for numerical recursion (2.10). In the case of pure list recursion we build the term on the right hand side from basic list functions and expect the arguments to be lists. As we have seen above we often mix computation involving lists, numbers and S-expression in general. Also, as with numerical recursion, recursive definition of functions on lists does not in general give rise to total functions and some care must be used to make sure your program will terminate. We will have more to say about proving termination in Chapter III.

4. Simple S-expression recursion.

In the case of recursion on the structure of S-expressions, the simplest situation is when the value of the function is immediate for atoms, and for non-atoms depends only on the values for the a-part and the d-part of the argument. Thus *subst* was defined by

4.1) $subst[x, y, z] \leftarrow \mathbf{if}\ \mathbf{at}z\ \mathbf{then}\ [\mathbf{if}\ z\ \mathbf{eq}\ y\ \mathbf{then}\ x\ \mathbf{else}\ z]\ \mathbf{else}\ subst[x, y, \mathbf{a}z]\ .\ subst[x, y, \mathbf{d}z]$.

More generally we have recursive definitions of the form

4.2) $f[x] \leftarrow \mathbf{if}\ \mathbf{at}x\ \mathbf{then}\ g[x]\ \mathbf{else}\ h[\mathbf{a}x, \mathbf{d}x, f[\mathbf{a}x], f[\mathbf{d}x]]$.

Here again the recursive definition of functions parallels the description of the domain. The definitions of *size* (the number of atoms in an S-expression) and *fringe*

4.3) $size[x] \leftarrow \mathbf{if}\ \mathbf{at}x\ \mathbf{then}\ 1\ \mathbf{else}\ size\ \mathbf{a}x + size\ \mathbf{d}x$

4.4) $fringe\ x \leftarrow \mathbf{if}\ \mathbf{at}x\ \mathbf{then}\ \langle x \rangle\ \mathbf{else}\ fringe\ \mathbf{a}x * fringe\ \mathbf{d}x$

are further examples of this form of definition.

The S-expression analogue of (2.6) and (3.8) (primitive S-expression recursion) is

4.5) $$f[x, y] \leftarrow \text{if } \mathrm{at}\, x \text{ then } g[x, y] \text{ else } h[\mathrm{a}x, \mathrm{d}x, y, f[\mathrm{a}x, ja[x, y]], f[\mathrm{d}x, jd[x, y]]]$$

and the following definition of *equal* is of this form

4.6) x *equal* $y \leftarrow$ if at x then [if at y then x eq y else NIL] else if at y then NIL else ax *equal* ay and dx *equal* dy.

Note that using the fact that eq is in fact defined on non-atoms in actual LISP implementations and properties of the propositional functions, we can simplify the definition of *equal* to

4.7) $$x \textit{ equal } y \leftarrow x \text{ eq } y \text{ or } [\neg \mathrm{at}\, x \text{ and } \neg \mathrm{at}\, y \text{ or } \mathrm{a}x \textit{ equal } \mathrm{a}y \text{ and } \mathrm{d}x \textit{ equal } \mathrm{d}y]$$

as was given in Chapter I.

Although the above definition of *equal* is an instance of the schema (4.5) it is more natural to think of it as a parallel recursion through the two arguments rather than thinking of one of them as a parameter. This could of course be elaborated to carry along parameters as well as recurring in parallel through some collection of arguments. In later chapters we will see more examples of programs that recur on more than one argument. (For example the *samefringe* program discussed in Chapter IV). The important point to check in such programs is that the collection of arguments being recurred on is always "simpler" in recursive calls to the program being defined.

The definition of *flatten* is an example of a double recursion.

4.8)
$$\textit{flatten}[x] \leftarrow \textit{flat}[x, \text{NIL}]$$
$$\textit{flat}[x, y] \leftarrow \text{if } \mathrm{at}\, x \text{ then } x \,.\, y \text{ else } \textit{flat}[\mathrm{a}x, \textit{flat}[\mathrm{d}x, y]]$$

Although *flatten* computes the same function as *fringe*, it is more efficient because the *append* operation used by *fringe* copies unnecessarily. As we shall see in a later chapter, *flatten* is also an improvement over *fringe* in that it is partially amenable to tail recursive evaluation. The kind of double recursion used in *flatten* is often useful. One way of thinking of this form of recursion is that results are computed from one side of the S-expression (the d side in this case) and then passed along to the other side. Thus we obtain a flow of information from side to side as well as top to bottom. The technique of writing a simple and easily understandable recursive definition and then later optimizing by the kind of transformation made in going from *fringe* to *flatten* is often useful.

Note that the definition of *flat* does not fit the general primitive recursive schema (4.5). Thus we have an example where one definition of a function clearly fits one of the simple recursion schemas and thus defines a total function, while for another definition of the function this fact is not immediate. Of course showing that both definitions compute the same function also requires some work. We could add to our list of schemas some which allow multiple recursions such as exhibited by *flatten*. However no matter how many such forms for defining total functions we allow, we will eventually have to resort to the general form (2.10) which is also the general form for S-expression recursion.

5. Other structural recursions.

When S-expressions are used to represent a class of expressions that are defined inductively then functions of these expressions often have a recursive form closely related to the inductive definition of the expressions. For example the arithmetic interpreter *numval* (I.9.4) computes directly the value for constants and variables and for sums and products it computes the value in terms of the values of the subexpressions from which the sum or product is constructd. The function *diff* (I.11.3) which symbolically differentiates the same class of expressions has a similar recursive structure in that it knows the answers immediately for constants and variables and for sums and products it uses the results for subexpressions to obtain the final answer. If we consider the arithmetic expressions where sum and product are simple binary operators we can write a simple recursion schema as follows:

```
                f[x, y] ←
                    if num[x] then gnum[x, y] else
5.1)                if isvar[x] then gvar[x, y] else
                    if issum[x] then gsum[s1 x, s2 x, y, f[s1 x, y], f[s2 x, y]]
                    else if isprod[x] then gprod[p1 x, p2 x, y, f[p1 x, y], f[p2 x, y]]
```

Then the evaluation function *numval*

```
                numval[x, y] ←
                    if isnum[x] then cval[x] else
5.2)                if isvar[x] then lookup[x, y] else
                    if issum[x] then sum[numval[s1 x, y] numval[s2 x, y]]
                    else if isprod[x] then prod[numval[p1 x, y] numval[p2 x, y]]
```

is an instance of this schema where *cval*, *lookup*, *sum* and *prod* are given.

A more complicated example of an interpreter with recursive structure based on the expression language structure is the LISP interpreter *eval* (I.13.1).

In general if we have a domain defined by starting with a base set (with computable test for membership) and a collection constructors with which we build elements of the domain by repeated application starting with elements of the base set, then there is a corresponding schema for recursive definition of functions on this domain. We will see more examples and have more to say about this in later chapters.

6. General tree recursion.

A tree structure is either a leaf or a node together with a collection of immediate successors which are also trees. Many data structures can be viewed as trees. For example, S-expressions are trees with leaves as atoms, and each non-leaf tree has two successors given by the *car* and the *cdr*. More generally, the structures described in section §5 can be thought of as tree-like structures with the basic elements as leaves, and each non-leaf tree having immediate successors given by the selector functions for the particular type of node. These structures all have two

things in common. Namely, (i) each kind of node has a fixed number of successors and (ii) these successors are already present as parts of the original tree.

Another kind of tree is given by an initial position together with a *successors* function from which we may obtain the immediate successors of any position (of relevance). A node may have any number of successors, and parts of the trees are built as needed. Functions on such trees typically involve two processes, one for recognizing and dealing with leaves and one for processing a collection of succesors. (Of course the two processes interact recursively.)

Consider the problem for searching a tree for a position having some particular property. We can write a program that describes a depth first search independent of the property or kind of tree. It can be used to search specific trees by defining the three auxiliary functions *successors*, *ter*, and *lose* for the particular problem. We have

6.1) *search p* ← **if** *lose p* **then** LOSE **else if** *ter p* **then** *p* **else** *searchlis*[*successors p*]

where

6.2) *searchlis u* ← **if** n *u* **then** LOSE **else** [λx: **if** *x* **eq** LOSE **then** *searchlis* d *u* **else** x][*search* a *u*].

In the applications, we start with a position $p0$, and we are looking for a win in the successor tree of $p0$. Certain positions lose and there is no point looking at their successors. This is decided by the predicate *lose*. A position is a win if it doesn't lose and it satisfies the predicate *ter*. The successors of a position are given by the function *successors*, and the value of *search p* is the winning position. No non-losing position should have the name LOSE or the function won't work properly.

One application is finding a path from an initial node to a final node in a graph represented by a list structure as described in chapter I. A position is a path starting from the initial node and continuing to some intermediate node and is represented by a list of its nodes in reverse order. The three functions for this application are

6.3) *lose p* ← a *p* ∈ d *p* ,
6.4) *ter p* ← [a *p* eq *final*] ,
6.5) *successors p* ← *mapcar*[λx: *x* . *p*, d*assoc*[a *p*, *graph*]].

The node we are proposing to visit next loses if it is already in the path, as that means we have been here before. The successors to a position are all of those positions immediately reachable from the current node. In our list notation for a graph the immediately reachable nodes are those on the list associated with the current node in *graph*. A position is terminal if the current node is the desired goal *final*.

search is used when we want to search a tree of possibilities for a solution to a problem. Naturally we can do other things with tree search recursion than just search. For example we may want to find all solutions to a problem. This can be done with a function *allsol* that uses the same *lose*, *ter*, and *successors* functions as does *search*. The simplest way to write *allsol* is

6.6) *allsol p* ← **if** *lose p* **then** NIL **else if** *ter p* **then** <*p*> **else** *mapapp*[*allsol*, *successors p*].

where

6.7) $mapapp[fn,\ u] \leftarrow$ **if** n u **then** NIL **else** $fn[a\,u] * mappap[fn,\ d\,u]$.

This form of the function is somewhat inefficient because of all the *appending*. A more efficient form uses an auxiliary function as follows:

$allsol\ p \leftarrow allsola[p,\ NIL]$

$allsola[p,\ found] \leftarrow$
    if *lose* p **then** *found*
    else if *ter* p **then** p . *found*
    else $allsolb[successors\ p,\ found]$.

$allsolb[u,\ found] \leftarrow$ **if** n u **then** *found* **else** $allsolb[d\,u,\ allsola[a\,u,\ found]]$.

The recursive program structure that arises here is common when a list is to be formed by recurring through a list structure.

We will give further applications of and variations on this form of tree search recursion in Chapter VII.

7. Non-structural recursions.

Not all recursions are structural. Any recursion may be put in the general form

$$f(x) \leftarrow \text{if } p(x) \text{ then } g(x) \text{ else } f(h(x)),$$

where several variables may have to be *cons*ed together to make the one variable x, but unless the functions g and h have some special relation, there is no reason to suppose that the computation will terminate.

For example,

$$f(n) \leftarrow \text{if } n=1 \text{ then } 1 \text{ else if } even\ n \text{ then } f(n/2) \text{ else } f(3\ n + 1)$$

terminates for all positive integer values of n that have been tested, but no-one has been able to determine whether it always terminates. If you want to experiment, try small values of n get an idea of how the function seems to behave, and then try $n = 27$.

The function

$sack[x, y] \leftarrow$ **if** a x **then** x . y **else if** a y **then** $sack[a\,x,\ d\,x]$ **else** $sack[d\,x,\ sack[x\ ,\ d\,y]]$

is an S-expression analog of the function *ack* defined in section §2 on numerical recursion. Unlike *flat* the occurrence of double recursion ($sack[\ldots, sack[\ldots]]$) is crucial. Like *ack* this function cannot be defined by any collection of simple recursive schemata. It is a kind of structural recursion.

Takeuchi defined the function

$$tak(x, y, z) \leftarrow \text{if } x \leq y \text{ then } z \text{ else } tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y))$$

to test LISP systems on a program that would run a long time but not use much storage or stack. It has been shown to always terminate for integer arguments. Most likely it also always terminates for real arguments, but this seems harder to show.

The interpreter *eval* is itself a non-structural recursion, and whether it terminates depends on the expressions it is given to evaluate.

8. Solving a LISP programming problem.

In this section we present a programming problem that is somewhat more complex than the examples of earlier sections and work out the solution in detail, the purpose being to help you get started thinking recursively and to show how a typical problem might be approached. After describing the function we wish to compute, we will write programs for two simpler but related functions in order to better understand various aspects of the control structure and information flow needed in the main computation. This is a useful technique in problem solving in general, the trick of course is to find a simplification which produces a problem that you can solve and whose solution is useful in solving the original problem. Having written programs for these simpler functions, we then use the insight gained to to solve the main problem.

The problem has to do with "lists of lists" or L-lists. An L-list is either the empty list, NIL, or and atom *cons*ed onto the front of an L-list, or an L-list *cons*ed onto the front of an L-list. Thus (A (A B) C) is an L-list, but (A (B . C) D) is not. Although this class of S-expressions is less general than the usual LISP notion of list, it is more uniform. The computation of a function on L-lists may use the value of the function on the *car* of the L-list (when it is non-atomic) as well as on the *cdr* as both are again L-lists.

The function *allsubsub* returns a list of all occurrences of an L-list u as a sublist or as a sublist of a sublist of an L-list v. For example (A A) occurs three times as a subsublist of (A A A B A A) and (A) occurs 4 times in (A (B (A A)) (C (((A))))).

We begin with a simpler function restricted to lists of atoms. Thus we wish to compute the function *allsub* that returns a list of all occurrences of a list of atoms u as a sublist of another list of atoms v. The first thing to do is decide on the representation of an occurrence of one list as a sublist of another list. In the case of lists of atoms, a fairly natural solution is to represent an occurrence of a particular sublist as the number n corresponding to position in the list of the beginning of that occurrence. Thus

$$allsub[(A\ A), (A\ A\ A\ B\ A\ A)] = (1\ 2\ 5).$$

To compute $allsub[u, v]$, if v is NIL then the answer is NIL Otherwise, imagine that we know the answer for dv. Then there are two possiblities. Either u "agrees with" v (v may be longer than u, but up to the end of u the elements of v and u are the same), or not. If not, the answer is just the result for dv otherwise we add the current position to the result for dv. This analysis suggests that we will need an additional argument to keep track of the current position. Thus we define a function $allsub1[u, v, p]$ where the argument p is intended to correspond to the

position of the argument v in the initial list. If v is NIL then the result is NIL. Otherwise suppose we know the value of *allsub1*$[u, dv, p+1]$ then we either add p onto that value or return it unchanged according to whether or not u agrees with v. Assuming we have a suitable definition of *agrees*, the above analysis leads to the following definitions:

8.1)
$$allsub[u,\ v] \leftarrow allsub1[u,\ v,\ 1]$$
$$allsub1[u,\ v,\ p] \leftarrow$$
$$\quad \textbf{if } n\,v \textbf{ then NIL}$$
$$\quad \textbf{else if } agrees[u,\ v] \textbf{ then } p \,.\, allsub1[u,\ d\,v,\ 1 + p]$$
$$\quad \textbf{else } allsub1[u,\ d\,v,\ 1 + p].$$

It remains for us to write the definition of *agrees*$[u, v]$ which determines whether or not u matches the beginning of v. This is easy. If u is NIL then we have gotten to the end of u with out finding a disagreement and we return T. If v is NIL (and u is not) then we return NIL and u does not agree with the beginning of v. Otherwise the result is T only if $a\,u = a\,v$ and the *cdr*s of u and v agree. Thus

8.2) $agrees[u,\ v] \leftarrow$ **if** $n\,u$ **then** T **else if** $n\,v$ **then** NIL **else** $a\,u$ **eq** $a\,v$ **and** $agrees[d\,u,\ d\,v]$.

The internal forms of the above definitions are

```
(DEFUN ALLSUB (U V) (ALLSUB1 U V 1))

(DEFUN ALLSUB1 (U V P)
       (COND ((NULL V) NIL)
             ((AGREES U V) (CONS P (ALLSUB1 U (CDR V) (ADD1 P))))
             (T (ALLSUB1 U (CDR V) (ADD1 P)))))

(DEFUN AGREES (U V)
       (COND ((NULL U) T)
             ((NULL V) NIL)
             (T (AND (EQUAL (CAR U) (CAR V))
                     (AGREES (CDR U) (CDR V))))))
```

Now we return to the *allsubsub* problem. As with the simpler case, we must first decide how to represent an occurrence of an L-list as a subsublist of an L-list. Again we need only represent the position where the list comparison begins. Consider an arbitrary point, p, in an L-list. Either it is an element of the list, or is contained in one of the elements of the list. In the first case the position n of the element is sufficient, in the second case we need the position n of the element together with the position of the point p relative to that element. Thus in the first case we take the list containing only the number n and in the second case we add n onto the list representing the position of p in the nth element of the list. Using this representation we have

allsubsub[(A),(A (B (A A)) (C (((A))))))] = ((1) (2 2 1) (2 2 2) (3 2 1 1 1)).

To construct the desired list of positions we check for a agreement of u at each position and add matching positions to the result. (Actually we will see that some positions can be ruled out with out explicitly checking for agreement.) In particular, we will need to pass around sufficient information to be able to construct a representation of the current position whenever an agreement is found. Let us consider the simpler problem of computing *allpos*$[v]$, a list of all the

positions in v. A program for this can easily be modified to list only those positions satisfying the "agrees" condition. A definition of *allpos* can be obtained by recalling the discussion about of how a position is to be represented. Namely if n is the position of v in the list of which v is a tail then (i) if v is NIL then there are no positions and the answer is NIL, (ii) a v is an atom then add $\langle n \rangle$ to the list of positions in d v passing $n+1$ as the current position number (iii) otherwise compute the list of positions in a v (relative to a v), tack n onto the front of each position, append this onto the list of positions in d v (again passing $n+1$) and finally add the current position, $\langle n \rangle$, to the result. Thus we have

```
                allpos v ← allpos1[v, 1]
8.3)
                allpos1[v, n] ←
                    if n v then NIL
                    else if at a v then <n> . allpos1[d v, 1 + n]
                    else <n> . [tack[n, allpos a v] * allpos1[d v, 1 + n]]

8.4)            tack[n, w] ← if n w then NIL else [n . a w] . tack[n, d w]
```

Now we modify to *allpos* and *allpos1* to carry around the parameter u and rule out those positions that don't agree. In particular we only add $\langle n \rangle$ to the list if *agrees*$[u, v]$ is satisfied, and in this case we don't look for agreement in a v as the top level match makes this impossible (recall our lists are finite tree-like structures). So we arrive at the following:

```
                allsubsub[u, v] ← allsubsub1[u, v, 1]
8.5)
                allsubsub1[u, v, n] ←
                    if n v then NIL
                    else if agrees[u, v] then <n> . allsubsub1[u, d v, 1 + n]
                    else if at a v then allsubsub1[u, d v, 1 + n]
                    else tack[n, allsubsub[u, a v]] * allsubsub1[u, d v, 1 + n]
```

An alternate solution for the *allpos* and the *allsubsub* problems is to pass along a complete representation of the current position. When we pass it in the d v direction we need to add 1 to the last element and when we pass it in the a v direction we need to add a 1 to the end of the list. Since all of the action is at the end of the list it is easier to pass along the reverse of the position list and re-reverse it when returning the position as an answer. Thus we get the following definitions

```
                allpos v ← allpos1[v, (1)]
8.6)
                allpos1[v, p] ←
                    if n v then NIL
                    else if at a v then reverse p . allpos1[d v, [1 + a p] . d p]
                    else reverse p . [allpos1[a v, 1 . p]] * allpos1[d v, [1 + a p] . d p]]
```

allsubsub[*u*, *v*] ← *allsubsub*1[*u*, *v*, (1)]

8.7)

*allsubsub*1[*u*, *v*, *p*] ←
    if n *v* then NIL
    else if *agrees*[*u*, *v*] then *reverse p* . *allsubsub*1[*u*, d *v*, [1 + a *p*] . d *p*]
    else if at a *v* then *allsubsub*1[*u*, d *v*, [1 + a *p*] . d *p*]
    else *allsubsub*1[*u*, a *v*, 1 . *p*]] * *allsubsub*1[*u*, d *v*, [1 + a *p*] . d *p*]

Notice that in the first solution, the construction of the position representation was taken care of somewhat automatically by the recursive structure of the program, while in the second case we had to worry about the details of constructing the position representations.

In both cases the computation works by passing information both down and across the list structure and getting results back. What is passed on and the interpretation of the result returned depends on the direction. In this example there is no exchange of information between the subcomputations. In a more complicted situation this might also be necessary.

9. Lots of LISP functions to program.

The following are descriptions of functions on lists and S-expressions. You should write LISP programs to compute these functions. A description of the form "*foo*[*x*] is true if and only if ..." means your program should return T in the case that *foo*[*x*] is true and NIL otherwise. Write your solutions out in external form, if you like, then convert them to internal form and try them out in whatever LISP system you have access to. You should convince yourself by some informal process of reasoning that your solutions are indeed correct. You will be asked to prove various facts about your programs after you have studied Chapter III. This should encourage you to write clean, understandable programs and document them carefully so you will remember why you thought they were correct to begin with and what tricks you may have employed.

The problems are classified according to the intended interpretation of the lists or S-expressions. The classes include lists, S-expressions, arithmetic expressions, polynomials, (finite) sets, permutations, trees, and graphs. Each group begins with some simple problems (with one line solutions) to get you going. For the more complicated functions, you will find that defining auxiliary functions is useful. Towards the end the programs may require a fair amount of thinking to get things organized correctly, but none of the problems require really lengthy programs. Pay attention to getting the trivial cases (e. g. when some of the arguments are NIL or atomic or otherwise to be considered basic) correct. It is in general important to understand the trivial cases in the computation of a function.

The observant reader will no doubt notice that some of these functions are defined in later chapters. Thus you will have some cases where you can compare your solution to those given. (Pointers to these definitions can be found in the function index.)

Lists

1. *samelength*[*u*, *v*] tests whether the lists *u* and *v* have the same length.

samelength[(A B C), (D E F)] = T
samelength[(A B C), (A C)] = NIL

Do not use any operations or tests on numbers to write your program.

2. *prup*[*u*, *v*] is the list formed by pairing the corresponding elements of the lists *u* and *v*. Thus

prup[(X Y Z), (1 A (FOO BAR))] = ((X . 1) (Y . A) (Z FOO BAR)).

3. *istail*[*u*, *v*] is true if and only if the list *u* is a tail of *v*. That is when *cdr*ing through *v* you will eventually get to *u*.

4. *commontail*[*u*, *v*] is the longest common sublist of *u* and *v* ending with the ends of the lists. Thus

commontail[(A B C D E),(A C D E)] = (C D E).

5. *upto*[*u*, *v*] when *u* is a tail of *v*, is the list of elements of *v* upto the point where *u* begins. Thus

upto[(C D E),(A B C D E)] = (A B).

6. *mapapp*[*f*, *u*] appends together the lists obtained by applying *f* to successive elements of *u*.

7. *mapchoose*[*p*, *u*] forms a list of those elements of *u* satisfying *p*.

8. *partition*[*u*, *n*] is the list of partitions of the list *u* into *n* sublists *u*1, ... *un* such that *u*1*[*u*2*...*un*]=*u*. If *n* is greater than the length of *u* then your program should return an error message of some kind. Thus

partition[(A B C),2]=((A) (B C))((A B) (C)))

Write a program *testpart* to test the result returned by *partition*. For each partition, *testpart* should append together the lists *ui* and check that the result is *u*.

S-expressions

9. *mirror*[*x*] returns the mirror image of an S-expression *x*. Thus

mirror[((A.B).(C.D))]=((D.C).(B.A)).

10. *occur*[*x*, *y*] is true if and only if the atom *x* occurs in the S-expression *y*. For example

occur[B, ((A.B).C)] = T.

11. *multiplicity*[*x*, *y*] is the number of times the atom *x* occurs in the S-expression *y*.

A path in an S-expression *x* is a list of A's and D's such that you can apply a succession of a's or d's to *x* (according to whether the next list element is A or D) without trying to apply a or d to an atom. We say that the resulting subexpression is at the end of the path or that the path leads to that subexpression. Thus (A D) is a path in ((X . (Y . Z)) . F) but (D A) is not. Also (Y . Z) is at the end of the path (A D) in ((X . (Y . Z)) . F).

12. *ispath*[*p*, *x*] is true if and only if *p* is a path in *x*.

13. *depth*[*x*] is the length of the longest path leading to an atom in the S-expression *x*.

depth[(((A . B) . C) . D] = 3.

We say that an S-expression is balanced if it is an atom or if *depth*[a*x*] and *depth*[d*x*] differ by at most 1 and a*x* and d*x* are both balanced.

14. *balanced*[*x*] is true if and only if the S-expression *x* is balanced.

15. *balance* x is balanced and has the same fringe as x

16. *get*[*y*, *p*] is the subexpression at the end of the path *p* in the S-expression *y*. Thus

get[(A ((B) C) (B)), (D A A)] = (B).

17. *point*[x, y] is the path in the S-expression y leading to the left-most occurrence of the S-expression x in the S-expression y. Thus

point[(B), (A ((B) C) (B))] = (D A A).

18. *allpoint*[x, y] is a list of all paths p such that *get*[y, p] = x. Thus

allpoint[(B), (A ((B) C) (B))] = ((D A A) (D D A)).

19. *commons* x is a list of the subexpressions of the S-expression x that occur more than once in x, each followed by a list of the points (paths leading to the points) where it occurs.

Symbolic arithmetic and algebra

20. Let a polynomial in x be represented by a list of its coefficients in order of ascending powers of x. Thus x^3+x+5 is represented by (5 1 0 1).

    20.1. *sum*[u, v] is the sum
    20.2. *prod*[u, v] is the product
    20.3. *quot*[u, v] is the quotient
    20.4. *rem*[u, v] is the remainder

    of the polynomials u and v in the same notation.

21. Consider arithmetic expressions as represented in this chapter. Namely an expression is

    (i) a number (satisfies *numberp*),
    (ii) a variable (not a number and satisfies at),
    (iii) a sum : PLUS . < list of expressions > or
    (iv) a product : TIMES . < list of expressions >.

    (For simplicity, assume the sum and product lists always have at least 2 elements.)

    The function *sop* converts such expressions into sum of products form, eg. the resulting expression is either a monomial or a sum of monomial terms which has the form PLUS . <list of monomials>. A monomial is either a number, a variable, or a product of the form TIMES . < list of numbers or variables >.

    Try your simplifier on expressions returned by *diff*.

Sets

22. Lists can be thought of as representing finite sets using the the program *member*[x, u] defined earlier, to compute the membership relation. Write programs to compute the functions union, $u \cup v$, intersection, $u \cap v$, and the set difference, $u-v$, for lists u and v viewed as representing sets. For example:

(A B C) ∪ (B C D) = (A B C D),
(A B C) ∩ (B C D) = (B C),

and

(A B C) - (B C D) = (A).

[Remark: There are a couple of possible conventions for representing finite sets as lists. One is to require S-expressions representing elements of a set to occur in the representing list exactly once. The other is to allow multiple occurrences of an S-expression representing an element of the set. You should decide which convention you wish to adopt, and make sure your programs are consistent with that convention.]

23. Suppose x takes numbers as values and u takes as values lists of numbers in ascending order, e.g. (2 4 7). Write a function *insert*[x, u] whose value is obtained from that of u by putting x in u in its proper place. Thus

$$insert[3, (2\ 4)] = (2\ 3\ 4),$$

and

$$insert[3, (2\ 3)] = (2\ 3\ 3).$$

24. Write functions giving the union, intersection, and set difference of ordered lists; the result is wanted as an ordered list.

Note that computing these functions of unordered lists takes a number of comparisons proportional to the square of the number of elements of a typical list, while for ordered lists, the number of comparisons is proportional to the number of elements.

25. Using *insert*, write a function *sort* that transforms an unordered list into an ordered list.

26. Write a function *goodsort* that sorts a list using a number of comparisons proportional to $n \log n$, where n is the length of the list to be sorted.

Permutations

27. *rcycle*[u] is obtained from the list u by moving the last element to the first position. Thus

$$rcycle[(A\ B\ C\ D)] = (D\ A\ B\ C).$$

28. *lcycle*[u] is obtained from the list u by moving the first element to the last position. Thus

$$lcycle[(A\ B\ C\ D)] = (B\ C\ D\ A).$$

We can think of a permutation on n objects as a function which maps a list (I1 I2 ... IN) of n distinct elements to a list containing the same elements in a different order. Thus (3 4 1 2) is a permutation of (1 2 3 4) and (FOO BAR BAZ) is a permutation of (BAR BAZ FOO). The permutation itself can be represented as a list in several ways. We describe two possible representations of a permutation *pi*.

REP1: *pi* is represented as a list of numbers (j1 j2 ... jN) which is itself a permutation of the list (1 2 ... N) and the result of applying such a permutation to a list u of length n is the list u' where the kth element of u' is the j*k*th element of u. Thus the permutation (2 3 4 1) applied to the list (A B C D) is the list (B C D A). (1 2 3 4) is the identity permutation.

REP2: *pi* is represented as a list of disjoint cycles, where a cycle is a a list of length at most n of numbers between 1 and n (with no two elements the same). The permutation represented

by such a list of cycles is the permutation that results from applying each of the cycles. (The order of application doesn't matter since the cycles are disjoint.) The result of applying a cycle (j1 j2 ... jk) to a list u of lenght n is the list u' where the element in position jm in u' is the element in position $jm-1$ in u for $1 \le m \le k$ (taking $j0$ to be jk). Elements in positions not mentioned in the cycle are unchanged. Thus the cycle (1 4 2) applied to (A B C D) gives (B D C A). The empty list of cycles is the identity permutation in this representation.

29. *isperm*1[*pi*, *n*] (*isperm*2[*pi*, *n*]) is true just if the list *pi* represents a permutation on *n* objects according to REP1 (REP2).

30. *sameperm*2[*pi*1, *pi*2] is true if and only if *pi*1 and *pi*2 represent the same permutation. (Note that the representation by REP2 is not unique while in REP1 it is.)

31. *rho*12[*pi*] maps a list *pi* representing a permutation according to REP1 to a list representing the same permutation according to REP2. *rho*21[*u*] is the inverse map.

32. *compose*1[*pi*1, *pi*2] represents the permutation that results if *pi*2 is applied followed by *pi*1, all represented according to REP1. *compose*2[*u*, *v*] does the same but using REP2.

33. *invert*1[*pi*] (*invert*2[*pi*]) is the inverse to the permutation *pi*, in REP1 (REP2). Thus *compose*1[*invert*1[*pi*], *pi*] is the identitiy permutation.

Note that one way to solve the above problems would be to write the programs for one representation and use the transformations *rho*12 and *rho*21 to obtain those for the other representation. However part of the purpose of the exercise is to see how the different representations behave and this solution would defeat that purpose.

Trees

Consider a tree structure represented as a list in the following way. A leaf is any list of the form (LEAF e) where e is any S-expression. A tree is either a leaf, or a list of trees.

34. *leaves* t is a list of all the leaves in the tree *t*, in the same order as they appear in the tree. (Compare to *fringe* for S-expressions.)

35. Let *leafval* be an evaluation function on leaves. (Assume *leafval returns* integers.) *bestleaf t* is a leaf of *t* having a maximal value according *leafval*. That is no other leaf of *t* has a better value.

36. *bestleaf*1 *t* is a leaf of *t* of maximal value and among those of the same value, *bestleaf*1 *t* has least depth (is closest to the root).

graphs

Let *g* be a directed graph represented as a list of lists as described earlier in this chapter.

37. *isnext*[*u*, *v*, *g*] is true if and only if *g* contains an edge from *u* to *v*.

38. *successors*[*u*, *g*] is the list of vertices, *v*, in *g* such that *isnext*[*u*, *v*, *g*].

39. *predecessors*[*u*, *g*] is the list of vertices *v*, in *g* such that *isnext*[*v*, *u*, *g*].

40. *undir*[*g*] is true if and only if *g* is undirected. That is, if for every edge from *u* to *v* in *g* there is also and edge from *v* to *u*.

41. *mkundir*[*g*] is the graph *g*1 with the same vertices as *g*, and such that there is an edge from *u* to *v* in *g*1 if and only if *g* has either an edge from *v* to *u* or and edge from *u* to *v*.

42. If *g* is undirected then *delete_vertex*[*v*, *g*] returns a graph *g*1 with vertices those of *g* omitting *v*, and edges the same as *g* omitting those connecting *v* to another vertex.

43. If *g* is undirected then *complement*[*g*] returns a graph *g*1 with vertices the same as *g*, but vertices *v* and *w* are joined by an edge in *g*1 if and only if they are not joined by an edge in *g*.

44. For any graph *g*, *reachable*[*u*, *v*, *g*] is true if and only if there is a sequence of vertices *u*1, *u*2, ... ,*un* in *g* with *u* = *u*1, *v* = *un* and *isnext*[*ui*, *ui*+1, *g*] for 1≤*i*≤*n*−1.

45. For any graph *g*, *conn*[*g*] is true if and only if the directed graph *g* is connected in the sense that every vertex is reachable from every other vertex.

NB: Graphs in general have cycles, so when you are recurring through a graph you need to keep track of where you have been in order to avoid a forever looping program.

Chapter III

PROVING FACTS ABOUT LISP PROGRAMS

In this chapter we begin the "Proving" part of the book. In particular we will explain a method for proving *extensional* properties of a restricted but powerful class of LISP programs which we call *clean*, *pure* LISP programs. We will give a number of simple examples to illustrate the method. The following chapter is devoted to a few more complex examples. We will see in later chapters how the proof techniques can be extended to cover a larger class of properties and a larger class of programs.

While the ultimate aim of proving programs is to replace debugging by computer-checked proofs of program correctness, we aren't there yet. However, even hand-checked proofs of correctness of small programs increase out understanding of why programs work and improve the clarity and quality of the programs we write.

An *extensional property* is one that depends only on the function computed by the program. The fact that two sort programs compute the same function and that *append* is associative are extensional facts, but that one sort program does n^2 comparisons and another $n \log n$ comparisons or that one program for *reverse* does fewer *cons*es than another are not. These latter facts are examples of *intensional* properties. *Clean* LISP programs have no side effects (our methods require the freedom to replace a subexpression by an equal expression), and equality refers to the S-expressions and not to the list structures. *Pure* LISP involves only recursive function definitions and doesn't use assignment statements. So far we have only discussed how to write clean and pure LISP programs. The constructs to be explained in Chapter V will take us out of this realm.

The basic idea is to represent both programs and the properties we want to prove about them as sentences in first order logic. The methods of first-order logic are well developed and understood. Although we present the proofs somewhat informally, they can easily be fully formalized. After a brief introduction to the idea of formalization, we give an informal exposition of first order logic and describe the theories of S-expressions, lists and natural numbers. Next come techniques for proving properties of programs known in advance to always terminate. After this come the more difficult techniques required when termination cannot be assumed and termination or non-termination must be proved.

We attempt to motivate and intuitively justify the methods of representation of programs, but no formal proof of soundness is given. Much of the material in this chapter is based on [[Cartwright and McCarthy 1979]]. An extensive treatment of older program proving techniques is given in [[Manna 1974]].

1. About Formalizing.

The main reason for developing formal methods is to be able to make ideas and arguments clear and precise. This wins in several ways. For the logicians the development of formal systems and methods of reasoning allowed them to deal with paradoxes which arose in informal

arguments and also to prove many interesting things about their systems. For the computer scientist formalization provides a way of making your notions and reasoning precise enough to be understood and manipulated by a computer. One goal, of course, it to have the computer do as much of the work for you as possible.

In a formal system there are notions of syntax, semantics, and rules for manipulating syntactic entities in a semantics preserving manner. Thus there are expressions formed from some given collection of symbols in a prescribed manner, a means of assigning meaning to these expressions, and a notion of deduction.

For example, a programming system is a formal system. It has a language consisting of the programs, usually specified by some sort of grammar and recognized by a parser.

For reasoning about programs we will use a formal system based on first order logic. Understanding properties of a system based on first order logic can be often be reduced to understanding properties of the underlying logic. This underlying logic is powerful and in general well understood. Such systems are determined by three basic components:

1. a domain with some designated subdomains (called sorts),
2. a language whose function and predicate symbols are interpreted in the domain and
3. a collection of facts that are true of the domain.

Given the basic components, the expressions of the language, interpretation of these expressions in the domain and rules for (syntactically) deducing additional facts are fixed as we will explain below.

Such a system can be mechanized in the following sense. We can write a program that is capable of understanding a description of it, build a data structure representing the various components, check that statements are well-formed expressions of the language, and that an alleged proof is indeed a valid deduction from the known facts. Given such a program, we can describe to it our system for proving facts about LISP programs and it will be able to check that our proofs are correct. Such a program exists at Stanford [[Weyhrauch 1977]]. Other programs exist which are designed to automatically prove facts about LISP programs. For example see [[Boyer and Moore 1978]].

2. The theory of S-expressions.

We begin by simply presenting a formal theory, namely the theory of S-expressions. We present it as formalization and abstraction of the the discussion of S-expressions given in Chapter I. The universe or domain in question contains S-expressions and possibly other yet unspecified objects. The goal is to formalize properties of and syntactically characterize some sub-domains such as S-expressions, lists, and natural numbers (viewed as a subdomain of the atomic S-expressions).

We first describe the language, some simple "boolean" facts about the domain of S-expressions and give axioms describing the algebraic properties of basic operations on S-expressions. We then explain a principle of induction on S-expressions and represent this

principle by the schema of S-expression induction. We deduce some additional properties of S-expressions as examples of (semi-)formal proofs based on the formal description.

Our precise notion of formal first order theory will be explained in the next section before we complete our description of the formal system for reasoning about LISP programs.

The elementary theory of *car*, *cdr*, and *cons*.

The domain of S-expressions we will denote by *SEXP* and the subdomains consisting of the atoms and the non-atomic S-expressions we denote by *ATOM* and *PAIR*, respectively. The fact that S-expressions are the disjoint union of atoms and non-atoms is expressed formally by the *domain relations*:

$$SEXP = ATOM \cup PAIR$$
$$ATOM \cap PAIR = \phi$$

As in the external notation for LISP programs, we will use list and/or dot notation for S-expression constants. We mention two particular atoms T and NIL. Formally we specify that they are of sort *ATOM* by the *domain membership facts*:

$$\mathtt{T} \in ATOM$$
$$\mathtt{NIL} \in ATOM$$

In order to make general statements about S-expressions we need variables. We will use X, Y, Z as general variables. They stand for abitrary elements of the entire domain. There will also be variables for each of the sub-domains. x, y, z will stand for (range over) S-expressions, xx, yy, zz will stand for non-atomic S-expressions and a, b, c will stand for atoms. This is expressed formally by the additional domain membership facts:

$$x, y, z \in SEXP$$
$$xx, yy, zz \in PAIR$$
$$a, b, c \in ATOM$$

The basic functions on S-expressions are *car*, *cdr* and *cons* which we will denote by a, d and infix "." (or occasionally by cons) as in the external notation for programs. That *cons* applied to any two S-expressions is a non-atomic S-expression and *car* and *cdr* applied to non-atomic S-expressions produce S-expressions is expressed formally by the *domain mappings*:

$$\mathtt{a}: PAIR \to SEXP$$
$$\mathtt{d}: PAIR \to SEXP$$
$$\mathtt{cons}: SEXP \times SEXP \to PAIR$$

We will treat *SEXP*, *PAIR* and *ATOM* as unary predicates on the domain with the obvious meaning.

Now some facts about *car*, *cdr* and *cons*. Recall that when an S-expression is constructed from two given S-expressions, the result has the first S-expression in the a-part and the second in the d-part. This is stated formally by the axioms:

$$\mathtt{CAR}: \forall x\, y.\mathtt{a}[x \,.\, y] = x$$
$$\mathtt{CDR}: \forall x\, y.\mathtt{d}[x \,.\, y] = y$$

Since x, and y are specified to range over S-expression, the quantifier $\forall x\ y.$ means "for all S-expressions x and for all S-expressions y".

Also given a non-atomic S-expression xx, the S-expression constructed from $a\,xx$ and $d\,xx$ is equal to xx (although probably not eq to xx). Thus

$$\text{CONS:}\quad \forall xx.[a\,xx\ .\ d\,xx] = xx$$

Since xx is specified to range over non-atomic S-expressions, the quantifier $\forall xx.$ means "for all non-atomic S-expressions xx" or "for all pairs xx". The axiom CONS also expresses the fact that two S-expressions are equal if and only if their a-parts are equal and their d-parts are equal. That is:

$$\text{EQPAIR:}\quad \forall xx\ yy.[[a\,xx = a\,yy] \wedge [d\,xx = d\,yy] \equiv xx = yy]$$

We can even give a formal proof that EQPAIR is is true in the theory developed so far. To prove that a statement holds for all xx, yy it is enough to prove that the unquantified statement holds without making any special assumptions about xx or yy. Thus we want to prove

$$[a\,xx = a\,yy] \wedge [d\,xx = d\,yy] \equiv xx = yy$$

To prove that a formula of the form $A \equiv B$ is true it is sufficient to prove that $A \supset B$ and $B \supset A$ are both true since equivalence is really an abbreviation of the conjunction of the two implications. So we must prove

$$[a\,xx = a\,yy] \wedge [d\,xx = d\,yy] \supset xx = yy$$

and

$$xx = yy \supset [a\,xx = a\,yy] \wedge [d\,xx = d\,yy]$$

To prove $A \supset B$ we assume A and show that B follows. For the first implication, assume

$$[a\,xx = a\,yy] \wedge [d\,xx = d\,yy].$$

Then by substitition of equals for equals we have

$$[a\,xx\ .\ d\,xx] = [a\,yy\ .\ d\,yy]$$

and using CONS twice we obtain the desired conclusion

$$xx=yy.$$

For the second implication, assume

$$xx=yy.$$

Then

$$[a\,xx = a\,yy] \wedge [d\,xx = d\,yy]$$

follows by subsititution. This completes the proof.

The above proof was carried out in a "natural deduction" style and could be completely formalized (for example so as to be accepted by a proof checker for natural deduction style proofs) without much additional detail.

[Remark: A first order theory deduced by formal proof from given axioms will hold in any domain where the given axioms hold. Thus we are really proving things about a class of domains, not just one.]

The principle of S-expression induction.

With the axioms and domain facts given so far we can use the rules of logic to prove many addiditonal facts about particular S-expressions and a few general facts, but there are many facts about S-expressions that can not be proved. For example in the domain of S-expressions we have

NO-RPLACS: $\forall x\ y\colon x \neq [x\ .\ y]$

but this is not provable in the theory developed so far. Indeed there are domains where the facts given so far are true, but $\exists x\ y\colon x = [x\ .\ y]$ is also true. (The significance of the label NO-RPLACS will become clear in Chapter V where we discuss operations that create circular list structures.) That says that the algebraic and domain facts are insufficient to characterize S-expressions. Indeed Goedel's famous theorem extends to tell us that no theory in which all proofs can be checked admits proofs of all true statements about S-expressions. However, all practically interesting sentences involving pure LISP programs are provable in the above theory supplemented by a suitable principle of mathematical induction.

Recall the "inductive" definition of S-expression: an S-expression is either an atom or it is the result of applying *cons* to the *car* and the *cdr*. Further more each S-expression is the result of a finite number of *cons*es begining with a finite number of atoms. Thus if some property is true of all atoms and whenever it is true of x and y it is true of $[x\ .\ y]$ then it will be true of all S-expressions. This is expressed formally by the S-expression induction schema:

SEXPINDUCTION: $\forall a\colon \Phi\ a \wedge \forall xx\colon [\Phi\ a\,xx \wedge \Phi\ d\,xx \supset \Phi\ xx] \supset \forall x\colon \Phi\ x\ .$

Here Φ is a predicate parameter. The schema is used by subsituting a unary predicate expression (a formula with a designated variable which is to be replaced by the argument to Φ) for Φ. Thus the schema stands for an infinite collection of axioms, one for each formula and designated (free) variable that formula.

Using S-expression induction we can prove NO-RPLACS: $\forall x\ y\colon x \neq [x\ .\ y]$. Instantiating SEXPINDUCTION with $\Phi\ x \equiv \forall y\colon x \neq [x\ .\ y]$ we obtain the particular axiom

$$\forall a\ y\colon a \neq [a\ .\ y] \wedge \forall xx\colon [\forall y\colon a\,xx \neq [a\,xx\ .\ y] \wedge \forall y\colon d\,xx \neq [d\,xx\ .\ y] \supset \forall y\colon xx \neq [xx\ .\ y]] \supset \forall x\ y\colon x \neq [x\ .\ y].$$

Thus to prove $\forall x\ y\colon x \neq [x\ .\ y]$ it is sufficient to prove the two clauses in the hypothesis. The clause for the ATOM case is $\forall a\ y\colon a \neq [a\ .\ y]$. We prove it by contradiction. That is we assume the negation, $a = [a\ .\ y]$, holds for some atom a and S-expression y. Then we have by the domain relations, function mappings and the fact that a ranges over atoms the following:

$$ATOM\ a \wedge PAIR\ [a\ .\ y] \wedge ATOM \cap PAIR = \phi.$$

Clearly a contradiction.

The clause for the non-ATOM case is

$$\forall xx: [\forall y: a\,xx \neq [a\,xx\ .\ y] \wedge \forall y: d\,xx \neq [d\,xx\ .\ y] \supset \forall y: xx \neq [xx\ .\ y]].$$

It is proved by assuming $\forall y: a\,xx \neq [a\,xx\ .\ y] \wedge \forall y: d\,xx \neq [d\,xx\ .\ y]$ (the induction hypothesis) an deducing $\forall y: xx \neq [xx\ .\ y]$. The latter is again done by contradiction. Assume for some S-expression y that $xx=[xx\ .\ y]$. Then by substitution and CAR we have $a\,xx=a[xx\ .\ y]=xx$. Substituting again we get $a\,xx=[a\,xx\ .\ y]$ which contradicts the induction hypothesis, completing the proof.

S-expression induction is on example of a class of induction principles known as "structural" induction. Such principles are based directly on the inductive definition of the domain to which they apply. We will later add principles of list and numcerical induction which are also structural induction principles.

S-expression induction could be introduced as a proof rule rather than an axiom schema. To prove that some formula $\Phi\,x$ holds for all S-expressions, x (i.e. to prove $\forall x: \Phi\,x$) it is sufficient to

(i) prove the ATOM case: $\forall a: \Phi\,a$
and
(ii) assume the induction hypotheses: $\Phi\,a\,xx$ and $\Phi\,d\,xx$ and prove from them $\Phi\,xx$.

This is essentially what we did in carrying out the above proof.

3. Summary of notion of formal theory.

Now we will make precise our notion of first order theory of which the theory of S-expressions is an example. As mentioned before, the basic components of such a theory are a domain with some designated subdomains (called sorts) a language whose predicate and function symbols are interpreted in the domain and a collection of facts that are true of the domain. What we need to do is make clear the process used to describe the basic components and to show how this description determines the well-formed expressions of the language, their interpretation in the domain of interest and say what the rules for (syntactically) deducing additional facts.

We assume that the domain of interest, its subdomains, and a collection of functions and predicates on the domain are known (given by some independent means). Our description of a theory will have the following basic outline:

(i) List the sort symbols and specify the subdomain denoted by each.

(ii) Present the domain relations. (See the discussion of facts.)

(iii) List the constants for each sort, and specify the object denoted by each one.

(iv) List the variable ranging over each sort.

(v) List the function symbols, specifying the function denoted.

(vi) Give domain maps for each function symbol.

(vii) List the predicate symbols, giving the predicate denoted by each.
Sort symbols are considered to be unary predicate symbols with the obvious denotation.
"=" is assumed to be a binary predicate symbol with the standard interpretation

(viii) Give the axioms.

The non-logical symbols are then the sort symbols, the constant and variable symbols, the function symbols and the additional predicate symbols. If ξ is a variable of sort D then we will also use $\xi_0, \xi_1, \ldots$ as variables of that sort. Note that each function and predicate symbol has a fixed number of arguments called the arity. (This is determined when the function or predicate denoted by the symbol is specified.)

The above form of description defines the basic theory. This theory will be extended from time to time by adding additional function and predicate symbols and defining axioms.

The Language.

Now we describe how the expressions of the language are built once the non-logical symbols have been given. There are four classes of expressions: terms, formulas, function expressions, and predicate expressions. Terms denote objects (individuals) in the domain, formulas denote truthvalues, function and predicate expressions denote functions and predicates on the domain. We define the four classes of expressions inductively. Terms are used in making formulas, and formulas occur in terms so that the definitions are *mutually recursive*. Function and predicate expressions are also involved in the mutual recursion.

Terms: Constants are terms, and variables are terms. If ψ is a function expression taking n arguments, and $t_1, \ldots, t_n$ are terms, then $\psi[t_1, \ldots, t_n]$ is a term. If A is a formula and t_1 and t_2 are terms, then $[IF\ A\ THEN\ t_1\ ELSE\ t_2]$ is a term. Some examples of terms in the theory of S-expressions are:

(i) NIL
(ii) x
(iii) ax
(iv) $IF\ ATOM\ x\ THEN\ x\ ELSE\ dx$

Function expressions: A function symbol is a function expression. If $\xi_1, \ldots, \xi_n$ are general variables and t is a term, then $[\lambda\xi_1, \ldots, \xi_n; t]$ is a function expression.

Predicate expressions: A predicate symbol is a predicate expression. If $\xi_1, \ldots, \xi_n$ are general variables and A is a formula, then $[\lambda\xi_1, \ldots, \xi_n; A]$ is a predicate expression.

Some examples of function and predicate expressions are:

(v) $\mathbf{a}$
(vi) $SEXP$
(vii) λX: $IF\ ATOM\ X\ THEN\ X\ ELSE\ \mathbf{d}X$

Formulas: If ϕ is a predicate expression taking n arguments and $t_1, \ldots, t_n$ are terms, then $\phi[t_1, \ldots, t_n]$ is a formula. In particular, if t_1 and t_2 are terms then $t_1 = t_2$ is a formula. If A, B, C are formulas, then $\neg A$, $A \wedge B$, $A \vee B$, $A \supset B$, $A \equiv B$, and $IF\ A\ THEN\ B\ ELSE\ C$ are formulas. If $\xi_1, \ldots, \xi_n$ are variables, and A is a formula, then $[\exists \xi_1 \ldots \xi_n: A]$ and $[\forall \xi_1 \ldots \xi_n: A]$ are formulas. Some examples of formulas in the theory of S-expressions are:

(viii) $\mathbf{a}x = \mathbf{d}x$
(ix) $ATOM\ X$
(x) $IF\ \neg ATOM\ x\ THEN\ \mathbf{a}x = \mathbf{d}x\ ELSE\ T = T$
(xi) $\forall x$: $[PAIR\ x \supset \exists y\ z: x = y\ .\ z]$

An occurrence of a variable ξ is classified as *bound* or *free* according to the following rule. ξ is bound in an expression of one of the forms $[\lambda \xi_1 \ldots \xi_n: t]$, $[\lambda \xi_1 \ldots \xi_n: A]$, $[\forall \xi_1 \ldots \xi_n: A]$ or $[\exists \xi_1 \ldots \xi_n: A]$ if it is one of the numbered ξ's. An occurrence of ξ in an expression e is bound if that occurence is bound in some subexpression of e. Otherwise it is free. A formula having no free variables is called a *sentence*. Thus X is bound in example (vii) but not in (ix). The formula of example (xi) is a sentence.

We note that conditional expressions and λ-expressions are not ordinarily included in a first order language. These extensions do not change the logical strength of the theory, because, as we shall see, every formula that includes conditional expressions or λ-expressions can be transformed into an equivalent formula without them. However, the extensions are practically important, because they permit us to use recursive definitions directly as formulas of the logic.

Assigning meaning to expressions.

Next we explain how to determine the meaning of an expression in the domain of interest (or any domain where the non-logical symbols have been assigned an appropriate meaning). The meaning of expressions containing free variables can not be determined, in general, without assigning a value to each of the variables. We say that an assignment is "allowed" if the variable is general or if it ranges over a subdomain D and the value assigned is in that subdomain. So, we will imagine a fixed but arbitrary interpretation assigning allowed values to each variable of the language. Meanings are then relative to such and interpretation. Meanings are determined inductively in a manner parallel to the construction of expressions.

Terms: The meaning of a constant is the object that it denotes. The meaning of a variable is the value assigned to it by the interpretation. If ψ is a function expression taking n arguments, and $t_1, \ldots, t_n$ are terms, then the meaning of $\psi[t_1, \ldots, t_n]$ is the value of the function assigned to ψ when applied to the values assigned to the terms t_i. *If* A is a formula and t_1 and t_2 are terms, if A is true then the value assigned to $IF\ A\ THEN\ t_1\ ELSE\ t_2$ is the value assigned to t_1, otherwise it is assigned the value assigned to t_2.

In the theory of S-expressions, if the value assigned to x is (A . B) then the meanings of the terms given in examples (i)–(iv) above are:

(i) NIL
(ii) (A . B)
(iii) A
(iv) B

Function and predicate expressions: Function and predicate symbols are assigned the function or predicate denoted by the symbol. The expression $[\lambda\xi_1, \ldots, \xi_n; e]$ is assigned a function or predicate according to whether e is a term or formula. The value of $[\lambda\xi_1, \ldots, \xi_n; e][t_1, \ldots, t_n]$ the value assigned to e relative to an interpretation which is modified by assigning the values of the ts to the corresponding ξs.

This is our reason for only allowing λ-binding of general variables. Namely if we allowed binding of sorted variables then some terms would not have well defined values, and this is not in the spirit of first order logic. Another approach would be to allow λ-binding of any variable, but only allow application of such expressions to terms of the appropriate sort. This makes the class of well-formed expressions undecidable which is undesirable if we wish to represent our system is in a computer.

Formulas: If ϕ is a predicate expression taking n arguments and $t_1, \ldots, t_n$ are terms, then $\phi[t_1, \ldots, t_n]$ is true exactly when the tuple of values assigned to the ts satisfies the predicate assigned to ϕ. In particular, if t_1 and t_2 are terms then $t_1 = t_2$ is true exactly when t_1 and t_2 are assigned the same value. If A, B, C are formulas, then $\neg A$, is true exactly when A is false, *IF A THEN B ELSE C* is true if A is true and B is true or if A is false and C is true and false otherwise. The values of $A \wedge B$, $A \vee B$, $A \supset B$, and $A \equiv B$ are determined from the values assigned to A and B according to the following truthtable:

A	B	$\wedge$	$\vee$	$\supset$	$\equiv$
true	true	true	true	true	true
true	false	false	true	false	false
false	true	false	true	true	false
false	false	false	false	true	true

Table 4. The semantics of logical connectives.

If $\xi_1, \ldots, \xi_n$ are variables, and A is a formula, then $[\forall\xi_1 \ldots \xi_n; A]$ is true exactly if A is true for all allowed interpretations of the variables differing from the fixed one only in assignments made to the variables ξ_r. $[\exists\xi_1 \ldots \xi_n; A]$ is true exactly if there is some allowed interpretation of the variables differing from the fixed one only in assignments made to the variables ξ_r.

In the theory of S-expressions, if the value assigned to x is (A . B) then the meanings of the formulas given in examples (viii)–(x) above are:

(viii) false
(ix) false
(x) false

while if the value assigned to x is (A . A) then the meanings of the formulas given in examples (viii)–(x) above are:

(viii) true
(ix) false
(x) true

The formula of example (xi) is true for any interpration of the variables. Being a sentence, its truth or falsity doesn't depend on the values assigned to the variables.

Those who are familiar with the lambda calculus should note that λ is being used here in a very limited way. Namely, the variables in a lambda-expression take only elements of the domain as values, whereas the essence of the lambda calculus is that they take arbitrary functions as values. We may call these restricted lambda expressions *first-order lambdas*.

Facts

Facts come in two flavors: domain facts and axioms. Axioms are simply sentences of the language that we are asserting to be true in the domain. There are three way of expression domain facts: as domain relations, as domain membership specifications and as domain mappings for functions. The use of domain expressions is a matter using a more natural notation. As we will see each expression of a domain fact has an equivalent axiom.

Domain terms are formed from sort symbols and finite subdomain descriptions of the form $\{\xi_1, \ldots \xi_n\}$ using the binary boolean operations $\cup$ and $\cap$ meaning union and intersection. Domain relations are formed from domain terms using the binary relations symbols $=$ and $\subset$ meaning equality and containment. The axioms corresponding to the domain relations for the theory of S-expressions are:

$$SEXP = ATOM \cup PAIR \quad \leftrightarrow \quad \forall X: [SEXP\ X \equiv ATOM\ X \vee PAIR\ X]$$
$$ATOM \cap PAIR = \phi \quad \leftrightarrow \quad \forall X: \neg[ATOM\ X \wedge PAIR\ X]$$

Domain membership facts have the form:

$$\xi \in D \text{ or } \xi \in D$$

where ξ denotes a constant symbol, ξ denotes a variable symbol and D denotes a sort symbol. The corresponding axioms are:

$$D\ \xi \text{ or } \forall \xi\colon D\ \xi.$$

Function mappings have the general form:

$$F: D_1 \times \ldots \times D_n \rightarrow D_0$$

which corresponds to the axiom:

$$\forall X_1 \ldots X_n{:}[D_1\ X_1 \wedge \ldots \wedge D_n\ X_n] \supset D_0\ F[X_1 \ldots X_n]] \quad .$$

where the D_i denote sort symbols and F denotes a function symbol taking n arguments.

Rules

There are many possible formal proof systems (collections of rules for deducing facts). The natural deduction system mentioned in § 2 is one possibility. Basically any rule that only allows deduction from given facts of facts that are true in any domain where all of the given facts are true is a valid rule. We will not specify any particular proof rules. However, the proofs presented will be, for the most part, informal versions of "natural" deduction proofs augmented by extended substitution rules and tautologies. In fact most of the proofs have been carried out formally in such a system and confirmed by a proof-checker. Since the use of sorted variables, conditional expressions and first order lambdas are not standard, we give some rules below for treating them.

Manipulating quantifiers over sorted variables.

In order to shorten formulas, we use sorted variables. Thus we can write $\forall xx{:}[a\,xx\ .\ d\,xx = xx)$, taking advantage of the fact that the variable xx ranges only over non-atomic S-expressions, instead of having to write $\forall x.[\neg ATOM\ x \supset a\,x\ .\ d\,x = x]$. The proofs are also shortened, since we can avoid having to prove $\neg ATOM\ xx$ every time the fact is needed.

The usual rules given for manipulating quantifiers (generalization, specializtion, etc.) apply to variables that range over the entire domain. In order to handle variables restricted to range over some subset we must either modify the rules or show how to eliminate the restricted variables from a formula. Since we have given no rules to modify, we will indicate how quantifiers over sorted variables can be eliminated. Given a particular set of rules, it would be better to extend them to handle sorted variables using the elimination rules as a guide. Otherwise the purpose of introducing them is defeated.

Suppose the variable ξ is restricted to range over the subdomain D and the variable ξ_1 is a general variable. For example, in the theory of S-expressions we have the subdomain $SEXP$, with the variable $x \in SEXP$ and X is a general variable. The formula $\forall\xi{:}A$ is then equivalent to $\forall\xi_1{:}[D\ \xi_1 \supset A]$ and the formual $\exists\xi{:}A$ is equivalent to $\exists\xi_1.[D\ \xi_1 \wedge A]$.

Rules for Conditional Expressions.

All the properties we shall use of conditional terms follow from the relation

3.1) $$[A \supset [IF\ A\ THEN\ t_1\ ELSE\ t_2] = t_1] \wedge [\neg A \supset [IF\ A\ THEN\ t_1\ ELSE\ t_2] = t_2]\ .$$

It is worthwhile to list separately some properties of conditional terms. First we have the obvious

3.2) $$[IF\ \text{true}\ THEN\ t_1\ ELSE\ t_2] = t_1$$
$$[IF\ \text{false}\ THEN\ t_1\ ELSE\ t_2] = t_2.$$

Next we have a *distributive law* for functions applied to conditional terms, namely

3.3) $$f[IF\ A\ THEN\ t_1\ ELSE\ t_2] = IF\ A\ THEN\ f[t_1]\ ELSE\ f[t_2]\ .$$

This applies to predicates as well as functions and can also be used when one of the arguments of a function of several arguments is a conditional term. It also applies when one of the terms of a conditional term is itself a conditional term.

Thus

3.4) $$IF\ [IF\ A\ THEN\ B\ ELSE\ C]\ THEN\ t_1\ ELSE\ t_2 =$$
$$IF\ A\ THEN\ [IF\ B\ THEN\ t_1\ ELSE\ t_2]\ ELSE\ [IF\ C\ THEN\ t_1\ ELSE\ t_2]$$

and

3.5) $$IF\ A\ THEN\ [IF\ B\ THEN\ t_1\ ELSE\ t_2]\ ELSE\ t_3 =$$
$$IF\ B\ THEN\ [IF\ A\ THEN\ t_1\ ELSE\ t_3]\ ELSE\ [IF\ A\ THEN\ t_2\ ELSE\ t_3]\ .$$

When the expressions following THEN and ELSE are formulas, then the conditional expression can be replaced by a formula according to

3.6) $$[IF\ A\ THEN\ B\ ELSE\ C] = [A \wedge B] \vee [\neg A \wedge C]\ .$$

These rules permit eliminating conditional expressions from formulas by first using distributivity to move the conditionals to the outside of any functions or predicates and then replacing the conditional expression by a Boolean expression. They could be added to our formal proof system either as rules or as axiom schemata.

More facts about conditional terms are given in [McCarthy 1963] including a discussion of canonical forms that parallels the canonical forms of boolean terms. Any question of equivalence of conditional terms is decidable by truth tables analogously to the decidability of propositional sentences.

Lambda-expressions.

The only rule required for handling lambda-expressions in first order logic is called *lambda–conversion*. Essentially it is

$$[\lambda\xi\colon e][t] = \text{< the result of substituting } t \text{ for } \xi \text{ in } e \text{ >}.$$

As examples of this rule, we have

$$[\lambda x\colon a\,x\ .\ y][u\ .\ v] = [a[u\ .\ v]]\ .\ y\ .$$

However, a complication requires modifying the rule. Namely, we can't substitute for a variable ξ an expression e that has a free variable ξ_1 into a context in which ξ_1 is bound. Thus it would be wrong to substitute $\xi_1 + \xi_2$ for ξ in $\forall\xi_1\colon [\xi + \xi_1 = \xi_2]$ or into the term $[\lambda\xi_1\colon \xi + \xi_1][u + v]$. Before doing the substitution, the variable ξ_1 would have to be replaced in all its bound occurrences by a new variable.

Lambda–expressions can always be eliminated from sentences and terms by lambda–conversion, but the expression may increase greatly in length if a lengthy term replaces a variable that occurs more than once in e. It is easy to make an expression of length proportional to n

whose length is proportional to 2^n after conversion of its n nested lambda-expressions. For example $\lambda\xi_1: [\xi_1.\xi_1]\mathbf{I} \ldots [\lambda\xi_n: [\xi_n.\xi_n]\mathbf{I}A]] \ldots]$

becomes

```
(A . A),

((A . A) . (A . A)),
```

or

```
((((A . A) . (A . A)) . ((A . A) . (A . A)))
  . ((A . A) . (A . A)) . ((A . A) . (A . A))))
```

for n = 1, 2, or 4 respectively.

The use of λ-expressions in writing programs also may improve efficiency by specifying some the expression be evaluated once and the value used in several places. Where there are side-effects, of course more than efficiency is effected.

4. Lists, natural numbers and LISP program primitives.

We now extend the theory of S-expressions in order to talk about the subdomain of lists and natural numbers. We also define functions corresponding to the basic LISP programs other that *car*, *cdr* and *cons* that were described in Chapter I.

The Theory of lists.

The domain of lists is a subdomain of the domain of S-expressions which we denote by *LIST*. It is the union of the one element domain *NULL* and the domain of non-empty lists *NELIST*. These domain facts are given by

$$LIST \subset SEXP$$
$$NELIST \subset PAIR$$
$$NULL = \{\mathtt{NIL}\}$$
$$LIST = NULL \cup LIST$$
$$NULL \cap NELIST = \phi$$

The variables u, v, w will range over lists and uu, vv, ww will range over non-empty lists.

$$u, v, w \in LIST$$
$$uu, vv, ww \in NELIST$$

The behaviour of *car*, *cdr* and *cons* on the domain *LIST* is given by the function mappings:

$$\mathbf{a}: NELIST \to SEXP$$

d: $NELIST \to LIST$
cons: $SEXP \times LIST \to NELIST$

Since lists are a subdomain of S-expressions and non-empty lists a subdomain of non-atomic S-expressions the axioms CAR, CDR and CONS apply to lists also. For lists we have the following structural induction principle.

LISTINDUCTION: $\Phi\ \mathrm{NIL} \wedge \forall uu: [\Phi\ d\,uu \supset \Phi\ uu] \supset \forall u: \Phi\ u$.

Natural numbers.

We will treat the domain of natural numbers as a subdomain of the atoms. This reflects the way LISP systems treat numbers and makes it possible to discuss mixed symbolic and numeric expressions conveniently. The natural numbers will be denoted by *NATNUM* and is the union of the one element domain *ZERO* and the domain of positive numbers *POSP*.

$NATNUM \subset ATOM$
$ZERO = \{0\}$
$NATNUM = ZERO \cup POSP$
$ZERO \cap POSP = \phi$

We will use the ordinary numerals 0, 1, 2, etc. to denote numbers. The variables l, n, m will range over numbers and ll, mm, nn will range over positive numbers.

$l, n, m \in NATNUM$
$ll, mm, nn \in POSP$

The basic functions on numbers are *successor* and *predecessor* which we will denote by *add*1 and *sub*1 corresponding to names of the LISP programs that compute these functions. *add*1 maps numbers to wo positive numbers and *sub*1 maps positive numbers to numbers.

*add*1: $NATNUM \to POSP$
*sub*1: $POSP \to NATNUM$

SUB1: $\forall n$: *sub*1 *add*1 $n = n$
ADD1: $\forall nn$: *add*1 *sub*1 $nn = nn$

Our form of the induction principle for natural numbers is:

NUMINDUCTION: $\Phi\ 0 \wedge \forall nn: [\Phi\ sub1\ nn \supset \Phi\ nn] \supset \forall n: \Phi\ n$.

This ends the description of the basic theory of S-expressions, lists and numbers. Our theory is a dynamic one, however, and we will introduce new function and predicate symbols with defining axioms as we proceed.

[Remark: there are domains other than S-expressions where the facts we have specified hold. For example we have not specified the value of *car* and *cdr* on atoms, and there are many consistent possiblilities. The additional facts we can prove using any valid proof rules will also be true in a such domain.]

Axioms for basic LISP programs.

As advertised, we are going to extend the basic theory by defining new functions and predicates. In order to represent LISP programs as functions in a first order theory, we first need to represent the basic programs. We have already introduced *car*, *cdr* and *cons*. For the rest we add the ternary function symbol if, to represent the conditional. We will generally write

$$\text{if } t_0 \text{ then } t_1 \text{ else } t_2 \text{ instead of } \text{if}[t_0, t_1, t_2]$$

as in the external notation for programs. We also add the function symbols equal, at, n, not, and, or corresponding to external notation. The axioms characterizing these functions are:

IF: $\forall x\, y\, z$: [if x then y else z] = *IF* $x \neq$ NIL *THEN* y *ELSE* z
EQUAL: $\forall x\, y$: [x equal y] = *IF* [$x = y$] *THEN* T *ELSE* NIL
ATOM: $\forall x$: at x = *IF ATOM* x *THEN* T *ELSE* NIL
NULL: $\forall x$: n x = *IF NULL* x *THEN* T *ELSE* NIL
NOT: $\forall x$: not x = if x then NIL else T
AND: $\forall x\, y$: [x *and* y] = if x then y else NIL
OR: $\forall x\, y$: [x *or* y] = if x then x else y

For purposes of talking about LISP programs we define the subclass of terms in our language known as LISP term. Variables and S-expression constants are LISP terms. If f is one of the basic LISP functions a, d, cons, equal, at, n, not, and, or or, or the name of a function representing a LISP program, or a "new" function symbol and the arity of f is n, and $t_0, t_1, \ldots, t_n$ are LISP terms then $f[t_1, \ldots, t_n]$ and $[\lambda x_1 \ldots x_n; t_0][t_1, \ldots, t_n]$ are LISP terms. And those are all the LISP terms. Notice that these correspond to the terms of LISP program definitions omitting the label construct and programs as arguments.

The restriction to S-expression constants may seem vacuous now, as those are all we have mentioned so far, but but there will be more.

5. Representing programs known to terminate.

Now we are ready to explain how (some) LISP programs can be represented. First we consider only programs known to terminate. There are several syntactic criteria defining classes of such programs. They are said to terminate on form. We will consider three forms: list recursion, S-expression recursion and numeric recursion. They include a large number of useful programs and the forms are easy to recognize. We give some simple examples to illustrate the basic style and method of proof. Later we will show how to represent programs not known to terminate, and how to prove termination in some cases. We will be able to prove termination of the classes of programs discussed in this section, so this will turn out to be a special case of the more general method. We start with the special case as it is easier to explain and allows us to get to the proving part with less fuss.

Consider a LISP program given by a definition of the form

5.1) $$f[x] \leftarrow \tau[f, x]$$

where τ is a LISP term involving the variable x, the program name f and the basic LISP programs S-expression constants and perhaps names of previously defined programs. (See §4 for more complete definition of LISP term.) This program partially defines a function also called f, on S-expressions such that whenever x is such that the program terminates with result y then $f\,x=y$. Using the rules for computation given in Chapter I we can see that for x such that the computation of $f[x]$ terminates we have $f[x] = \tau[f, x]$. Due to the double use of symbols as external notation for programs and to denote functions in our theory, and the manner in which the basic LISP function were axiomatized, this equation holds whether it is viewed as an equation between program terms or as an equation in the logic (for those x such that the computation halts). Thus, given a domain on which the program f is known to terminate there is a function f completely characterized on that domain by the functional equation corresponding to (5.1). We can add a new function symbol to the language naming this function and the axiom obtain from (5.1) by replacing "$\leftarrow$" by "$=$", replacing the function parameter by the new function symbol and universally quantifying over x. Clearly this reasoning applies to programs with more than one input parameter.

In Chapter II we discussed various forms of recursive definitions. Some of these forms we claimed had the property that any program having that form terminates for all allowed input. One such form was list recursion:

5.2) $$f[u, \xi] \leftarrow \text{if } n\,u \text{ then } g[\xi] \text{ else } h[a\,u,\ d\,u,\ \xi,\ f[d\,u,\ j[a\,u,\ d\,u,\ \xi]]]$$

The symbols g, h and j occuring in these definitions must denote programs definable in terms of previously defined programs (or basic programs), must be known to terminate on appropriate domains with appropriate ranges. In particular if g terminates on input in domain D_0 returning results in D_1, j terminates on input in $SEXP \times LIST \times D_0$ returning results in D_0 and h terminates on input in $SEXP \times LIST \times D_0 \times D_1 \cup D_2$ returning results in D_2 then the program f will terminate returning results in D_1 on argurnents in $NULL \times D_0$ and in D_2 for arguments in $NELIST \times D_0$.

[Remark: For simplicity we have included just a single parameter ξ in each of the above forms. In general we allow any number of parameters (including none) to appear, and they may be restricted to whatever domains we know about.]

For such a program, if the programs g, h, j are represented by functions gg, hh, jj then we will have the domain mappings:

$$gg\colon D_0 \to D_1$$
$$jj\colon SEXP \times LIST \times D_0 \to D_0$$
$$hh\colon SEXP \times LIST \times D_0 \times D \to D_2$$

and if ff is the function representing f we will add the domain mappings

$$ff\colon LIST \times D_0 \to D_1 \cup D_2$$
$$ff\colon NULL \times D_0 \to D_1$$
$$ff\colon NELIST \times D_0 \to D_2$$

and the axiom characterizing ff will be:

$$\forall u\ \xi\colon ff[u, \xi] = \text{if } n\,u \text{ then } gg[\xi] \text{ else } hh[a\,u, d\,u, \xi,\ ff[d\,u, jj[a\,u, d\,u, \xi]]]$$

where ξ ranges over D_0.

The progam for *append* has the form of a list recursion.

APPEND: $u*v \leftarrow$ **if** n u **then** v **else** a u . [d u * v].

Namely, ξ is v, D_0, D_1, D_2 are *LIST*, g is the identity, j returns its third argument and $h[x, w, v, u]$ is x . u. which has range *NELIST*. Note that if v is replace by vv then all of the component programs return values in *NELIST*. We can thus introduce the function also denoted by infix * by giving the domain mappings and equational axiom as follows:

$$*: LIST \times LIST \rightarrow LIST$$
$$*: NULL \times LIST \rightarrow LIST$$
$$*: NELIST \times LIST \rightarrow NELIST$$
$$*: LIST \times NELIST \rightarrow NELIST$$

and the axiom characterizing *ff* will be:

APPEND-DEF: $\forall u\ v$: $u*v$ = **if** n u **then** v **else** a u . [d u * v].

Two immediate consequences of the defining axiom are:

APPEND-NIL: $\forall v$: NIL$*v = v$,
APPEND-UU: $\forall uu\ v$: $uu*v$ = a uu . [d uu $*v$],

To prove APPEND-NIL we substitute NIL for u in APPEND-DEF and use the axioms NULL and IF. Thus

NIL$*v$=**if** n NIL **then** v **else** a NIL . [d NIL * v] ; APPEND-DEF
=**if** T **then** v **else** a NIL . [d NIL * v] ; NULL
= v ; IF

The proof of APPEND-UU is similar. Here we need to know that $\forall uu$: $\neg NULL\ uu$ which is a consequence of the domain facts $uu \in NELIST$ and $NULL \cap NELIST = \phi$.

$uu*v$=**if** n uu **then** v **else** a uu . [d uu * v] ; APPEND-DEF
=**if** NIL **then** v **else** a uu . [d uu * v] ; NULL and domain facts
= a uu . [d uu * v] ; IF

In what follows we will not continue to give proofs of such elementary facts, but just state them as immediate corollaries of the defining axiom.

The main reason for restricting the second argument of *append* to *LIST* is that the intent is to stick two lists together to get a new one. Also, when so restricted the function has nice mathematical properties.

RID-APPEND: $\forall u$: $u*$NIL = u,
ASSOC-APPEND: $\forall u\ v\ w$: $u*[v*w] = [u*v]*w$.

APPEND-NIL and RID-APPEND express the fact is that NIL is the "zero" of the append operation. The standard mathematical terminology is to say that NIL is both a left identity and a right identity for *append*. ASSOC-APP expresses the fact that *append* is associative - like addition and multiplication of numbers. [Unlike addition and multiplication of numbers, *append* is not commutative, since (A B)*(C D) = (A B C D) ≠ (C D)*(A B)]. Associativity allows us to omit brackets and write $u*v*w$, because it doesn't matter how the expression is parenthesized.

The proofs of APPEND NIL, and APPEND UU were untypically easy. As we remarked in § 2 not many interesting general properties about S-expressions or lists can be proved just by substituting in function definitions and using known properties of S-expressions and the elementary LISP functions. In fact, when we want to prove something about a function defined by recursion, we almost always have to use some corresponding induction principle to carry out the proof. Proving that NIL is a right identity (RID-APPEND) is an example. The induction principle corresponding to list recursion is list induction. Recall that list recursion divides the computation into two cases. In the first case the recursion argument is NIL and the result is computed directly from the parameters. In the second case the recursion argument is a non-empty list uu and the result can be computed directly once it is known for duu Similarly to prove a property $\Phi[u]$ using list induction we consider two cases. In the first case u is NIL and we must prove Φ[NIL] using facts and definitions already known. In the second case u an a non-empty list uu and we prove $\Phi[uu]$ using in addition to known facts, the induction hypothesis $\Phi[duu]$. This analysis is expressed in the schema LISTINDUCTION given in § 4. It can also be viewed as a proof rule.

Let's see how this works to prove $\Phi\, u = u*\text{NIL}=u$. In the NIL case we must prove NIL*NIL=NIL, which is an instance of APPEND-NIL. In the non-NIL case we have the induction hypothesis $duu*\text{NIL} = duu$, and

$$\begin{aligned} uu*\text{NIL} &= auu\ .\ [duu * \text{NIL}] && ;\ \text{APPEND-UU} \\ &= auu\ .\ duu \\ &= uu && ;\ \text{CONS} \end{aligned}$$

which is the desired proof of $\Phi[uu]$ assuming $\Phi[duu]$.

A similar (though slightly more complicated) argument will prove the associativity of *append* (ASSOC-APPEND). In particular we want to prove $\Phi\ u = u*[v*w]=[u*v]*w$. In the NIL case we must prove $\text{NIL}*[v*w]=[\text{NIL}*v]*w$, which follows easily using two applications of APPEND-NIL. In the non-NIL case we must prove $uu*[v*w]=[uu*v]*w$. We assume the induction hypothesis: $duu*[v*w]=[duu*v]*w$ We also need the two simple facts

APPEND-CAR: $a[uu * v] = a[auu\ .\ [duu]*v] = auu$
APPEND-CDR: $d[uu * v] = d[auu\ .\ [duu]*v] = [duu]*v$

which are direct consequences of APPEND-UU, domain facts and the S-expression facts CAR and CDR. We have

$$\begin{aligned} uu*[v*w] &= auu\ .\ duu*[v*w] && ;\ \text{APPEND-UU} \\ &= auu\ .\ [[duu*v]*w] && ;\ \text{induction hypothesis} \\ &= a[uu*v]\ .\ [d[uu*v]*w] && ;\ \text{APPEND-CAR,CDR} \\ &= [uu*v]*w] && ;\ \text{APPEND-UU} \end{aligned}$$

completing the proof of ASSOC-APPEND. The algebra of this proof is rather typical. We used the definition of the function involved several times, we used the induction hypothesis once, and we used the elementary algebraic facts about conditional expressions and the basic functions of LISP. Also in this proof it does not matter whether we take $\Phi[u]$ to be $u*[v*w]=[u*v]*w$ or $\forall v\ w{:}u*[v*w]=[u*v]*w$. The difference being that in the first case the induction hypothesis is only refers to the particular v and w of the conclusion while in the second case it applies to any lists v, w. When proving properties of recursively defined functions, the unquantified form of Φ is generally good enough when the function definion does not modify the parameters (non-recursion arguments) in recursive calls. If it passes functions of the parameters in recursive calls then it may be necessary to use the quantified version of Φ.

Exercises.

Recall that *reverse* and *reversel* are defined by

reversel $u \leftarrow$ *if* n u then NIL else [*reversel* d u] * <a u>

reverse $u \leftarrow$ *rev*[u, NIL]

rev[u, v] $\leftarrow$ if n u then v else *rev*[d u, a u . v]

Show that the programs *reversel* and *rev* have the form of list recursion and give the domain mappings and axioms for the corresponding functions.

Prove the following properties of *reverse*

1. $\forall u$: *reverse* u = *reversel* u

2. $\forall u\ v$: *reverse*[u * v] = [*reverse* v] * [*reverse* u]

3. $\forall u$: *reverse reverse* u = u.

The first proof requires inventing a suitable sentence on which to do an induction.

Now we consider two other classes of programs that terminate on form. Namely, those defined by S-expression rescursion and those defined by numerical recursion. (The latter is generally known as primitive recursion.) Recall from Chapter II that S-expression recursion has the form

5.3) $f[x,y] \leftarrow$ if at x then $g[x,y]$ else $h[a x, d x, y, f[a x, ja[a x, d x, y]], f[d x, jd[a x, d x, y]]]$

and numerical recursion has the form

5.4) $f[n,y] \leftarrow$ if n equal 0 then $g[y]$ else $h[n-1, y, f[n-1, j[n-1, y]]]$.

As for list recursion, the symbols g, h, j, ja, jd occuring in these definitions must be denote explicitly definable programs known to terminate on appropriate domains and having suitable ranges. In particular for S-expression recursion, if the programs g, h, ja, jd are represented by functions gg, hh, jja, jjd and we will have the domain mappings:

$$gg\colon SEXP \times D_0 \to D_1$$
$$hh\colon SEXP \times SEXP \times D_0 \times D \times D \to D_2$$
$$jja\colon SEXP \times SEXP \times D_0 \to D_0$$
$$jjd\colon SEXP \times SEXP \times D_0 \to D_0$$

and if ff is the function representing f then we add the domain mappings

$$ff\colon SEXP \times D_0 \to D_1 \cup D_2$$
$$ff\colon ATOM \times D_0 \to D_1$$
$$ff\colon PAIR \times D_0 \to D_2$$

and the axiom characterizing ff will be:

$$\forall x\ \xi\colon ff[x,\xi] = \text{if } at\,x \text{ then } gg[x,\xi] \text{ else } hh[ax, dx, \xi, ff[ax, jj1[ax, dx, \xi]], ff[dx, jj2[ax, dx, \xi]]]$$

where ξ is of sort D_0.

and in the case of numeric recursion, we will have the domain mappings:

$$gg\colon D_0 \to D_1$$
$$hh\colon NATNUM \times D_0 \times D \to D_2$$
$$jj\colon NATNUM \times D_0 \to D_0$$

and if ff is the function representing f we will add the domain mappings

$$ff\colon NATNUM \times D_0 \to D_1 \cup D_2$$
$$ff\colon ZERO \times D_0 \to D_1$$
$$ff\colon POSP \times D_0 \to D_2$$

and the axiom characterizing ff will be:

$$\forall n\ \xi\colon ff[n,\xi] = \text{if } n \text{ equal } 0 \text{ then } gg[\xi] \text{ else } hh[n-1, \xi, ff[n-1, jj[n-1, \xi]]]$$

where ξ is of sort D_0.

[Remark: As with list recursion, we have formulated S-expression and numeric recursion with a single parameter. The extension to several parameters should be clear.]

Example

To illustrate the use of the S-expression recursion principle, we will introduce functions *size*, *fringe* and *length* representing the programs of the same names and prove

$$\forall x\colon length\ fringe\ x = size\ x$$

(We assume at this point that "+" and simple properties of numbers are known and will mainly concentrate on techniques for proving facts about functions on lists and S-expressions.)

Recall the programs for *size*, *fringe* and *length*:

SIZE: *size* $x \leftarrow$ **if** ax **then** 1 **else** *size* ax + *size* dx,
FRINGE: *fringe* $x \leftarrow$ **if** ax **then** <x> **else** *fringe* ax * *fringe* dx,
LENGTH: *length* $u \leftarrow$ **if** nu **then** 0 **else** 1 + *length* du

size and *fringe* are defined by S-expression recursion and *length* by list recursion. The domain mappings and axioms for the corresponding functions are:

SIZE-DEF: $\forall x$: *size* x = **if** ax **then** 1 **else** *size* ax + *size* dx
size: *SEXP* → *POSP*

FRINGE-DEF: $\forall x$: *fringe* x = **if** ax **then** <x> **else** *fringe* ax * *fringe* dx
fringe: *SEXP* → *NELIST*

LENGTH-DEF: $\forall u$: *length* u = **if** nu **then** 0 **else** 1+*length* du
length: *LIST* → *NATNUM*
length: *NULL* → *ZERO*
length: *NELIST* → *POSP*

As immediate corollaries of the definitions we have

SIZE-ATOM: $\forall a$: *size* a = 1
SIZE-PAIR: $\forall xx$: *size* xx = *size* axx + *size* dxx

FRINGE-ATOM: $\forall a$: *fringe* a = <a>
FRINGE-PAIR: $\forall xx$: *fringe* xx = *fringe* axx * *fringe* dxx

LENGTH-NIL: *length* NIL = 0
LENGTH-UU: $\forall uu$: *length* uu = 1 + *length* duu

Since the definitions of *fringe* and *size* are S-expression recursions, S-expression induction is appropriate to carry out the proof. (Recall the rule for S-expression induction given in § 2.) We wish to prove $\forall x$: Φ x where:

$$\Phi[x] \equiv [\textit{length fringe } x = \textit{size } x].$$

In the ATOM case we must prove $\forall a$: *length fringe* a = *size* a. We have

length fringe a = *length* <a>	by FRINGE-ATOM
= 1+*length* d<a>	by LENGTH-UU and domain facts
= 1+*length* NIL	by CDR
= 1	by LENGTH-ZERO and properties of +.
= *size* a	by SIZE-ATOM

which proves the ATOM case. In the non-ATOM case we assume the induction hypothesis:

$$[\textit{length fringe } a\textit{x} = \textit{size } a\textit{x}] \wedge [\textit{length fringe } d\textit{x} = \textit{size } d\textit{x}]$$

and prove *length fringe* xx = *size* xx. We have: ,

length fringe xx = *length*[*fringe* a*xx* ∗ *fringe* d*xx*] by FRINGE-PAIR

and

size xx = *size* a*xx* + *size* d*xx* by SIZE-PAIR
= [*length fringe* a*xx*] + [*length fringe* d*xx*] by induction hypohesis

By the domain facts *fringe* a*xx* and *fringe* d*xx* are of sort *LIST*, so we are reduced to proving something of the form *length*[*u*∗*v*]=*length u* + *length v*. This suggests that we prove a general lemma to that effect and then the proof will be complete. We prove

$$\forall u\ v{:}[\mathit{length}[u{*}v]{=}\mathit{length}\ u + \mathit{length}\ v].$$

Since the definition of *append* is by list recursion we will use list induction for the proof. In the NIL case we must show

$$\forall v{:}[\mathit{length}[\mathrm{NIL}{*}v]{=}\mathit{length}\ \mathrm{NIL} + \mathit{length}\ v].$$

We have

length[NIL∗*v*] = *length v*	by APPEND-NIL
= 0 + *length v*	by properties of 0 and "+".
= *length* NIL + *length v*	by LENGTH-NIL.

proving the NIL case. In the non-NIL case we assume the induction hypothesis

$$\mathit{length}\ [\mathrm{d}\,uu]{*}v = \mathit{length}\ \mathrm{d}\,uu + \mathit{length}\ v$$

and prove *length*[*uu*∗*v*]=*length uu*+*length v*. Thus

length[*uu*∗*v*] = 1+*length* d[*uu*∗*v*]	by LENGTH-UU and domain facts
= 1 + *length* [d*uu*]∗*v*	by APPEND-CDR
= 1 + [*length* d*uu* + *length v*]	by induction hypothesis.
= *length uu* + *length v*	by associativity of + and LENGTH-UU

this completes the proof of the lemma and thus of the original theorem.

As we mentioned earlier, there are many programs that terminate for all allowed input, but do not fit into one of the standard forms we have given so far. We could give more general forms. [A particularly nice example can be found in Boyer and Moore[1978]]. Instead we will generalize our technique to provide a method of representing programs defined by a general recursion and show how to prove termination.

Exercises

1. You were asked to write definitions for the function *mirror* in the exercises of Chapter II §9. Use the previously given definitions of *fringe* and *reverse* (and any facts proved about them so far) to show that your program satisfies

$$\forall x{:}[\mathit{reverse}\ \mathit{fringe}\ x = \mathit{fringe}\ \mathit{mirror}\ x].$$

6. Representing programs not known to terminate.

The method of the preceding section for representing programs known to terminate works well and could be extended to cover a large class of programs. However it is not possible to give a syntax that allows exactly those programs that always terminate, because deciding whether a program terminates is one of the famous unsolvable problems. Also, it is well known that LISP programs do not always terminate. (You have probably written a few such programs yourself!) Many interesting programs terminate on some but not all values of the arguments (for example *eval*) and we would like to be able to say things about these programs. In this section we will show how LISP programs defined by a general recursion schema can be represented and how termination can expressed and proved. The method extends the method of representing terminating programs.

Recall the general form of LISP program given in the previous section

6.1) $$f[x] \leftarrow \tau[f, x]$$

where τ is a LISP term involving f as a "new" program name. Let f also denote the partial function computed by this program. Thus $f\,x=y$ for those x such that the program terminates with result y. Also, for such x the equation $f\,x = \tau[f, x]$ holds as an equation among program terms and as an equation in the theory if f is interpreted as any function agreeing with the program whenever it halts. Since first order logic has the built in assumption that a function has a unique value for each of value of its argument, we can't represent the program by a partial function on S-expressions directly. Suppose we try to choose a total function on S-expressions extending the partial function and satisfying the the functional equation. The following example shows that this is not generally possible.

Omega example: S-expressions aren't enough.

Consider the program *omega* defined by:

OMEGA: $$omega\ x \leftarrow omega\ x\ .\ \mathrm{NIL}$$

It is easy to see that this program never terminates. (In a computer you would probably get an overflow of some kind as the evaluator attempted to construct an infinite tree of NILs.) Suppose we now form the corresponding equation

6.2) $$omega\ x = omega\ x\ .\ \mathrm{NIL}$$

and further assume that we can assign S-expression values to $omega[x]$ such that the equation is satisfied. Thus by CAR and substitution we have a $omega\ x=omega\ x$. But this contradicts the theorem NO-RPLACS proved in §2.

Among other things this tells us that if we are to represent programs by functions satisfying the functional equation, we must have objects in the domain other than S-expressions. What we need are non-S-expression objects to assign as values of the function when the program doesn't terminate. It turns out that one additional object will do. Thus we extend the domain of S-expressions by adding the non-Sexpression object $\perp$, called "bottom". The extended domain will

be denoted *ESEXP*. The value $\perp$ will be assigned to $f\,x$ when ever the corresponding computation does not terminate. We extend the basic LISP functions **a, d, cons, equal, at, n, not** to *ESEXP* by requiring the value to be $\perp$ if any of the arguments is $\perp$. We also require functions representing programs to have the value $\perp$ if any of the arguments is $\perp$. This corresponds to "call by value" evaluation which is what LISP does for programs defined in the manner described in previous chapters. It is also known as the "strict" extension. Other extensions will work, but we won't treat them here. The only non-strict functions we have are **if** and **and** or. **if** is $\perp$ if its first argument is $\perp$, but no restriction is made on the second and third arguments. This represents the specification that **if** only evaluates one of the second or third arguments depending on the outcome of the evaluating the first argument. **and**, and **or**, are extended by extending their definitions in terms of **if**.

[Remark: A complete description of the theory of extended S-expressions, and LISP programs is collected in the last section of this chapter.]

Given the program (6.1) the representing function f has the following properties:

(i) $f\,X = \tau[f, X]$ for all arguments.

(ii) f is the "least defined" such function. That is, if g is any other function satisfying (i), then whenever $f\,X$ is defined (eg. not $\perp$) then $g\,X$ is also defined and $f\,X = g\,X$.

(iii) $f\,x = y$ for some S-expression y if and only if the program f on input x terminates with result y.

If the program always terminates then the equation completely determines the function. Otherwise we need to express in the theory the fact expressed by (ii). This is done by a *minimization schema* which will be described later. If we can prove that $SEXP\,f\,x$ using only the functional equation and other previously known facts, then we will have shown that the program terminates on input x.

To summarize we represent a program defined by the recursion schema (6.1) by function on the domain *ESEXP* whose value is that determined by the program when it terminates and $\perp$ otherwise. This function is partially characterized by adding the axiom obtained from the functional equation by quantifying over the intended domains of definition and adding the domain mappings requiring that the value be $\perp$ if any of the arguments are. The characterization is completed by addition of the minimization schema for the recursion equation.

We can further generalize the form of definition by allowing multiple arguments and/or mutually recursive definitions. Thus programs

$$\begin{aligned} f_1[x_1, \ \ldots\ x_n] &\leftarrow \tau_1[f_1, \ \ldots\ , f_m, x_1, \ \ldots\ , x_n] \\ &\vdots \\ f_m[x_1, \ \ldots\ x_n] &\leftarrow \tau_m[f_1, \ \ldots\ , f_m, x_1, \ \ldots\ , x_n] \end{aligned} \tag{6.3}$$

where the τ_i are LISP terms involving the f_i as new program names are represented by adding to

the theory new function symbols nameing the programs, axioms and domain mappings corresponding to the program definitions and an appropriate minimization schema.

Example: Termination and correctness of *flatten*.

As an example of the method we represent the program *flatten*, which was advertised to compute the fringe of an S-expression, prove that it is terminates for all S-expressions and show that *flatten* does indeed compute the same function as *fringe*. The latter proof will show how to treat programs defined explicitly in terms of an auxiliary program that uses additional parameters. Recall the definitions

FLATTEN: $flatten[x] \leftarrow flat[x, \mathrm{NIL}]$
FLAT: $flat[x, u] \leftarrow$ if $\mathrm{at}\,x$ then $x.u$ else $flat[\mathrm{a}\,x, flat[\mathrm{d}\,x, u]]$.

We add to our theory the axioms

FLATTEN-DEF: $\forall x{:}\,flatten[x] = flat[x, \mathrm{NIL}]$
FLAT-DEF: $\forall x\ u{:}\,flat[x, u] =$ if $\mathrm{at}\,x$ then $x.u$ else $flat[\mathrm{a}\,x, flat[\mathrm{d}\,x, u]]$.
$flat{:}\ BOTTOM \times ESEXP \rightarrow BOTTOM$
$flat{:}\ ESEXP \times BOTTOM \rightarrow BOTTOM$

and as immediate corollaries we have

FLAT-ATOM: $\forall a\ u{:}\,flat[a, u] = a\ .\ u.$
FLAT-PAIR: $\forall xx\ u{:}\,flat[xx, u] = flat[\mathrm{a}\,xx, flat[\mathrm{d}\,xx, u]]$

First we will prove

$$\forall x\ u{:}\ NELIST\ flat[x, u].$$

This tells us that the progam for *flat* terminates for all input pairs $[x, u]$ and further that the range is the domain *NELIST*. Since the definition is an extended form of S-expression recursion we try S-expression induction as a method of proof.

In the ATOM case we have $\forall a\ u{:}\ NELIST\ flat[a, u]$ by FLAT-ATOM and the domain mappings for cons. In the non-ATOM case, we have the induction hypothesis

$$\forall u{:}\ NELIST\ flat[\mathrm{a}\,x, u] \wedge \forall u{:}\ NELIST\ flat[\mathrm{d}\,x, u].$$

By FLAT-PAIR we have $flat[xx, u] = flat[\mathrm{a}\,xx, flat[\mathrm{d}\,xx, u]]$. Using the d-part of the hypothesis we then can apply the a-part with u replaced by $flat[\mathrm{d}\,xx, u]$ and this completes the proof. An immediate consequence of this proof and FLATTEN is that $\forall x{:}\ NELIST\ flatten\ x$.

Now we will prove $\forall x{:}\ flatten\ x = fringe\ x$. By FLATTEN this is the same as $\forall x{:}\ flat[x, \mathrm{NIL}] = fringe\ x$. The simple principles we have used so far suggest that we proceed directly with a proof by S-expression induction on x. This unfortunately arrives at a dead end. (If you had trouble provinge $\forall u{:}\ reverse1\ u = reverse\ u$ this may be why!) The difficulty is that we begin with the second argument NIL, but after expanding the definition once this is no longer the case. Just as we have to carry along an extra variable in the computation we also do in the proof. There are several possible routes to take. We will prove the following more general statement

$$\forall x\ u\colon flat[x, u] = fringe\ x * u.$$

Clearly the statement we started to prove is a direct consequence of this one. and now we may proceed by a straightforward S-expression induction. In the ATOM case we have:

$flat[a, u] = a\ .\ u$	by FLAT-ATOM
$= \langle a \rangle * u$	by APPEND-NIL,APPEND-UU, since $\langle x \rangle = x\ .\ NIL$
$= fringe\ a * u$	by FRINGE-ATOM

In the non-ATOM case we have the induction hypothesis:

$$\forall u\colon flat[a\,xx, u] = fringe\ a\,xx * u \wedge \forall u\colon flat[d\,xx, u] = fringe\ d\,xx * u.$$

and we have

$flat[xx, u] = flat[a\,xx, flat[d\,xx, u]]$	; FLAT-PAIR
$= flat[a\,xx, fringe\ d\,xx * u]$	; induction hypothesis
$= fringe\ a\,xx * [fringe\ d\,xx * u]$	; induction hypothesis
$= fringe\ xx * u$	; ASSOC-APPEND,FRINGE-PAIR

This completes the proof. Notice that in order to apply the induction hypothesis and the associativity of *append* it is necessary to check the domain facts for *fringe*.

Example: termination of programs defined by list recursion.

Suppose we have a program defined by list recursion as follows

F: $f[u, \xi] \leftarrow$ if $n\,u$ then $g[\xi]$ else $h[a\,u, d\,u, \xi, f[d\,u, j[a\,u, d\,u, \xi]]]$

and that the programs g, h, and j are represented by functions having the same names such that we have the domain mappings:

$$g\colon D_0 \to D_1$$
$$j\colon SEXP \times LIST \times D_0 \to D_0$$
$$h\colon SEXP \times LIST \times D_0 \times D \to D_2$$

where D_0, D_1, D_2 are domains and $D = D_1 \cup D_2$, and ξ ranges over D_0. Then the program f is represented by a function f satisfying the axiom:

F-DEF: $\forall u\ \xi\colon f[u, \xi] = $ if $n\,u$ then $g[\xi]$ else $h[a\,u, d\,u, \xi, f[d\,u, j[a\,u, d\,u, \xi]]]$

which has the following immediated corollaries:

F-NIL: $\forall \xi\colon f[NIL, \xi] = g[\xi]$
F-UU: $\forall u\,u\ \xi\colon f[u\,u, \xi] = h[a\,u\,u, d\,u\,u, \xi, f[d\,u\,u, j[a\,u\,u, d\,u\,u, \xi]]]$

Using list induction we can prove $\forall u\ \xi\colon D\ f[u, \xi]$. In the NIL case we have $\forall \xi\colon D\ f[NIL, \xi]$ by F-NIL and the domain mappings for g. In the non-NIL case we assume the induction hypothesis

$\forall \xi$: $D\,f[d\,uu, \xi]$. By F-UU we have $f[uu, \xi] = h[a\,uu, d\,uu, \xi, f[d\,uu, f[a\,uu, d\,uu, \xi]]]$. Using the domain mapping for f and the induction hypothesis we have $D\,f[d\,uu, f[a\,uu, d\,uu, \xi]]$ and using the domain mapping for h we have $D\,f[uu, \xi]$ as desired. Using this we can easily show the further domain properties claimed for f, $\forall \xi$: $D_1\,f[\mathrm{NIL}, \xi]$ and $\forall uu\ \xi$: $D_2\,f[uu, \xi]$. Thus representation of programs defined by list recursion is a special case of the more general method. Showing that the S-expression recursion and numeric recursion principles are also special cases is similar. We need only replace list induction by S-expression or numeric induction respectively.

7. Recursively Defined Predicates.

In Chapter I we introduced propositional constructs *and*, *or* and *not* into the programing language with the intent that they should behave like the usual logical operators. We also treated some of the recursive definitions as though they defined predicates rather than functions (for example *member* and *equal*). In this section we will show how to represent programs by recursively defined predicates. For many programs written using the propositional constructs there is a corresponding predicate on S-expressions satisfying the equivalence obtained by replacing "←" by "≡" and the propositional programs by the corresponding logical connectives or predicates. In general it is not safe to make such a translation directly. To see why consider the following program:

LOSE: $lose\ x \leftarrow \mathrm{not}\ lose\ x$.

This would translate into

LOSE-DEF: $\forall x$:[$lose\ x \equiv \neg lose\ x$] .

which clearly loses by leading to a contradiction! We avoided a similiar problem for function definitions by adding the element $\perp$ to the domain thus providing a non S-expression value to assign to a term when the computation did not produce a value. Our solution is here the following. We continue to view recursive definitions as defining functions. Some of these functions will be intended to compute predicates in the sense that the predicate is true only for those values of the arguments for which the value of the function is an S-expression other that NIL. This is most naturally carried out in terms of a predicate *True* characterized by the axiom:

TRUTH: $\forall X$: [$True\ X \equiv SEXP\ X \wedge X \neq \mathrm{NIL}$]

from which we easily deduce

TRUTH1: $\forall x$: [$True\ x \equiv x \neq \mathrm{NIL}$]

For example consider the recursive program

SUBEXPF: $subexpf[x, y] \leftarrow$ [x equal y] or [not at y and [$subexpf[x, a\,y]$ or $subexpf[x, d\,y]$]].

Our intent is to define a predicate *subexp* satisfying the equivalence

SUBEXP: $\forall x\ y$: [*subexp*[x, y] = [$x = y$] $\vee$ [$\neg$*ATOM* y $\wedge$ [*subexp*[x, ay] $\vee$ *subexp*[x, dy]]]].

but we have no rule that allows changing a recursive program immediately into an equivalence relation for a predicate. Such a rule would lose as described above. Therefore, we first represent the program as a function obtaining the axiom

SUBEXPF-DEF: $\forall x\ y$:*subexpf*[x, y]=[x equal y]or[not at$y$ and [*subexpf*[x, ay] or *subexpf*[x, dy]]]

and the two immediate corollaries:

SUBEXPF-ATOM: $\forall x\ a$: *subexpf*[x, a] = x equal a
SUBEXPF-PAIR: $\forall x\ yy$: *subexpf*[x, yy] = x equal yy or [*subexpf*[x, ayy] or *subexpf*[x, dyy]].

We prove that *subexpf* is total

S-SUBEXP: $\forall x\ y$: *SEXP subexpf*[x, y].

using S-expression induction on the second argument. Thus we wish to prove $\forall y$: $\Phi\ y$ where:

$$\Phi[y] = \forall z\text{: } SEXP\ subexpf[z, y].$$

This works because recursive calls to *subexpf* in the definition all involve ay or dy as the second argument. For the ATOM case $\forall z\ a$: *SEXP subexpf*[z, a] follows from SUBEXPF-ATOM and the domain mapping of equal. In the non-ATOM case we have the induction hypotheses

$$\forall z\text{: } SEXP\ subexpf[z, ayy] \text{ and } \forall z\text{: } SEXP\ subexpf[z, dyy]$$

and $\forall z$: *SEXP subexpf*[z, yy] follows from this together with SUBEXPF-PAIR and the domain mappings for equal and or.

If we define the predicate *subexp* by

SUBEXP-DEF: $\forall X\ Y$: [*subexp*[X, Y] = *True subexpf*[X, Y]

we can prove the original sentence proposed for *subexp*.

First we give characterizations of the propositional functions in terms of the predicate *True* and the logical operations and predicates that the functions are indended to imitate. They follow directly from the defining axioms.

EQNOT: $\forall x$: [*True* not x = $\neg$*True* x]
EQAND: $\forall x\ y$: [*True*[x and y]= [*True* x $\wedge$ *True* y]]
EQOR: $\forall x\ y$: [*True*[x or y] = [*True* x $\vee$ *True* y]]
EQEQUAL: $\forall x\ y$: [*True*[x equal y] = $x = y$]
EQATOM: $\forall x$: [*True*[at$x$] = *ATOM* x]
EQNULL: $\forall x$: [*True*[nx] = *NULL* x]

The proof of SUBEXP is just a sequence of simplifications based on the use the EQ- lemmas, the definitions and the totality of *subexpf*. In particular, by SUBEXP-DEF we need only show

$$\forall x\ y\text{:}[True\ subexpf[x, y] = [x=y] \vee [\neg ATOM\ y \wedge [True\ subexpf[x, ay] \vee True\ subexp[x, dy]]]]$$

In the ATOM case by SUBEXPF-ATOM

$$\forall x\ a\colon [True[x \text{ equal } a] \equiv x = a]$$

which is true by EQEQUAL. In the non-ATOM case we let $y=yy$ and by SUBEXPF-PAIR, S-SUBEXPF we need only show

$$\forall x\ yy\colon [True\ [\ [x \text{ equal } yy] \text{ or } [subexpf[x, a\,yy] \text{ or } subexpf[x, d\,yy]]\ ]$$
$$\equiv [x = yy] \vee True\ subexpf[x, a\,yy] \vee True\ subexpf[x, d\,yy]\ ]$$

But this follows from S-SSUBEXPF, EQOR, and EQEQUAL.

[Remark: Our method of interpreting functions as predicates was chosen to correspond with the usual LISP convention. Other conventions are possible. For example we could restrict the functions that are to be interpreted as predicates to have values in the domain {T, NIL, ⊥}. The axioms for and etc. would be slightly different, but not much else changes.]

Exercise

You were asked to write the definition for the predicate *istail* in the exercises of Chapter II §9. Prove the following facts about *istail*.

$$\forall v\colon istail[\mathtt{NIL}, v]$$
$$\forall u\ v\colon [\neg NULL\ u \wedge istail[u, v] \supset istail[d\,u, v]]$$
$$\forall u\ v\ w\colon [istail[u, v] \wedge istail[v, w] \supset istail[u, w]]$$

First translate your definition of *istail* into a functional equation for the function *istailf*. Show *istailf* is total and that *istail* satisfies the equivalence you intended. All of this parallels the *subexp* example above. For the rest you are on your own.

8. Additional induction principles for proving.

When we presented the theory of LISP we introduced princples of induction for S-expressions, lists, and numbers based on the structure of the respective domains. These principles work well for proving facts about functions defined by the corresponding recursions. They are not generally easy to use when other forms of recursion are used in the function definition. In this section we will present some additional principles of induction.

Induction on well-founded orderings

The general idea of induction on some domain involves a notion of "smaller" on that domain which is *well-founded*. By well-founded we mean that every sequence of elements in which each element is smaller than the previous one must terminate after a finite number of steps. Given a well-founded notion of smaller and a property Φ, if it is the case that whenever Φ holds for all elements smaller than a given element then it also holds for that element, then we have Φ holds for all elements of the domain. This can be formally expressed by the schema

$$\forall \xi\colon [\forall \xi_1\colon [R[\xi_1, \xi] \supset \Phi\ \xi_1] \supset \Phi\ \xi] \supset \forall \xi\colon \Phi\ \xi$$

where R is a well-founded relation on the domain and ξ, ξ_1 are variables ranging over the domain.

To see that this induction principle is "correct" suppose the hypothesis holds and suppose there is some ξ such that $\neg\Phi\ \xi$. Then since R is well-founded, there is a smallest ξ such that $\neg\Phi\ \xi$. Otherwise we could find smaller counter examples ad-infinitum, forming a nonterminating sequence of elements each of which is smaller (as measured by R) than its predecessor. This contradicts the well-foundedness of R. If ξ is the smallest counter example then $\forall \xi_1\colon R[\xi_1, \xi] \supset \Phi\ \xi_1$ therefore $\Phi\ \xi$ must hold after all. Hence there is no such counter example.

As examples of well-founded relations on domains of immediate interest we have

$<$ the usual order relation on numbers
$<_1$ on lists where $u <_1 v \equiv \exists w w\colon v = w * u$, e.g. u is a proper tail of v.
$<_S$ on S-expressions where $x <_S y \equiv x \neq y \wedge subexp[x, y]$,
e.g. x is a proper subexpression of y.

Corresponding to these relations we have the following induction schemata:

NUMBINDUCTION-CVI: $\forall n\colon [\forall m\colon [m < n \supset \Phi\ m] \supset \Phi\ n] \supset \forall n\colon \Phi\ n$
LISTINDUCTION-CVI: $\forall u\colon [\forall v\colon [v <_1 u \supset \Phi\ v] \supset \Phi\ u] \supset \forall u\colon \Phi\ u$
SEXPINDUCTION-CVI: $\forall x\colon [\forall y\colon [y <_S x \supset \Phi\ y] \supset \Phi\ x] \supset \forall x\colon \Phi\ x$

Given domains D_1 and D_2 with well-founded orderings $<_1$, $<_2$ respectively there are a number of ways to construct new orderings on combinations of these domains. For example the lexicographic ordering $<_{12}$ on $D_1 \times D_2$ is defined by

$$(\xi_1, \xi_2) <_{12} (\xi_1', \xi_2') \equiv [\xi_1 <_1 \xi_1' \vee [\xi_1 = \xi_1' \wedge \xi_2 <_2 \xi_2']]$$

Another means of defining a well-founded ordering is to induce one by a mapping into a domain that has a well-founded ordering. Thus if $\rho\colon D_1 \to D_2$ and $<_2$ is a well-founded ordering on D_2 then the ordering $<_r$ on D_1 defined by $\xi <_r \xi' \equiv \rho\ \xi <_2 \rho\ \xi'$ is well-founded.

Rank functions and Induction on rank.

Suppose f is a recursively defined program on some domain D and let ξ be a variable of sort D. Suppose further that the recursive calls to f in the definition have arguments of the form $g_1\ \xi$, $g_2\ \xi$, ... $g_n\ \xi$. If a recursion is to succeed the recursive calls should be with arguments which are smaller in some sense. One way to make the notion of smaller precise is to find a "rank function" $\rho\colon D \to NATNUM$ such that $\rho\ g_i\ \xi < \rho\ \xi$ for $1 \leq i \leq n$. This rank function can be used to prove properties of the function compute by f using the principle of "induction on ρ". Informally, this principle says that if whenever $\Phi\ \xi_1$ holds for all ξ_1 of lesser rank than ξ then ξ satisfies Φ, then all ξ satisfy Φ. Formally we have the schema:

RANKIND: $\forall \xi: [\forall \xi_1: [\rho\ \xi_1 < \rho\ \xi \supset \Phi\ \xi_1] \supset \Phi\ \xi] \supset \forall \xi: \Phi\ \xi$

One way to justify the schema is as an instance of induction on a well-founded ordering. Namely the ordering $<_r$ defined by

$$\xi_1 <_r \xi \equiv \rho\ \xi_1 < \rho\ \xi$$

is well-founded. (Any strictly decreasing $<_r$ sequence induces a corresponding strictly decreasing $<$ sequence.) It can also be deduced in our theory from an appropriate instance of NATNUMINDUCTION-CVI. Before we prove that, we show how "rank induction" can be applied.

Consider the program *gopher* defined by

GOPHER: *gopher* $x \leftarrow$ **if** ax **then** x **else** *gopher* [aa$x$. [da$x$. dx]].

We represent this program by the function *gopher* satisfying the axiom:

GOPHER-DEF: $\forall x$: *gopher* x = **if** ax **then** x **else** *gopher* [aa$x$. [da$x$. dx]].

which has the immediate consequences:

GOPHER-ATOM: $\forall xx$: [*ATOM* a$xx \supset$ *gopher* xx=xx]
GOPHER-PAIR: $\forall xx$: [*PAIR* a$xx \supset$ *gopher* xx=*gopher*[aaxx . [daxx . dxx]]]

gopher digs up the left most atom of its argument, stacking the d-parts of successive *cars* as it goes. If you are familiar with operations on binary trees, *gopher* can be viewed as performing successive clock-wise rotations. These operations preserve the symmetric order on leaves of the tree. This corresponds to preserving the fringe of an S-expression.

From the definition we see that *gopher* makes sense only on non-atomic S-expressions. In the recursive call *gopher*[aa$x$. [da$x$. dx]] we see that what is getting "smaller" is ax. In particular

GOPHER-RANK: $\forall xx$: [*PAIR* a$xx \supset$ *size* a[aaxx . [daxx . dxx]] < *size* axx.]

So ρ xx=*size* axx is an appropriate rank function for proving facts about gopher. We will prove the following simple facts:

PAIR-GOPHER: $\forall xx$: *PAIR gopher* xx
SIZE-GOHPER: $\forall xx$: *size gopher* xx = *size* xx

To prove $\forall xx$: *PAIR gopher* xx we assume the rank induction hypothesis

$$\forall yy: [\textit{size}\ a\,yy < \textit{size}\ a\,xx \supset \textit{PAIR gopher}\ yy]$$

There are two cases corresponding two parts of the definition of *gopher*. If *ATOM* axx then *PAIR gopher* xx follows from GOPHER-ATOM. If *PAIR* axx then *PAIR gopher* xx follows from GOPHER-PAIR, GOPHER-RANK and the induction hypothesis. This completes the proof of PAIR-GOPHER.

The proof of $\forall xx$: *size gopher* xx = *size* xx is similar. We need a lemma about *size*:

$$\forall xx: [PAIR\ a\ xx \supset size\ xx = size\ [aa\ xx\ .\ [da\ xx\ \ dx\ xx]]]$$

which follows from SIZE-PAIR and the associativity of "+". We assume the rank induction hypothesis

$$\forall yy: [size\ a\ yy < size\ a\ xx \supset size\ gopher\ yy = size\ yy]$$

If $ATOM\ a\ xx$ then *size gopher* xx=*size* xx follows from GOPHER-ATOM. If $PAIR\ a\ xx$ then *size gopher* xx=*size* xx follows from GOPHER-PAIR, GOPHER-RANK, the size lemma and the induction hypothesis. This completes the proof of SIZE-GOPHER.

We have presented the idea of a rank function taking only one argument for notational simplicity. This idea and the corresponding induction principle generalize easily to multiple arguments.

To conclude our discussion of induction principles we will show how RANKIND can be derived from NUMBINDUCTION-CVI. In particular let D be some domain, ξ, ξ_1 variables ranging over D and let ρ be a rank function $\rho: D \to NATNUM$ and let Ψ be a predicate parameter. Then we want to prove:

RANKIND: $\forall \xi: [\forall \xi_1: [\rho\ \xi_1 < \rho\ \xi \supset \Psi\ \xi_1] \supset \Psi\ \xi] \supset \forall \xi: \Psi\ \xi$

We instantiate NUMBINDUCTION-CVI taking $\Phi[n] \equiv \forall \xi: \rho\ \xi = n \supset \Psi\ \xi$ obtaining

RANK-NUM: $\forall n: [\forall m: [m < n \supset \forall \xi: [\rho\ \xi = m \supset \Psi\ \xi]] \supset \forall \xi: [\rho\ \xi = n \supset \Psi\ \xi]\ ] \supset \forall n: \forall \xi: [\rho\ \xi = n \supset \Psi\ \xi]$

To prove RANKIND we assume

RANKHYP: $\forall \xi: [\forall \xi_1: [\rho\ \xi_1 < \rho\ \xi \supset \Psi\ \xi_1] \supset \Psi\ \xi]$

and show $\forall \xi: \Psi\ \xi$. Since the range of ρ is $NATNUM$, by RANK-NUM it is sufficient to show

$$\forall n: [\forall m: [m < n \supset \forall \xi: [\rho\ \xi = m \supset \Psi\ \xi]] \supset \forall \xi: [\rho\ \xi = n \supset \Psi\ \xi]\ ].$$

Thus we assume

RNUM-HYP: $\forall m: [m < n \supset \forall \xi: [\rho\ \xi = m \supset \Psi\ \xi]]$

and prove $\forall \xi: [\rho\ \xi = n \supset \Psi\ \xi]\ ]$ Assuming

RHO-N: $\rho\ \xi = n$

by RANKHYP, RHO-N it is sufficient to show $\forall \xi_1: [\rho\ \xi_1 < n \supset \Psi\ \xi_1]$. Assuming

RHO-M: $\rho\ \xi_1 < n$

by RNUM-HYP with $m = \rho\ \xi_1$ and RHO-M we have $\Psi\ \xi_1$ which completes the proof.

Exercises

1. Prove the following additional properties of *gopher* using induction on the rank.

CAR-GOPHER-FRINGE: $\forall xx$: a*fringe* xx = a*gopher* xx
CDR-GOHPER-FRINGE: $\forall xx$: d*fringe* xx = *fringe* d*gopher* xx

2. Derive the schema SEXPINDUCTTION from the schema NUMBINDUCTION-CVI by devising a suitable rank function.

[Remark: the point of this exercise and the above proof is to give you the idea that the various induction principles are just different ways of saying the same thing. In a practical proving system it is useful to have a lot of redundancy an many ways to express various notions.]

9. Partial functions and the Minimization Schema.

So far we have dealt only with recursive programs that defined total functions on S-expressions. In these cases the functional equation corresponding to the recursive definition completely characterized the function on S-expressions. Frequently we are interested in a program that does not always terminate and in this case there are likely to be many functions that satisfy the corresponding functional equation. Whatever we prove using this equation will be true for all of these functions, but there are also facts about the program we are representing that will not be provable using only the functional equation. For example, the program

9.1) $$loop\ x \leftarrow loop\ x$$

leads to the sentence

9.2) $$\forall x: [loop\ x = loop\ x]$$

which provides no information (all functions satisfy it) although the function *loop* corresponding to the program is undefined for all x. This is not always the case. Recall the program

9.3) $$omega\ x \leftarrow [omega\ x]\ .\ \mathtt{NIL}$$

which has the functional equation

9.4) $$\forall x: [omega\ x = [omega\ x]\ .\ \mathtt{NIL}]\ .$$

We showed in § 6 that assuming $SEXP\ omega\ x$ leads to a contradiction thus we can show $\forall x$: $[\neg SEXP\ omega\ x]$, using the functional equation and induction.

In order to characterize recursive programs, we need some way of expressing the fact that the function we mean is the least defined solution of the functional equation. We will do this by introducing, in addition to the functional equation, a schema called the *minimization schema* which expresses the fact that every function satisfying the equation on the domain of definition is defined at least every where that the function assigned to the program is defined, and when both are defined, the have the same value.

If the program f is defined by

9.5) $$f\,\xi \leftarrow \tau[f, \xi]$$

we have a function f satisyfying the functional equation

9.6) $$\forall \xi: f\,\xi = \tau[f, \xi]$$

where ξ is a variable ranging over the domain D. The minimization schema for this form of recursive definition has the form

9.7) $$\forall \xi: [D\,\tau[F, \xi] \supset F\,\xi = \tau[F, \xi]] \supset \forall \xi: [D\,f\,\xi \supset f\,\xi = F\,\xi]\,.$$

Here F is a function parameter. This schema is really *schema schema*, since for any particular term τ and function symbol f we produce from the schema a formula that still contains the paramater F. We then must substitute some function expression of our language for F to produce an axiom. (This is similar to the induction schemas, except there we substituted formulas for predicate parameters.)

The simplest application of the minimization schema is to show that the program for *loop* given above computes the totally undefined function. The minimization schema for loop is

9.8) $$\forall x:[SEXP\ F\ x \supset F\ x = F\ x] \supset \forall x: [SEXP\ loop\ x \supset loop\ x = F\ x]\,.$$

If we let F be the function expression $\lambda X: \perp$ then we obtain the axiom

$$\forall x:[SEXP\ \perp \supset \perp = \perp] \supset \forall x: [SEXP\ loop\ x \supset loop\ x = \perp]\,.$$

The left side of the implication is identically true since $SEXP \cap BOTTOM = \phi$ or $\neg SEXP\ \perp$. Thus we have

$$\forall x: [SEXP\ loop\ x \supset loop\ x = \perp]\,.$$

If we assume $SEXP\ loop\ x$ we obtain SEXP $\perp$ which is a contradiction. Thus we have shown

$$\forall x: \neg SEXP\ loop\ x$$

as desired.

The minimization schema can sometimes be used to show partial correctness. For example, the well known 91-function is defined by the recursive program over the natural numbers

$$f_{91}\ n \leftarrow \tau_{91}[f_{91}, n]$$

where

$$\tau_{91}[F, n] = \text{if } n > 100 \text{ then } n - 10 \text{ else } F\ F\ [n + 11]\,.$$

The corresponding minimization schema is

$$\forall n: [NATNUM\ \tau_{91}[F, n] \supset F\ n = \tau_{91}[F, n]] \supset \forall n: [NATNUM\ f_{91}\ n \supset f_{91}\ n = F\ n]\,.$$

The goal is to show that

$$\forall n: [f_{91}\ n = F_{91}\ n$$

where

$$\forall n: [F_{91}\ n = \text{if } n > 100 \text{ then } n - 10 \text{ else } 91] .$$

Using the minization schema we can easily show that

$$\forall n: [NATNUM\ f_{91}\ n \supset f_{91}\ n = F_{91}\ n .$$

Namely, we need only show

$$\forall n: [NATNUM\ \tau_{91}[F_{91}, n] \supset F_{91}\ n = \tau_{91}[F_{91}, n]]$$

which follows by direct calculation from the definitions of F_{91} and τ_{91} by considering the three cases $n>100$, $89<n\leq 100$ and $n\leq 89$. The hypothesis of the implication is not needed here.

The method of *recursion induction* is also an immediate application of the minimization schema. If we show that two functions satisfy the schema of a recursive program, we show that they both equal the function computed by the program wherever the function is defined.

For example we can use the minimization schema for the program

REV: $rev[u, v] \leftarrow$ **if** n u **then** *qv* **else** *rev*[d u, a u . *v*]

and the fact that $\forall u\ v$: *LIST* *rev*$[u, v]$ to show that

EQ-REV: $\forall u$: *rev*$[u,$ NIL$]$=*reverse*$[u]$ *where* REVERSE: *reverse* u $\leftarrow$ **if** n u **then** NIL **else** [reverse d u]*<a u>. *The* minimization schema for the program *rev* is

$$\forall u\ v: [LIST\ \tau_{rev}[F, u, v] \supset F[u, v] = \tau_{rev}[F, u, v]] \supset \forall u\ v: [LIST\ rev[u, v] \supset rev[u, v] = F[u, v]] .$$

Taking $F[u, v]$=*revrse*$[u]$*v, we need only show $F[u, v]=\tau_{rev}[F, u, v]$ as this gives $\forall u\ v$: *rev*$[u,$ *v*$]$=*reverse*$[u]$*v and EQ-REV is a special case. By the definitions we need only prove

$$reverse[u]*v = \text{if } n\ u \text{ then } v \text{ else } reverse[d\ u]*[a\ u\ .\ v].$$

In the case u is NIL both sides of the equation simplify to v. In the case u is some non-empty list uu we by REVERSE we need only show

$$[reverse[d\ uu]*<a\ uu>]*v = reverse[d\ uu]*[a\ uu\ .\ v].$$

But this is an easy consequence of ASSOC-APPEND, APPEND-NIL, APPEND-UU.

The utility of the minimization schema for proving partial correctness or non-termination depends on our ability to name suitable comparison functions. *loop* and f_{91} were easily treated, because the necessary comparison functions could be given explicitly without recursion. Any extension of the language that provides new tools for naming comparison functions, e.g. going to higher order logic, will improve our ability to use the schema in proofs.

10. Theory of LISP: Algebraic Axioms.

In this appendix we collect the language specification, domain facts and axioms for the theory of extended S-expressions and the subdomains of S-expressions, lists and numbers, and the definitions of functions representing the basic LISP programs.

Domains

ESEXP, *BOTTOM*, *SEXP*, *PAIR*, *ATOM*, *LIST*, *NELIST*, and *NULL* *NATNUM*, *POSP*, and *ZERO*.

Domain relations:

$$ESEXP = SEXP \cup BOTTOM$$
$$SEXP \cap BOTTOM = \phi$$

$$SEXP \subset ESEXP$$
$$SEXP = ATOM \cup PAIR$$
$$ATOM \cap PAIR = \phi$$

$$LIST \subset SEXP$$
$$NELIST \subset PAIR$$
$$NULL = \{\mathrm{NIL}\}$$
$$LIST = NULL \cup LIST$$
$$NULL \cap NELIST = \phi$$

$$NATNUM \subset ATOM$$
$$ZERO = \{0\}$$
$$NATNUM = ZERO \cup POSP$$
$$ZERO \cap POSP = \phi$$

Constants:

$$\mathrm{T} \in ATOM$$
$$\mathrm{NIL} \in NULL$$
$$\perp \in BOTTOM$$
$$0 \in ZERO$$
$$1, 2, \ldots \in POSP$$

Variables:

$$X, Y, Z, U, V, W, M, N \in ESEXP$$
$$x, y, z \in SEXP$$
$$xx, yy, zz \in PAIR$$
$$a, b, c \in ATOM$$

$u, v, w \in LIST$
$uu, vv, ww \in NELIST$

$l, n, m \in NATNUM$
$ll, mm, nn \in POSP$

[Remark: If ξ is a variable ranging over domain D then we may also use $\xi_0, \xi_1, \ldots$ as variables ranging over D].

Function symbols and domain mappings:

The basic function symbols are: **a, d, cons,** *add*1, *sub*1, **at, n, if, not, and, or.**

a: $BOTTOM \to BOTTOM$
a: $PAIR \to SEXP$
a: $NELIST \to SEXP$

d: $BOTTOM \to BOTTOM$
d: $PAIR \to SEXP$
d: $NELIST \to LIST$

cons: $BOTTOM \times ESEXP \to BOTTOM$
cons: $ESEXP \times BOTTOM \to BOTTOM$
cons: $SEXP \times SEXP \to PAIR$
cons: $SEXP \times LIST \to NELIST$

*add*1: $BOTTOM \to BOTTOM$
*add*1: $NATNUM \to POSP$

*sub*1: $BOTTOM \to BOTTOM$
*sub*1: $POSP \to NATNUM$

Algebraic Axioms

CAR: $\forall x\, y.\mathbf{a}[x\ .\ y] = x$
CDR: $\forall x\, y.\mathbf{d}[x\ .\ y] = y$
CONS: $\forall xx.[\mathbf{a}\,xx\ .\ \mathbf{d}\,xx] = xx$
EQPAIR: $\forall xx\ yy.[[\mathbf{a}\,xx = \mathbf{a}\,yy] \wedge [\mathbf{d}\,xx = \mathbf{d}\,yy] \supset xx = yy]$

SUB1: $\forall n$: *sub*1 *add*1 $n = n$
ADD1: $\forall nn$: *add*1 *sub*1 $nn = nn$

Induction schemata

SEXPINDUCTION: $\forall a{:}\Phi\ a \wedge \forall xx{:}[\Phi\ \mathbf{a}\,xx \wedge \Phi\ \mathbf{d}\,xx \supset \Phi\ xx] \supset \forall x{:}\ \Phi\ x\ .$
LISTINDUCTION: Φ NIL $\wedge \forall uu{:}\ [\Phi\ \mathbf{d}\,uu \supset \Phi\ uu] \supset \forall u{:}\ \Phi\ u\ .$
NUMINDUCTION: $\Phi\ 0 \wedge \forall nn{:}\ [\Phi$ *sub*1 $nn \supset \Phi\ nn] \supset \forall n{:}\ \Phi\ n.$

Defining axioms for program primitives

IF: $\forall X\ Y\ Z$: if$[X, Y, Z] = IF\ \neg SEXP\ X\ THEN\ \perp\ ELSE\ IF\ X \neq$ NIL $THEN\ Y\ ELSE\ Z$
$\forall x\ y\ z$: if$[x, y, z] = IF\ x \neq$ NIL $THEN\ y\ ELSE\ z$
if: $SEXP \times SEXP \times SEXP \to SEXP$

EQUAL: $\forall X\ Y$: X *equal* $Y = IF\ \neg[SEXP\ X \wedge SEXP\ Y]\ THEN\ \perp\ ELSE\ IF\ X = Y\ THEN$ T $ELSE$ NIL
$\forall x\ y$: [x equal y] $= IF\ [x = y]\ THEN$ T $ELSE$ NIL
equal: $SEXP \times SEXP \to SEXP$

ATOM: $\forall X$: at $X = IF\ \neg SEXP\ X\ THEN\ \perp\ ELSE\ IF\ ATOM\ X\ THEN$ T $ELSE$ NIL
$\forall x$: at $x = IF\ ATOM\ x\ THEN$ T $ELSE$ NIL
at: $SEXP \to SEXP$

NULL: $\forall X$: n $X = IF\ \neg SEXP\ X\ THEN\ \perp\ ELSE\ IF\ NULL\ X\ THEN$ T $ELSE$ NIL
$\forall x$: n $x = IF\ NULL\ x\ THEN$ T $ELSE$ NIL
n: $SEXP \to SEXP$

NOT: $\forall X$: not X = if$[X,$ NIL, T]
$\forall x$: not x = if x then NIL else T
not: $SEXP \to SEXP$

AND: $\forall X\ Y$: X and Y = if$[X, Y, X]$
$\forall x\ y$: x and y = if x then y else NIL
and: $SEXP \times SEXP \to SEXP$

OR: $\forall X\ Y$: X or Y = if$[X,\ X,\ Y]$
$\forall x\ y$: x *or* y = if x then x else y
or: $SEXP \times SEXP \to SEXP$

11. Exercises

The following are properties to prove about programs that you have written. The functions and predicates referred to are described in the Exercises of Chapter II §9. You should carry out the proofs for your definitions of the indicated functions or predicates. You may decide to rewrite your definition because (1) you find that your function definition is not correct or (2) another definition is cleaner and more amenable to proofs. This is quite natural if you are not thinking in terms of proving it correct when you write the definition.

1. Lists

1.1. $\forall u.\ samelength[u, reverse\ u]$

1.2. $\forall u\ v: istail[commontail[u, v], v]$
1.3. $\forall u\ v: istail[commontail[u, v], u]$

1.4. $\forall u\ v:\ commontail[u, v] = commontail[v, u]$

1.5. $\forall u\ v: [append[upto[commontail[u, v], v], commontail[u, v]] = v]$
1.6. $\forall u\ v: [append[upto[commontail[u, v], u], commontail[u, v]] = u]$

Note that *istail* has already been worked on in earlier exercises.

2. Sexpressions

2.1. $\forall x\ y: [x = get[y, point[x, y]]]$

2.2. $\forall x: [balanced[balance[x]]\ samefringe[balance[x], x]]$

3. Algebra and arithmetic

3.1. $\forall u\ v:\ poly\ u \wedge poly\ v \supset sum[prod[quot[u, v], v], rem[u, v]$

3.2. $\forall x:\ arith\ x \supset numval[sop\ x]] = numval[x]$

Part of the exercise is to invent suitable predicates *poly* and *arith which* characterize the domain on which the functions are valid.

4. Sets

4.1. $\forall x\ u\ v: [x \in [u \cup v] \supset x \in u \vee x \in v]$

4.2. $\forall x\ u\ v: [[x \in u] \supset \neg x \in [v - u]]$

5. Permutations

5.1. $\forall u: [lcycle\ rcycle\ u = u]$

5.2. $\forall u: [isperm1\ u \supset [compose1[p, invert1[p]] = id1$

5.3. $\forall u{:}[isperm1\ u \supset [compose1[invert1[p]_{\circ}\ p] = id1$

5.4. $u{:}\ [isperm1\ u \supset [invert1\ invert1\ p = p]]$

5.5. $\forall u\ v{:}[isperm2\ u \wedge isperm2\ v \supset rho21[compose2[u, v]] = compose1[rho21\ u, rho21\ v2]]$

Chapter IV

PROOFS: EXAMPLES

In this section we present some non-trivial examples illustrating the methods developed in Chapter III.

1. The SAMEFRINGE problem.

As a substantial application of the techniques described in Chapter III, we will give a proof of correctness of the predicate *samefringe* which has been presented as a solution of the SAMEFRINGE problem. This is the problem of determining whether or not two S-expressions have the same fringe using a minimum of space. We will be concerned only with the correctness of *samefringe* since the efficiency is not an *extensional* property.

We have already studied two programs for computing the fringe of an S-expression, namely *fringe* and *flatten*, which were proved to compute the same function from S-expressions to non-empty lists. The predicate *samefringe* is computed by the LISP program

SAMEFR: *samefringe*[x, y] ←
    x equal y or [not at x and not at y and *same*[*gopher* x, *gopher* y]]
where
SAME: *same*[x, y] ← a x equal a y and *samefringe*[d x, d y]
and
GOPHER: *gopher* x ← if at a x then x else *gopher* [aa x . [da x . d x]].

What we want to show is that *samefringe* says "yes" for input x, y just when *fringe* x=*fringe* y. To do this we use the ideas of Chapter III §7. We first represent the above programs by functions *samefringef*, *samef*, and *gopher* satisfying the axioms:

SAMEFR-DEF: ∀x y: *samefringef*[x, y] =
    x equal y or [not at x and not at y and *same*[*gopher* x, *gopher* y]]
SAME-DEF: ∀x y: *same*[x, y] = a x equal a y and *samefringef*[d x, d y]
GOPHER-DEF: ∀x: *gopher* x ← if at a x then x else *gopher* [aa x . [da x . d x]].

We then define the predicate *samefringe* by

SAMEFRINGE-DEF ∀x y: [*samefringe*[x, y] = *True* *samefringef*[x, y]]

and the statement of correctness becomes

SAMEFRINGE-COR: ∀x y: [*samefringe*[x, y] = *fringe* x=*fringe* y].

The program computing *samefringe* divides pairs x, y of S-expressions into two cases: ATOMXY where *ATOM* x ∨ *ATOM* y and PAIRXY where *PAIR* x ∧ *PAIR* y. And we have two immediate corollaries of SAMEFR-DEF and SAME-DEF.

SFF-ATOMXY: $\forall x\ y$: [*ATOM* $x \vee$ *ATOM* y] $\supset$ *samefringef*[x, y] = x equal y
SFF-PAIRXY: $\forall xx\ yy$: *samefringef*[xx, yy] =
a *gopher* xx equal a *gopher* yy and *samefringef*[d *gopher* xx, d *gopher* y]

Since recursive calls occur only in the PAIRXY case and are of the form *samefringef*[d *gopher* xx, d *gopher* yy] this suggests that to prove properties of *samefringef* (and of *samefringe*) we need to find a rank function $\rho[x, y]$ such that

$$\forall xx\ yy\colon \rho[\text{d } gopher\ xx, \text{d } gopher\ yy] < \rho[xx, yy].$$

Proof can then be done using induction on the rank as explained in Chapter III §8.

Recall that we proved in Chapter III §8

PAIR-GOPHER: $\forall xx$: *PAIR gopher* xx
SIZE-GOHPER: $\forall xx$: *size gopher* xx = *size* xx

Thus either $\rho[x, y]$=*size* x+*size* y or simply *rho*$[x, y]$ = *size* x will do as a rank function for *samefringe*. Since for either version of ρ we have

$$\forall xx\ yy\colon \rho[\text{d } gopher\ xx, \text{d } gopher\ yy] < \rho[xx, yy]$$

we see that to prove $x\ y$: $\Phi[x, y]$ by on rank it is sufficient to prove

PHI-ATOMXY: $\forall\ x\ y$: [*ATOM* $x \vee$ *ATOM* y]$\supset\Phi[x, y]$
and
PHI-PAIRXY: $\forall xx\ yy$: Φ[d *gopher* xx, d *gopher* yy] $\supset$ $\Phi[xx, yy]$

To prove that correctness of *samefringe* we first prove that the program terminates on all S-expression pairs, thus giving up a complete description of the predicate. Thus we prove by induction on the rank ρ that

S-SAMEFR: $\forall x\ y$: *SEXP samefringef*[x, y].

which according to the above discussion reduces to proving

S-ATOMXY: $\forall\ x\ y$: [*ATOM* $x \vee$ *ATOM* y]$\supset$*SEXP samefringef*[x, y]
and
S-PAIRXY: $\forall xx\ yy$: *SEXP samefringef*[d *gopher* xx, d *gopher* yy] $\supset$ *SEXP samefringef*[xx, yy]

S-ATOMXY follows directly from SFF-ATOMXY and the domain map for equal and S-PAIRXY follows directly from SFF-PAIRXY, PAIR-GOPHER, and the domain maps for equal and and.

We now obtain a characterization of *samefringe* in the two cases *ATOMXY* and *PAIRXY*.

SF-ATOMXY: $\forall x\ y$: [[*ATOM* $x \vee$ *ATOM* y] $\supset$ *samefringe*[x, y] $\equiv$ $x = y$]
SF-PAIRXY: $\forall xx\ yy$: [*samefringe*[xx, yy] $\equiv$
a *gopher* xx = a *gopher* $yy \wedge$ *samefringe*[d *gopher* xx, d *gopher* y]]

These facts follow directly from SAMEFRINGE-DEF,SFF-ATOMXY,SFF-PAIRXY,S-SAMEFR,PAIR-GOPHER, and the EQ- theorems given in Chapter III §7.

Now we are ready to prove SAMEFRINGE-COR. Using the same induction principle as for S-SAMEFR we see that we need only prove

COR-ATOMXY: $\forall x\ y$: [[*ATOM* $x \vee$ *ATOM* y]⊃[*samefringe*[x, y]=*fringe* x=*fringe* y]]
and
COR-PAIRXY: $\forall xx\ yy$: [[*samefringe*[d *gopher* xx, d *gopher* yy]=
fringe d *gopher* xx=*fringe* d *gopher* yy] ⊃
[*samefringe*[xx, yy]=*fringe* xx=*fringe* yy]]

In order to complete the proof we need some lemmas about *fringe* and *gopher*:

FRINGE-ATOMXY: $\forall x\ y$: [[*ATOM* $x \vee$ *ATOM* y]⊃[*fringe* x=*fringe* y=x=y]]
CAR-GOPHER-FRINGE: $\forall xx$: a*fringe* xx = a *gopher* xx
CDR-GOHPER-FRINGE: $\forall xx$: d*fringe* xx = *fringe* d *gopher* xx

CAR-GOPHER-FRINGE and CDR-GOHPER-FRINGE were given as exercises in Chapter III §8. FRINGE-ATOMXY follows easily from $\forall x\ a$: [*fringe* x=*fringe* a=x=a] Which can be proved as follows. If x is an atom Then by FRINGE-ATOM we need only show $\langle x\rangle$=$\langle a\rangle$=x=a which is follows from EQPAIR (proved in Chapter III §2). If x is not an atom then *ATOM*∩*PAIR*=ϕ tells us that $x \neq a$ and we need only show *fringe* $x \neq \langle a\rangle$. One way to see this is to observe that d$\langle a\rangle$=NIL and *NELIST* d*fringe* x. We leave the details to the reader.

Now to complete the proof of COR-SAMEFRINGE. COR-ATOMXY follows directly from SF-ATOMXY and FRINGE-ATOMXY. To prove COR-PAIRXY we assume

samefringe[d *gopher* xx, d *gopher* yy]=*fringe* d *gopher* xx=*fringe* d *gopher* yy]

then by SF-PAIRXY it is sufficient to prove

[a *gopher* xx=a *gopher* $yy \wedge$ *fringe* d *gopher* xx=*fringe* d *gopher* yy]=*fringe* xx=*fringe* yy

Using CAR-GOPHER-FRINGE and CDR-GOPHER-FRINGE this reduces to:

[a*fringe* xx=a*fringe* $yy \wedge$ d*fringe* xx=d*fringe* yy]=*fringe* xx=*fringe* yy

which is just EQPAIR.

Another version of *samefringe* without any auxiliary functions is

```
SAMEFR1: samefringe[x, y] ←
           x equal y or
           [not at x and not at y and
             [if at a x then [if at a y then a x equal a y and samefringe[d x, d y]
                                        else samefringe[x, a a y. [d a y . d y]]]
              else samefringe[a a x . [d a x . d x], y].
```

Note that a recursive call to *samefringe* does one of the following three things:

1) decreases *size* x + *size* y
2) leaves *size* x + *size* y and *size* ax invariant and decreases *size* ay
3) leaves *size* x + *size* y and *size* ay invariant and decreases *size* ax .

This can lead to a choice of an induction axiom schema in at least two ways. If in the NUMINDUCTION-CVI schema we let n and m range over all ordinals less than a given one it becomes a schema of transfinite induction. Ordinary induction is obtained as a special case where the bounding ordinal is ω the least transfinite ordinal. If we take the bounding ordinal to be ω^{ω} then the SAMEFRINGE theorem for the above version of *samefringe* can be proved using the predicate

$$\Phi\, n \;=\; \forall x\, y\colon [[\mathit{size}\ x + \mathit{size}\ y]\omega + \mathit{size}\ \mathrm{a}x + \mathit{size}\ \mathrm{a}y = n \supset \mathrm{THM}[x,\ y]]$$

Alternately one could axiomatize the lexicographic ordering of triples of numbers by

$$\forall l_1\ m_1\ n_1\ l\ m\ n\colon [(l_1,\ m_1,\ n_1) < (l,\ m,\ n) \equiv [l_1 < l] \vee [l_1 = l \wedge m_1 < m] \vee [l_1 = l \wedge m_1 = m \wedge n_1 < n]]$$

This ordering is well-founded (has no infinite decreasing sequences) and so we have a schema analogous to NUMINDUCTION-CVI given by

$$\forall l\ m\ n\colon [\forall l_1\ m_1\ n_1\colon [(l_1,\ m_1,\ n_1) < (l,\ m,\ n) \supset \Phi(l_1,\ m_1,\ n_1)] \supset \Phi(l,\ m,\ n)] \supset \forall l\ m\ n\colon \Phi(l,\ m,\ n)$$

The SAMEFRINGE theorems can now be proved using this schema with the predicate

$$\Phi(l,\ m,\ n) \equiv \forall x\ y\colon [l = \mathit{size}\ x + \mathit{size}\ y \wedge m = \mathit{size}\ \mathrm{a}y \wedge n = \mathit{size}\ \mathrm{a}x \supset \mathrm{THM}[x,\ y]]$$

The minimization schema provides an alternate method for proving the correctness of *samefringe*. To simplify matters we eliminate the auxiliary function *same* from the definition of *samefringe* thus eliminating the problem of having to deal with mutually recursive programs. The functional defining *samefringe* now becomes

τ_{sf} = $\lambda f\colon \lambda x\ y\colon$ [[x equal y] or [[not at x and not at y] and
[[a *gopher* x equal a *gopher* y] and f[d *gopher* x, d *gopher* y]]]] .

Now we form an axiom schema from the minimization schema by using the τ_{sf} given above and *samefringe* for f.

$$\forall x\ y\colon [SEXP\ \tau_{sf}[F, x, y] \supset F[x, y] = \tau_{sf}[F, x, y]] \supset \forall x\ y\colon [SEXP\ \mathit{samefringe}[x, y] \supset \mathit{samefringe}[x, y] = F[x, y]] .$$

We instantiate this schema with

$$F[x,\ y] = \mathit{fringe}\ x \text{ equal } \mathit{fringe}\ y \ .$$

The proof then consists of the following steps.

Prove

1) $\forall x\, y$: $[F[x, y] = \tau_{sf}[F][x, y]]$

2) $\forall x\, y$:$[SEXP\ samefringef[x, y]$

conclude from 1) - 2) and the minimization schema instantiatio

3) $\forall x\, y$: [*fringe* x equal *fringe* y = *samefringef*$[x, y]$]

and from the definition of *samefringe* 2) and 3) conclude

5) $\forall x\, y$: [*samefringe*$[x, y]$ = *fringe* x = *fringe* y]

as desired.

2. Correctness of a program to partition lists.

Consider the problem of determining all partitions of a list into some number of (non-empty) sublists. We say that a list w is a partition of another list u into n parts if:

1. each element of w is a nonempty list,
2. the length of w is n, and
3. the result of *appending* the elements of w together (in order) is u.

Thus `((A B) (C) (D))` is a partition of the list `(A B C D)` into 3 parts, and

```
(((A B) (C) (D)) ((A) (B C) (D)) ((A) (B) (C D)))
```

is the list of all partitions of `(A B C D)` into 3 parts. Note that for lists the order of the elements of the partition is important, in contrast the usual notion of partitions of a natural number into a list of summands.

In order to make the discussion of partitions easier, we will introduce some new sorts: *UULIST* (lists whose members are non-empty lists) made up of the subsorts *NEUULIST* (non-empty lists of non-empty lists) and *NULL*, *PNLIST* (partition lists, e.g. the members are of sort *UULIST*) made up of the subsorts *NEPNLIST* and *NULL*, and *PNNLIST* (non trivial partition lists, e.g. the members are of sort *NEUULIST*) made up of the subsorts *NEPNNLIST* and *NULL*. We will have additional variables ranging over the new sorts.

$uul, vvl, wwl \in UULIST$
$uull, vvll, wwll \in NEUULIST$
$pl, ql, rl \in PNLIST$
$pll, qll, rll \in NEPNLIST$
$pnnl, qnnl, rnnl \in PNNLIST$
$pnnll, qnnll, rnnll \in NEPNNLIST$

And we will have definition of programs by recursion on the new domains (analogous to ordinary list recursion) and corresponding structural induction principles. A formal description of the new sorts is given at the end of this section.

In order to formalize the notion of a partition, we need to specify what is meant by the phrase "appending elements of a list together". The program *combine* appends together the elements of a partition and is defined using *UULIST* recursion by:

combine uul ← **if** *n uul* **then** NIL **else** *a uul* * *combine d uul*

and represented by the function *combine* which satisfies the axiom:

COMBINE: $\forall uul$: *combine uul* = **if** *n uul* **then** NIL **else** *a uul* * *combine d uul*

from which we easily deduce

COMBINE-NIL: *combine* NIL = NIL
COMBINE-ULL: $\forall uull$: *combine uull* = *a uull* * *combine d uull*

If we restrict the domain of *append* to *UULIST PNLIST* or *PNNLIST* we obtain the additional domain mappings for *append*:

$*: UULIST \times UULIST \to UULIST$
$*: PNLIST \times PNLIST \to PNLIST$
$*: PNNLIST \times PNNLIST \to PNNLIST$

Using the facts specfied by the function mappings for *append* we have

combine: $UULIST \to LIST$
combine: $NEUULIST \to NELIST$

Now we can give a formal definition of what we mean by partition. We introduce the predicate *Ispartn* with the defining axiom

ISPARTN-DEF:
$\forall U\ N\ W$:[*Ispartn*[W, U, N] = $\exists uul$.(W = *uul* $\wedge$ *length uul* = N $\wedge$ *combine uul* = U]]

which is a direct formalization of the three properties of a partition given above. Given a list u and a number n we want a list pl of all partitions of u into n parts.

Before we give the program computing pl, we analyze some properties of partitions of a list. First, the degenerate cases where u is NIL or n is 0. If u = NIL then it can not be partitioned into non-empty lists, so the only possibility is the partition which itself is NIL partitioning of u into 0 parts. If u is non-empty then there must be at least 1 part in the partition, e.g. $n \neq 0$. This is expressed by the following lemmas. Also note that in the definition of *Ispartn*, if $n \neq 0$ we may choose the *UULIST* to be non-empty.

ISPARTN-NIL-ZERO: $\forall W$:[*Ispartn*[W, NIL, 0] = W = NIL]
ISPARTN-NIL-NN: $\forall nn\ W$:¬*Ispartn*[W, NIL, nn]
ISPARTN-UU-ZERO: $\forall uu\ W$:¬*Ispartn*[W, uu, 0]
ISPARTN-DEF-NN:
$\forall uu\ nn\ W$:[*Ispartn*[W, uu, nn]=$\exists uull$:[W=*uull*∧*length uull*=nn∧*combine uull*=uu]]

These lemmas are easily proved using the definitions of *Ispartn*, *length* and *combine*, the fact that the only the empty list has length 0 and only the empty list of non-empty lists combines into NIL.

Now we consider non-trivial partitions of non-empty lists. Consider a partition *uull* of the non-empty list *uu* into *nn* parts. Let *ww*=a*uull*. Note that *ww* must be and initial segment of *uu*. There are two posibilities. Either *ww* is the one element list <a*uu*>, or it is a list of length greater than one (e.g. a*ww*=a*uu* and d*ww*≠NIL. In the first case d*uull* is a partition of d*uu* into *sub*1 *nn* parts. In the second case the list formed by replacing *ww* by d*ww* (in *W*) is a partition of d*uu* into *nn* parts. Thus we have

ISPARTN-UU-NN:
$$\forall uu\ nn\ W{:}[Ispartn[W, uu, nn] \equiv$$
$$\exists uull{:}[W{=}uull \wedge a\,uull{=}\langle a\,uu\rangle \wedge Ispartn[d\,uull, d\,uu, sub1\ nn]] \vee$$
$$\exists uull{:}[W{=}uull \wedge aa\,uull{=}a\,uu \wedge Ispartn[[da\,uull\ .\ d\,uull], d\,uu, nn]]$$

The proof of ISPARTN-UU-NN uses just the definitions of *Ispartn*, *length*, and *combine* together with basic facts about numbers and S-expressions. By ISPARTN-DEF-NN it is sufficient to show

$$\exists uull{:}[W{=}uull \wedge length\ uull{=}nn \wedge combine\ uull{=}uu]$$
$$\equiv \exists uull{:}[W{=}uull \wedge a\,uull{=}\langle a\,uu\rangle \wedge Ispartn[d\,uull, d\,uu, sub1\ nn]] \vee$$
$$\exists uull{:}[W{=}uull \wedge aa\,uull{=}a\,uu \wedge Ispartn[[da\,uull\ .\ d\,uull], d\,uu, nn]]$$

First assume

ISUUNN1: $W{=}uull \wedge length\ uull{=}nn$ *and* $combine\ uull{=}uu$.

Then if we subsitute *combine uull* for *uu* and use COMBINE-ULL, APPEND-UU and the basic facts about S-expressions we obtain a*uull*=a*uu* . da*uull* In the case: da*uull*=NIL we will show

$$\exists vull{:}[W{=}vull \wedge a\,vull{=}\langle a\,uu\rangle \wedge Ispartn[d\,vull, d\,uu, sub1\ nn]]$$

and in the case: da*uull*≠NIL we will show

$$\exists vull{:}[W{=}vull \wedge aa\,vull{=}a\,uu \wedge Ispartn[[da\,vull\ .\ d\,vull], d\,uu, nn]]$$

thus proving the => direction of the equivalence. Assume da*uull*=NIL then if we let *uul*=*qd* uull we have by APPEND-NIL, APPEND-UU, LENGTH-UU, COMBINE-ULL and basic properties of numbers and S-expressions

$$uul{=}d\,uull \wedge length\ uul{=}sub1\ nn \wedge combine\ uul{=}d\,uu$$

and by ISPARTN-DEF

$$W{=}uull \wedge d\,uull{=}[a\,uu\ .\ NIL] \wedge Ispartn[d\,uull, d\,uu, sub1\ nn]$$

completing the first case. Now assume da*uull*≠NIL Let da*uull*=*ww* and [*ww* .d*uull*]=*wwll*. Then as above we have

$$cons(cdr\ car\ uull, cdr\ uull){=}wwll \wedge length\ wwll{=}nn \wedge combine\ wwll{=}d\,uu$$

and by ISPARTN-DEF

$$W{=}uull \wedge aa\,uull{=}a\,uu \wedge Ispartn[da\,uull\ .\ d\,uull, d\,uu, nn]$$

completing the second case and the proof of the => direction.

To prove the <= direction there are also two cases:

$\exists uull{:}[W = uull \wedge a\ uull = \langle a\ uu\rangle \wedge Ispartn[d\ uull, d\ uu, sub1\ nn]] \supset$
$\exists uull{:}[W = uull \wedge length\ uull = nn \wedge combine\ uull = uu]$
and
$\exists uull{:}[W = uull \wedge aa\ uull = a\ uu \wedge Ispartn[[da\ uull\ .\ d\ uull], d\ uu, nn]] \supset$
$\exists uull{:}[W = uull \wedge length\ uull = nn \wedge combine\ uull = uu].$

If we assume

$$W = uull \wedge a\ uull = \langle a\ uu\rangle \wedge Ispartn[d\ uull, d\ uu, sub1\ nn]$$

or

$$W = uull \wedge aa\ uull = a\ uu \wedge Ispartn[[da\ uull\ .\ d\ uull], d\ uu, nn]]$$

then by ISPARTN-DEF, APPEND-NIL, APPEND-UU, LENGTH-UU, COMBINE-ULL we have

$$W = uull \wedge length\ uull = nn \wedge combine\ uull = uu$$

Which completes the proof of ISPARTN-UU-NN.

Notice that ISPARTN-UU-NN implies that the partitions of *uu* can be expressed in terms of the partitions of d *uu*. Namely we take the partitions of d *uu* into *sub*1 *nn* parts and add <a *uu*> as the first element of each partition. To these we *append* the result of taking the partitions of d *uu* into *nn* parts and adding a *uu* to the first element of each partition. These two tacking procedures are implemented by the programs *tack*1 and *tack*0 defined by *PNLIST* and *PNNLIST* recursion respectively.

TACK1: *tack*1[*vv*, *pl*] ← if n *pl* then NIL else [*vv* . a *pl*] . *tack*1[*vv*, d *pl*]
TACK0: *tack*0[*x*, *pnnl*] ← if n *pnnl* then NIL else [[*x* . aa *pnnl*] . da *pnnl*] . *tack*0[*x*, d *pnnl*]

they are represented by the functions *tack*0 and *tack*1 satisfying

TACK0-DEF: $\forall vv\ pl{:}tack1[vv, pl] =$ if n *pl* then NIL else [*vv* . a *pl*] . *tack*1[*vv*, d *pl*]
*tack*0: *PNNLIST* → *PNNLIST*

TACK1-DEF: $\forall x\ pnnl{:}tack0[x, pnnl] =$ if n *pnnl* then NIL
else [[*x* . aa *pnnl*] . da *pnnl*] . *tack*1[*x*, d *pnnl*]
*tack*1: *PNLIST* → *PNNLIST*

and we have the immediate corollaries:

TACK1-NIL: $\forall vv{:}tack1[vv, NIL] =$ NIL
TACK1-PNNLL: $\forall vv\ pll{:}\ tack1[vv, pll] = [vv\ .\ a\ pll]\ .\ tack1[vv, d\ pll]$

TACK0-NIL: $\forall x{:}tack0[x, NIL] =$ NIL
TACK0-PNNLL: $\forall x\ pnnll{:}\ tack0[x, pnnll] = [[x\ .\ aa\ pnnll]\ .\ da\ pnnll]\ .\ tack1[x, d\ pnnll]$

The following facts about *tack*1 and *tack*0 correspond to the clauses of ISPARTN-UU-NN.

MEM-TACK1: $\forall vv\ pl\ W{:}[member[W, tack1[vv, pl]] \equiv$
$\exists uull{:}[W = uull \wedge a\ uull = vv \wedge member[d\ uull, pl]]\ ]$
MEM-TACK0: $\forall x\ pnnl\ W{:}[member[W, tack0[x, pnnl]] \equiv$
$\exists uull{:}[W = uull \wedge a\ a\ uull = x \wedge member[da\ uull\ .\ d\ uull, pnnl]]\ ]$

The proofs of MEM-TACK1 and MEM-TACK0 are similar. We will prove only MEM-TACK1. It is proved by using the PNLISTINDUCTION schema. In the case pl=NIL we must show

$$\forall vv\ W{:}[member[W, tack1[vv, NIL]] \equiv \exists uull{:}[W = uull \wedge a\ uull = vv \wedge member[d\ uull, NIL]]\ ].$$

$tack1[vv, NIL]$=NIL by TACK1-NIL and $\neg member[W, NIL]$ by MEMBER-NIL so both sides of the equivalence are false.

In the case $pl \neq$ NIL we let $pl = pll$ and we have

$member[W, tack1[vv, pll]]$
$\equiv W = [vv\ .\ a\ pll] \vee member[W, tack1[vv, d\ pll]]$ MEMBER-UU, TACK1-PNLL, SEXP
$\equiv W = [vv\ .\ a\ pll] \vee \exists uull{:}[W = uull \wedge a\ uull = vv \wedge member[d\ uull, d\ pll]\ ].$ induction
$\equiv \exists uull{:}[W = uull \wedge a\ uull = vv \wedge d\ uull = a\ pll] \vee$
$\exists uull{:}[W = uull \wedge a\ uull = vv \wedge member[d\ uull, d\ pll]$ $[vv\ .\ a\ pll] \in NEUULIST$
$\equiv \exists uull{:}[W = uull \wedge a\ uull = vv \wedge member[d\ uull, pll]]\ ]$ logic of $\exists$ and MEMBER-UU

Which completes the proof.

The program *partn* is given by the definition

PARTN: $partn[u, n] \leftarrow$
if n u then [if n equal 0 then <NIL> else NIL]
else [if n *equal* 0 then NIL else
$tack1[<a\ u>, partn[d\ u, sub1\ n]] * tack0[a\ u, partn[d\ u, n]]$

which is represented by the function *partn* satisfying

PARTN-DEF: $\forall u\ n{:}partn[u, n] =$
if n u then [if n equal 0 then <NIL> else NIL]
else [if n *equal* 0 then NIL else
$tack1[<a\ u>, partn[d\ u, sub1\ n]] * tack0[a\ u, partn[d\ u, n]]$

with the immediate corollaries

PARTN-NIL-ZERO: $partn[NIL, 0] = <NIL>$
PARTN-NIL-NN: $\forall nn{:}\ partn[NIL, nn] = NIL$
PARTN-UU-ZERO: $\forall uu{:}\ partn[uu, 0] = NIL$
PARTN-UU-NN: $\forall uu\ nn{:}\ partn[uu, nn] =$
$tack1[<a\ uu>, partn[d\ uu, sub1\ nn]] * tack0[a\ uu, partn[d\ uu, nn]]$

The correctness of *partn* is stated by

ISPARTN-COR: $\forall u\ n{:}\exists pl{:}[partn[u, n] = pl \wedge \forall W{:}[member[W, pl] \equiv ispartn[W, u, n]]]$.

We prove ISPARTN-COR by list induction.

In the case u=NIL there are two cases. If n=0 then $partn[u, n]$=<NIL>. Since <NIL> is of sort $PNLIST$ we need only show

$$\forall W:[member[W, \langle NIL\rangle]=Ispartn[W, NIL, 0]]].$$

which follow from MEMBER-NIL, MEMBER-UU, and ISPARTN-NIL-ZERO. If $n\neq 0$ then $partn[u, n]$=NIL which is also of sort $PNLIST$, so we need only show

$$\forall W:[member[W, NIL]=Ispartn[W, NIL, n]]]$$

which follows from MEMEBER-NIL, and ISPARTN-NIL-NN. This takes care of the case u=NIL.

In the case $u\neq$ NIL we let $uu=u$. If n=0 then $partn[uu, n]$=NIL and as NIL is of sort $PNLIST$ we need only show

$$\forall W:[member[W, NIL]=Ispartn[W, uu, 0]]]$$

which follows from MEMBER-NIL and ISPARTN-UU-ZERO.

If $n\neq$ NIL we let $n=nn$ and assume the induction hypothesis

$$\forall n:\exists pl:[partn[\mathbf{d}\,uu, n]=pl \wedge \forall W:[member[W, pl]=Ispartn[W, \mathbf{d}\,uu, n]]].$$

Now,

$$partn[uu, nn]=tack1[\langle \mathbf{a}\,uu\rangle, partn[\mathbf{d}\,uu, sub1\ nn]] * tack0[\mathbf{a}\,uu, partn[\mathbf{d}\,u, nn]]$$

and using the induction hypothesis we have for some pl and ql

$$partn[\mathbf{d}\,uu, sub1\ nn]=pl \wedge \forall W:[member[W, pl]=Ispartn[W, \mathbf{d}\,uu, sub1\ nn]]$$

and

$$partn[\mathbf{d}\,uu, nn]=ql \wedge \forall W:[member[W, ql]=Ispartn[W, \mathbf{d}\,uu, nn]]].$$

By ISPARTN-DEF-NN we have that every member of ql is of sort $NEUULIST$, so we may take $ql=qnnl$ to be of sort $PNNLIST$. Thus

$$partn[uu, nn]=tack1[\langle \mathbf{a}\,uu\rangle, pl] * tack0[\mathbf{a}\,uu, qnnl]$$

which is of sort $PNLIST$ using function mappings for $tack1$, $tack0$ and $append$. So we need only show

$$\forall W:[member[W, partn[uu, nn]]=Ispartn[W, uu, nn]]]$$

to complete the proof. This follows by the following argument:

member[*W*, *partn*[*uu*, *nn*]]
 =*member*[*W*, *tack*1[<a *uu*>, *pl*]] ∨ *member*[*W*, *tack*0[a *uu*, *qnnl*]] ;by MEMBER-UV
 =[∃*uull*:[*W* =*uull*∧a *uull*=<a *uu*>∧*member*[d *uull*, *pl*]] ∨
  ∃*uull*:[*W* =*uull*∧aa *uull*=a *uu*∧*member*[da *uull* . d *uull*, *qnnl*]]] ;by MEM-TACK1,MEM-TACK8
 =[∃*uull*:[*W* =*uull*∧a *uull*=<a *uu*>∧*Ispartn*[d *uull*, d *uu*, *sub*1 *nn*]] ∨
  ∃*uull*:[*W* =*uull*∧aa *uull*=a *uu*∧*Ispartn*[da *uull* . d *uull*, d *uu*, *nn*]]] ;by induction hypothesis
  =*Ispartn*[*W*, *uu*, *nn*]] ;by ISPARTN-UU-NN

This completes the proof. Note that we did not prove *partn* to be total as a separate step. This is because the clause ∃*pl*. *partn*[*u*, *n*]=*pl* says among other things that *PNLIST partn*[*u*, *n*].

Formal description of domains *UULIST*, *PNLIST*, *PNNLIST*

Thus we have the subdomain relations:

$$UULIST = NULL \cup NEUULIST$$
$$NULL \cap NEUULIST = \phi$$
$$UULIST \subset LIST$$
$$NEUULIST \subset NELIST$$

$$PNLIST = NULL \cup NEPNLIST$$
$$NULL \cap NEPNLIST = \phi$$
$$PNLIST \subset LIST$$
$$NEPNLIST \subset NELIST$$

$$PNNLIST = NULL \cup NEPNNLIST$$
$$NULL \cap NEPNNLIST = \phi$$
$$PNNLIST \subset UULIST$$
$$NEPNNLIST \subset NEUULIST$$

We will have additional variables ranging over the new sorts.

$$uul,\ vvl,\ wwl \in UULIST$$
$$uull,\ vvll,\ wwll \in NEUULIST$$
$$pl,\ ql,\ rl \in PNLIST$$
$$pll,\ qll,\ rll \in NEPNLIST$$
$$pnnl,\ qnnl,\ rnnl \in PNNLIST$$
$$pnnll,\ qnnll,\ rnnll \in NEPNNLIST$$

The function mappings for *car*, *cdr*, and *cons* on the new sorts is as follows:

a: $NEUULIST \to NELIST$
d: $NELIST \to LIST$
cons: $NELIST \times UULIST \to NEUULIST$

a: $NEPNLIST \to UULIST$
d: $NEPNLIST \to PNLIST$
cons: $UULIST \times PNLIST \to NEPNLIST$

a: *NEPNNLIST* → *NEUULIST*
d: *NEPNNLIST* → *PNNLIST*
cons: *NEUULIST* x *PNNLIST* → *NEPNNLIST*

Since *UULISTs* *PNLISTSs* and *PNNLISTs* are analogous to *LISTs* Thus we have recursion and induction on these domains. In particular we have the induction schemata:

UULISTIND: Φ NIL∧∀*uull*:[Φ d*uull* ⊃ Φ *uull*] ⊃ ∀*uul*:Φ *uul*
PNLISTIND: Φ NIL ∧ ∀*pll*:[Φ d*pll* ⊃ Φ *pll*] ⊃ ∀*pl*:Φ *pl*
PNNLISTIND: Φ NIL ∧ ∀*pnnll*:[Φ d*pnnll* ⊃ Φ *pnnll*] ⊃ ∀*pnnl*:Φ *pnnl* .

3. Exercises

This collection of exercises treats properties of association lists, substitutions, and pattern matching. These topics are strongly interrelated and the ability to treat properties of such programs is fundamental to the general area of symbolic computation.

1. Association lists

Prove the following facts about association lists.

1.1. ∀*u v*: [*alist u* ∧ *alist v* ⊃ *alist*[*u* * *v*]

1.2. ∀*z u v*: *alist u* ∧ *alist v* ⊃
[*assoc*[*z*, *u***v*] = if n*assoc*[*z*, *u*] then *assoc*[*z*, *v*] else *assoc*[*z*, *u*]]

where

alist u ← n*u* ∨ [¬at a*u* ∧ *alist* d*u*]]]

assoc[*x*, *s*] ← if n*s* then NIL else if *x* = aa*s* then a*s* else *assoc*[*x*, d*s*]

2. Properties of substitutions and substitution lists.

With function definitions as given below, *subst*[*x*, *y*, *z*] is the result of replacing the variable *y* by the S-expression *x* wherever *y* occurs in *z*. If *s* is a list of substitutions of the form *y* . *x* then *sublis*[*z*, *s*] is the result of "simultaneously" performing all of the substitutions on the list to *z*. If *s*1 and *s*2 are lists of substitutions then *s*1 • *s*2 is the "composition" of the two. Prove the following properties.

2.1. ∀*x y z*: *subst*[*x*, *y*, *z*] = *sublis*[*z*, <*y* . *x*>]

2.2. ∀*z s*1 *s*2: *sublis*[*z*, *s*1 • *s*2] = *sublis*[*sublis*[*z*, *s*1], *s*2]

2.3. ∀*z s*1 *s*2 *s*3: *sublis*[*z*, *s*1 • [*s*2 • *s*3]] = *sublis*[*z*, [*s*1 • *s*2] • *s*3]

2.4. ∀*x x*1 *y y*1 *z*: ((*y* ≠ *y*1 ∧ ¬*occur*[*y*, *x*1]) ⊃

$$subst[x1, y1, subst[x, y, z]] = subst[subst[x1, y1, x], y, subst[x1, y1, z]])$$

where

$$subst[x, y, z] \leftarrow \textbf{if } atz \textbf{ then } [\textbf{if } y = z \textbf{ then } x \textbf{ else } z] \textbf{ else } subst[x, y, az] \,.\, subst[x, y, dz]$$

$$sublis[x, s] \leftarrow \textbf{if } atx \textbf{ then } \{assoc[x, s]\}[\lambda z: \textbf{if } nz \textbf{ then } x \textbf{ else } dz] \textbf{ else } sublis[ax, s] \,.\, sublis[dx, s]$$

$$s1 \bullet s2 \leftarrow subsub[s1, s2] * s2$$

$$subsub[s1, s2] \leftarrow \textbf{if } ns1 \textbf{ then NIL else } [aas1 \,.\, sublis[das1, s2]] \,.\, subsub[ds1, s2]$$

3. Pattern matching

The function *inst* is one sort of "pattern matcher". We say that an S-expression x is an instance of another S-expression y if y can be transformed into x by replacing some of the atoms of y which satisfy *isvar* by suitable values. (All occurrences of the same variable should be replaced by the same value.) $inst[x, y, NIL]$ is NO if x is not and instance of the pattern y, and otherwise is the list of substitions which will convert y into x. In the latter case we have $x = sublis[y, inst[x, y, NIL]]$. Write a definition for *inst* and prove that

$$\forall x\ y\ u: (inst[x, y, u] \neq NO \supset x = sublis[y, inst[x, y, u]])$$

4. Unification

(This is a fairly substantial exercise.)

$unify[x, y]$ attempts to find the most general pattern which is an instance of both x and y. If no such pattern exists the it returns NO, otherwise it returns a list of substitutions which will convert both x and y into that pattern. Write a definition of *unify* and prove

4.1. $\forall x\ y: (unify[x, y] \neq NO \supset sublis[x, unify[x, y]] = sublis[y, unify[x, y]])$

4.2. $\forall x\ y: (unify[x, y] = NO \supset \forall s: sublis[x, s] \neq sublis[y, s])$

4.3. $\forall x\ y\ s: (sublis[x, s] = sublis[y, s] \supset \exists s1: \forall z: sublis[z, s] = sublis[z, unify[x, y] \bullet s1])$

Chapter V

LISP PROGRAMS WITH SIDE EFFECTS

Pure clean LISP programs such as we described in Chapters I and II simply compute some function of their arguments and have no effect on the environment in which they are executed. They admit the elegant mathematical characterization described and applied in Chapter III. Specifically, each recursively defined pure clean LISP program can be translated into a functional equation and minimization schema in a first order language, and the equation and schema can be used to prove that the program meets its extensional specifications or other mathematical properties of the function computed by the program.

However this is not the whole story. LISP systems also provide for the definition of sequential programs in addition to the recursive definition mechanism. A sequential program operates by performing a sequence of operations modifying the "state". Thus LISP systems also provide a variety of features for examining and modifying the state of the LISP world. These include assigning values to variables, more generally modifying the "property list" of an atom, destructive (as opposed to *con*structive) operations on list structure, even examining and modifying the LISP control mechanisms.

A sequential program can be viewed as computing a function by designating some property of the final state of the computation (such as the value of some variable) as the output. In general a LISP program will be used both to compute a function and to cause "side-effects". By using programs with side-effects in a larger program one can often compute answers more conveniently than in pure LISP. Side-effects can be used as a means of communication among a group of programs acting jointly to carry out a computation.

The mathematical properties of general side-effects are not well understood, although some techniques for treating certain classes of "well-structured" sequential programs have been developed. Programs that take totally uninhibited advantage of side-effects are usually difficult to understand, debug and modify. This has led to experienced programmers to use these features in moderation. That such programs are not accessible to present proof techniques is of secondary interest to most applied programmers. In the future we expect that language designers will reduce the need for side-effect programming by providing more direct means for achieving their results, and programmers will come to see debugging a correctness proof as a more worthwhile activity than debugging a program by examples, because it terminates with a conclusive proof that the program meets its specifications.

In this chapter we shall describe sequential programs in LISP and some additional features for side-effect programming such as arrays, property list operations, and destructive list structure operations. Every computation that can be done with these features can be done in pure clean LISP, but nevertheless there are often good reasons for using these features. We shall examine the features themselves and also the criteria that determine when they should be used in preference to pure clean LISP. Ideas for mathematical treatment of sequential programs and side-effects will be presented in a later chapter. We also postpone discussion of features for examining and modifying the LISP control structure until Chapter VI.

1. Sequential (ALGOL-like) programs.

Sequential programs are impure (by definition), but can be clean - provided certain restrictions are observed. The external notation for sequential programs is adapted from that of ALGOL 60.

As an example, we might write *reverse* as follows:

1.1)
```
reverse u ←
    program[[v]
        v ← NIL;
    a:  if nu then return v;
        v ← au . v;
        u ← du;
        go to a].
```

The internal form of the same program is

```
(DEFUN REVERSE (U)
    (PROG (V)
        (SETQ V NIL)
        A
        (COND ((NULL U) (RETURN V)))
        (SETQ V (CONS (CAR U) V))
        (SETQ U (CDR U))
        (GO A)))
```

where the paragraphing is only for the reader's convenience.

Sequential programs are introduced in LISP by allowing as a term an expression of the (external) form

program[[<variable list>] <statement list>],

where <variable list> is a list of variables local to the program, and <statement list> is a list of the statements of the program. In external notation, as in ALGOL 60, the statements are separated by semicolons, and any statement may be preceded by a label followed by a colon.

The statements are of the following kinds:

1. Assignment statements of the form

<left hand side> ← <right hand side>,

where <left hand side> is a variable, possibly subscripted, and <right hand side> is a LISP expression that can be evaluated.

2. go to statements of the form

go to <label>

where <label> is an atom or an expression that evaluates to a label. Since labels are atoms in internal notation, any expression evaluating to an atom may be used, but the usual case is a conditional expression wherein the second element of each pair is an atom. Should the resulting expression not be an atom or should that atom not be used as a label, an error message will be generated.

3. Conditional statements which have the form

if <p1> then <s1> else if ... else if <pn> then <sn>,

where the <pi> are propositional terms having truth values, and the <si> are any statements. Notice that conditional statements terminate with a then clause as do conditional expressions in general in internal form If none of the propositional terms are true the statement has no effect. (Unless of course there were side effects in the evaluation of the propositional terms.)

4. return statements of the form

return <expression>

where <expression> is an arbitrary LISP term. The effect of executing this statement is to return from the program giving the program as a term the value of <expression>.

In general the internal form of a program term is the following:

(PROG <variable list> <s1> ... <sn>)

and the four types of statements have the forms:

(SETQ <variable> <expression>)
(GO <label>)
(COND (<p1> <s1>) ... (<pn> <sn>))
(RETURN <expression>)

In addition a PROG statement may be an atom which is interpreted as a label. (GO <label>) transfers control to the list of statements beginning with the statement following <label>. Thus internal notation is again more uniform than external notation in treating labels as elements of the statement list rather than as attachments to statements.

In some LISP implementations arrays are treated specially and subscripted variables are not allowed as first arguments to SETQ. The STORE command must be used in order to change an array element. We will say more about arrays in the next section.

As further examples of how program might be used, here are some ways we might write *append*:

1.2) $u * v \leftarrow$ program[[] return if nu then v else au . [$du * v$]],

is just a trivial rewrite of the recursive definition.

```
u * v ←
    program[[w]
        if n u then return v;
        w ← a u . [d u * v];
        return w]
```

1.3)

is almost as close to the pure LISP form. If we want to replace the recursion by a loop, we can write

```
u * v ←
    program[[u1, v1, w]
        w ← NIL;
        u1 ← u;
        v1 ← v;
    a:  if n u1 then go to b;
        w ← a u1 . w;
        u1 ← d u;
        go to a;
    b:  if n w then return v1;
        v1 ← a w . v1;
        w ← d w;
        go to b].
```

1.4)

This corresponds to the recursive program

```
u * v ← app[u, v, NIL]

app[u, v, w] ←
    if n u then [if n w then v else app[u, a w . v, d w]]
    else app[d u, v, a u . w]
```

1.5)

which would have essentially the same "computation sequence" as the sequential program if it is compiled or interpreted iteratively (without using the stack).

As a final example we have a sequential program for computing the partitions of a list. A recursive program was given and proved correct in Chapter III. This example illustrates the difference between "top-down" (recursive) and "bottom-up" (sequential) thinking. Recall that the criterion for W to be a partition of a list u into n parts is

$$\forall W\ u\ n.(Ispartn[W, u, n] \equiv \exists uul.(W=uul \wedge length\ uul=n \wedge combine\ uul=u)).$$

To write the recursive program $partn[u, n]$ we listed the partitions for that base cases where $u=$NIL or $n=0$. Then we figured out how each partition could be constructed from a partition of a sublist of u (namely $d\,u$) and put it all together in a recursive definition. An alternate approach is to define a linear ordering on the set of partitions of a list u into n parts and then write a

program to enumerate them in order. For example, we could begin with the smallest and program a "successor" operation giving the next partition in the ordering. Since there are only a finite number of partitions, the program will eventually terminate with the largest.

We order partitions as follows: the smallest is obtained by making the first $n-1$ elements of the partition lists of length 1 and the nth element what ever is left after deleting the first $n-1$ members of u. The largest partition is a list in which the the first element is the list containing the first m elements of u, where m=*length* $u-(n-1)$ and the remaining elements are one element lists. In general, the partition w is greater than the partition v if at the first element where they disagree the element in w is longer than that of v. The program *prtn* given below enumerates partitions in this order. The auxiliary *maxpart* implements the definition of biggest partition given above. The loop A completes an initial fragment of a partition by adding lists of one element until the nth element is reached. The remainder of u (stored in v) is then added and the completed partition is added to the collector pl. The loop B is used to back up thru a partition until the first position where the length of the entry can be incremented is found.

```
prtn[u, n] ←
    program [pl, p, v, m]
        if n u ∧ n = 0 then return <NIL>
        else if n = 0 then return NIL
        else if length u < n then return NIL
        pl ← NIL
        p ← NIL
        v ← u
        m ← n
    A:  if m = 1 then [pl ← reverse v . p . pl, go to B]
1.6)    p ← <a v> . p
        v ← d v
        m ← sub1 m
        go to A
    B:  if maxpart[a pl, u, n] then return pl
        else if m < length v then
        [p ← [a p * <a v>] . d p, v ← d v, go to A]
        v ← [a p * v]
        p ← d p
        m ← add1 m
        go to B

maxpart[w, u, n] ← length u < length a w + n
```

In a later chapter we will see how this program can be proved to satisfy the same specification as the recursive version.

As promised, here are some circumstances in which sequential programs might be preferred to functional programs.

1 In some cases and for some people, it is easier to think about how to go from the initial conditions step-by-step to the final result than to think about how the final result can be obtained from easier cases. The two programs for computing partitions are examples of these two styles of

thinking and programming. There seems to be a matter of taste to a substantial extent, though the functional form seems to have clear conceptual advantages for functions like *append* and *subst*.

2. When we are considering a program that interacts with an environment instead of producing an answer, the sequential form often seems more natural.

3. When there are a large number of variables, it can turn out that the necessary functions have an unwieldy number of arguments. Frequently in such cases only a few of the variables actually change in any "loop", so it is inefficient to pass them all around.

4. A program for carrying out a test by searching a space of test cases may make a decision at some intermediate point and wish to abort the search at this point. Without the aid of special control mechanisms (such as *catch* and *throw* to be discussed in Chapter VI) the explicit loop structure of a sequential program is needed to do this efficiently.

Remarks:

1. Although return and go to statements in LISP have no meaning unless they occur inside a program, an assignment statement may be regarded as an program term. Thus $[x \leftarrow ay] \cdot z$ is an abbreviation for program[[] $x \leftarrow ay$; return x] . z.

2. Besides SETQ, most LISPs allow assignments of the form (SET <exp1> <exp2>), where <exp1> is evaluated to determine to what variable the assignment is to be made. If <exp1> doesn't evaluate to a variable, an error is signalled.

3. LISPs generally have special form of the PROG construct for frequently used special cases. For example

(PROG1 <s1> <s2> . . . <sn>)

does statements 1 to n in order and returns the value returned by <s1> and

(PROGN <s1> <s2> . . . <sn>)

does statements 1 to n in order and returns the value returned by sn.

4. In the conditional statement it is convenient to allow a list of statements as the then clause rather than a single statement. This often allows a more natural organization of the program and causes no difficulty in the rewriting of clean sequential programs as functional programs. In fact LISP systems usually allow this more general form of conditional, thus the conditional has the general form

(COND <p1 e11 ... e1k> ... <pn en1 ... enm>)

The value returned in the case one is wanted is the value of the last <eij> evaluated. This allows the simulation of a flowchart description of a sequential program without explicitly labeling the sequence of instructions that is to follow a particular branch.

5. In writing programs in external form we will often use the convention that each

statement is written on a separate line. This means that the ";" separator in not needed and may be omitted in this case.

2. Arrays in LISP

As mentioned in section §1 in addition to simple variables which may be assigned values using the assignment statement, LISP admits arrays. An array may in principle have any number of dimensions. If *foo* is a 2-dimensional (say 3 x 4) array then *foo*[*i*,*j*] can be thought of as a complicated variable name. It evaluates to the jth element of the ith row of *foo* and it can be assigned a value using the function *store*. Thus *store*[*foo*, <2, 3>, APPLE] assigns to row 2 column 3 of *foo* the value APPLE. Recall that in external form we allowed subscripted variables in assignment statements. Thus *store*[*foo*, <*i*,*j*>, *e*] can also be written *foo*[*i*,*j*] ← *e*. (*foo*[*i*,*j*] appearing on the left of an assignment denotes the position in the array while in other contexts it denotes the contents of that position.) It would not in principle be difficult to modify the SETQ of LISP to accept subscripted variables, although there may be practical reasons for not doing so.

Before using an array, it must be declared to LISP. (ARRAY FOO T 3 4) tells the LISP system that the atom FOO is to denote a 3 x 4 array of S-expressions. (The third item T specifies the type of the array. There are other possibilities for type such as FIXNUM or FLONUM in MACLISP. The form of the array declaration may also vary with implementation.) The internal forms for the above expressions for accessing and storing into arrays are (FOO I J) and (STORE (FOO 2 3) 'APPLE). Note that accessing an array element has the same form as a function call. LISP knows that FOO is an array because you declared it to be one using the ARRAY command. This is analogous to telling LISP about a program name using the DEFUN command.

In addition to accessing and storing primitives for operating on arrays, LISP usually provides a means of convertings lists to arrays and vice versa. In MACLISP they are called FILLARRAY and LISTARRAY. The following conversation with MACLISP will declare a fixnum array TIC and fill it with alternating 1s and 0s.

```
user:     (ARRAY TIC FIXNUM 3 3)
LISP:     TIC

user:     (FILLARRAY 'TIC '(1 0 1 0 1 0 1 0 1))
LISP:     TIC

user:     (LISTARRAY 'TIC)
LISP:     (1 0 1 0 1 0 1 0 1)

user:     (TIC 1 0)
LISP:     0

user:     (TIC 1 1)
LISP:     1
```

Notice that (i) the value of FILLARRAY is the name of the array and not the array and (ii) the indices start at 0. Note also that general arrays (of type T) can arbitrary entries including array names and arrays pointers.

Arrays are are practical for two reasons. One is quick access to individual elements, the other is that they are updated rather than copied which saves space. One could compare the use of lists vs arrays for storing data to the use of tapes vs disks. Lists are somewhat more flexible than tape in that you can splice and chop somewhat more readily but the sequential access vs direct access analogy is good.

In LISP arrays might be used to represent game boards in game playing programs, or to represent a picture as a "pixel" array. We will see an exmple of the former in Chapter VII. As to the latter, the program *picture find* scans a *picture* represented as an array (say of colors) for all and returns a list of all locations where a given picture specification *pat* is satisfied. The program *pmatch* checks to see if the specification *pat* is satisfied at *pic*[*i*,*j*]. A specification is a list of triples <*rd cd pred*>. It is satisfied at *pic*[*i*,*j*] if each of the triples is. A triple is satisfied if the location <*i*+*rd*,*j*+*cd*> is within the picture and *pred*[*pic*[*i*+*rd*,*j*+*cd*]] holds.

```
pmatch[pat, pic, i, j] ←
    program [s, p, r, c, m, n]
        m ← sub1 adarraydims pic
        n ← sub1 addarraydims pic
        p ← pat
    LOOP:
        if np then return T
        s ← ap
        as
        if r < 0 ∨ m < r then return NIL
        c ← j + ads
        if c < 0 ∨ n < c then return NIL
        if apply[adds, <apply[pic, <r, c>]>] then [p ← dp, go to LOOP]
        return NIL
```

2.1)

```
picture find[pic, pat] ←
    prog [i, j, m, n, locs]
        m ← sub1 adarraydims pic
        n ← sub1 addarraydims pic
        i ← 0
        j ← 0
        locs ← NIL
    LOOP:
        if pmatch[pat, pic, i, j] then locs ← [i . j] . locs
        if j < n then [j ← add1 j, go to LOOP]
        if i < m then [i ← add1 i, j ← 0, go to LOOP]
        return locs
```

For example if we define the array PIX as follows

```
(ARRAY PIX T 6 6)

(FILLARRAY 'PIX
        (W R W G W B)
        (R W G W B W)
        (W G W B W R)
        (G W B W R W)
        (W B W R W G)
        (B W R W G W) )
```

and the specification PAT as follows

```
(SETQ S1 '(0 0 (FUNCTION (LAMBDA (X) (EQ X 'W))) ) )
(SETQ S2 '(0 1 (FUNCTION (LAMBDA (X) (EQ X 'R))) ) )
(SETQ S3 '(1 0 (FUNCTION (LAMBDA (X) (EQ X 'R))) ) )
(SETQ S4 '(-1 0 (FUNCTION (LAMBDA (X) (EQ X 'B))) ) )
(SETQ S5 '(0 -1 (FUNCTION (LAMBDA (X) (EQ X 'B))) ) )

(SETQ PAT (LIST S1 S2 S3 S4 S5))
```

then the list of locations satifying the specification is obtained by

```
(PATTERN FIND 'PIX PAT)

((4 . 2) (3 . 3) (2 . 4))
```

We can begin to characterize arrays in LISP by modifying the characterization of assignment and contents functions on state vectors given in McCarthy [1962b]. Thus we have functions $a[array, pos, exp]$ and $c[array, pos]$ where a returns the array resulting from $store[array, pos, exp]$ (in LISP *store* returns the value of exp) and c returns the contents of *array* at *pos* (like writing $array[pos]$). The functions a and c satisfy the following relations

2.2) $c[a[array, pos, exp], pos1] = \text{if } pos=pos1 \text{ then } exp \text{ else } c[array, pos1]$

2.3) $array = a[array, pos, c[array, pos]]$

2.4) $a[a[array, pos0, exp0], pos1, exp1] = \text{if } pos0=pos1 \text{ then } a[array, pos0, exp1]$ else $a[a[array, pos1, exp1], pos0, exp0]$

These equations characterize the abstract notion of array. We also need to extend the notion of equality to arrays in the obvious way. (Namely two arrays are equal if and only if the they have the same set of allowed positions and the contents of any allowed position is the same for both arrays.) They do not deal with the fact that the *store* function of LISP destroys the old array in the process of creating the new version. They can be used to prove propertied of programs that have array type arguments.

For additional array functions and precise specification of features the reader should consult the programmer's manual for the version of LISP being used.

3. Defining macros in LISP

A macro is a form of shorthand or "syntactic sugar". A macro definition tells LISP that when it sees the defined form it should first "decode" it obtaining the intended expression and then evaluate the result. For example, we might wish to have a genuine if-then-else conditional. Thus we would write

```
(IF (ATOM X) (LIST X) 'NOT-AN-ATOM-X)
```

and the expression evaluated would be

```
(COND ((ATOM X) (LIST X)) (T 'NOT-AN-ATOM-X)).
```

The reason for using macros is generally to write programs that are more compact and suggestive of their intent. For example defining

```
(BODY <e>)
```

which returns the body of a λ-expression to be

```
(CADDR <e>)
```

and other such mnemonic devices might make *eval* more readable. You might ask why not just define the appropriate functions? There are a couple of reasons. One is that compilers can be made to expand simple macros before compiling the code, the eliminating one or more levels of function call for each instance. Also in the case of IF a function definition (of the ordinary sort) would not work as only one of the latter two arguments to IF should be evaluated not both. It is also the case that the treatment of macros reflects the intent of such definitions more accurately that function definitions.

The basic macro facility of LISP is very simple. A definition has the form

```
(DEFUN <macname> MACRO (<var>) <macbody>).
```

This has the effect of associating with <macname> a MACRO property which is a λ-expression with a single variable <var> and body <macbody>. When LISP encounters a term whose first element is an atom with a MACRO property then <macbody> is evaluated with <var> bound to the entire term. This produces the expression whose value is desired and that expression is evaluated to determine the value of the term. For example the BODY macro is defined by

```
(DEFUN BODY MACRO (L) (LIST 'CADDR (CADR L)) )
```

and the IF macro by

```
(DEFUN IF MACRO (L) (LIST 'COND (LIST (CADR L) (CADDR L)) (LIST T (CADDDR L))))
```

Then when LISP encounters

```
(BODY '(LAMBDA (X) X))
```

it will macro expand this to

```
(CADDR '(LAMBDA (X) X))
```

which then evaluates to X. When it encounters

```
((LAMBDA (X) (IF (ATOM X) (LIST X) 'NOT-AN-ATOM)) 'AB)
```

It will evaluate

```
(COND ((ATOM X) (LIST X)) (T 'NOT-AN-ATOM))
```

in an environment with X bound to AB and return (AB). As a slightly more complex example consider extending *cons* to an arbitrary number of arguments by right associating. Thus *rcons*[*x*1, *x*2, *x*3] stands for *cons*[*x*1, *cons*[*x*2, *x*3]]. This can be done by the following macro definition:

```
(DEFUN RCONS MACRO (L)
  (COND ((NULL (CDDR L)) (CADR L))
        (T (LIST 'CONS (CADR L) (CONS 'RCONS (CDDR L)))) ))
```

Several of the built-in LISP functions that take arbitrary numbers of arguments are in fact implemented as macros based on functions taking 2 arguments. PLUS and TIMES are two examples.

Although the macro facility is very powerful (we have not even touched the surface with these examples) it is rather clumsy to use for the simple kinds of macros we have been discussing. A simple but useful form of macro definition is the following:

```
(MACDEF <macform> <macbody>).
```

Where <macform> is a list with the macro name as the first element and parameters following. When a macro defined in this way is encountered the actual arguments are substituted for the parameters in <macbody> and the resulting expression evaluated for the value. The BODY and IF macros can be defined this way as follows:

```
(MACDEF (IF A B C) (COND (A B) (T C)))

(MACDEF (BODY E) (CADDR E))
```

Much easier! The MACDEF feature can be implemented as a macro-macro which converts the higher level definition into a low level definition (now we are beginning to see some of the power) as follows:

```
(DEFUN MACDEF MACRO (L)
  (LIST 'DEFUN
        (CAADR L)
        'MACRO
        '(%L)
        (LIST 'SUBLIS
              (LIST 'PRUP (LIST 'QUOTE (CDADR L)) (LIST 'CDR '%L) )
              (LIST 'QUOTE (CADDR L))) ))
```

where PRUP is the program for converting a pair of lists into an a-list given in Chapter I §13.

A somewhat more sophisticated macro-macro has the form

(PMACDEF <macpattern> <macbody>)

Where <macpattern> has the form (<macname> <p1> . . . <pn>) and the <pi> are elementary patterns which are either atoms whose pname begins with ?, which match an arbitrary atom, atoms whose pname begins with * which match list segments or any other atoms which match only themselves. When a term having a macro definition of this form is encountered, the argument list is matched to the pattern sequence. Elements matching ? variables are substituted for the corresponding variable in <macbody> while list segments matching * variables are spliced into lists where the corresponding variable occurs in <macbody>. For example

```
(PMACDEF (LET *vars BE *vals IN ?body) ((LAMBDA (*vars) ?body) *vals))
```

implements the usual LET construct for variable scoping.

The PMACDEF facility can be implemented as a macro-macro written in terms of a suitable pattern matcher and pattern instantiater. Clearly the more sophisticated the pattern matcher is, the more sophisticated this macro definition form will be. We will have more to say about this in the chapter on pattern matching.

[Remark: MACDEF and PMACDEF are simplified versions of macro-macros available in MACLISP.]

As we see, macros provide the programmer with the ability to implement both data structures and high level program constructs. One possible point of view in language design is to provide a very simple basic language (such as pure LISP) and a powerful macro definition facility interms of which a higher level languages can be implemented. With a good compiler this need not reduce the efficiency of compiled programs and provides the programmer with great flexibility. The RABBIT compiler for SCHEME [Steele 1978] implements this point of view.

Exercises

1. What is the value of (RCONS 'A 'B 'C)?

2. Write a definition for the macro LCONS that applies CONS to an arbitrary number of arguments using left association rather than right association. Thus *lcons*[*x*1, *x*2, *x*3]=*cons*[*cons*[*x*1, *x*2], *x*3].

4. Property lists

Atoms were introduced as the basic elements from which more complex S-expressions are CONStructed. Apparently as the name suggests atoms are without any finer structure. However, when used in the S-expression representation of LISP programs, symbolic atoms (e.g. non-numeric atoms, henceforth called symbols) have various kinds of information associated with them. Symbols representing variables need associated values, those naming functions or macros need associated definitions. LISP keeps track of this information in a simple and uniform manner using what is know as a property list. Each symbol has an associated property list which can be thought of as an ordinary list associating property names with property values. For example the value of a variable can be stored as the value associated with the symbols VALUE property, while the definition of a function can be stored under the EXPR property. Other properties typically used by LISP include PNAME (print name), FEXPR, LEXPR, SUBR, FSUBR, LSUBR, (various flavors of function definitions), MACRO and ARRAY. The collection of symbols and their property lists are LISPs version of a symbol table.

We will say more about the properties used by LISP when we discuss the *eval* component of LISPs Top-level in Chapter VI. We will also discuss the computer representation of symbols in more detail in that chapter.

For the present we will concentrate on how the programmer can make use of property lists. We have already used the operations DEFUN and DEFPROP for introducing function definitions. The basic operations for examining and modifying property lists are:

```
(PLIST <atom>)
(GET <atom> <property>)
(PUTPROP <atom> <value> <property>)
(REMPROP <atom> <property>)
```

where <atom> should evaluate to an atom, <property> must evaluate to an atom which names the property, and <value> can be any LISP expression that can be evaluated. [In some LISPs a property list is also an accepted value for <atom>.] The exact form of these terms is implementation dependent, in particular the order in which the arguments are expected may vary.

(PLIST 'A) is the property list of the atom A.

GET returns the <property> value (if any) associated with <atom>. PUTPROP puts a property name <property> followed by the property value <value> on the property list of <atom>, and REMPROP removes a property. DEFPROP is like PUTPROP except that the arguments are not evaluated. Thus if (PUTPROP 'C 12 'AT-WT) is executed then (GET 'C 'AT-WT) will return 12. If (REMPROP 'C 'AT-WT) is executed then (GET 'C 'AT-WT) will return NIL as the AT-WT property is no longer on the property list of C. If (DEFPROP APPLE RED COLOR) is executed then (GET 'APPLE 'COLOR) will return RED. Thus DEFPROP can be used to put arbitrary properties on property lists. The difference between DEFPROP and PUTPROP is that DEFPROP does not evaluate its arguments, but PUTPROP does.

From a mathematical point of view PUTPROP acts like an assignment statement and REMPROP like a special kind of an assignment statement. GET gets the value of a variable. However, the mathematical properties of these LISP operations haven't been systematically studied.

One use of property lists in programming is in the creation and modification of networks of data. Values attached to atoms point to expressions that include other atoms. A program manipulating chemical structures might use property lists to associate such information as valence, atomic weight, nuclear spin, etc. with each chemical atom of interest. It could also treat fragments as quasi atoms (chemical atoms, that is) by naming them, putting the structure of the fragment on the property list, as well as analogs to other atomic properties. These properties can then be used as required by programs for building or analyzing complex molecules. Another example is the program used to print the LISP programs appearing in this book in external form. Each special LISP atom (such as CAR, COND, APPEND, QUOTE, ...) has on its property list the name of the program used to compile the construct it signals, type information, external print name (so that CAR prints as a for example) and other control information. This is an example of "data driven" programming. Another example of a data driven program would be a compiler that looked up the program to compile special constructs on the COMPFN property of the atom signalling that construct. Such a compiler is quite flexible and allows you to extend the language easily. Another example of this style of program is the *editor* given in Chapter VI.

5. Pseudo-functions that modify list structures.

In pure LISP, list structure is never changed. *car* and *cdr* merely move about in a list structure, and *cons* creates new list structure from the free storage list. Of course, a variable can get a new value that corresponds, for instance, to putting an element on the end of a list, but in pure LISP this can only be accomplished by creating new list structure. In this section we will discuss operations that actually modify list structure. They often have efficiency advantages, but their use interferes with saving memory by using merging list structure, and techniques for proving correctness of such programs are quite undeveloped.

The simplest way to express operations that modify list structure is in the form of assignments to *car–cdr* chains of variables, e.g. we may write $\mathrm{ada}\, x \leftarrow y$. z. The effect of executing this statement is to replace the ada part of the list structure pointed to by x with the value of the right hand side. As a term, the statement takes the updated value of the first argument. In internal notation, we use the pseudo-functions (RPLACA X Y) and (RPLACD X Y) which correspond to the statements $\mathrm{a}\, x \leftarrow y$ and $\mathrm{d}\, x \leftarrow y$ respectively.

RPLACA and RPLACD are highly unclean. The effect of evaluating an expression involving RPLACA or RPLACD depends on the state of list structure and not merely on the S-expressions that the variables have as values. Consider the programs *test*1 and *test*2 given by

```
test1[] ←
    program[[]
        x ← (A B);
        y ← (A B);
        z ← [adx ← C] ]

test2[] ←
    program[[]
        x ← (A B);
        y ← x;
        z ← [adx ← C]]
```

After executing *test*1 we have x=(A C), y=(A B), and z=(C), while after executing *test*2 we have x=(A C), y=(A C), and z=(C). This is because in the first case x and y are different list-structures, so modifying doesn't change y. In the second case x and y are the same list-structures (e.g. x eq y) so any modification of x also affects y. In fact if x and y are not the same, but share some substructure, any modification of that structure will affect both x and y. The uncleanliness of RPLACA and RPLACD means that the programmer must know exactly what list structures merge.

Here are the internal forms of the test programs in case you are unsure of the notations:

```
(DEFUN TEST1 ()
  (PROG ()
    (SETQ X '(A B))
    (SETQ Y '(A B))
    (SETQ Z (RPLACA (CDR X) 'C))
  ))

(DEFUN TEST2 ()
  (PROG ()
    (SETQ X '(A B))
    (SETQ Y X)
    (SETQ Z (RPLACA (CDR X) 'C))
  ))
```

A typical application is the pseudo-function *nconc* defined by

```
nconc[u, v] ←
    program[[w]
        if n u then return v;
        w ← u;
loop:
        if n d w then [d w ← v; return u];
        w ← d w;
        go to loop].
```

5.1)

nconc can also be given a definition that has the form of a recursive program, namely

5.2)
```
nconc[u, v] ← if n u then v else nconc1[u, u, v]

nconc1[u, w, v] ←
    if n d w then [λside: u][rplacd[w, v]]
    else nconc1[u, d w, v].
```

The value of *nconc*[*u*, *v*] is just $u * v$, but it achieves this result by modifying the last element of *u* to point to *v*. It is typically used when no other variable points to a list structure that merges into *u*, and the old value of *u* will not be used again. In that case, *nconc* is advantageous, because it does no *cons*es, and thus can't initiate garbage collection. No formal methods exist at present for proving that these conditions are met.

RPLACD can also be used to insert an element into the middle of a list. Suppose, for example, that we wish to insert the atom B after every occurrence of the atom A in a list *u*. We can define a pure LISP function that gives a list obtained from the list *u* by inserting such B's as follows:

5.3) *insertb u* ← if n *u* then NIL else if a *u* eq A then A . [B . *insertb* d *u*] else a *u* . *insertb* d *u*.

Computing *insertb u* does not change the value of *u*, because new list structure is manufactured. On the other hand, if we define

5.4)
```
insertb u ←
    program[[w];
        w ← u;
    loop:
        if n w then return u;
        if a w eq A then [d w ← B . d w; w ← d w];
        w ← d w;
        go to loop],
```

we get a program that does the actual insertions. It is more efficient because it uses fewer *cons*es, but it will damage any list structure that merges with the structure representing *u*, and it is mathematically recalcitrant.

Finally we have two examples of programs destructively removing items from a list. The program *remv* deletes all occurrences of *x* from the list *u*. To remove something from a list destructively the *cdr* of the previous element is rplaced by the pointer to the next element. If there is no previous element, the list is replaced by its *cdr*. Thus the two loops of *remv*.

```
remv[x, u]
    program[[v]
    LEADERS:
        if n u then return u;
        if a u eq x then [u ← d u, go to LEADERS];
        v ← u;
    INTERIORS:
5.5)    if n d v then return u;
        if [ad v eq x] then [d v ← dd v; go to INTERIORS];
        v ← d v;
        go to INTERIORS]
```

The program *prune* deletes all elements of the list u that are members of the list *seen*. Instead of using two loops, we tack a dummy onto u so there will always be a previous element.

```
prune[u, seen]
    program[[v]
        u ← NIL . u;
        v ← u;
    ploop:
5.6)    if n d v then return d u;
        if [ad v ∈ seen] then [d v ← dd v; go to ploop];
        v ← d v;
        go to ploop]
```

In order to make the removals go uniformly we begin by tacking a dummy element NIL onto u. Thus the list is non-empty and the next element of interest is the "second" element of the list currently pointed to. An example of a program using destructive operations in a controlled and "clean" manner is the *editor* given in Chapter VI. Here the program to be edited is initially copied, then destructively edited to save repeated and unnecessary reconstruction.

Exercise

1. Consider the following test programs:

```
test3[] ←
    program[[]
        x ← (A B);
        y ← x;
        return x equal [λside.x][a y ← C]]

test4[] ←
    program[[]
        x ← (A B);
        y ← x;
        return copy x equal [λside.x][a y ← C]]

copy x ← if at x then x else [copy car x . copy cdr x]
```

We have *copy* x equal x =T and *test*3[]=T but *test*4[]=NIL. Explain this.

6. Re-entrant List Structure.

So far we have only considered list structure in which all *car–cdr* chains terminate in atoms. Mathematically, infinite list structures are also possible. They can't be represented in a computer, but since they can satisfy the LISP algebraic axioms, we have had to exclude them in our axiomatization of LISP by axiom schemata of induction. Another possibility is the re-entrant list structure. These can be created by the RPLACA and RPLACD operations. Figure 9 shows some examples of re-entrant list structures. For example, if the variable x has value (A), executing the statement $ax \leftarrow x$ gives rise to a circular structure (i) while if we execute $dx \leftarrow x$ we get the structure (ii). If we execute $dx \leftarrow x$ with x as in (i) or $ax \leftarrow x$ with x as in (ii) we get the doubly re-entrant structure (iii). Notice that in (i) we have x eq ax, in (ii) x eq dx and in (iii) we have x eq ax and x eq dx. These equations contradict the induction schemata for lists and S-expressions. Thus list structures with destructive operations are not a model of the theory of lists and S-expressions given in Chapter III.

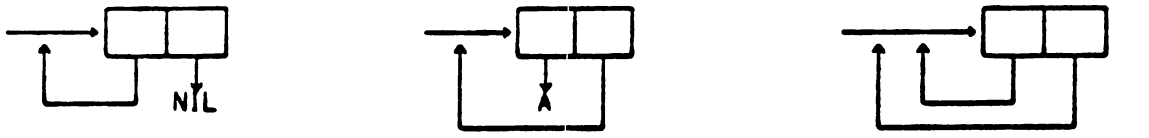
We can simulate recursive programs by replacing calls to the program by a pointer to the lambda expression defining the program. For example if we execute *mkleft* then the resulting value of *left* will have the structure shown in Figure 10.

```
mkleft[]
    program[[]
        left ← (LAMBDA (X) (COND ((ATOM X) X) (T (LEFT (CAR X))) ));
        aadaddadd left ← left;
        return LEFT]
```

Also *apply*[*left* <*x*>] = *left*[*x*] where

6.1) $$left\ x \leftarrow \text{if } atx \text{ then } x \text{ else } left\ ax.$$

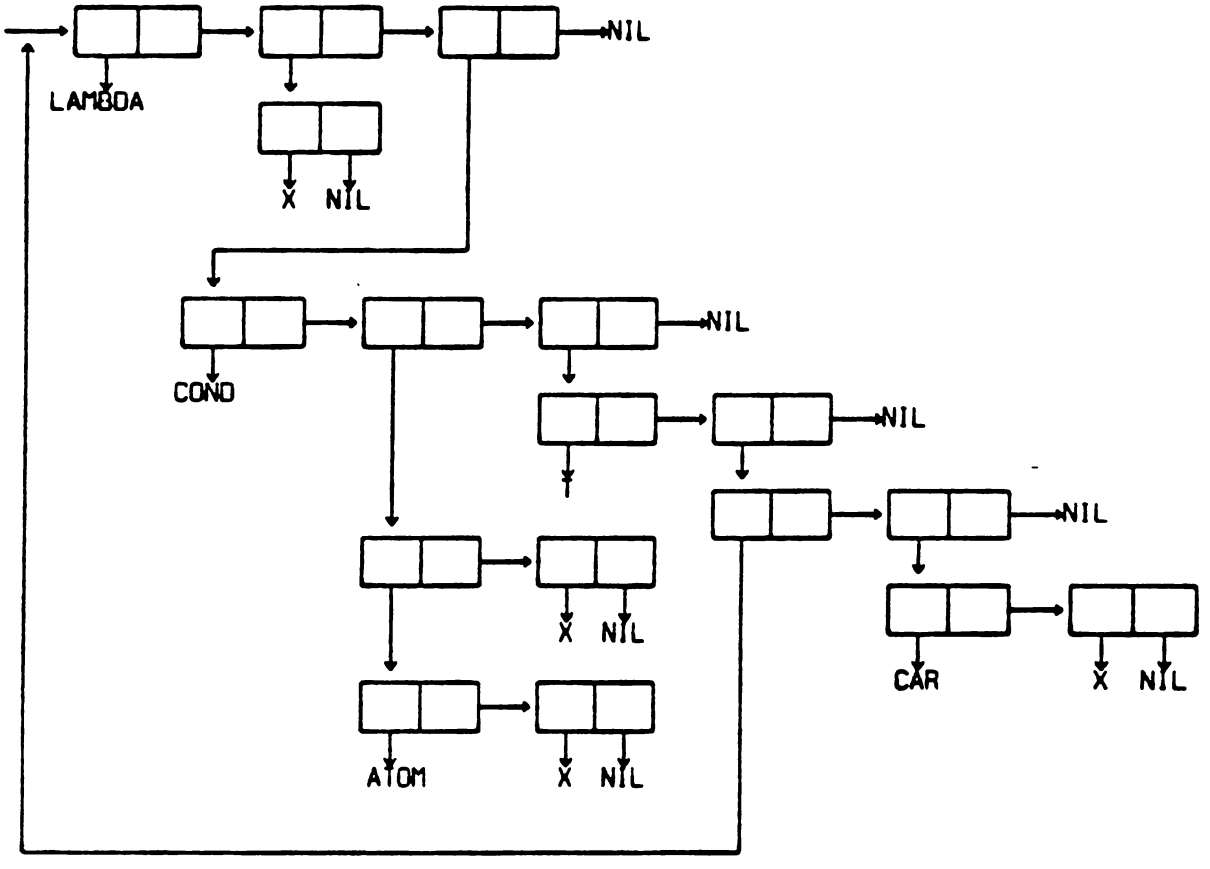
The reason for encasing the construction in a program even though the assignments were made to non-prog variables was that most LISPs (including MACLISP) do not check for re-entrant structures and an attempt to print a re-entrant list will cause an infinite loop.



(RPLACA X X) (RPLACD X X) (RPLACA (RPLACDA X X))

(i) (ii) (iii)

Figure 9. Examples of re-entrant list structures.




```
(SETQ LEFT '(LAMBDA (X) (COND ((ATOM X) X) (T (LEFT (CAR X))) )))
           (RPLACA (CADR (CADDR (CADDR LEFT))) LEFT)
```

Figure 10. Another re-entrant list structure.

As we have just noted, re-entrant list structures do not satisfy the induction axioms given for S-expressions, and the correctness of most of programs given in this book depends on the data being non-re-entrant. Re-entrant structures can be represented by S-expressions provided certain atoms are reserved for use as labels and a suitable convention is adopted for distinguishing ordinary atoms from labels and pointers to the labels. For example, the structure of figure 9 (iii) might be represented by `((LABEL A).((POINT A).(POINT A)))`.

Programs that compute with re-entrant structures need to keep lists of vertices they have visited so that they can tell when they are about to re-enter. Thus the equivalence of merging list structure can be tested by the predicate *equiv* defined by:

```
equiv[x, y] ← not [equiv1[x, y, NIL] eq LOSE]

equiv1[x, y, u] ←
    if u eq LOSE then LOSE
    else if x eq y or match[x, y, u] then u
    else if atx or aty or unmatch[x, y, u] then LOSE
    else equiv1[ax, ay, equiv1[dx, dy, [x . y] . u]]

match[x, y, u] ← not nu and [[x eq aau and y eq dau] or match[x, y, du]]

unmatch[x, y, u] ← not nu and [x eq aau or y eq dau or unmatch[x, y, du]]
```

6.2)

The argument u in *equiv1* is a list of positions pairs seen so far in the parallel traversal of x and y. It is necessary to carry around information of this sort in order to avoid repeating some part of the computation forever. Note also the use of eq. The test we want to make is "have we been here before?" not "have we seen this S-expression before?" as the program *equal* for testing the latter would loop on re-entrant structures.

Exercise

1. Consider the following definition of the fibonnaci function.

```
fibon n ← if n eq 0 or n eq 1 then 1 else fibloop[n, (1 1)]

fibloop[n, l] ←
    if nddl then
        [if n eq 2 then adrplacd[dl, <al + adl>]
         else fibloop[sub1 n, rplacd[dl, <al + adl>]]]
    else if n eq 2 then addl else fibloop[n-1, dl]
```

6.3)

After evaluating *fibon* 5 if we examine the definition on the property list of *fibon* it corresponds to the definition

```
fibon n ← if n eq 0 or n eq 1 then 1 else fibloop[n, (1 1 2 3 5 8)].
```

Thus *fibon* seems to be getting smarter. What has happened?

7. Exercises.

Sequential programs

1. Extend *eval* (I.13.1) to handle PROG, SETQ, GO, and RETURN. Is it necessary to change the evaluation of CONDs? There are several possible ways of handling labels. Try to think of at least two and consider the advantages and disadvantages of each.

Property list and Special properties.

2. The association-list argument of *eval* I.13.1 provides a way of associating a value property with any atom. This simple version of *eval* does not distinguish between values that are intended to be applied as functions and simple values. Replace the a-list argument by a more general structure capable of associating arbitrary properties with an atom. Write appropriate accessing and modifying functions (*get*, *putprop*, and *remprop*) and modify *eval* where necessary, including a clause for DEFUN.

3. How might you extend the more sophisticated *eval* above to handle PROG and related constructs. Are there now new ways of treating statement labels?

Macro definitions

4. The macro definition feature of LISP described earlier is some what clumsy to use in general. Write your own macro definition maker that takes a macro name, a parameter specification and a definition body and produces an appropriate definition. The parameter specification could be a pattern containing variables that are to match various kinds of expressions such as atoms, segments of lists, arbitrary expressions, expressions satisfying some predicate. The pattern would also contain constants, etc.. The macro definition produced would then generate code by matching the call to the pattern to obtain values of the variables and then filling them in where they occur in the body. For example given a definition such as

```
(MACRO-MAKER FOR (?I IN ??LIST DO *STMTS)
    (PROG (?I L)
        (SETQ L ??LIST)
       LOOP
        (COND ((NULL L) (RETURN NIL)))
        (SETQ ?I (CAR L))
        (SETQ L (CDR L))
        *STMTS
        (GO LOOP)) )
```

A call to this macro such as

```
(FOR X IN '(A B C) DO
  (COND ((GET X 'FN) (APPLY (GET X 'FN) NIL)))
  (PRINT X))
```

would result in the following code to be evaluated

```
(PROG (X L)
    (SETQ L '(A B C))
   LOOP
    (COND ((NULL L) (RETURN NIL)))
    (SETQ X (CAR L))
    (SETQ L (CDR L))
```

```
(COND ((GET X 'FN) (APPLY (GET X 'FN) NIL)))
(PRINT X)
(GO LOOP))
```

The task then is to write the program *macro-maker* which will put a suitable macro definition on the property list of the indicated atom.

Structure modifying functions

5. Write a function that reverses a list by reversing the pointers and thus not copying the list. How could you prove your function correct? (The latter is a non-trivial task to carry out formally.)

Re-entrant list structures.

6. Write a program to list the atoms in a possibly re-entrant list structure.

7. Write a program to translate a graph description from the notation of Chapter I to a re-entrant list structure. Write a program to translate the other way.

8. Write a function *mk-label* which takes a name (atom) and an expression as arguments and replaces all occurrences of the name by a pointer to the expression itself. When applied to a lambda-expression this creates a self-referential expression which when applied to an appropriate list of arguments returns the same value as if *label*[*name*, *expression*] had been applied to the same arguments thus simulating the *label* construct.

9. Write functions to translate in each direction between possibly re-entrant structures and the S-expressions corresponding to them using the LABEL, POINTER notation described above.

10. Write an axiom schema of induction for finite possibly re-entrant list structures. Hint: The schema can be based on an assertion of convergence of a program that scans a list structure keeping track of the vertices it has already visited.

Chapter VI

HOW LISP WORKS

This chapter explains what it takes to make LISP actually run on a computer. It isn't a manual for any particular system, but understanding the basic components of LISP as a interactive programming system will help understand and use any LISP system. We discuss components common to essentially all LISP systems and give some hints about the variety of possible implementations.

The organization of the chapter is "bottom up". We begin with as discussion of LISP objects -- the basic data structures understood and manipulated by a LISP system. Next we describe the three main components -- *read*, *eval* and *print* -- which make up the "top-level" of LISP. This "read-eval-print loop" is generally what you talk to when running LISP interactively. In principle this is all that is needed. However, in order to interact productively with a programming system we require of it more than just the ability to run programs. Most LISP systems have facilities for making output more readable -- known as pretty-printing, for editing programs, for detecting and reporting errors, for aiding in the debugging process and for interacting with the host system -- external file manipulation for example. The ability to run programs interpretively and the very simple syntax of LISP make it possible to have very elegant and flexible editing, error-handling, and debugging facilities as an integral part of the system. We present a simple editor based on the MACLISP editor and indicate several possible extensions. We discuss how tracing and stepping through the execution of a program can be done in LISP. Finally there is a brief section on I/O. This aspect of LISP (or any system) is necessarily implementation dependent. However there are some basic operations common to most LISPs and we present (an abstraction of) them.

[Remark: in the description of the basic LISP structures, we refer occasionally to programs which aren't directly available to the user. They may or may not be explicitly implemented. Mainly, they are convenient for purposes of explanation.]

1. LISP objects.

A LISP system deals with a variety of objects in the process of representing S-expressions as list-structures and carrying out computations on these structures. These include pnames (corresponding to the character string representation of atoms) numeric and symbolic atoms, oblists (LISPs version of a symbol table) and pointers of various flavors. In this section we will discuss the construction and basic operations on these objects.

Pnames.

Each atom has a print name (pname for short) which is the link between the internal representation of atoms and the external representation as a character string. The key component of the LISP read program is *readatom* which converts the character string representing an atom externally to the corresponding pname. The pname is used to determine the internal

representation of the atom. Correspondingly, the key component of the LISP print program is *printatom* which converts the pname to the corresponding external (character string) representation. Clearly if the mapping between representations is to be correct we must have, for atoms a and character strings c, *readatom*[*printatom*[a]]=a and *printatom*[*readatom*[c]]. Generally pnames are not directly accessible to the user. They are however, important properties of atoms and several of the basic operations are conveniently described using pnames as intermediate objects.

[Remark: *readatom* and *printatom* are not programs typically available directly to the user. See remark at the end of the introduction to this chapter.]

Numeric Atoms.

LISP distinguishes two kinds of atoms, numeric and symbolic. Numeric atoms are generally assumed to denote particular numbers and can be recognized, represented and printed without a lot of additional information. Thus, although numbers are classed as atoms (e.g. at 1 is true) the representation of numbers in LISP is usually different than that of symbolic atoms. Numbers are not entered on the oblist and they do not have property lists. Their pnames and "values" are computed from the representation and cannot be accessed in the usual way. In particular, you cannot set the "value" of a numeric atom (which is probably just as well) so they really are constants. One possible representation of numbers is by a cell the a-part of which says this is an atom, and d-part which is a pair consisting of the type and a pointer to actual machine representation of the numerical value. Sufficiently small integers can be represented more compactly by a cell the a-part of which is a special flag and the d-part of which is a machine representation of the numerical value. Another possibility is to store all numbers of a particular kind in a particular area and represent a number by a pointer directly to the machine representation of its value. As we mentioned in Chapter I, LISP can treat arbitrarily large integers by representing them as a list "digits" with respect to some large base. Typically the first element of the list is a flag saying "I am a bignum" and a sign. In any case the reading, printing, and other primitive programs of LISP must pay attention to whether or not the object being dealt with is a symbolic atom or a numerical atom or some non-atomic object.

Symbolic Atoms: property lists

Symbolic atoms do not in general have a meaning fixed by the system. They can be names of variables, programs, macros, or arrays and as such must be capable of having values, program code (compiled or in S-expression form), macro definitions, and array descriptions (pointers) associated with them. In addition arbitrary user defined properties can be associated with an atom. In the simplest case LISP treats all properties of an atom (including its pname) uniformly by putting them on a property list. In this case the atom is represented by a pointer to the property list and all properties can be accessed and set uniformly by special operations on property lists. In order to distinguish an atom (e.g. a property list) from other list-structure, the a-part of the first element contains data that is recognized as an "I am an atom" flag. This uniformity is simple and elegant. Information stored on a property list can be retrieved for a particular atom in a time that depends on the length of the property list of the atom, and this is usually quite short. The time is independent of the number of atoms. There is some inefficiency when frequently used properties, like value, are not accessed directly but must be looked up like user defined properties. Also the uniformity allows the programmer to manipulate the state of the

system in arbitrarily complicated and unmanageable ways. Another possible representation of an atom is a pointer to a block of memory with system properties such as pname, value, (possibly) program descriptions, and a pointer to the list for user defined properties stored in fixed locations relative the block pointer. There may also be restricted access to some of the system properties, thus imposing a small amount of discipline on the programmer.

Property lists are usually realized as ordinary lists. Each "property" has an atom as its name, and the name of the property is ordinarily followed by a pointer to the information in question. Because this information can be a string of characters, a floating point number or a pointer to a subroutine, the property list is ordinarily not a proper LISP list, and programs that operate on proper lists can cause errors if applied to property lists. For example, a program that interpreted a floating point number as a pair of pointers would interpret random places in memory as list structure. Thus, as we discussed in Chapter V LISP provides special operations such as GET, PUTPROP, and REMPROP for manipulation of property lists.

Symbolic atoms: Oblists.

LISP keeps track of atoms explicitly known by name by making entries in the "oblist" (called obarray in some implementations). The oblist is used mainly by the read program to look up atoms having a given pname. If an atom is known (has been mentioned by name previously and not explicitly forgotten) then a pointer to the atom can be obtained by looking up the oblist entry corresponding to its pname. Otherwise an atom having that pname must be created and entered in the oblist. This protocol insures that separate mention of atoms having the same external representation are represented by the same internal structure.

The oblist could be an ordinary list manipulated by *car*, *cdr*, *cons* and even *rplac*'s. Some implementations use arrays rather than lists to store the oblist, hence the name "obarray". Often the oblist is "hashed" in order to speed up the search for a particular entry. The idea of a hashed list (or array) is that it is divided up into a fixed number n of sublists called "hash buckets", and there is a program *hash* which computes a number between 1 and n for each possible element of the list. If *hash*[e] is k then e is in the kth bucket and the search for e can be restricted to searching that bucket rather than the whole list. In the case of oblists the argument to *hash* is a pname. If the hashed list is to be an improvement over a simple linear list as far as searching efficiency is concerned, the hash program should distribute the elements of the list evenly among the buckets. This is not always easy to do. For a further discussion of hashing as a general searching method see Knuth[1968b].

There are two basic operations for modifying an oblist, *intern* and *remob*. *intern* takes an atomic object (a pointer to it) and returns a pointer to an atom with the same pname which is a member of the oblist. If no atom with the same pname is on the oblist then *intern* puts the given atom there, and the returned pointer will thus be equal to the input pointer. If an atom of the same name is already on the oblist, then the pointer to that atom is returned and it may or may not be equal to the input pointer. (A cleaner explanation of *intern* would require an oblist as an argument also. In LISP many operations use some part of the LISP environment as implicit arguments. Of course they also may update the environment when executed.) *remob* is the atom forgetting program. It takes an atomic object as an argument and removes it from the oblist. *remob* doesn't destroy the atom, it just can no longer be referenced by name. *remob* and *intern* are usually available directly to the user. As with any destructive operation the use of *remob* can get you into trouble if you are not careful, and are not quite sure of the state of the world when you do it.

The basic primitive for creating atoms is *mkatom* which takes a pname and returns an atom with that pname and with no other properties set. *mkatom* is not directly available to the user, but is used (possibly implicitly) by other system programs that manipulate atoms.

Making atoms: Examples.

LISP makes available to the programmer several operations on atoms and oblists. These include *maknam*, *implode* and *gensym* for making atoms, *explode* for analyzing pnames and *intern* and *remob* for updating the oblist. The oblist operations were described above. (You should consult the manual for the LISP you are using, for details about the precise names and conventions for using these programs.) In the following we attempt to explain the subtleties and uses of the programs for making atoms.

First we consider why the ability to make atoms (other than as needed when the are read) is useful. In some sense the usefulness of this and other features is limited only by the imagination of the programmer. One class of programs that use the atom making ability are programs that need to invent labels for some reason. Compilers are in this class. The typical output of a compiler is a list of symbolic assembly instructions some of which are labeled with symbolic addresses which are referenced in other instructions. Other members of the class include program generating and program transforming programs that need to introduce new variable names, statement labels, etc.. Another example is a program that uses temporary files as work space and the number of files needed is not known in advance. Therefore it is convenient to be able to generate file names. Finally in a system of programs for manipulating logical formulas or λ-expressions a program that substitutes an expression for occurrences of a bound variable generally needs to be able to rename bound variables in order to avoid conflict. This is most conveniently done by providing a means of generating new variable names as the need arises.

The operation *maknam* takes a list of characters (atoms whose pname is a single character) and returns a new atom whose pname is the concatenation of this list of characters. It does not enter the atom on the oblist, or look to see if an atom of the same pname is already known. Thus it always produces an atom that is not eq any already existing atom.

gensym behaves similarly, but it takes no arguments. It determines the pname from a prefix and a count stored in some global location. The usual prefix is G and the resulting pname has the form Gdddd where each d is a decimal digit. Each time *gensym* is evaluated the count is incremented by 1. Thus if we type (GENSYM) to LISP and it types back G0069 then typing (GENSYM) again will result in G0070 being typed back. (No prediction is made as to what will happen when the symbol reaches G9999). (LISP implementations usually provide the user a mechanism for resetting the GENSYM count and prefix character(s).) Note the non-functional aspect of *gensym*. Namely, *gensym*[]≠*gensym*[]. (Try typing (EQ (GENSYM) (GENSYM)) to LISP.) *gensym* is useful in places where you need a label, but don't care what it is called.

implode, like *maknam* takes a list of characters and returns an atom whose pname is the concatenation of the characters in the list. However, unlike *maknam* it first looks to see if an atom with that pname already known (in the oblist) and if so returns that atom (not a new one) If not, a new atom is made with the appropriate pname and entered in the oblist. Thus *implode* is the same as applying *readatom* to the pname computed from the character list. It behaves more like a function than do *maknam* and *gensym*.

explode is essentially an inverse to *implode*. It takes an atom and returns the character list corresponding to the pname of the atom.

One reason for using *gensym* and *maknam* has to do with the fact that an atomic symbol that is put on the oblist never goes away (unless you explicitly *remob* it), while atoms generated by *gensym* and *implode* can be garbage collected as soon as no one is pointing to them any more. Thus if you have a program that is going to generate a lot of temporary symbols which will not be wanted after some it is efficient spacewise to not put these symbols on the oblist.

If you are confused at this point, some examples of the behavior of *maknam*, *implode*, *gensym*, *intern*, *remob*, and *explode* my help. Your are strongly encourage to go to your nearest LISP and try some experiments of your own to be sure you understand what is happening.

```
user:    (EQ (MAKNAM '(B A M)) (MAKNAM '(B A M)))
LISP:    NIL

user:    (EQ (IMPLODE '(B A M)) (IMPLODE '(B A M)))
LISP:    T

user:    (SETQ X (MAKNAM '(F O O)))
LISP:    FOO

user:    (EXPLODE X)
LISP:    (F O O)

user:    (EXPLODE 'FOO)
LISP:    (F O O)

user:    (EQ X (MAKNAM '(F O O)))          ;maknam makes a new atom
LISP:    NIL

user:    (EQ X (IMPLODE '(F O O)))         ;maknam didn't intern x
LISP:    NIL

user:    (EQ 'FOO (IMPLODE '(F O O)))      ; implode interns FOO
LISP:    T

user:    (EQ X 'FOO)
LISP:    NIL

user:    (SETQ Y (IMPLODE '(F O X)))
LISP:    FOX

user:    (EQ Y (IMPLODE '(F O X)))         ;implode interned y
LISP:    T

user:    (EQ Y (INTERN Y))
LISP:    T

user:    (SETQ Y (MAKNAM '(F O X)))
LISP:    FOX
```

```
user:     (EQ Y (INTERN Y))                    ;an atom having same pname
LISP:     NIL                                  ;but not eq y is on the oblist

user:     (REMOB Y)
LISP:     NIL

user:     Y                                    ;y is still a prefectly good atom
LISP:     FOX

user:     (EQ Y (IMPLODE '(F O X)))            ; but not on the oblist any more
LISP:     NIL

user:     (SETQ X (GENSYM))
LISP:     G0123

user:     (EQ X 'G0123)                        ;gensym also makes new
LISP:     NIL                                  ;uninterned atoms
```

List structure.

Now that we can build and recognize atoms, the next step is to be able to construct representations of arbitrary S-expressions. Thus we need to represent list structures and implement the operations *car*, *cdr*, and *cons*. LISP sets aside a portion of memory available to it in an area called list space. This space contain of cons-cells. As explained in Chapter I, a cons-cell must be big enough to hold two pointers and could be a single machine word with the two halves being pointers to the a- and d- parts or if the size of a word is too small, a pair of words. In either case *car* and *cdr* can be fairly simply operations (in the case of the usual PDP-10 implementation a single machine instruction).

At any particular time some of the list space will be in use and some of it will be free. The free space is linked into a single list called the "free storage list" and LISP keeps a pointer to the beginning of that list. When a *cons* is executed, a word is removed from the free storage list, the a-part and the d-part are filled in and a pointer to this word is returned. When there is no more free storage left (more accurately when the amount left reaches some preset minimal amount) then LISP attempts to obtain more free space. One way is to request it from the host system, but this is undesirable in general, since a lot of the list space is taken up by structures which are no longer accessible and may as well be "recycled". This recycling process is done by what is known as a "garbage collector". Thus when *cons* discovers that it is out of space, it calls the garbage collecter to find some. The garbage collector must first find out which parts of list space are in use (accessible) and which are not. This is known as the marking phase. In the sweep phase it collects the unused space, adding the recovered cons-cells to the free storage list. If it succeeds in finding unused space, *cons* can proceed, if not the computation usually grinds to a halt, printing some suitable error message.

To determine what parts of list structure are currently accessible the garbage collector starts with all of the immediately accessible parts and follows all possible *car* and *cdr* chains until reaching an atom or something previously seen (marked). The immediately accessible parts of list structure include the property lists of all atoms (including the contents of value cell in case this is kept separately), structures on whatever stacks LISP is using for control, contents of any list type

arrays, plus probably a few special locations or registers that LISP may use for special purposes. There are a number of algorithms of doing the marking and collection of unused space. It is a technique generally useful in situations where storage gets allocated and reclaimed. For more details the reader is referred to Knuth[1968a] or Baker[1978] and references therein.

Other LISP objects.

The structures and operations we have discussed above are sufficient for a LISP system that is only going to deal with interpreted code and will only allow S-expressions as data for programs. Most systems provide at least two additional features. One is the ability to run compiled code, and the other is the addition of arrays to the allowed types of data. Thus a space known as binary program space is usually set aside for storing compiled code. Sometimes arrays are kept in the same space, and sometimes they are kept in an additional separate space. In any event, this means that there are two additional objects that some of the basic LISP routines must recognize and deal with. One is the "subr" object which is the address of a compiled program and is associated with the atom naming the program via the SUBR property. The other is an array descriptor which will be associated via the ARRAY property with the atom naming that array.

In addition to the structures that make up the data manipulated by a LISP program, the LISP system must maintain structures describing the current state of a computation. In general there will be one or more stacks (often called Push-Down-Lists or pdls) where information is kept as to current and/or old variable bindings (binding contexts or environments), knowledge about the expression currently being evaluated, and what to do next (control stack). Information on these stacks is accessed via pdl-pointers, context pointers etc.. The LISP system usually provides programs to allow the user to examine control stacks and environments. We will say something about this later.

Exercises

1. Write a program *saveup* that takes a list of atoms and a list of properties and collects for each atom and association list of property/value pairs. *saveup* returns a list of atom/alist pairs.

2. Write a program *removem* that takes the same arguments as *saveup* and removes all the properties from the property lists of all the atoms.

3. Write a program *restorem* that takes a list of the form output by *saveup* and puts back all of the properties on the property lists of all the atoms.

The above collection of functions are useful for switching environments in a system written in LISP.

2. The Top level of LISP.

The top level of LISP can be described in terms of the programs *read*, *eval* and *print* for reading, evaluating and printing S-expressions. Namely,

toplevel[] ← program[[] *top*: *print eval read*[]; go to *top*].

The *read* program gets from the input stream the next string of tokens corresponding to the external representation of an S-expression and constructs the corresponding list structure representation. *eval* computes the value denoted by the expression in the current environment, and *print* converts a list structure into the external representation of the corresponding S-expression and sends it to the output stream.

In this Chapter we will present programs that embody the recursive structure of the process of reading and printing S-expressions in both of the notations (dot and list) discussed in Chapter I. The "slashifying" of special characters is also discussed. We have already discussed how a simplified version of *eval* works (Chapter I §13). Here we discuss some of the decisions that must be made in designing a LISP interpreter, how they effect the results computed by your programs, and how LISP uses the "system properties" stored on the property lists of atoms.

Reading and Printing: "dot" notation

Consider first reading and printing S-expressions in "dot" notation. Rather than deal with the issue of representing character strings, we read an print a listified external notation which is a list of atoms. The special atoms LP, RP and DOT represent the delimiter tokens "(", ")" and "." respectively and all other atoms represent themselves.

Let *DOTLIST* be the sort of lists that correspond to "dot" notation representations of S-expressions. They can be characterized as follows:

(i) < <other-atom> > is a *DOTLIST* representing the atom <other-atom>, where <other-atom> is any atom not in the list of special atoms LP, DOT, RP.

(ii) if *da* and *dd* are *DOTLIST*s then <LP> * *da* * <DOT> * *dd* * <RP> is a *DOTLIST* representing [*x* . *y*] where *da* represents *x* and *dd* represents *y*.

(iii) Those are all of the lists of sort *DOTLIST*.

The program *prindot* "print"s a list of sort *DOTLIST* representing the S-expression given as an argument.

The definition of *prindot* is:

2.1) *prindot*[*e*] ← *prina*[*e*, NIL]

2.2)
```
prina[e, l] ←
    if at e then e . l
    else LP . prina[a e, DOT . prina[d e, RP . l]]
```

thus

```
prindot[(PLUS (TIMES A B) C)] =
        (LP PLUS DOT LP LP TIMES DOT LP A DOT LP B
         DOT NIL RP RP RP DOT LP C DOT NIL RP RP RP),
```

Notice that the recursive structure of *prindot* is very much like that of *flatten* (I.8.8). It simply inserts the delimiter characters in the appropriate places.

The program *readdot* "read"s a list of atoms. If the list is of the form $dl * u$ for some *DOTLIST*, dl and list u then the result is $e \,.\, u$ where e is the S-expression represented by dl. Otherwise the atom READ-ERROR is returned.

The definition of *readdot* is:

```
readdot u ←
        if at u then READ-ERROR
        else if a u equal LP then
2.3)        {readdot d u}[λw:                                  ;read a-part
                if at w ∨ ¬[ad w equal DOT] then READ-ERROR     ;check for error
                else {readdot dd w}[λv:                        ;read d part
                    if at v ∨ ¬[ad v equal RP] then READ-ERROR  ;check for error
                    else [a w . a v] . dd v]]                  ;cons up the result
        else u
```

thus

```
a readdot[(LP PLUS DOT LP LP TIMES DOT LP A DOT LP B
                DOT NIL RP RP RP DOT LP C DOT NIL RP RP RP)] =
          (PLUS (TIMES A B) C).
```

Reading and Printing: mixed "list dot" notation

Now we treat the more general mixed "list-dot" notation. Let *LDOTLIST* be the sort of lists that correspond to "list-dot" notation representations of S-expressions. They can be characterized as follows:

(i) < <other-atom> > is a *LDOTLIST* representing the atom <other-atom>, where <other-atom> is any atom not in the list of special atoms LP, DOT, RP.

if di $1 \le i \le n$ and dd are *LDOTLIST*s representing, respectively, the S-expressions xi $1 \le i \le n$ and xd then

(iia) <LP RP> is an *LDOTLIST* representing ( ) or NIL

(iib) <LP> * $d1$ * ... * dn * <RP> is an *LDOTLIST* representing <$x1$ ... xn>

(iic) <LP> * $d1$ ... dn * <DOT> * dd * <RP> is a *DOTLIST* representing [$x1$ [... [xn . xd] ...]]

(iii) Those are all of the lists of sort *LDOTLIST*.

The program *prinlis* "print"s an S-expression in "list" notation. This is the extreme form of list-dot notation where dd following DOT in clause (iic) must be an atom. To print non-atomic

S-expression in this style, start by printing LP, then *cdr* through the S-expression printing the *car* until an atom is reached. If this atom is NIL the S-expression is indeed a list and the printing is terminated with RP. Otherwise the printing is terminated with DOT followed by the atom followed by RP. *prinlis* is defined by:

2.4) *prinlis*[*e*] ← *prinb*[*e*, NIL]

prinb[*e*, *l*] ←
    if a*t* *e* then *e* . *l*
2.5) else LP . [if n*d* *e* then *prinb*[a*e*, RP . *l*]
        else if a*t*d*e* then *prinb*[a*e*, DOT . [d*e* . [RP . *l*]]]
        else *prinb*[a*e*, d*prinb*[d*e*, *l*]]]

thus

prinlis[(PLUS (TIMES A B) C)] = (LP PLUS LP TIMES A B RP C RP).

and

prinlis[(A . (B . (C . D)))] = (LP A B C DOT D RP).

prinlis also has the same basic structure as *prindot*, but makes some intermediate tests in order to omit all but the essential delimiter characters.

The program *read* gobbles up the initial segment, *dl*, of a list *dl* * *u* and returns [*e* . *u*] where *e* is the S-expression represented by *dl* in list-dot notation. The auxiliary program *reada* does the work. It the uses the auxiliary variable *l* to stack partial results. The basic idea is the following: if the next atom is LP then gobble up 0 or more S-expressions from the remaining list, stacking them on *l*, until a delimiter (DOT or RP) is reached. If the delimiter is RP then the result is the *reverse* of *l*. If the delimiter is DOT then gobble up one more S-expression and return *l* reversed onto that. The definitions of *read* and *reada* are as follows:

2.6) *read* *u* ← if a*u* equal LP then *reada*[d*u*, NIL] else *u*

reada[*u*, *l*] ←
    if n*u* then *revl*[*l*, ERROR] . NIL
2.7) else if a*u* equal RP then *reverse* *l* . d*u*
    else if a*u* equal LP then {*reada*[d*u*, NIL]}[λ*w*: *reada*[d*w*, a*w* . *l*]]
    else if a*u* equal DOT then {*reada*[d*u*, NIL]}[λ*w*: *revl*[*l*, aa*w*] . d*w*]
    else *reada*[d*u*, a*u* . *l*]

2.8) *revl*[*u*, *v*] ← if n*u* then *v* else *revl*[d*u*, a*u* . *v*]

thus

a*read*[(LP PLUS LP TIMES A B RP C RP)=(PLUS (TIMES A B) C)

We note that *read* and *reada* don't check the syntax completely, so some non-wellformed lists will be "read" without complaint, although the result may be strange.

We end the discussion of reading and printing with some remarks on the treatment of special characters in LISP. Implicit in the functioning of the LISP scanner in the fact that certain characters such as the delimiters "(", ")", "." and " " (<space>) are treated specially and cannot directly be included in the names of atoms. LISP systems usually provide a way of including these special characters in an atom name by designating an additional special character to serve as an escape character. Thus any character following the escape character in an input string is treated as an ordinary letter. This is ofter called "slashifying" as the designated escape character is often the "/" character. Thus (AB/.D) is a list of one element, the atom whose pname is "AB.D" whereas (AB.D) is a pair, the a-part being AB and the d-part being D. One result of this "slashifying" convention is that many of the pname manipulating and printing functions have several flavors according to whether the name or the slashified version is being dealt with. We mention the fact only to make you aware of it. You should consult your LISP manual for further details. You should also be aware of the fact that "/" itself (or whatever the designated escape character is) being a special character is gobbled up by the LISP scanner, thus if you really want the symbol "/" you must type "//".

Evaluation

In most current LISP systems the evaluator doesn't require an a-list (environment) argument. Instead it uses the property lists and environment (binding context) stacks or some equivalent structure to store the corresponding information. The simple *eval* was given as a recursive function and thus much of the control structure is automatic in the recursion structure. Actual interpreters are implemented iteratively, and use control stacks to keep track of what they are doing and what is left to be done. There are several important decisions to be made in the design of an interpreter. These include the mechanism for making, saving and restoring variable bindings, how to evaluate or interpret the expression occuring in functional position of an application term, when and how to bind the free variables in function expressions. (This relates to the infamous "funarg" problem.) If you only write simple clean pure LISP programs in which no free variables occur in function expressions then these decisions probably won't effect the meaning of your programs. However, most programs are like that. In the following we discuss some possible decisions and explain the use of system properties. You should consult the manual for LISP you plan to use to find out what decisions have been made in designing the interpreter. A detailed analysis of binding and control mechanisms in LISP can be found in Allen[1979]. The iterative SCHEME interpreter described in Sussman and Steele [1978] handles environment stacks elegantly.

When LISP evaluates the application of a λ-expression to a list of arguments it must first associate the variables in the variable list of the expression with the values of the corresponding arguments. Since this evaluation may occur in the midst of the evaluation of a bigger expression where the same variable names occur in the outer expression, the old values must be saved and restored when the current evaluation is completed. Similarly when a program is entered, the current values of the program variables must be saved and restored when the program is exited. (The interpreter may initialize these variables or leave them unbound.) The necessary decisions include: where to keep the current binding, how to save and restore bindings, and how to access outer bindings of variables. These are not independent. Thus the current binding of a variable could be kept always as the VALUE property. Whenever a new binding is made, the old one is saved on the binding context stack and restored when the program or λ is exited. Another possibility is to consider the VALUE property as the top level, global environment and put new bindings on the stack. Access to value is slower, but restoring is faster.

For expressions occuring in the "function" position of a term, the interpreter must decide what program is denoted. If the expression is a λ-expression then the λ-variables are bound to the values of the arguments and the body is evaluated. If the expression is an atom there many possible conventions. One is to always look for an applicative property (as discussed below) on the property list of the atom and use that value. If there are several, the reasonable choice is the first one found as it is the most recent property set. Another possibility would be to check the local environment first to see if the atom is bound to an applicable expression. If none is found then look on the property list. Some LISPs use the VALUE property for every thing and in the case where an applicable object is needed, will repeat the evaluation process until such an object is found, or it can't go further for some reason.

The problem of when and how to assign values to the free variables in a function-expression is the subject of much controversy in the LISP world. The question is where to "trap" the free-variables. It is a problem that plagued the logicians in the early days of the development of formal logic and also of λ-calculus. It arises when you define what it means to substitute an expression for occurrences of a bound variable in some other expression. The logicians solution is to not allow trapping to occur. Namely, outer bound variables are renamed if they occur free in the expression being substituted. Thus free variables remain free. In computation there are times when it is convenient to allow trapping of free-variables. For example the function-expression passed to a program may just be a piece of code extracted from that program to make it easier to read and you intend that the same name in both pieces of code refer to the same thing. The original LISP interpreter provided maximal trapping in the sense that variables function-expressions were always assumed to refer the the most recent binding. (This was more of an oversight than a conscious decision.) Current LISPs usually provide both alternatives to some degree. In MACLISP evaluating

```
(FUNCTION (LAMBDA (X) (PLUS X Y)))
```

produces the undecorated λ-expression

```
(LAMBDA (X) (PLUS X Y))
```

which when applied will use the value of Y at the time of application, while evaluating

```
(*FUNCTION (LAMBDA (X) (PLUS X Y)))
```

produces the "funarg" term

```
(FUNARG (LAMBDA (X) (PLUS X Y)) <current-bcp>)
```

where <current-bcp> is a pointer to the current binding context. When applied the context indicated by the 3rd element of a funarg is used instead of the binding context in force at application time. There are many subtleties involved in solving the "funarg" problem. For example, what if the binding context referred to in a MACLISP funarg expression has disappeared before it is used? This is a "funval" problem in the sense that it is when the funarg expression is passed up as a value of a computation to be used later the binding context is likely to have disappeared (popped from the stack). We won't pursue the problem further.

[References?]

We conclude the discussion of evaluation with a brief description of the use of some of the standard system properties other that PNAME and VALUE. An atom with an EXPR property is assumed by LISP to denote an interpretable program and the value of the EXPR property should be an application term (such as a program name, a λ-expression, or a label-expression) suitable for being interpreted. Recall that this property can be set using one of the operations DEFUN or DEFPROP described in Chapter I. An atom with a SUBR property denotes a compiled program which may be directly executed. The value of the SUBR property is a subroutine pointer which points to the compiled program. Atoms denoting built in program such as REVERSE or APPEND have a SUBR property automatically. When a compiled program is loaded into LISP the name of that program gets a SUBR property. The properties FEXPR and LEXPR are similar to EXPR except that the arguments to the program are treated differently. In the EXPR case a fixed number of arguments are expected and they are evaluated before executing the program. In the FEXPR case the list of arguments (e.g.. the d-part of the expression) is passed unevaluated to the program as a single entity. In the LEXPR case the number of arguments is arbitrary, the arguments are evaluated and saved (typically on the stack), the number of arguments is passed to the program and the program must use a special auxiliary program *args* to access the argument values. The three SUBR cases are analogous to those for EXPR. We can express the difference between DEFPROP and PUTPROP explained above by saying that DEFPROP is an FSUBR and PUTPROP is a SUBR.

If an atom has a MACRO property then that property should be a application term taking one argument. When such an atom is encountered as the a-part of an expression being evaluated, the corresponding macro definition is applied to the entire expression, as the single argument, and the value is a new expression that is evaluated in place of the original one. An atom can be given the MACRO property using DEFUN with type MACRO specified or some other macro-definition facility provided by the system or defined by the user. See Chapter V §3 for more details.

An atom with an ARRAY property denotes an array. The value of the ARRAY property is an array pointer which provides LISP with a means of accessing the array. The ARRAY property can be set using the ARRAY command as explained in Chapter V §2. The array pointer includes type, bounds, and location information.

Exercises

1. Modify *read* and *readα* to do some syntax checking and return "error messages" is case of an ill-formed argument.

2. Rewrite *readdot* as a sequential program with out any recursive calls.

3. Editing LISP programs.

An important part of any LISP system is an interactive editor. With the help of the editor you can write and modify programs. Some editors allow you to evaluate expressions with out leaving the editing environment. They may also provide facility for editing S-expressions other than programs. The fact that LISP programs are S-expressions with a very simple syntax means that it is easy to write a simple but powerful LISP program editor in LISP.

In this section we will present a simple interactive editor based on the MACLISP editor. Before describing this program let us consider what we want from a program editor. First it should be able to lookup the current program given its name and be able to replace it by a modified version. Second it should be able to move around in the program being edited thus changing its notion of the *current expression* (known as the CE). Third it should be able to insert and delete sub-expressions (elements of the CE). This is sufficient capability to convert any program into any other. It is also somewhat essential to be able to look at the CE, thus the editor should be able to print the CE. Other features which are useful include moving parentheses "in and out", wrapping a subsequence of a list into a list -- e.g. inserting a par of parentheses, or unwrapping an element of a list and splicing its elements into the containing list -- e.g. erasing a pair of parentheses, finding the first occurrence of some expression -- e.g. changing the CE to be that occurrence or maybe the list containing that occurrence, replacing all occurrences of an expression by another expression -- either at the top level of the CE or at all levels, transposing members of the CE, moving an expression from on place to another, or reversing the CE. These are only a few ideas. We implement a representative selection. The reader should be able to implement any others without much difficulty. Some editors also have file manipulation capability, so old definitions can be looked up in files and new ones saved or used to update a file. We will not treat this aspect of editing.

The program *editor* takes program name as an argument. When invoked, *editor* looks up the program by getting the EXPR property, initializes the global parameters *fn top*, *ce* and *chain* and then goes into a command reading - executing loop. The parameter *fn* is just to save the program name. *top* is the program being edited. *ce* the current expression (*top* initially). *chain* the chain of expressions leading from *top* to *ce*. *If top=ce* the *chain=*NIL otherwise the first element of *chain* is a pair [*n* . *e*] where *e* is the expression immediately containing *ce* and *n* is the position of *ce* in *e*.

There are two kinds of commands those handled directly by the editor and those which the editor uses to look up what should be done. The direct commands are Q, OK, or <n> for any number n. Q means exit the editor, OK means replace the old version of the program being edited by the new one and then exit. <n> means move to the nth element of the *ce*, making that the new *ce*. There are two type of lookup commands, atomic and list. For atomic commands the editor looks up the program to execute on the ATOMIC-EDIT-FNS property of that atom and runs it. For list commands the editor looks up the program to execute on the LIST-EDIT-FNS property of a *command* and applies it to d*command*. If in either case the property is not found then the command is evaluated as an ordinary term and the result printed. This style of programming is sometimes called "data directed". It makes the editor extremely easy to extend or modify. Since the data structures manipulated are quite simple and easily explained, it can even be explained to a user in sufficient detail that the user can implement his own favorite commands. The reason that the numeric commands are treated directly is that numbers don't have property lists. The editor could be made more uniform by giving these commands symbolic names, but moving down into an expression via such commands is sufficiently convenient to make it worth implementing the special case. We take some care in the editor and command codes to make sure the command will work in the context it has been given. If not an error message is printed and no action occurs. Many of the commands work destructively. This is safe because the expression to edit is a "copy" of the original definition. In fact one might consider it essential to edit destructively. Since otherwise, after each change the expression would have to be rebuilt. At least it is in the spirit of editing to make destructive changes, since the point of editing in to change the expression.

The program *editor* and its auxiliary functions are defined as follows.

```
editor l (FEXPR) ←
   program [fn, top, ce, chain, command, efn]
      fn ← a l
      top ← copy get[fn, EXPR]
      if n top then [errmsg0[]; return NO-EDIT]
      ce ← top; chain ← NIL
 EDLOOP:
      print ϵ
      command ← read[]
      if command = Q then return BYE
      else if command = OK then return putprop[fn, top, EXPR]
      else if numberp command then
3.1)       [if at ce ∨ command > length ce then [errmsg1[], go to EDLOOP],
            chain ← [command . ce] . chain,
            ce ← nth[ce, command],
            go to EDLOOP]
      if at command then efn ← get[command, ATOMIC-EDIT-FN]
       else efn ← get[a command, LIST-EDIT-FN]
      if efn = NIL then [print eval command, go to EDLOOP]
       else if at command then [eval efn, go to EDLOOP]
       else if ¬at command then
          [apply[efn, d command], go to EDLOOP]


errmsg0[] ← [print fn, princ 'not an EXPR']
errmsg1[] ← [terpri[], princ '> length of CE']
errmsg2[] ← [terpri[], princ 'Unknown command']
errmsg3[] ← [terpri[], princ 'You are at the top']
errmsg4[] ← [terpri[], princ 'You are at the left edge']
errmsg5[] ← [terpri[], princ 'You are at the right edge']
errmsg6[] ← [terpri[], princ 'CE is atomic']

3.2)   nth[u, n] ← if n > 1 ∧ ¬n d u then nth[d u, sub1 n] else a u

3.3)   pos[u, n] ← if n > 1 ∧ ¬n d u then pos[d u, sub1 n] else u

3.4)   copy x ← if at x then x else copy a x . copy d x
```

We have implemented the following atomic commands:

P ; print the *ce*,
UP ; move up one level in the chain,
RT ; move one to the right,
LI ; move the left parenthesis in one
thus removing the first element of the *ce* and inserting it before the *ce*), and
RO ; move the right parenthesis out one
thus making the right sibling of the *ce* the last element of the *ce*.

The expressions put on the ATOMIC-EDIT-FN properties are shown below. If you have trouble

figuring out how some of the pointer manipulations work we suggest drawing a list structure (boxes and pointers) of some CE and following through the sequence of assignments.

```
3.5)   p:ATOMIC-EDIT-FN ← print ce                    ;print the ce

       up:ATOMIC-EDIT-FN ← program []                 ;ce ← parent expression
3.6)            if nchain then return errmsg3[]
                ce ← dachain
                chain ← dchain

       rt:ATOMIC-EDIT-FN ← program [n]                ;ce ← expression to right
                if nchain then return errmsg3[]
3.7)            n ← add1 aachain
                if n > length dachain then return errmsg5[]
                ce ← nth[dachain, n]
                aachain ← n

       li:ATOMIC-EDIT-FN ← program [cetmp, pos]       ;move left paren in
                if nchain then return errmsg3[]
                if atce then return errmsg6[]
3.8)            pos ← pos[dachain, aachain]          ;point to ce
                dpos ← dce . dpos                    ;insert dce after ce
                apos ← ace                           ;replace ce by ace
                ce ← dce
                aachain ← add1 aachain

       ro:ATOMIC-EDIT-FN ← program [cetmp, pos, pos1] ;move right paren out
                if nchain then return errmsg3[]
                if atce ∧ ¬nce then return errmsg6[]
                pos ← pos[dachain, aachain]
                pos1 ← dpos
3.9)            if npos1 then return errmsg5[]
                cetmp ← NIL . ce                     ;in case ce is NIL
                nconc[cetmp, pos1]                   ;add next element to end of ce
                dpos ← dpos1                         ;delete it from parent list
                dpos1 ← NIL
                ce ← dcetmp
                apos ← ce                            ;make sure new ce is in parent list
```

We have implemented the following list commands:
(I <n> <exp>) ; insert <exp> before the <n>th element of the *ce*,
(D <n>) ; delete the <n>th element of the *ce*.

The programs for executing these commands are given by

```
        i:LIST-EDIT-FN ←                                  ;insert x before the nth element
          λn x.[program [pos, tmp]
                if n < 0 then return errmsg2[]
                 else if n = 1 then
3.10)                 if nchain then top ← [ce ← x . ce]
                      else [ce ← x . ce ;
                            apos[dachain, aachain] ← ce]
                pos ← pos[ce, sub1 n]                     ;point to position for insertion
                tmp ← x . NIL                             ;make list containing x
                dtmp ← dpos                               ;splice it in
                dpos ← tmp ]

        d:LIST-EDIT-FN ←                                  ; delete the nth element of the ce
            λn.[program [pos, tmp]
                 if n < 0 then return errmsg2[]
                  else if n = 1 then
3.11)                 if nchain then top ← [ce ← dce]
                      else [ce ← dce ;
                            apos[dachain, aachain] ← ce]
                  else if n > length ce then return errmsg2[]
                 pos ← pos[ce, sub1 n]
                 dpos ← ddpos ]
```

Exercises

1. Write programs to execute the atomic commands LO, and RI (see the description of RO and LI).

2. Modify the list commands to take a negative argument which is interpreted as count from the end rather that the beginning of the *ce*. 3. Write a programs for executing list-commands of the form (R <e1> <e2>) (which means to replace occurrences of <e1> as elements of the *ce* by <e2>) and (TR <e1> <e2>) (which means to replace occurrences of <e1> as subexpression (at any level) of *ce* by <e2>). Note that it is advisable to make a new copy of <e2> for each replacement otherwise in later editing a change to one occurrences may have the undesired side effects of changing all occurrences.

4. Write a program to execute the command (MV <m> <n>) which moves the element in <m>th position to the <n>th position.

5. Consider implementing an UNDO feature in the editor which would undo the last command (or last n commands). One way to do this is to notice that commands that make changes have inverses. For example of LI is LO, of (I <n> <exp>) is (D <n>), etc. All we need is for the editor to setup a list for storing undoing commands and require that each command that makes a change stack its inverse on the undo list.

4. Error Handling.

A robust and helpful LISP system will do all it can to keep you from destroying it. This includes noticing when you have asked it to evaluate an ill formed or illegal expression. In this class we include attempting to evaluate a variable that hasn't been bound (SETQed or bound as a LAMBDA or PROG variable), taking the *car* or *cdr* of an atom or other non list structure object such as an array pointer, attempting to append a non-list to some S-expression, applying a function to the wrong number of arguments, etc. When such an error is noticed LISP will complain, printing a (hopefully informative) message, and take some appropriate action. Other errors such as an infinite recursion can't be prevented by inspection of the expression to be evaluated, but will be noticed when some abnormal state is reached such as when the control stack overflows or an attempt is made to read or write in an illegal location. Again LISP will complain by printing a message saying what abnormal condition was noticed and take appropriate action. Of course there is always the possibility of a truly disastrous error from which there is no recovery and here there is not much to do but restart and proceed with caution. The question remains as to what to do when an error is noticed. Typically LISP will enter a state where a debugging package can take over and help you find out what is amiss. In this state you will be able to evaluate expressions, possibly back up the computation some number of steps, have the computation proceed (after providing whatever information was lacking and caused the error) or abandon ship and return to the top level to start again.

In order to aid in error recovery and generally provide the user with the ability to access information about the state of a computation, LISP provides various means of altering the normal progress of evaluation and examining and modifying the environment. One such means is the break loop. Evaluating *break*[*name*, *test*] causes LISP to evaluate *test* and if it is not NIL to go in to a break loop. The *name* argument is printed out as a message and then a pseudo read-eval-print loop is entered, where the expression read is tested to see if it is one of a few special "escape from the loop" commands. If so the loop is exited in the manner indicated by the command. Otherwise the form is evaled, printed and we are back at the top of the loop. There are at least two modes of return from a break loop. One is just to return to the top level, aborting whatever computation the break occured in, and the other is to resume the computation (possibly having altered the state of the world before doing so).

To get a complete picture of the state of the computation it is useful to know the sequence of events that led up to the current state. This is made possible by stacking such information as evaluation proceeds. Just how this is done and what information is kept depends on the particular implementation. For the sake of example we will base our description on the MACLISP implementation. Imagine that there is a stack (called the specpdl) where each time the interpreter begins to evaluate an S-expression it pushes onto the stack the following information: the current specpdl pointer, the expression to be evaluated, an a pointer to the current variable binding environment, (MACLISPs version of the a-list argument to *eval*). The current stack pointer is then reset to the new stack top and evaluation continues. When evaluation of the expression is complete that collection of information is popped from the stack. There are two primitive operations associated with this stack. One, *evalframe*[*pdl*], returns a list of the information at the stack position indicated by pdl. Thus we can use this information to move up and down the stack and evaluate expressions in various environments. The other operation causes the interpreter to change its notion about the current top of the stack. This is called *freturn* (for forced return). If *freturn*[*pdl*, *exp*] is evaluated then the interpreter behaves as though *pdl* were the top of the stack and the value of *exp* were the result of evaluating the

expression stored there. It thus "returns" from that state and proceeds. This is a highly non-local form of go to and should be used with caution, however it is most useful for building error-recovery and debugging routines.

Finally there are primitives for inducing and trapping errors. The main primitives of interest are the pair of functions *error* and *errset*. In the simplest case *error* takes one argument which is a message to be printed. Evaluation of *error msg* causes the value of *msg* to be printed and a LISP error to be signaled. The result of signalling an error is popping up to the nearest *errset* or to the top level of LISP if there is no enclosing *errset*. The value returned is NIL. *errset* has two arguments, the first is an expression to be evaluated and the second is a flag. If an error occurs while evaluating the expression, the *errset* "traps" the error and returns the value returned by the error function, otherwise *errset* returns a list of one element, the value of the expression. If the flag is off (e.g., if it is NIL) then no error messages are printed. The reason for *errset* returning a list containing the value of the successfully evaluated expression is to distinguish the value NIL from the atom NIL which says that an error has occurred. An alternate form of error generation is *err* which takes a single argument. Evaluation of *err x* causes an error to be signaled and the value of *x* is returned. There are many variations and extensions of the above error functions which you should learn about for your particular implementation of LISP.

In addition to trapping errors, the *err*, *errset* mechanism can be used to return from the midst of a computation when the answer has been found and it is not necessary to process the result further. For example if you are looking for a particular atom in an S-expression by doing a recursive search, then if the atom is found, executing *err* pops directly back to the calling *errset* without doing the intermediate book keeping necessary to return through many levels of recursive calls. MACLISP provides a "structured" form of such non-local returns in the form of the *catch/throw* construct. Each takes two arguments an expression to evaluate and a tag. *catch*[*e*, *tag*] is evaluated by evaluating *e*. If an expression of the form *throw*[*x*, *tag1*] is evaluated in the course of evaluating *e* and *tag=tag1* then the evaluation of *e* is abandonded an the value of *x* is returned as the value of the *catch*. If the evaluation of *e* is completed then its value is returned as the value of the *catch*. We will have more to say about this and other mechanisms of controlling evaluation of expressions in a later chapter.

5. Debugging Aids in LISP.

In some cases looking at the list of function calls leading up to an error and at the values of some of the variables is sufficient to pin point an error in a program, however it is often useful to have more powerful debugging facilities available. We will look at two kinds of debugging aids. The first is the sort of debugging package that modifies function definitions to print information about the state of the computation and perhaps interact with the user. This includes such features as tracing and breaking. The second is the sort of program that will take over after a break loop has been entered and help you figure out what has happened.

When a function is traced, each time it is called information is printed out giving the function name and the values of the arguments. When it returns a value an additional line is printed out giving the name of the function and the value being returned. In order to make the output somewhat more readable, the printed lines could be indented according to the depth of nesting of calls to a traced function. To trace a function one replaces the the original function

definition with a program that does the initial printing, applies the original definition to the arguments, does the final printing and returns the value. This program is also responsible for updating any global variables that are used to keep track of the recursion depth of a call.

Thus we could build a small trace package using the following programs.

```
        trace fn ←
            program [def]
                if get[fn, TRACED] then return ALREADY-TRACED
                def ← get[fn, EXPR]
                if ndef then return NOT-DEFINED
                if ¬[adef = LAMBDA] then return NOT-LAMBDA
                putprop[fn, T, TRACED]
                putprop[fn, def, !OLDDEF]
5.1)            putprop[fn,
                        subst[fn,
                            ?FN,
                            subst[adef,
                                ?ARGS,
                                subst[LIST . adef,
                                    *ARGS,
                                    get[!TRACE, !PATTERN]]]],
                        EXPR]
                return OK

        untrace fn ←
            program [def]
                if ¬get[fn, TRACED] then return NOT-TRACED
                def ← get[fn, !OLDDEF]
5.2)            putprop[fn, NIL, TRACED]
                remprop[fn, !OLDDEF]
                putprop[fn, def, EXPR]
                return OK
```

In order for the *trace* routine to work we define the !PATTERN property by evaluating

```
defprop[!TRACE,
        (LAMBDA ?ARGS
          (PROG (!VAL)
            (TERPRI) (MARKS !ILEVEL)
            (PRINC (QUOTE Entering )) (PRINC (QUOTE ?FN))
            (PROG (ARGL)
              (SETQ ARGL (QUOTE ?ARGS))
              L1
              (COND ((NULL ARGL) (RETURN NIL)))
              (TERPRI) (INDENT (PLUS !ILEVEL 2))
              (PRINC (CAR ARGL)) (PRINC (QUOTE  = )) (PRINC (EVAL (CAR ARGL)))
              (SETQ ARGL (CDR ARGL))
              (GO L1))
            (SETQ !VAL
                  ((LAMBDA (!ILEVEL)
                     (APPLY (GET (QUOTE ?FN) (QUOTE !OLDDEF)) *ARGS))
```

```
            (ADD1 !ILEVEL)))
        (TERPRI) (INDENT !ILEVEL)
        (PRINC (QUOTE Returning from )) (PRINC (QUOTE ?FN))
        (PRINC (QUOTE  with )) (PRINC !VAL)
        (TERPRI)
        (RETURN !VAL))),
    !PATTERN]
```

The programs *indent* and *marks* are used by a traced program to print n blanks or n marks say ">". We give the definition of *indent*. The definition of *marks* is similar.

```
indent n ←
    program[i]
        i←1
    loop
5.3)    if i>n then return NIL
        princ " "
        i←i+1
        go to loop
```

Before evaluating a traced function it is necessary to initialize the global variable !ILEVEL.

For example if we say to LISP

```
(!TRACE READ)
(!TRACE READA)
(SETQ !ILEVEL 0)
(READ '(LP LP A RP A DOT B RP))
```

the following will be printed

```
Entering READ
  U = (LP LP A RP A DOT B RP)
>Entering READA
   U = (LP A RP A DOT B RP)
   L = NIL
>>Entering READA
    U = (A RP A DOT B RP)
    L = NIL
>>>Entering READA
     U = (RP A DOT B RP)
     L = (A)
   Returning from READA with ((A) A DOT B RP)
  Returning from READA with ((A) A DOT B RP)
>>Entering READA
    U = (A DOT B RP)
    L = ((A))
>>>Entering READA
     U = (DOT B RP)
     L = (A (A))
>>>>Entering READA
      U = (B RP)
      L = NIL
>>>>>Entering READA
       U = (RP)
```

```
      L = (B)
    Returning from READA with ((B))
   Returning from READA with ((B))
  Returning from READA with (((A) A . B))
 Returning from READA with (((A) A . B))
Returning from READA with (((A) A . B))
Returning from READ with ((A) A . B)

((A) A . B)
```

The other case of debugging by modification of function definitions is breaking. The idea is to modify a function by putting "break points" in the code. The breaks may be conditional or unconditional, they may be inserted at the beginning or before or after certain expressions in the code. Thus we might have a program *mkbreaks*[*fn*, *clauses*] which saves the original definition of *fn*, sets a flag saying it is broken, makes the modifications indicated by the list of break clauses, and makes the new definition. Each break clause describes a position to place a break and the condition under which the break is to occur (the test expression).

Given the ability to "break" the evaluation of an expression we now need some useful routines for examining the state of the world. The functions *evalframe* and *freturn* (or analogous functions) provide us with the necessary primitives, but not it the most useful form. A typical *helper* program might include operations for moving up and down the stack, printing the current expression, evaluating expressions in the current environment, and forcing the broken computation to continue. Then in a break loop *helper* could be invoked, and would go into a command reading loop. Notice that moving down the stack (backwards in time) can be done by alternately applying *evalframe* and selecting the pdl pointer from the list. However moving back up is not so easy. The *helper* program will need to maintain its own stack of pdl pointers.

Exercises

1. Add some additional features to the tracing program. For example it would be useful to have only selected arguments to a function printed at each call. Also printing the value of some additional expression before or after, (or both) the evaluation the function application. You may think of other features you would like to have in a tracing package.

2. Write a program that adds break points to function definitions. You will need to decide how the break points are to be specified.

3. Write code to execute some *helper* commands. Use the *evalframe function* described in section §4. Assume that NIL represents the pointer to the current top of the stack. Use and maintain the global variables POINT and PTLIST which represent the current position in the stack and the list of stack positions from here to the top (to allow you to go up as well as down). Implement the commands UP <n> and DOWN <n> for moving up and down the stack <n> levels, NEXT <fn> which moves to the next occurrence of a call to <fn> down the stack, CE which prints the expression being evaluated at the current position, VAL <exp> which evaluates <exp> in the binding environment corresponding to the current position and prints the result, and FREES which finds all free variables in the current expression and prints the variable names and values in the environment corresponding to the current position. FREES should check that the variable is actually bound in the current environment and if indicate that fact rather that causing an error by trying to evaluate it.

6. Exercises.

1. Write a program to "pretty print" function definitions. The idea is to be able to print with a suitable division into lines and with indentation so that the structure of the program is easier to discover. The program may be similar to *prinlis*. You will need additional special characters, say SPACE and NEWLINE. The program should print subexpressions on a single line when ever possible, align arguments in a function call, and pairs of a conditional when they will not fit on a line. You will probably wish to experiment to see how much indentation is desired in various cases. You will probably be able to think of other conventions that will make the resulting output easier to read.

2. A standard aid in the debugging machine language programs is the ability to single step to the execution of a program. In this mode the processor halts after the evaluation of each instruction, allows the programmer to examine and modify the contents of registers, memory locations, the program counter etc.. We can imagine the process of interpreting and S-expression and being composed of indivisible steps such and applying primitive functions building an environment to evaluate the body of a lambda expression, a branching decision, etc.. Write a program *stepper* which "single-steps" through the evaluation of an expression. A simple version would prints out each expression as it begins to evaluate it, and print out values as they are obtained. A fancier version might go into a read-eval-print loop (maybe a break loop) so that the programmer can examine the state of the computation, modify things, and then continue the single steping. You may want options to bail out now, or evaluate the current expression in fast mode (returning to single stepping if there is any computation pending.

Chapter VII

LISP PROGRAMS FOR SEARCHING

Many programs use the technique of exploring a "search space" of possible solutions in order to solve a problem. It is often convenient to represent the search space as a tree structure with a starting position and some method of getting to successor positions. In this case the search can be controlled by some standard algorithm for visiting all nodes of a tree.

In this chapter we will examine various algorithms for searching. The key feature in all cases is the ability to separate the problem into a basic recursive control program that describes the search and a collection of problem dependent functions. We give examples of specific problems and the collection of functions that characterize the problem for each case.

Searching is an important tool in Artificial Intelligence research and much work has been done on various techniques. Most books treating AI discuss searching. For an introduction to the general topic of searching techniques see Nilsson[1971].

1. Depth First Tree Search.

One algorithm for searching a tree like structure is the depth first search. It consists of taking the first (in some fixed ordering) untried branch at each level until a leaf (terminal position) is reached or until no more branches are left. In either case pop up one level and continue the search until all nodes have been visited.

Simple S-expression recursion can be regarded as a special case of depth first recursion. It is special in that there are exactly two branches, but even more important, the tree is the tree of parts of the S-expression and is present at the beginning of the calculation.

In the general case of tree search recursion, the tree is generated by a *successors* function. We can describe the tree search in a problem independent way by the general depth first tree search function *search*. We have already seen a simple application of *search* in the problem of finding a path to a particular node in a graph (Chapter II §6). We recall here the definition of *search*.

1.1) *search* $p \leftarrow$ if *lose* p then LOSE else if *ter* p then p else *searchlis*[*successors* p]

where

1.2) *searchlis* $u \leftarrow$ if n u then LOSE else {*search* a u}[λx: if x = LOSE then *searchlis* d u else x].

In order to solve a search problem using *search* we must be able to characterize the problem in terms of the auxiliary functions *successors*, *ter*, and *lose* used by *search*.

As a nontrivial example we show how to solve the so-called Instant Insanity puzzle using *search*. First we must describe the puzzle. There are four cubical blocks, and each face of each

block is colored with one of four colors. The object of the puzzle is to build a tower of all four blocks such that each vertical face of the tower involves all four colors. In order to use the above defined function *search* for this purpose, we must decide on a representation of positions and give the functions *lose*, *ter*, and *successors*. A position will be represented by a list of lists - one for each face of the tower. Each sublist is the list of colors of the faces of the blocks showing in that face. Initially the tower is empty. The successors of a position are obtained by adding the next unused block in all possible orientations, eg. by adding the appropriate color to each of the four sublists. We shall assume that the blocks are described in the following longwinded but convenient way. (We'll take up precomputing this description later.) For each block there is a list of the 24 orientations of the block where each orientation is described as a list of the colors around the vertical faces of the block when it is in that orientation. Thus the puzzle is described by a list of lists of lists which we shall call *puzz*.

We now have

1.3) *lose p* ← *orlis*[λ*u*: ¬n *u* ∧ a *u* ∈ d *u*, *p*],
1.4) *ter p* ← [*length* a *p* = 4],
1.5) *successors p* ← *mapcar*[λ*x*: *mapcar*2[λ*yz*: *z.y*, *p*, *x*], *nth*[*puzz*, 1 + *length* a *p*]]

where the functions *mapcar*2 and *nth* are given by

1.6) *mapcar*2[*f*, *u*, *v*] ← if n *u* then NIL else *f*[a *u*, a *v*] . *mapcar*2[*f*, d *u*, d *v*].
1.7) *nth*[*u*, *n*] ← if *n*=1 then a *u* else *nth*[d *u*, *n*−1].

nth picks out the list repesenting the next block in *puzz*. The outer mapping functon runs through the list of orientations returned by *nth* and makes a new position for each one. This is done by the inner mapping function which runs simultaneously through the list tower faces and block faces, putting the block face on the tower face.

A solution to the puzzle is obtained by assigning a value to *puzz* and evaluating *search*[*p*0] with

*p*0 = (NIL NIL NIL NIL).

Obtaining *puzz* in the desired form is as complicated a computation as the actual tree search. It can be conveniently done by a sequence of assignment statements starting with the following description of the blocks:

*puzz*1 ← ((G B B W R G) (G G B G W R) (G W W R B R) (G G R B W W)).

Here each block is represented by a list of the colors of the faces starting with the top face, going around the sides in a clockwise direction and finishing with the bottom face.

We need to go from this description of the blocks to a list of the possible cycles of colors on the vertical faces for the 24 orientations of the block. This not easy, because the order in which we have given the colors is not invariant under rotations of the block. An easy way out is to start with a block whose faces are assigned the numbers 1 thru 6 starting with the top, going clockwise around the sides and finishing with the bottom. We write down one cycle of side colors for each choice of the face put on top and get the list of all 24 cycles by appending the results of generating the cyclic permutations of the cycles. All this is accomplished by the assignment,

puzz2 ← *cycles*[(2 3 4 5)] * *cycles*[(2 5 4 3)] * *cycles*[(1 2 6 4)]
* *cycles*[(1 4 6 2)] * *cycles*[(1 3 6 5)] * *cycles*[(1 5 6 3)].

where the function *cycles* is defined by

1.8) *cycles u* ← *maplist*[λ*v*: *v* * *upto*[*u*, *v*], *u*]
1.9) *upto*[*u*, *v*] ← if *v* = *u* then NIL else a*u* . *upto*[d*u*, *v*].

Next we create a list of substitution lists, one for each block, expressing the colors of the six numbered faces. This is done by the assignment

puzz3 ← *mapcar*[λ*v*: *mapcar2*[*cons*, (1 2 3 4 5 6), *v*], *puzz1*].

We use these substitutions to get for each block the list of 24 orientations of the block. Thus

puzz4 ← *mapcar*[[λ*s*: *mapcar*[[λ*u*: *mapcar*[[λ*x*: d*assoc*[*x*, *s*]], *u*]], *puzz2*]], *puzz3*]

puzz4 has all 24 orientations of the first block while for symmetry reasons we need only consider three as distinct, say the first, ninth, and seventeenth. So we finally get

puzz ← <*nth*[a *puzz4*, 1], *nth*[a *puzz4*, 9], *nth*[a *puzz4*, 17]> . d *puzz4*.

The program when compiled runs about 11 seconds on the KA-10. Interpreted it runs in 2-3 seconds on the KL-10. And we have

search[*p*0] = ((G W R B) (R W G B) (B R G W) (W B G R))

The descriptions for making *puzz3* and *puzz4* make heavy use of mapping functions, which sometimes obscures what is really going on. Alternate descriptions for these constructions are given by the following assignments

puzz3a ← *mapcar*[[λ*v*: *prup*[(1 2 3 4 5 6), *v*]], *puzz1*]

where

prup[*u*, *v*] ← if n*u* then NIL else [a*u* . a*v*] . *prup*[d*u*, d*v*].

and

puzz4a ← *mapcar*[[λ*s*: *sublis*[*puzz2*, *s*]], *puzz3*]

where

sublis[*z*, *a*] ←
    if at*z* then {*assoc*[*z*, *a*]}[λ*zz*: if n*zz* then *z* else d*zz*]
    else *sublis*[a*z*, *a*] . *sublis*[d*z*, *a*].

Of course we have *puzz3* = *puzz3a* and *puzz4* = *puzz4a*.

A more sophisticated representation of face cycles and partial towers makes a factor of ten speedup without changing the basic search algorithm. Here a face cycle is represented by 16 bits in a word, four for each face, a particular one of which being turned on tells us the color of the face. If we bool-or these words together for the blocks in a partial tower we get a word which tells us for each face of the tower what colors have been used. A particular face cycle from the next block can be added to the tower if the bool-and of its word with the word representing the tower

is zero. We represent a position by a list of words representing its partial towers with 0 as the last element and the highest partial tower as the first element. The virtue of this representation is that it makes the description of the algorithm short. The initial position is (0). The new *puzz* can be formed from the old one by the assignment

puzza ← *mapcar*[λu: *mapcar*[*bool-cycle*, *u*], *puzz*]

where

```
1.10)   bool-cycle v ← if n v then 0 else bool-color a v + 16 • bool-cycle d v
1.11)   bool-color x ← if x = R then 1 else if x = W then 2 else if x = G then 4 else 8.
```

Now we need new versions of *lose*, *ter*, and *successors*. These are

```
1.12)   lose p ← false ,
1.13)   ter p ← [length p = 5],
1.14)   successors p ← mapchoose[[λx: [a p bool-and x] = 0],
                                 [λx: [a p bool-or x] . p], nth[puzza, length p]]
```

where

```
        mapchoose[pred, fn, u] ←
             if n u then NIL
1.15)        else if pred a u then fn[a u] . mapchoose[pred, fn, d u]
             else mapchoose[pred, fn, d u].
```

lose is trivial, because the *mapchoose* is used to make sure that only non-losing new positions are generated by *successors*. This version runs in a little less than one second on the KA-10. A greater speedup can be made by the application of some mathematics. In fact, with enough mathematics, extensive tree search is unnecessary in this problem.

2. Game trees.

The positions that can be reached from an initial position in a game form a tree, and deciding what move to make often involves searching this tree. However, game trees are searched in a different way than the trees we have looked at previously, because the opposing interests of the players makes it not a search for a joint line of play that will lead to the first player winning, but rather a search for a strategy that will lead to a win regardless of what the other player does. In this section we discuss two methods for searching a game tree. The first is a simple minimax search which examines all lines of play before making a decision. The second makes use of a heuristic known as the $\alpha\beta$-heuristic to prune the search space. In both cases the basic recursive structure of the computation is presented in a game independent fashion by letting the game be characterized by a collection of functions which satisfy some general conditions. The game of 2-dimensional Tic Tac Toe will be used as an example application of these game tree searching techniques.

In the simplest situation the game is characterized by a function *successors* that gives the positions that can be reached in one move, a predicate *ter* that tells when a position is to be

regarded as terminal for the given analysis, and a function *imval* that gives a number approximating the value of a terminal position to one of the players. We shall call this player the maximizing player and his opponent the minimizing player. Usually, the numerical values arise, because the search cannot be carried out to the end of the game, and the analysis stops with reasonably static positions that can be evaluated by some rule. Naturally, the function *imval* is chosen to be easy to calculate and to correlate well with the probability that the maximizing player can win the position.

Consider the game of Tic Tac Toe played in 2-dimensions. A position is determined by knowing which of the nine squares contain X's, O's, or blanks. The successors of a position in the case that it is the X-players turn to move are all those positions obtained by putting an X in a blank square. A position is terminal if there are three X's or three O's on a line or diagonal, or if there are no blank squares. If we take the X-player to be the maximizing player then we could assign values to the terminal positions as follows. In the case of three X's the value is 1, in the case of three O's the value is -1 and otherwise it is 0.

The simple minimax rule for finding the correct move in a position uses a pair of programs (*vmin*, *vmax*). *vmax* gives the value of a position to the maximizing player by using *imval* if the position is terminal and taking the max of the values of the successor positions to the minimizing player otherwise. *vmin* similiarly gives the value of a position to the minimizing player.

For this we want programs for getting the maximum or the minimum of a function on a list, and they are defined as follows:

2.1) $$maxlis[u, f] \leftarrow \text{if } n\,u \text{ then } -\infty \text{ else } max[f[a\,u], maxlis[d\,u, f]]$$

2.2) $$minlis[u, f] \leftarrow \text{if } n\,u \text{ then } \infty \text{ else } min[f[a\,u], minlis[d\,u, f]]$$

Here the symbols $-\infty$ and ∞ represent numbers that are smaller and larger than any actual values that will occur in evaluating f. If these numbers are not available, then it is necessary to prime the program with the values of f applied to the first element of the list, and the programs are then meaningless for null lists. Iterative versions of these programs are generally better; we give only the iterative version of *maxlis*, namely

2.3)
$$maxlis[u, f] \leftarrow maxlisa[u, -\infty, f],$$
$$maxlisa[u, x, f] \leftarrow \text{if } n\,u \text{ then } x \text{ else } maxlisa[d\,u, max[x, f[a\,u]], f]$$

We now have

2.4) $$vmax\ p \leftarrow \text{if } ter\ p \text{ then } imval\ p \text{ else } maxlis[successors\ p, vmin]$$

2.5) $$vmin\ p \leftarrow \text{if } ter\ p \text{ then } imval\ p \text{ else } minlis[successors\ p, vmax]$$

The best move is determined by the pair of programs (*movemax*, *movemin*). The key to the computation is the pair of programs (*bestmaxa*, *bestmina*) which determine the optimal move from a list of moves. They call each other recursively and differ only in that one is for when the maximizing player is to move and the other is for when the minimizing player is to move. The argument *bestmove* is the best move found so far and *bestval* is value of the resulting position as computed by f. Here are the definitions.

2.6) *movemax p* ← *bestmax*[*succesors p*, *vmin*],

2.7) *movemin p* ← *bestmin*[*succesors p*, *vmax*],

where

bestmax[*u*, *f*] ← *bestmaxa*[d *u*, a *u*, *f*[a *u*], *f*],

2.8)

bestmaxa[*u*, *bestmove*, *bestval*, *f*] ← if n *u* then *bestmove*
else if *f*[a *u*] ≤ *bestval* then *bestmaxa*[d *u*, *bestmove*, *bestval*, *f*]
else *bestmaxa*[d *u*, a *u*, *f*[a *u*], *f*],

and

bestmin[*u*, *f*] ← *bestmina*[d *u*, a *u*, *f*[a *u*], *f*],

2.9)

bestmina[*u*, *bestmove*, *bestval*, *f*] ← if n *u* then *bestmove*
else if *f*[a *u*] ≥ *bestval* then *bestmina*[d *u*, *bestmove*, *bestval*, *f*]
else *bestmina*[d *u*, a *u*, *f*[a *u*], *f*].

This straight minimax computation of the best move does much more computation in general than is necessary. The simplest way to see this is from the move tree of Figure 11.

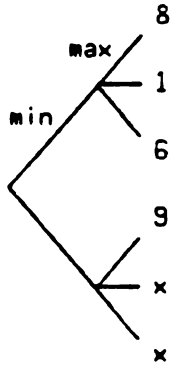


Figure 11. Move tree showing how simple minimax does too much work.

We see from this figure that it is unnecessary to examine the moves marked × because their values have no effect on the value of the game or on the correct move since the presence of the 9 is sufficient information at this point to show that the move leading to the vertex preceding it should not be chosen by the minimizing player.

The general situation is that it is unnecessary to examine further moves from a position once a move is found that refutes moving to that position in the first place. This idea is called the αβ-heuristic. As an example of how this heuristic might be used, we describe three collections

of programs: (*valmax*, *valmin*), (*linemax*, *linemin*) and (*treemax*, *treemin*). The programs (*valmax*, *valmin*) return the value of the game to the corresponding player, (*linemax*, *linemin*) return the value and a line of play obtaining that value, and (*treemax*, *treemin*) return the value and a "proof" that this is the optimal value for the corresponding player. The "proof" consists of two trees of moves and terminal values giving a strategy guaranteeing in one case that the value of the game is at least the value returned and in the other case that it is at most the value returned. Each pair of programs uses an auxiliary pair of programs which do the processing of successor lists. Thus in each group there are four mutually interacting programs.

The use of the αβ-heuristic is the same in each group, they differ in the information that they are carrying around as the search progresses. As before we assume that the functions *successors*, *ter*, and *imval* characterize the game. For purposes of determining properties of *valmax* and friends, the program *rectify* can be assumed to compute the identity function. It is included in order to allow a more efficient representation of the game and will be discussed in the next section. *ext* is used to extract from a position the move leading to that position.

The programs (*valmax*, *valmin*) each take three arguments: a position *p*, and the parameters *alpha* and *beta*. *alpha* is the maximum value that can be guaranteed the maximizing player based on the search so far, and *beta* is the least value that can be guaranteed the minimizing player based on the search so far. Initially *alpha* is essentially $-\infty$ and *beta* is ∞. *valmax* works as follows. It checks to see if the position is terminal, and if so returns *imval p* directly, otherwise it asks *vmaxlis* for the best value (with respect to *alpha* and *beta*) among the successor positions. *vmaxlis* goes through a list of positions, computes the value of each one to the minimizing player, using *valmin*, and does one of three things. If the value is not greater than *alpha*, it is ignored as we are guaranteed to be able to do as well or better by a previously found move. If the value is not less than *beta* then the computation is aborted as *beta* is the best that the maximizing player can hope for and the minimizing player would not allow this line of play if it produced a higher value. If the value is greater than *alpha* but less than *beta* then this is a better choice for the maximizing player and *alpha* is updated. When the list is exhausted *alpha* represents the maximum value that the maximizing player can be guaranteed starting from the position that gave rise to the initial list of moves. *valmin* works in a dual fashion. The definitions are:

```
                valmax[p, alpha, beta] ←
2.10)               if ter[rectify p, alpha, beta] then imval p
                    else vmaxlis[successors p, alpha, beta]

                vmaxlis[u, alpha, beta] ←
                    if n u then alpha
                    else {valmin[a u, alpha, beta]}[λs:
2.11)                   if ¬[s > alpha] then vmaxlis[d u, alpha, beta]
                        else if s < beta then vmaxlis[d u, s, beta]
                        else beta]
```

```
                valmin[p, alpha, beta] ←
2.12)               if ter[rectify p, alpha, beta] then imval p
                    else vminlis[successors p, alpha, beta]

                vminlis[u, alpha, beta] ←
                if n u then beta
                else {valmax[a u, alpha, beta]}[λs:
2.13)                   if ¬[s > alpha] then alpha
                        else if s < beta then vminlis[d u, alpha, s]
                        else vminlis[d u, alpha, beta]]
```

We see that (*valmax*, *valmin*) are related to (*vmax*, *vmin*) as follows:

```
                valmax[p, alpha, beta]=
                        if vmax[p]≤alpha then alpha
2.14)                   else if alpha<vmax[p]<beta then vmax[p]
                        else if beta≤vmax[p] then beta

                valmin[p, alpha, beta]=
                        if vmin[p]≤alpha then alpha
2.15)                   else if alpha<vmin[p]<beta then vmin[p]
                        else if beta≤vmin[p] then beta
```

Thus if we initialize *valmax* or *valmin* with a sufficiently small *alpha* and a sufficiently large *beta* we will get the same answer as given by the brute force search. If we miss, we can tell and try again with a better estimate.

The programs (*linemax*, *linemin*) take three arguments, the same as for (*valmax*, *valmin*). The additional argument *line* required by (*lmaxlis*, *lminlis*) corresponds to the first line of play (list of moves) found which terminates in a position of value *alpha* for *linemax* or *beta* for *linemin*. (*lmaxlis*, *lminlis*) either pass *line* along in the case that the value found is not an improvement, or construct a new *line* from the one returned by (*linemin*, *linemax*) in the case that it leads to an improved value. The result returned is the optimal value *cons*ed onto the list of moves giving the first line of play found attaining that value. The definitions are:

```
                linemax[p, alpha, beta] ←
2.16)               if ter[rectify p, alpha, beta] then <imval p>
                    else lmaxlis[successors p, alpha . ALPHA-CUTOFF, alpha, beta]

                lmaxlis[u, line, alpha, beta] ←
                    if n u then alpha . line
                    else {linemin[a u, alpha, beta]}[λs:
2.17)                   if ¬[a s > alpha] then lmaxlis[d u, line, alpha, beta]
                        else if a s < beta then lmaxlis[d u, ext a u . d s, a s, beta]
                        else beta . line]
```

```
                linemin[p, alpha, beta] ←
2.18)               if ter[rectify p, alpha, beta] then <imval p>
                    else lminlis[successors p, beta . BETA-CUTOFF, alpha, beta]

                lminlis[u, line, alpha, beta] ←
                if n u then beta . line
                else {linemax[a u, alpha, beta]}[λs:
2.19)                   if ¬[a s > alpha] then alpha . line
                        else if a s < beta then lminlis[d u, ext a u . d s, alpha, a s]
                        else lminlis[d u, line, alpha, beta]]
```

The programs (*treemax*, *treemin*) take the same arguments as the value and line versions do The programs (*tmaxlis*, *tminlis*) each take two additional arguments, *trmax* and *trmin*. These are move trees including terminal values, which are used to build the proofs previously mentioned. If a position p is non-terminal and u = *successors* p then *tmaxlis*[u, NIL, NIL, α, β] returns a list <*val*, *pfmax*, *pfmin*> where one of the following is true:

1.) $val \leq \alpha$ and *pfmin* is a proof that the value of p to the maximizing player is at most *val*.

2.) $\alpha < val < \beta$, *pfmax* is a proof that the value of p to the maximizing player is at least *val* and *pfmin* is a proof that it is at most *val*.

3.) $\beta \leq val$ and *pfmax* is a proof that the value of p to the maximizing player is at least *val*.

Similarly *treemin*[u, NIL, NIL, α, β] returns a list <*val*, *pfmax*, *pfmin*> where the value is now relative to the minimizing player.

Note that in the case $val \leq \alpha$ ($\geq \beta$) we make no requirement of *pfmax* (*pfmin*). This is in these cases the searches were probably incomplete and the resulting proof would make no sense. The definitions of are:

```
                treemax[p, alpha, beta] ←
2.20)               if ter[rectify p, alpha, beta] then {imval p}[λv: <v, <v>, <v>>]
                    else tmaxlis[successors p, alpha . ALPHA-CUTOFF, NIL, alpha, beta]

                tmaxlis[u, trmax, trmin, alpha, beta] ←
                    if n u then <alpha, trmax, trmin>
                    else {treemin[a u, alpha, beta]}[λs:
                        if ¬[a s > alpha] then
2.21)                       tmaxlis[d u, trmax, [ext a u . add s] . trmin, alpha, beta]
                        else if a s < beta then
                            tmaxlis[d u, ext a u . ad s, [ext a u . add s] . trmin, a s, beta]
                        else <beta, ext a u . ad s, NIL>]
```

```
        treemin[p, alpha, beta] ←
2.22)       if ter[rectify p, alpha, beta] then {imval p}[λv: <v, <v>, <v>>]
            else tminlis[successors p, NIL, beta . BETA-CUTOFF, alpha, beta]

        tminlis[u, trmax, trmin, alpha, beta] ←
            if n u then <beta, trmax, trmin>
            else {treemax[a u, alpha, beta]}[λs:
                if ¬[a s > alpha] then <alpha, NIL, ext a u . add s>
2.23)           else if a s < beta then
                    tminlis[d u, [ext a u . ad s] . trmax, ext a u . add s, alpha, a s]
                else tminlis[d u, [ext a u . ad s] . trmax, trmin, alpha, beta]]
```

We now describe how a move tree, *pf*, represents a proof of the sort required above. There are four cases.

1) *pf* is a proof that the value of *p* to the maximizing player is at least *val* if one of the following holds:

(i) *p* is terminal and *pf* = <*imval p*> and *imval p* ≥ *val*

(ii) *p* is not terminal and *pf* = *mv* . *pf'* where *mv* is a move leading to *p'* ∈ *successors p* and *pf'* is a proof that the value of *p'* to the minimizing player is at least *val*.

2) *pf* is a proof that the value of *p* to the minimizing player is at least *val* if one of the following holds:

(i) *p* is terminal and *pf* = <*imval p*> and *imval p* ≥ *val*.

(ii) *p* is not terminal and *pf* is a list of elements of the form *mv* . *pf'*, one for each *p'* ∈ *successors p*, such that *mv* is a move leading from *p* to *p'* and *pf'* is a proof that the value of *p'* to the maximizing player is at least *val*.

Cases 3) and 4) are obtained from 1) and 2) by interchanging the words maximizing and minimizing and reversing the sense of the inequalities throughout.

The αβ algorithm can also be used to in the case that there is a way to estimate the value of a position *p*. Suppose we estimate that *a* < *vmax p* < *b*. Then if *valmax*[*p*, *a*, *b*] = *val* with *a* < *val* < *b* then we win as we now have the correct value and hopefully the narrower range caused even more pruning than usual. If *val* ≤*a* then we must lower the estimate of the value and try again. Similarly if *val* ≥ *b* then we raise the estimate.

The reduction in number of positions examined given by the αβ algorithm over the simple minimax algorithm depends on the order in which the moves are examined. In the worst case, the moves happen to be examined in inverse order of merit in every position on the tree, i.e. the worst move first. In that case, there is no improvement over minimax. The best case is the one in which the best move in every position is examined first. If we look *n* moves deep on a tree that has *k* successors to each position, then minimax looks at k^n positions while αβ looks at about only $k^{n/2}$. Thus a program that looks at 10^4 moves with αβ might have to look at 10^8 with minimax. For this reason, game playing programs using αβ make a big effort to include as good

as possible an ordering of moves into the *successors* function. When there is a deep tree search to be done, the way to make the ordering is with a short look-ahead to a much smaller depth than the main search. Still shorter look-aheads are used deeper in the tree, and beyond a certain depth, non-look-ahead ordering methods are of decreasing complexity.

A version of $\alpha\beta$ incorporating optimistic and pessimistic evaluations of positions was first proposed by McCarthy about 1958. Edwards and Hart at M.I.T. about 1959 proved that the present version of $\alpha\beta$ works and calculated the improvement it gives over minimax. The first publication, however, was Brudno [1963]. For additional discussion of $\alpha\beta$ techinques see Nilsson[1971], Knuth and Moore[1975] or Winston[1977].

It is worth noting that $\alpha\beta$ was not used in the early chess playing programs in spite of the fact that it is clearly used in any human play. Its non-use, therefore, represents a failure of self-observation. Very likely, there are a number of other algorithms used in human thought that we have not noticed and incorporated in our programs. To the extent that this is so, artificial intelligence will be *a posteriori* obvious even though it is *a priori* very difficult.

3. The hidden board trick.

The program *rectify* is used to implement a particular method of representing a game known as the "hidden board trick". It basically relies on side effects to produce an alternate representation of a given position when it is needed. Here a position is represented by a list of moves leading to it with the most recent move at the head of the list and the first move of the game at the end of the list. This is an efficient representation with respect to storage for two reasons. One is that lists of positions will frequently have only the last few moves differing and can be represented as merging list structures. The second is that other representations frequently take the form of a collection of tables and the representation of each position requires a lot of space. However it is often easier to compute the successors, determine if a position is terminal, and compute its value using a more spacious representation. To take advantage of this fact we keep around a representation of the position of current interest, the hidden board, in a form convenient for doing these computations. A list of moves leading to that position is also kept. The function *rectify* is used to make a new position current. It returns the new position unchanged, but has the side effect of making it the current position as far as the alternate representation is concerned. This is accomplished by using *commontail* to find out where in the past the current and new positions meet , backing up to this position and then moving forward along the path of the new position. This requires the additional game dependent functions *revert* for taking back moves, and *update* for making moves. The programs for *rectify* and its auxiliaries follow.

```
                rectify p ←
                  program [z, q]
                     q ← commontail[p, p1]
                   L1: if q = p1 then go to L2
                     revert[]
3.1)                 go to L1
                   L2: z ← listsubt[p, p1]
                   L3: if nz then return p
                     update az
                     z ← dz
                     go to L3

3.2)            commontail[u, v] ← reverse commonhead[reverse u, reverse v]

                commonhead[u, v] ←
3.3)              if nu ∨ nv ∨ ¬[au = av] then NIL
                  else au . commonhead[du, dv]

                listsubt[u, v] ← listsubta[u, length u − length v, NIL]
3.4)
                listsubta[u, n, z] ←
                  if n = 0 then z else listsubta[du, sub1 n, au . z]
```

4. 2-dimensional Tic Tac Toe.

As a simple example of how the programs described in the previous two sections can be used we give a set of programs characterizing the game of 2-dimensional Tic Tac Toe and show the results of some computations using the programs employing the αβ-heuristic. This characterization of Tic Tac Toe improves the efficiency of the search routines by doing some precomputing of position values and thus pruning and ordering the list of successors for a given position.

The nine squares of the Tic Tac Toe board are numbered from left to right starting from the top row. The eight lines are numbered similarly with the horizontal lines first, then vertical, then the two diagonals. There are several global arrays and variables. These are initialized by the programs *commence* and *newgame*. The ith element of *lines* is a list of lines containing the ith square. *xcount* tells how many X's are on a given line, *ocount* does the same for O's. *p1* is the list of moves leading to the position represented by the state of the global variables. (A move is represented by the number of the square filled.) *xs*, *os*, and *bs* are lists of the squares containing X's, O's, and blanks respectively. *w* tells whose turn it is to move. If T then it is the O-players turn, if NIL then it is the X-players turn. *level* is the number of moves so far (the length of *p1*). *count* is the total number of moves examined.

For the X-player a winning position has value 10−*level*, any other terminal position has value 0. For the O-player a winning position has value *level*−10 and any other terminal position has value 0.

A position is considered terminal if there are no blank spaces left, or if $10 - level \leq \alpha$, or if $-10 + level \geq \beta$, or if it is the X-players turn to move and *xcount* is 3 for some line, or if it is the O-players turn and *ocount* is 3 for some line.

successors uses *sort* and its auxiliaries to prune and sort the list of possible moves from a non-terminal position according to the following rules. If some move is a win then return a list containing only that move, otherwise if some move is required to keep the opponent from winning then that is returned as the single possibility, otherwise if some move is a double-threat (creates two winning moves for the player) then it is returned as the only possibility, otherwise a list of all possible moves is returned with the threats (moves creating a winning move for the player) at the front of the list.

The programs *revert* and *update* take back and make moves in the obvious way.
Here are the definitions: (note that the numbers are octal)

```
        commence[] ←
          program []
              array[lines, T, 12]
              array[xcount, fixnum, 11]
              array[ocount, fixnum, 11]
              store[lines 1, (1 4 7)]
              store[lines 2, (1 5)]
4.1)          store[lines 3, (1 6 10)]
              store[lines 4, (2 4)]
              store[lines 5, (2 5 7 10)]
              store[lines 6, (2 6)]
              store[lines 7, (3 4 10)]
              store[lines 10, (3 5)]
              store[lines 11, (3 6 7)]

4.2)    ext p ← a p

        newgame[] ←
          program [n]
              n ← 0
            L: n ← add1 n
              store[xcount n, 0]
              store[ocount n, 0]
              if n < 10 then go to L
4.3)          p1 ← NIL
              xs ← NIL
              os ← NIL
              bs ← (1 2 3 4 5 6 7 10 11)
              w ← NIL
              level ← 0
              count ← 0
              return (NEW GAME)
```

```
ter[p, alpha, beta] ←
  ¬n p
  ∧ [level = 11
    ∨ 11 - level < alpha
    ∨ -11 + level > beta
4.4)  ∨ [program [n]
          n ← 0
        L2: n ← add1 n
          if 3 = [if w then xcount n else ocount n] then
            return T
          if n < 10 then go to L2
          return NIL]]

imval p ←
  if w then
    [program [n]
       n ← 0
     L3: n ← add1 n
4.5)    if 3 = xcount n then return 12 - level
        if n < 10 then go to L3 else return 0]
  else program [n]
         n ← 0
       L4: n ← add1 n
         if 3 = ocount n then return -12 + level
         if n < 10 then go to L4 else return 0

4.6)  successors p ← sort mapcar[[λx: x . p], bs]

revert[] ←
  program [a]
     level ← sub1 level
     bs ← a[if w then xs else os] . bs
     if w then xs ← dxs else os ← dos
     a ← lines a p1
4.7)  L5: if na then go to L6
     if w then store[xcount a a, sub1 xcount a a]
      else store[ocount a a, sub1 ocount a a]
     a ← da
     go to L5
   L6: w ← ¬w
     p1 ← dp1
     return NIL
```

```
          update m ←
            program [a]
                level ← add1 level
                if w then os ← m . os else xs ← m . xs
                bs ← delete[m, bs]
                p1 ← m . p1
4.8)            count ← add1 count
                a ← lines m
             L7: if n a then go to L8
                if w then store[ocount a a, add1 ocount a a]
                 else store[xcount a a, add1 xcount a a]
                a ← d a
                go to L7
             L8: w ← ¬w
                return[]

4.9)      sort u ← sorta[u, NIL, NIL]

          sorta[u, th, ord] ←
            if n u then [th * ord]
            else if win a u then <a u>
4.10)       else if answer a u then sortb[d u, a u]
            else if doubleth a u then sortc[d u, a u]
            else if threat a u then sorta[d u, a u . th, ord]
            else sorta[d u, th, a u . ord]

          sortb[u, m] ←
4.11)       if n u then <m> else if win a u then <a u> else sortb[d u, m]

          sortc[u, m] ←
            if n u then <m>
4.12)       else if win a u then <a u>
            else if answer a u then sortb[d u, a u]
            else sortc[d u, m]

          win p ← if w then orlis[[λx: 2 = ocount x], lines a p]
4.13)           else orlis[[λx: 2 = xcount x], lines a p]

          answer p ←
4.14)       if w then orlis[[λx: 2 = xcount x], lines a p]
            else orlis[[λx: 2 = ocount x], lines a p]

          doubleth p ←
            twolis[[λx:
4.15)             1 = [if w then ocount x else xcount x]
                ∧ orlis[[λw: x ∈ lines w], delete[a p, bs]]],
              lines a p]

          twolis[pred, u] ←
4.16)       ¬n u ∧ [[pred a u ∧ orlis[pred, d u]] ∨ twolis[pred, d u]]
```

```
                threat p ←
                  orlis[[λx:
4.17)                   1 = [if w then ocount x else xcount x]
                        ∧ orlis[[λw: x ∈ lines w], delete[a p, bs]]],
                       lines a p]

                delete[x, u] ←
                  if n u then NIL
4.18)             else if x = a u then d u
                  else a u . delete[x, d u]
```

Below we give some examples of the results returned by (*lmin*, *lmax*) and (*tmin*, *tmax*). In each case the situation is 2 moves into a game where the X player has moved first and hence it is the X players turn to move.

```
Position numbering:
        (1  2  3)
        (4  5  6)
        (7 10 11)

Example 1.

    Board:  (O X _)
            (_ _ _)
            (_ _ _)

lmax[p1, -1000, 1000] = (0 5 10 6 4 7 3 11)

tmax[p1,-1000, 1000] =
       (0
        (5 (10 6 (4 7 (3 11 0))))
        ((3 7 (4 11 (5 10 -2)))
         (4 5 (11 7 (3 6 (10 0))))
         (6 5 (11 3 (7 10 (4 0))))
         (7 5 (11 10 (4 6 (3 0)) (3 6 (4 0)) (6 3 (4 0))))
         (11 7 (4 5 (3 6 (10 0))))
         (5 10 (11 4 (7 3 (6 0)))
               (7 3 (11 6 (4 0)) (4 6 (11 0)) (6 4 (11 0)))
               (3 7 (4 11 -2))
               (4 6 (11 3 (7 0)) (7 3 (11 0)) (3 7 (11 0)))
               (6 4 (7 3 (11 0))) )
         (10 5 (11 7 (3 4 -2))) ) )


Example 2.

    Board:  (O _ X)
            (_ _ _)
            (_ _ _)

lmax[p1, -1000, 1000] = (3 11 6 7 5 10)

tmax[p1,-1000, 1000] =
       (3
        (11 (6 7 (5 10 3)))
        ((2 7 (4 11 (5 0)))
         (4 5 (11 6 (2 0) (10 0) (7 0)))
         (10 11 (5 7 (2 3)))
         (5 7 (4 6 (11 0) (10 0) (2 0)))
         (6 11 (5 7 (4 3)))
         (7 5 (11 6 (10 3)))
         (11 6 (7 5 (10 3))) ) )
```

Chapter VIII

COMPILING IN LISP

Compiling is an important example of symbolic computation and has received much study. Much of the study has been devoted to parsing which is essentially the transformation of an input string in the source language into an internal form. The internal form used depends on the compiler. Sometimes it is Polish prefix or postfix notation, sometimes it is list structure, and sometimes it consists of entries in a collection of tables.

When internal notation LISP is being compiled, the parsing is trivial, because the input is S-expressions and the internal form wanted is list structure, so what parsing there is is done by the ordinary LISP *read* routine. Therefore, compilers can be very compact and transparent in structure, and we can concentrate our attention on the code generation phase. This is as it should be, because, parsing is basically a side issue in compiling, and code generation is the matter of main scientific interest.

We shall describe two compilers in this chapter called LCOM0 and LCOM4 which compile S-expression LISP into machine language for the PDP-10 computer according to the conventions of MACLISP. Before describing the compilers, we must describe these conventions.

1. Some facts about the PDP-10.

The target language is called LAP for LISP assembly program. Each function is compiled separately into a separate LAP program, and these programs are written onto a disk file. These files can later be read into a LISP core image and assembled in place by the LAP assembler which is part of the LISP runtime routines. The compiler, on the other hand, is a separate LISP core image that can be instructed to compile several input files. For an input file called <name>, it produces an output file called <name>.LAP. All this is specific to the time-sharing systems for the PDP-10 and DECSYSTEM-20.

The LAP program produced is a list of words; the last word is NIL, and the first word is a header of the form (LAP *fname* SUBR) where *fname* is the name of the function compiled. This header tells the DSKIN program that it should read S-expressions till it comes to NIL and then submit what it has collected to the LAP assembler which will assemble it into core. The rest of the words are either atoms representing labels or lists representing single PDP-10 instructions.

The following facts about the PDP-10 may be of use:

The PDP-10 has a 36 bit word and an 18 bit address. In instructions and in accumulators used as index registers the address is found in the right part of the word where the least significant bits in arithmetic reside. The compilers we shall describe don't need to know the word length, since this is handled by LAP.

There are 16 general registers which serve simultaneously as accumulators (receiving the results of arithmetic operations), index registers (modifying the nominal addresses of instructions

to form effective addresses), and as the first 16 registers of memory (if the effective address of an instruction is less than 16, then the instruction uses the corresponding general register as its operand.) The fact that there are sixteen general registers is not known to the compiler, but it will produce non-working code if it is given functions to compile that have too many arguments.

All instructions have the same format and are written for the LAP assembly program in the form

(<op name> <accumulator> <address> <index register>).

Thus (MOVE 1 3 P) causes accumulator 1 to receive the contents of a memory register whose address is 3+c(P), i.e. 3+<the contents of general register P>. <address> may be a number, a label, or an S-expression constant in the form (QUOTE *e*) where *e* is an S-expression. The latter allows a quoted S-expression to be loaded into accumulator 1 by the instruction (MOVLI 1 (QUOTE *e*)). An addtional form that appears in the address field for some instructions is (% 0 0 *m m*). This is a literal with value a word containing *m* in the lefthand side and *n* in the righthand side. Which form of <address> is meaningful depends on the instruction.

To determine the effective address of an instruction the following rules apply. The value of <address> is combined with the contents of the index register to form a direct address. If no index register is specified, then just the value of <address> is used. If an @ occurs in the list as a separate symbol, then the memory reference is indirect, i.e. the effective address is the contents of the right half of the word directly addressed by the instruction (modified by the index register and indirect bit of that word).

In the following description of some instructions useful in constructing the compiler, <ef> denotes the effective address of an instruction and ac denotes the accumulator.

MOVE	c(ac) ← c(<ef>)
MOVEI	c(ac) ← <ef>
MOVEM	c(<ef>) ← c(ac)
HLRZ	right half of c(ac) ← left half of c(<ef>)
HRRZ	right half of c(ac) ← right half of c(<ef>)
(These two are used indirectly for car and cdr respectively.)	
ADD	c(ac) ← c(ac) + c(<ef>)
SUB	c(ac) ← c(ac) - c(<ef>)
JRST	go to <ef>
JUMPE	if c(ac) = 0 then go to <ef>
JUMPN	if c(ac) ≠ 0 then go to <ef>
CAME	if c(ac) = c(<ef>) then <skip next instruction>
CAMN	if c(ac) ≠ c(<ef>) then <skip next instruction>
PUSH	c(c(right half of ac)) ← c(<ef>); the contents of each half of ac is increased by one and if the contents of the left half is then 0, a stack overflow interrupt occurs. (PUSH P ac) is used in LISP to put c(ac) on the stack.
POP	c(<ef>) ←c(c(right half of ac)); the contents of each half of ac are then decreased by 1. This may be used for removing the top element of the stack.

Thus (POP P I) puts the top element of the stack in accumulator I. The i-th element of the stack is obtained by (MOVE I @ i P).

POPJ (POPJ P) is used for returning from a subroutine

These instructions are adequate for compiling basic LISP code with the addition of the subroutine calling pseudo-instrucion. The form of this instruction is (CALL *n* (QUOTE <subr>)) . It is used for calling the LISP subroutine <subr> with *n* arguments. The convention is that the arguments will be stored in successive accumulators beginning with accumulator 1, and the result will be returned in accumulator 1. In particular the functions ATOM and CONS are called with (CALL 1 (QUOTE ATOM)) and (CALL 2 (QUOTE CONS)) respectively. Programs produced by you will be called similarly when their names are referred to. The details of how CALL works are unimportant here; it actually traps to a machine language routine that checks whether the function is being traced and which can replace the CALL by PUSHJ for greater speed when tracing is definitely not wanted.

2. Code produced by LISP compilers.

We will discuss two compilers, a simple one called LCOM0 and a more optimising compiler called LCOM4. LCOM4 produces about half as many instructions for a given function as does LCOM0. Besides these, there are the standard PDP-10 LISP compiler written at M.I.T. by Richard Greenblatt and Stuart Nelson and modified by Whitfield Diffie and the MACLISP compiler NCOMPLR.

At the end of this section are examples of the output of each of the compilers mentioned above for the file containing:

```
(DEFPROP DROP
 (LAMBDA(U)
  (COND ((NULL U) NIL) (T (CONS (LIST (CAR U)) (DROP (CDR U)))))))
EXPR)
```

The following comments are with regard to these examples.

1. Note that all four compilers produce the same first line of heading. This is necessary for the LAP assembly program which also requires the NIL at the end as punctuation. The standard compiler and NCOMPLR also use a function called XCONS that has the same effect as CONS except that it receives its arguments in the reverse order which is sometimes convenient. This requires, of course, that the compiler be smart enough to decide where it is more convenient to put the arguments of the *cons* function. They also use NCONS rather than LIST to form a list of one element.

2. The two compilers, LCOM0 and LCOM4, are similar in structure, but LCOM4, the better one, which is twice as long, uses the following optimisations.

2.1. When the argument of CAR or CDR is a variable, it compiles a (HLRZ 1 @ *i* P) or

(IIRRZ 1 0 *i* P) which gets the result through the stack without first compiling the argument into an accumulator.

2.2. When it has to set up the arguments of a function in the accumulators, in general, the program must compute the arguments one at a time and save them on the stack, and then load the accumulators from the stack. However, if one of the arguments is a variable, is a quoted expression, or can be obtained from a variable by a chain of CARs and CDRs, then it need not be computed until the time of loading accumulators since it can be computed using only the accumulator in which it is wanted.

3. The Diffie compiler produces about 10 percent less code than LCOM4; the main difference seems to be that it sometimes notices when it has an expression that it will need later. Sometimes this feature leads it astray. For example, it may save *a u* for later use at greater cost than re-computing it.

4. NCOMPLR is noted particularly for its efficient code for numerical computations. Hence the "N". Except for some bookkeeping code initially, the Diffie compiler and NCOMPLR produce similar code for this example.

****

LCOM0 produces the following code for the MACLISP LAP assembler.

```
(LAP DROP SUBR)
(PUSH P 1)
(MOVE 1 0 P)
(PUSH P 1)
(MOVE 1 0 P)
(SUB P (% 0 0 1 1))
(CALL 1 (QUOTE NULL))
(JUMPE 1 G0003)
(MOVEI 1 0)
(JRST 0 G0002)
G0003
(MOVEI 1 (QUOTE T))
(JUMPE 1 G0004)
(MOVE 1 0 P)
(PUSH P 1)
(MOVE 1 0 P)
(SUB P (% 0 0 1 1))
(CALL 1 (QUOTE CAR))
(PUSH P 1)
(MOVE 1 0 P)
(SUB P (% 0 0 1 1))
(CALL 1 (QUOTE LIST))
(PUSH P 1)
(MOVE 1 -1 P)
(PUSH P 1)
(MOVE 1 0 P)
(SUB P (% 0 0 1 1))
(CALL 1 (QUOTE CDR))
(PUSH P 1)
(MOVE 1 0 P)
(SUB P (% 0 0 1 1))
(CALL 1 (QUOTE DROP))
(PUSH P 1)
(MOVE 1 -1 P)
(MOVE 2 0 P)
(SUB P (% 0 0 2 2))
(CALL 2 (QUOTE CONS))
(JRST 0 G0002)
G0004
G0002
(SUB P (% 0 0 1 1))
(POPJ P)
NIL
```

****

LCOM4 produces the following code for the MACLISP LAP assembler.

```
(LAP DROP SUBR)
(PUSH P 1)                          ~save argument, U, on stack
(MOVE 1 0 P)                        ~get U off stack
(JUMPE 1 G0003)                     ~if U=NIL then exit
(HLRZ 1 @ 0 P)                      ~car U
(CALL 1 (QUOTE LIST))               ~<car U>
(PUSH P 1)                          ~save <car U>
(HRRZ @ 1 -1 P)                     ~cdr U (U now 1 below top of stack
(CALL 1 (QUOTE DROP))               ~drop cdr U
(MOVE 2 1)                          ~set up arguments for cons
(MOVE 1 0 P)
(SUB P (% 0 0 1 1))                 ~reset stack to arglist
(CALL 2 (QUOTE CONS))               ~cons[<car u>,cdr U]
G0003
(SUB P (% 0 0 1 1))                 ~reset stack to position at entry
(POPJ P)                            ~return
NIL
```

****

The Diffie compiler produces the following code for the LISP1.6 assembler.

```
(LAP DROP SUBR)
       (PUSH P 1)
       (JUMPE 1 TAG1)
       (HLRZ@ 1 0 P)
       (CALL 1 (E NCONS) S)
       (PUSH P 1)
       (HRRZ@ 1 -1 P)
       (CALL 1 (E DROP) S)
       (POP P 2)
       (CALL 2 (E XCONS) S)
 TAG1  (SUB P (C 0 0 1 1))
       (POPJ P)
       NIL
```

****

The MACLISP compiler, NCOMPLR, produces the following code:

(The pseudo-instruction ARGS tells the assembler the number of arguments expected by the function. The instruction JSP is a special fast function call done by jumping rather than going through the regular calling mechanism. T denotes an accumulator designated to hold temporary values.)

```
(LAP DROP SUBR)
(ARGS DROP (NIL . 1))
(PUSH P 1)
(JSP T PDLNMK)
(JUMPE 1 G0001)
(HLRZ 1 @ 0 P)
(JSP T %NCONS)
(PUSH P 1)
(HRRZ 1 @ -1 P)
(CALL 1 'DROP)
(POP P 2)
(JSP T %XCONS)
G0001
(SUB P (% 0 0 1 1))
(POPJ P)
NIL
```

3. LCOM0.

The following is an annotated listing of the MACLISP version of LCOM0 in external notation. A listing in internal notation can be found in Appendix A.

compl is the user-callable driver. It is a FEXPR. It takes as an argument a single file name. EXPRs on a file called FILNAM will be compiled into LAP and written on the file FILNAM.LAP. Other types of function definitions and non-definitions are simply copied to output. It is, alas, dependent on the I/O structure of the implementation and need not be studied carefully. All the other functions are substantially independent of implementation.

```
        FEXPER:
        compl file ←
          [uwrite[],                                    ~Open a file for output
           apply[EREAD, file],                          ~Open input file
           select-disk-input[
            read-until-eof[with z do                    ~Read each expression in file
              if az = DEFUN ∨ [az = DEFPROP ∧ adddz = EXPR] then
               [program [prog]
3.1)                prog
                    ← [if az = DEFUN then comp[adz, addz, adddz]
                       else comp[adz, adaddz, addaddz]]
                    unselect-tty
                     select-disk-output mapc[print, prog]   ~ Print code in file
                    print <adz, length prog>]
              else unselect-tty select-disk-output print z],
           apply[UFILE, <afile, LAP>],                  ~Close and name file
           ENDCOMP]]
```

comp compiles a single function definition, returning a list of the LAP code corresponding to the definition. *fn* is the atomic name of the function being compiled. *vars* is the formal parameter list for the function. *exp* is the function body.

```
        comp[fn, vars, exp] ←
          {length vars}[λn:
            <<LAP, fn, SUBR>>
3.2)        * mkpush[n, 1]
            * compexp[exp, -n, prup[vars, 1]]
            * <<SUB, P, <%, 0, 0, n, n>>>
            * ((POPJ P) NIL)]
```

prup returns an a-list formed by pairing successive elements of *vars* with consecutive integers beginning with *n*.

```
        prup[vars, n] ←
3.3)      if n vars then NIL else [avars . n] . prup[dvars, add1 n]
```

mkpush returns a list of *n* (PUSH P *i*) instructions, where *i* runs from *n* to *m*+*n*−1. It is used to push arguments onto the stack.

```
      mkpush[n, m] ←
3.4)    if n < m then NIL else <PUSH, P, m> . mkpush[n, add1 m]
```

compexp is the heart of LCOM0. It determines precisely what an expression is, and compiles appropriate code for it. It returns a list of that code. *exp* is the expression to be compiled. *m* is minus the number of entries on the stack. When added to a value retrieved from the A-LIST *vpr*, it can be used to locate a variable on the stack. *vpr* is an A-LIST, associating variable names with numbers which, when added to *m*, give stack offsets. Both *m* and *vpr* maintain these definitions throughout. *gensym*[] is a pseudo-function of no arguments. Every time it is called, it returns a different symbol. The presence of *gensym*[] means that LCOM0 and LCOM4 are not pure LISP so that the techniques of Chapter 3 cannot be used to prove that they meet their specifications. It is possible to eliminate the use of *gensym*, but as of the present writing, such version hasn't been written. In any case, other unsolved problems remain before one can give a first order logic proof of the correspondence between *eval* and the code produced by the compilers.

```
      compexp[exp, m, vpr] ←
        if natom exp then ((MOVEI 1 0))                          ~NIL
        else if exp = T then ((MOVEI 1 (QUOTE T)))               ~T
        else if numberp exp then <<MOVEI, 1, <QUOTE, exp>>>      ~number
        else if atom exp then <<MOVE, 1, m + dassoc[exp, vpr], P>>   ~variable
        else if aatom exp = AND ∨ aatom exp = OR ∨ aatom exp = NOT then   ~boolean expression
          {gensym[], gensym[]}[λl1, l2:
            comboolean[exp, m, l1, NIL, vpr]
            * <(MOVEI 1 (QUOTE T)),
               <JRST, 0, l2>,
               l1,
               (MOVEI 1 0),
               l2>]
        else if aatom exp = COND then comcond[dexp, m, gensym[], vpr]   ~COND
3.5)    else if aatom exp = QUOTE then <<MOVEI, 1, exp>>         ~QUOTE
        else if atom aexp then                                   ~function call
          {length dexp}[λn:
            complis[dexp, m, vpr]
            * loadac[1 - n, 1]
            * <<SUB, P, <%, 0, 0, n, n>>>
            * <<CALL, n, <QUOTE, aexp>>>]
        else if aatom aexp = LAMBDA then                         ~λ-expression
          {length dexp}[λn:
            complis[dexp, m, vpr]
            * compexp[addaexp, m - n, prup[adaexp, 1 - m] * vpr]
            * <<SUB, P, <%, 0, 0, n, n>>>]
        else NIL                                                 ~oops!
```

complis compiles code to evaluate each expression in a list of expressions and to push those values onto the stack. It returns a list of that code. It is used to compile code to evaluate arguments to called functions or λ-expressions. *u* is a list of expressions.

```
      complis[u, m, vpr] ←
        if natom u then NIL
```

```
3.6)      else compexp[a u, m, vpr]
              * ((PUSH P 1))
              * complis[d u, sub1 m, vpr]
```

loadac returns a list of (MOVE *i j* P) instructions, loading consecutive accumulators from the top of the stack. *k* indexes the accumulator loaded. *n* indexes the stack offset.

```
          loadac[n, k] ←
3.7)       if n > 0 then NIL
           else <MOVE, k, n, P> . loadac[add1 n, add1 k]
```

comcond compiles a COND. *u* is a list of clauses in the COND. *l* is a label to be emitted at the end of all code for the COND.

```
          comcond[u, m, l, vpr] ←
           if n u then <l>
           else {gensym[]}[λl1:
3.8)        comboo1[aa u, m, l1, NIL, vpr]
            * compexp[ada u, m, vpr]
            * <<JRST, 0, l>, l1>
            * comcond[d u, m, l, vpr]]
```

combool compiles code for a single propositional expression. That is, the code generated evaluates the expression and branches somewhere, according to whether its value is true or false. *p* is the propositional expression. *l* is a label which represents the branch point. *flg* is a flag. If *flg* is NIL, code is to fall thru on non-NIL result and branch to *l* on NIL result. If *flg* is non-NIL, code is to fall thru on NIL result and branch to *l* on non-NIL result.

```
          combool[p, m, l, flg, vpr] ←
           if at p then                                          ~simple variable
            [compexp[p, m, vpr]
             * <<if flg then JUMPN else JUMPE, 1, l>>]
           else if a p = AND then                                ~conjunction
            [if ¬flg then compandor[d p, m, l, NIL, vpr]
             else {gensym[]}[λl1:
               compandor[d p, m, l1, NIL, vpr]
3.9)           * <<JRST, 0, l>>
               * <l1>]]
           else if a p = OR then                                 ~disjunction
            [if flg then compandor[d p, m, l, T, vpr]
             else {gensym[]}[λl1:
               compandor[d p, m, l1, T, vpr] * <<JRST, 0, l>> * <l1>]]
           else if a p = NOT then combool[ad p, m, l, ¬flg, vpr] ~negation
           else compexp[p, m, vpr]                               ~other expression
                * <<if flg then JUMPN else JUMPE, 1, l>>
```

compandor compiles code for lists of predicates connected conjunctively or disjunctively. *u* is a list of predicates. *l* is a label. *flg* is a flag. If *flg* is NIL, we are to fall thru on non-NIL results and branch to *l* on NIL results (AND case). If *flg* is non-NIL, we are to fall thru on NIL results and branch to *l* on non-NIL results (OR case).

```
        compandor[u, m, l, flg, vpr] ←
          if n u then NIL
3.10)     else comboo[a u, m, l, flg, vpr]
               * compandor[d u, m, l, flg, vpr]
```

4. LCOM4.

The following is a listing in external notation of the MACLISP version of the compiler LCOM4. A listing in internal notation can be found in Appendix A. Comments are added where this compiler differs from LCOM0. The main difference has to do with the classification of expressions into five classes that can be compiled differently for greater efficiency. The five classes are defined as follows:

0. constants NIL, T, and numbers,
1. variables -- any atom not considered a constant,
2. quoted S-expressions -- non-atomic constants,
3. *car* - *cdr* chains applied to a variable,
4. all others, except
5. the last class 4 element in the argument list.

```
FEXPR:
  compl file ←
   [uwritc[],
    apply[EREAD, file],
    select-disk-input[
     read-until-eof[with z do
      if a z = DEFUN ∨ [a z = DEFPROP ∧ addd z = EXPR] then
       [program [prog]
4.1)                prog
                    ← [if a z = DEFUN then comp[ad z, add z, addd z]
                       else comp[ad z, ad add z, add add z]]
                    unselect-tty
                     select-disk-output mapc[print, prog]
                    print <ad z, length prog>]
      else unselect-tty select-disk-output print z],
     apply[UFILE, <afile, LAP>],
     ENDCOMP]]
```

Instead of *appending* the instructions into a list like *compexp* of LCOM0 does, LCOM4 *conses* them into a tree form and flattens the final version into a list. In order that the items of the tree form all have the structure of a list of atoms, labels and the end mark NIL are put in a list flagged by LABEL and extracted by *flat*. The hope was to make the compiler faster by omitting all the calls to *append* but it does not seem to have made a lot of difference.

```
  comp[fn, vars, exp] ←
   {prup[vars, 1], length vars}[λvpr, n:
    flat[<<<LAP, fn, SUBR>>,
         mkpush[n, 1],
4.2)     compexp[exp, -n, vpr],
         substack n,
         ((POPJ P) (LABEL NIL))>,
        NIL]]
```

```
        flat[u, s] ←
          if nu then s
          else if nau then flat[du, s]
4.3)      else if au = LABEL then adu . s
          else if atau then u . s
          else flat[au, flat[du, s]]
```

substack creates code to move down the stack, but creates no code for the case $n = 0$ which is a no-op.

```
        substack n ←
4.4)      if n = 0 then NIL else <<SUB, P, <%, 0, 0, n, n>>>

        prup[vars, n] ←
4.5)      if nvars then NIL else [avars . n] . prup[dvars, add1 n]

        mkpush[n, m] ←
4.6)      if n < m then NIL else <PUSH, P, m> . mkpush[n, add1 m]

        compexp[exp, m, vpr] ←
          if nexp then ((MOVEI 1 0))
          else if exp = T ∨ numberp exp then
            <<MOVEI, 1, <QUOTE, exp>>>
          else if atexp then <<MOVE, 1, m + dassoc[exp, vpr], P>>
          else if aexp = CAR then                          ~compute car directly
           [if atadexp then
              <<HLRZ, 1, @, m + dassoc[adexp, vpr], P>>
            else <compexp[adexp, m, vpr], ((HLRZ 1 @ 1))>]
          else if aexp = CDR then                          ~compute cdr directly
           [if atadexp then
              <<HRRZ, 1, @, m + dassoc[adexp, vpr], P>>
            else <compexp[adexp, m, vpr], ((HRRZ 1 @ 1))>]
          else if aexp = AND ∨ aexp = OR ∨ aexp = NOT ∨ aexp = EQ
           then {gensym[], gensym[]}[λl1, l2:
            <combool[exp, m, l1, NIL, vpr],
4.7)         <(MOVEI 1 (QUOTE T)),
              <JRST, 0, l2>,
              <LABEL, l1>,
              (MOVEI 1 0),
              <LABEL, l2>>>]
          else if aexp = COND then comcond[dexp, m, gensym[], vpr]
          else if aexp = QUOTE then <<MOVEI, 1, exp>>
          else if ataexp then                              ~uses improved
           <complisa[dexp, m, vpr],                        ~argument evaluation
            <<CALL, length dexp, <QUOTE, aexp>>>>
          else if aaexp = LAMBDA then
           {length dexp}[λn:
            <stackup[dexp, m, vpr],
             compexp[addaexp, m − n, prup[adaexp, 1 − m] * vpr],
             substack n>]
          else NIL
```

```
        stackup[u, m, vpr] ←                                  ~ = complis of LCOM0.
          if nu then NIL
4.8)      else <compexp[au, m, vpr],
                ((PUSH P 1)),
                stackup[du, sub1 m, vpr]>
```

ccchain test whether an expression is a *car* - *cdr* chain applied to a variable.

```
        ccchain exp ←
4.9)      [aexp = CAR ∨ aexp = CDR] ∧ [atadexp ∨ ccchain adexp]
```

compc generates code (in reverse order) to load the value of a *car* - *cdr* chain into the accumulator designated by *n*2.

```
        compc[exp, n2, m, vpr] ←
          if atexp then error COMPC
          else if aexp = CAR then
           [if atadexp then
4.10)        <<HLRZ, n2, @, m + dassoc[adexp, vpr], P>>
             else <HLRZ, n2, @, n2> . compc[adexp, n2, m, vpr]]
          else if atadexp then
            <<HRRZ, @, n2, m + dassoc[adexp, vpr], P>>
          else <HRRZ, n2, @, n2> . compc[adexp, n2, m, vpr]
```

comcond is essentially the same as in LCOM0. It optimizes by considering the special cases where the next clause in the list of arguments to COND is of the form ((NULL *e*) NIL) or (T *e*) for some expression *e*.

```
        comcond[u, m, l, vpr] ←
          if nu then <<LABEL, l>>
          else if ¬ataau ∧ aaau = NULL ∧ nadau then
            <compexp[adaau, m, vpr],
             <<JUMPE, 1, l>>,
             comcond[du, m, l, vpr]>
4.11)     else if aau = T then <compexp[adau, m, vpr], <<LABEL, l>>>
          else {gensym[]}[λl1:
            <combool[aau, m, l1, NIL, vpr],
             compexp[adau, m, vpr],
             <<JRST, 0, l>, <LABEL, l1>>,
             comcond[du, m, l, vpr]>]
```

complisa classifies the expressions on the argument list *u*, calls *complis* to generate code for those arguments which must be computed and stored on the stack, *loadac* to generate code for loading the accumulators and *substack* to generate code to restore the stack.

```
        complisa[u, m, vpr] ←
          {classify u}[λz:
4.12)       <complis[z, m, 1, vpr],
             loadac[z, 1 – ccount z, 1, m – ccount z, vpr],
             substack ccount z>]
```

ccount of a list of classified expressions is the number of expressions on the list of class 4. That is the number of values that will have to be stacked when evaluating z as an argument list.

```
        ccount z ←
          if nz then 0
4.13)     else if aaz = 4 then add1 ccount dz
          else ccount dz
```

loadac compiles code to load accumulators n2 and up with the values of the expressions on the list z of classified expressions. m2 is the stack offset for the value of the next class 4 expression on the list.

```
        loadac[z, m2, n2, m, vpr] ←
          if nz then NIL
          else if aaz = 1 then
            <MOVE, n2, m + dassoc[daz, vpr], P>
            . loadac[dz, m2, add1 n2, m, vpr]
          else if aaz = 0 then
            <MOVEI, n2, <QUOTE, daz>>
            . loadac[dz, m2, add1 n2, m, vpr]
4.14)     else if aaz = 2 then
            <MOVEI, n2, daz> . loadac[dz, m2, add1 n2, m, vpr]
          else if aaz = 3 then
            <reverse compc[daz, n2, m, vpr],
             loadac[dz, m2, add1 n2, m, vpr]>
          else if aaz = 5 then loadac[dz, 1, add1 n2, m, vpr]
          else <MOVE, n2, m2, P>
               . loadac[dz, add1 m2, add1 n2, m, vpr]
```

complis generates code to stack the values of the class 4 expressions on the list of classified expressions z. k is the accumulator where the next value should go. The last class 4 expression is marked class 5. When this is evaluated it is loaded directly rather than being stacked. If a class 5 expression is encountered, *complis* can quit as it has no more work.

```
        complis[z, m, k, vpr] ←
          if nz then NIL
          else if aaz = 4 then
            <compexp[daz, m, vpr],
             ((PUSH P 1)),
4.15)        complis[dz, sub1 m, add1 k, vpr]>
          else if aaz = 5 then
            <compexp[daz, m, vpr],
             if k = 1 then NIL else <<MOVE, k, 1>>>
          else complis[dz, m, add1 k, vpr]
```

classify takes a list of expressions and returns a corresponding list of classified expressions of the form [n . e] where n is the class of e according to the rules given at the beginning of the listing. The last class 4 expression on the list is numbered 5.

```
4.16)   classify u ← class2[class1[u, NIL], NIL, T]
```

```
        class1[u, v] ←
          if nu then v
          else if atau then
            [if au = NIL ∨ au = T ∨ numberp au then
4.17)         class1[du, [0 . au] . v]
              else class1[du, [1 . au] . v]]
          else if aau = QUOTE then class1[du, [2 . au] . v]
          else if ccchain au then class1[du, [3 . au] . v]
          else class1[du, [4 . au] . v]

        class2[u, v, flg] ←
          if nu then v
4.18)     else if flg ∧ aau = 4 then class2[du, [5 . dau] . v, NIL]
          else class2[du, au . v, flg]

4.19)   mkjrst l ← <<JRST, 0, l>>
```

combool is essentially the same as in LCOM0 except it treats additional special cases where p is of the form T, (EQ $e1$ $e2$), or (NULL e). It also handles the jumping somewhat differently using two versions of *compandor*.

```
        combool[p, m, l, flg, vpr] ←
          if p = T then [if flg then mkjrst l else NIL]
          else if atp then
            <compexp[p, m, vpr],
             <<if flg then JUMPN else JUMPE, 1, l>>>
          else if ap = EQ then
            <complisa[dp, m, vpr],
             if flg then ((CAMN 1 2)) else ((CAME 1 2)),
             mkjrst l>
          else if ap = AND then
            [if ¬flg then compandor[dp, m, l, NIL, vpr]
4.20)         else {gensym[]}{λl1:
                <compandor1[dp, m, l1, l, NIL, vpr], <<LABEL, l1>>>]]
          else if ap = OR then
            [if flg then compandor[dp, m, l, T, vpr]
             else {gensym[]}{λl1:
               <compandor1[dp, m, l1, l, T, vpr], <<LABEL, l1>>>]]
          else if ap = NOT then combool[adp, m, l, ¬flg, vpr]
          else if ap = NULL then
            <compexp[adp, m, vpr],
             <<if flg then JUMPE else JUMPN, 1, l>>>
          else <compexp[p, m, vpr],
                <<if flg then JUMPN else JUMPE, 1, l>>>

        compandor[u, m, l, flg, vpr] ←
          if nu then NIL
4.21)     else <combool[au, m, l, flg, vpr],
                compandor[du, m, l, flg, vpr]>

        compandor1[u, m, l, l2, flg, vpr] ←
```

```
        if n u then mkjrst l2
4.22)   else if n d u then comboolⁱ[a u, m, l2, ¬flg, vpr]
        else <comboolⁱ[a u, m, l, flg, vpr],
              compandorⁱ[d u, m, l, l2, flg, vpr]>
```

Chapter IX

ABSTRACT SYNTAX

1. What is Abstract Syntax?

When we compute with symbolic expressions, whether they be computer programs, algebraic expressions, or formulas of logic, some aspects of their representation are irrelevant for many purposes. For example, the sum of a and b may be written $a + b$ as in the most common algebraic notation, (PLUS a b) as in LISP, $+ab$ in "Polish notation" or $ab+$ in the reverse Polish notation made popular by the Hewlett-Packard calculators. Mathematical logicians use representations of expressions by "Gödel numbers" in order to represent symbolic computations as numerical computations. One such Gödel numbering represents the sum of a and b as 2^a3^b. Of course, "the sum of a and b" is just another such notation which we have been using to talk about the others.

For many aspects of evaluating sums or compiling them, the details of the representation are irrelevant. Our programs must determine whether an expression is a sum, if so what the summands are, and it may need to make sums from constituent expressions. The predicate that tells whether an expression is a sum and the functions that extract the summands comprise the *analytic syntax* of sums. The function that makes a sum expression out of the summand expression gives the *synthetic syntax* of sums. The analytic and synthetic syntaxes are, of course, related to each other.

Abstract syntax, first described in [McCarthy 1962b], is a way of presenting only the information about a class of expressions that is relevant to computing with them, omitting irrelevant detail. Later we shall prove a small compiler correct without ever deciding on a definite notation. If we decide on a notation for some expressions and write programs to test whether an expression is of a given type, to extract subexpressions and to construct new expressions from old ones then we have give a *concrete syntax*. Any concrete syntax corresponding to our abstract syntax will do.

Example 1. Simplification of arithmetic expressions.

Consider arithmetic expressions built up from constants and variables by forming sums and products. The abstract syntax is given in table 5.

predicate	analytic operation	synthetic operation
isconst(*e*)		
isvar(*e*)		
issum(*e*)	*s1*(*e*), *s2*(*e*)	*mksum*(*e1*, *e2*)
isprod(*e*)	*p1*(*e*), *p2*(*e*)	*mkprod*(*e1*, *e2*)

Table 5. Abstract Syntax of Arithmetic Expressions.

The predicate column asserts that the expressions comprise constants, variables and binary sums and procucts, and that the predicates *isconst*, *isvar*, *issum*, and *isprod* enable one to classify each expression. The analytic operations column tells us that variables and constants are "atomic" expressions, while sums and products have subexpressions obtained by applying $s1$ and $s2$ in the case of sums, or $p1$ and $p2$ in the case of products.

The predicates and analytic operations comprise the *abstract analytic syntax* of these expressions, e.g. they allow us to analyse the expressions. The analytic syntax provides sufficient information to allow us to write an interpreter (evaluator) thus defining the semantics of our expression language. Thus we have

```
                value(e, ξ) ←
                    if isconst(e) then val(e)
1.1)                else if isvar(e) then c(e, ξ)
                    else if issum(e) then value(s1(e), ξ) + value(s2(e), ξ)
                    else if isprod(e) then value(p1(e), ξ) • value(p2(e), ξ)
```

where *val*(*e*) gives the numerical value of an expression *e* representing a constant, *c*(*e*, ξ) gives the value of the variable *e* in the state vector ξ and "+" and "•" are some binary operations. (It is natural to regard "+" and "•" as an operations that resembles addition and multiplication of real numbers, but our we shall need only a few properties of these operations.)

The synthetic operations column of table 5 says that sums can be constructed from a pair of expressions using the operation *mksum* and similarly products can be constructed from a pair of expression using *mkprod*. These operations form the *abstract synthetic syntax* of expressions, e.g. they allow us to construct new expressions from old ones.

Using the abstract syntax functions and predicates we can write a function that simplifies an expression by

1.1. removing 0's from sums,
1.2. removing 1's from products,
1.3. replacing a product containing 0 as a factor by 0.

We will assume the constants 0 and 1 are represented by "0" and "1", for simplicity. Then *simplify* is given by

```
                simplify(e) ←
                    if isvar(e) ∨ isconst(e) then e
                    else if issum(e) then
                        {simplify(s1(e)), simplify(s2(e))}
                        [λe1 e2. if e1=0 then e2
                                 else if e2=0 then e1
1.2)                             else mksum(e1, e2)]
                    else if isprod(e) then
                        {simplify(p1(e)), simplify(p2(e))}
                        [λe1 e2. if e1=0 ∨ e2=0 then 0
                                 else if e1=1 then e2
                                 else if e2=1 then e1
                                 else mkprod(e1, e2)]
```

Suppose we would like to prove something about our functions on expressions. For example we would like to know that the simplifier will indeed terminate (e.g. there are no infinite expressions). Then we need and a way of expressing the fact that the only expressions allowed are those that can be constructed from variables and constants by a finite number of applications of the sum and product constructions. The following schema is an induction principle for expressions essentially expressing this fact.

Suppose Φ is a predicate applicable to expressions, and suppose that for all expressions e, we have

1.3)
$$\begin{aligned} &isconst(e) \supset \Phi(e) \text{ and} \\ &isvar(e) \supset \Phi(e) \text{ and} \\ &issum(e) \wedge \Phi(s1(e)) \wedge \Phi(s2(e)) \supset \Phi(e) \text{ and} \\ &isprod(e) \wedge \Phi(p1(e)) \wedge \Phi(p2(e)) \supset \Phi(e). \end{aligned}$$

Then we may conclude that $\Phi(e)$ is true for all expressions e.

Using this principle we can show that the simplifier terminates by the techniques of Chapter III.

Now suppose we would like to show that the simplifier preserves the value of expressions. Then we will need some facts about the relations satisfied by the analytic and synthetic components of the syntax. Thus for expressions e, $e1$, $e2$ we have

1.4)
$$\begin{aligned} &issum(mksum(e1, e2)) \\ &s1(mksum(e1, e2))=e1 \\ &s2(mksum(e1, e2))=e2 \\ &issum(e) \supset e=mksum(s1(e), s2(e)) \end{aligned}$$

1.5)
$$\begin{aligned} &isprod(mkprod(e1, e2)) \\ &p1(mkprod(e1, e2))=e1 \\ &p2(mkprod(e1, e2))=e2 \\ &isprod(e) \supset e=mkprod(p1(e), p2(e)) \end{aligned}$$

Using these relations, the principle of induction, and some simple properties of the + and • operations involving 0 and 1 we can show that

1.6) $$value(e, \xi)=value(simplify(e), \xi)$$

far all expressions e and state vectors ξ.

Example 2. Abstract and Concrete Syntax of LISP terms.

Now we will give the abstract and concrete analytic syntax of pure (functional) LISP terms. We will see in the case of LISP the concrete syntax is very close to the abstract syntax. This is one reason that it is so easy to write LISP programs to manipulate LISP programs. We will also introduce in this example additional techniques which are useful for defining abstract syntax.

A pure LISP term is essentially any term that can occur at the right hand side of a defining equation as described in Chapter I (excluding labels). Thus LISP terms are either variables,

constants, conditional terms, list terms, or application terms, denoted by *lvar*, *lconst*, *lcond*, *llist* and *lappl* respectively.

We will use some further classes of expressions which are not LISP terms but are useful in analyzing the composite terms. These are the classes of function expressions: basic functions (*lbfun*), defined functions (*ldfun*), and lambda expressions (*llamb*) that occur in application terms.

Finally some of the components of the expressions will be lists of expressions. We will use *car*, *cdr* and *null* to analyse the lists of expressions. It should be emphasized that this use is in an abstract sense and does not depend on any properties of a particular representation of lists. There will be three special kinds of lists: the list of pairs of terms in a conditional term (*lpairlist*), the list of arguments in an application term (*ltermlist*), and the list of variables in a lambda expression (*lvarlist*).

The abstract analytic syntax of LISP terms is given in table 6.

predicate	associated functions
lconst(e)	
lvar(e)	
lbfun(e)	
ldfun(e)	
null(e)	
lcond(e)	*pairl(e)*
llist(e)	*terml(e)*
lappl(e)	*funx(e)*, *argl(e)*
llamb(e)	*varl(e)*, *body(e)*
lpairlist(e)	*ifpart(e)*, *thenpart(e)*, *elsepart(e)*
ltermlist	*car*, *cdr*
lvarlist	*car*, *cdr*

Table 6. LISP Absrtact Syntax.

In the usual S-expression representation of LISP terms (internal form) we would have for the basic predicates the following definitions

$$lconst\ x \leftarrow [\mathbf{at}\ x \wedge [numberp\ x \vee x{=}\mathbf{T} \vee x{=}\mathbf{NIL}]] \vee \mathbf{a}\ x{=}\mathbf{QUOTE}$$
$$lvar\ x \leftarrow \mathbf{at}\ x \wedge \neg[numberp\ x \vee x{=}\mathbf{T} \vee x{=}\mathbf{NIL}]$$
$$lbfun(e) \leftarrow [e{=}\mathbf{CAR} \vee e{=}\mathbf{CDR} \vee e{=}\mathbf{CONS} \vee e{=}\mathbf{ATOM} \vee e{=}\mathbf{EQ}]$$
$$ldfun(e) \leftarrow \mathbf{at}\ e \wedge \neg[numberp(e) \vee e{=}\mathbf{T} \vee e{=}\mathbf{NIL} \vee lbfun(e)]$$
$$null(e) \leftarrow \mathbf{n}\ e$$

The concrete syntax for conditional terms and the related auxiliary expressions is given by

$$lcond(e) \leftarrow \mathbf{a}\ e{=}\mathbf{COND} \wedge lpairlist(pairl(e))$$
$$pairl(e) \leftarrow \mathbf{d}\ e$$
$$lpairlist(e) \leftarrow \mathbf{n}\ e \vee [lterm(ifpart(e)) \wedge lterm(thenpart(e) \wedge lpairlist(elsepart(e))]$$

$ifpart(e) \leftarrow \mathbf{aa}e$
$thenpart(e) \leftarrow \mathbf{ada}e$
$elsepart(e) \leftarrow \mathbf{d}e$

For list terms we have

$llist(e) \leftarrow \mathbf{a}e = \text{LIST} \wedge lterm list(terml(e))$
$terml(e) \leftarrow \mathbf{d}e$
$lterm list(e) \leftarrow \mathbf{n}e \vee [lterm(car(e)) \wedge lterm list(cdr(e))]$

The concrete syntax for application terms and lambda expression is the following

$lappl(e) \leftarrow [lbfun(funx(e)) \vee ldfun(funx(e)) \vee llamb(funx(e))] \wedge lterm list(argl(e))$
$funx(e) \leftarrow \mathbf{a}e$
$argl(e) \leftarrow \mathbf{d}e$
$llamb(e) \leftarrow \mathbf{a}e = \text{LAMBDA} \wedge lvarlist(varl(e)) \wedge lterm(body(e))$
$varl(e) \leftarrow \mathbf{ad}e$
$body(e) \leftarrow \mathbf{add}e$
$lvarlist(e) \leftarrow \mathbf{n}e \vee [lvar(car(e)) \wedge lvarlist(cdr(e))]$

Exercises

1. Prove the statements made about the expression simplifier in example 1. That is, prove it terminates by defining a predicate *isexp* such that

$$isexp(e) \equiv isvar(e) \vee isconst(e) \vee issum(e) \vee isprod(e)$$

and showing that $isexp(simplify(e))$ for all expressions e. Then show that the statement (1.6) hold for all expressions.

2. Work out the synthetic syntax for LISP terms and state relations for each construct analagous to those given for the expressions in example 1. State an induction principle for LISP terms.

2. Correctness of a Compiler for Arithmetic Expressions.

In the following sections we prove the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language. The presentation is adapted from that of McCarthy and Painter[1967].

We begin with a brief outline of the methods used to formalize the problem. The analytic abstract syntax is given for both the arithmetic expressions (source language) and machine language programs (object language). An interpreter defined in terms of the analytic syntax is given for each language. The two interpreters are written in the same language, and thus define the semantics of the two languages in common terms. The interpreters each make use of a *state vector* to represent the values of variables (in the expression language case) or to store contents of

registers (in the machine language case). The synthetic syntax is also given for the machine language and the compiler is defined in terms of the analytic syntax of the source language and the synthetic syntax of the object language. Thus the compiler breaks down terms in the source language and rebuilds them in the object language. To show that the compiler is correct we must show that for an expression e the interpretation of e by the expression interpreter gives the same answer as the execution of the compiled code for e by the machine language interpreter, assuming that they start out with the same values for all of the variables in e.

The expressions of our source language are a subclass of those described in §1. The computer language into which these expressions are compiled is for a single address computer with an accumulator, called *ac*, and four instructions: *li* (load immediate), *load*, *sto* (store) and *add*. (Our simple language does not require the use of jump instructions.) The compiler produces code that computes the value of the expression being compiled and leaves this value in the accumulator. The above expression is compiled into code which in assembly language might look as follows:

load	x
sto	t
li	3
add	t
sto	t
load	x
sto	$t + 1$
load	y
sto	$t + 2$
li	2
add	$t + 2$
add	$t + 1$
add	t

Again because we are using abstract syntax there is no commitment to a precise form for the object code.

3. The source language.

As mentioned above the source language is a subset of the expression language described in §1. In particular, we will consider expressions that are variables, constants or sums. Products are omitted as they introduce nothing new into the problem and serve only to make everything longer. Thus we will use the relevant parts of table 5 for the abstract analytic syntax of source expressions. The semantics is given by the interpreter *value* (1.1) by simply ignoring the *isprod* clause.

For proving the correctness of compilers, we don't need a synthetic syntax for source language expressions since the interpreter and compiler use only the analytic syntax. The induction principle for this simpler class of expressions is obtained by deleting the *isprod* clause from the induction hypothesis (1.3). Thus

If Φ is a predicate applicable to expressions, and if for all expressions e, we have

3.1)
$$\begin{aligned} &isconst(e) \supset \Phi(e) \text{ and} \\ &isvar(e) \supset \Phi(e) \text{ and} \\ &issum(e) \wedge \Phi(s1(e)) \wedge \Phi(s2(e)) \supset \Phi(e). \end{aligned}$$

Then we may conclude that $\Phi(e)$ is true for all expressions e.

4. The object language.

We must give both the analytic and synthetic syntaxes for the object language because the interpreter defining its semantics uses the analytic syntax and the compiler uses the synthetic syntax. The analytic and synthetic syntaxes for instructions are given in the table 7.

operation	predicate	analytic operation	synthetic operation
li α	isli(s)	arg(s)	mkli(α)
load x	isload(s)	adr(s)	mkload(x)
sto x	issto(s)	adr(s)	mksto(x)
add x	isadd(s)	adr(s)	mkadd(x)

Table 7. Abstract Syntax of Machine Instructions.

A program is a list of instructions and $null(p)$ asserts that p is the null list. If the program p is not null then $first(p)$ gives the first instruction and $rest(p)$ gives the list of remaining instructions. We shall use the operation $p1*p2$ to denote the program obtained by appending $p2$ onto the end of $p1$. Since we have only one level of list we can identify a single instruction with a program that has just one instruction.

The synthetic and analytic syntaxes of instructions are related by the following.

4.1)
$$\begin{aligned} &isli(mkli(\alpha)) \\ &arg(mkli(\alpha)) = \alpha \\ &isli(s) \supset s = mkli(arg(s)) \\ &null(rest(mkli(\alpha))) \\ &isli(s) \supset first(s) = s \end{aligned}$$

4.2)
$$\begin{aligned} &isload(mkload(x)) \\ &x = adr(mkload(x)) \\ &isload(x) \supset x = mkload(adr(x)) \\ &null(rest(mkload(x))) \\ &isload(s) \supset first(s) = s \end{aligned}$$

$$
\begin{array}{ll}
 & issto(mksto(x)) \\
 & x = adr(mksto(x)) \\
4.3) & issto(s) \supset s = mksto(adr(s)) \\
 & null(rest(mksto(x))) \\
 & issto(s) \supset first(s) = s
\end{array}
$$

$$
\begin{array}{ll}
 & isadd(mkadd(x)) \\
 & x = adr(mkadd(x)) \\
4.4) & isadd(s) \supset s = mkadd(adr(s)) \\
 & null(rest(mkadd(x))) \\
 & isadd(s) \supset first(s) = s
\end{array}
$$

$$
\begin{array}{ll}
 & \neg null(p) \supset p = first(p) * rest(p), \\
4.5) & \neg null(p1) \wedge null(rest(p1)) \supset p1 = first(p1 * p2) \\
 & null(p1 * p2) \equiv null(p1) \wedge null(p2).
\end{array}
$$

The * operation is associative. (The somewhat awkward form of these relations comes from having a general concatenation operation rather than just an operation that prefixes a single instruction onto a program.)

A state vector for a machine gives, for each register in the machine, its contents. We include the accumulator denoted by *ac* as a register. Associatedwith state vectors are the functions *a* (for assign) and *c* (for contents). Thus

1. $c(x, \eta)$ denotes the value of the contents of register x in machine state η.

2. $a(x, \alpha, \eta)$ denotes the state vector that is obtained from the state vector η by changing the contents of register x to α leaving the other registers unaffected.

The relations satisfied by these functions are:

$$
\begin{array}{ll}
 & c(x, a(y, \alpha, \eta)) = \text{if } x = y \text{ then } \alpha \text{ else } c(x, \eta), \\
4.6) & a(x, \alpha, a(y, \alpha, \eta)) = \text{if } x = y \text{ then } a(x, \alpha, \eta) \text{ else } a(y, \alpha, a(x, \alpha, \eta)), \\
 & a(x, c(x, \eta), \eta) = \eta.
\end{array}
$$

[We note that the first argument to the *c* and *a* functions is a variable in the case of the expression state vector while in the case of the machine state vector it is a register. This causes no difficulty, the functions satisfy the same relations in both cases and can just be thought as polymorphic.]

Now we can define the semantics of the object language by

$$
\begin{array}{ll}
 & step(x, \eta) \leftarrow \\
 & \quad \text{if } isli(s) \text{ then } a(ac, arg(s), \eta) \\
4.7) & \quad \text{else if } isload(s) \text{ then } a(ac, c(adr(s), \eta), \eta) \\
 & \quad \text{else if } issto(s) \text{ then } a(adr(s), c(ac, \eta), \eta) \\
 & \quad \text{else if } isadd(s) \text{ then } a(ac, c(adr(s), \eta) + c(ac, \eta), \eta)
\end{array}
$$

which gives the state vector that results from executing an instruction and

4.8) $$outcome(p, \eta) \leftarrow \text{if } null(p) \text{ then } \eta \text{ else } outcome(rest(p), step(first(p), \eta))$$

which gives the state vector that results from executing the program p with state vector η.

The following lemma is left as an exercise for the reader.

4.9) $$outcome(p1 * p2, \eta) = outcome(p2, outcome(p1, \eta))$$

5. The compiler.

In order to compile an expression we need to know where the values of the variables occuring in the expression will be stored. We shall assume that the function *loc* maps each variable to the register where it will initially be stored in the machine state vector. If the comparison of the interpretation of an expression and the execution of its compiled code is to be meaningful, the initial state vectors must agree on the variables occuring in the expression. Thus we will assume that the relation

5.1) $$c(loc(v), \eta) = c(v, \xi)$$

between the state vector η before the compiled program starts to act and the state vector ξ of the source program holds.

Now we can write the compiler. It is

5.2)
$$\begin{aligned} &compile(e, t) \leftarrow \\ &\quad \text{if } isconst(e) \text{ then } mkli(val(e)) \\ &\quad \text{else if } isvar(e) \text{ then } mkload(loc(e)) \\ &\quad \text{else if } issum(e) \text{ then } compile(s1(e), t) * mksto(t) * compile(s2(e), t + 1) * mkadd(t) \end{aligned}$$

Here t is the number of a register such that all variables are stored in registers numbered less than t, so that registers t and above are available for temporary storage.

Before we can state our definition of correctness of the compiler, we need a notion of partial equality for state vectors

$$\zeta_1 =_A \zeta_2$$

where ζ_1 and ζ_2 are state vectors and A is a set of variables means that corresponding components of ζ_1 and ζ_2 are equal except possibly for values, of variables in A. Symbolically, $x \notin A \supset c(x, \zeta_1) = c(x, \zeta_2)$. Partial equality satisfies the following relations:

5.3) $\zeta_1 = \zeta_2$ is equivalent to $\zeta_1 =_{\{\}} \zeta_2$ where $\{\}$ denotes the empty set,

5.4) if $A \subset B$ and $\zeta_1 =_A \zeta_2$ then $\zeta_1 =_B \zeta_2$,

5.5) if $\zeta_1 =_A \zeta_2$, then $a(x, \alpha, \zeta_1) =_{A-\{x\}} a(x, \alpha, \zeta_2)$,

5.6) if $x \in A$ then $a(x, \alpha, \zeta) =_A \zeta$,

5.7) if $\zeta_1 =_A \zeta_2$ and $\zeta_2 =_B \zeta_3$, then $\zeta_1 =_{A \cup B} \zeta_3$.

In our case we need a specialization of this notation and will use

$\zeta_1 =_t \zeta_2$ to denote $\zeta_1 =_{\{x \mid x \geq t\}} \zeta_2$
$\zeta_1 =_\infty \zeta_2$ to denote $\zeta_1 =_{\{\infty\}} \zeta_2$ and
$\zeta_1 =_{t,\infty} \zeta_2$ to denote a$\zeta_1 =_{\{x \mid x = \infty \vee x \geq t\}} \zeta_2$.

The correctness of the compiler is stated in

THEOREM 1. If η and ξ are machine and source language state vectors respectively such that (5.1) holds then

$$outcome(compile(e, t), \eta) =_t a(ac, value(e, \xi), \eta)$$

It states that the result of running the compiled program is to put the value of the expression compiled into the accumulator. No registers except the accumulator and those with addresses $\geq t$ are affected.

6. Proof of Correctness Theorem.

The proof is accomplished by an induction on the expression e being compiled using the principle given in §3. We prove it first for constants, then for variables, and then for sums on the induction hypothesis that it is true for the summands. Thus there are three cases.

Statement		Justification
I. $isconst(e)$.		
$outcome(compile(e, t), \eta)$	$= outcome(mkli(val(e)), \eta)$	5.2
	$= step(mkli(val(e)), \eta)$	4.8, 4.1
	$= a(ac, arg(mkli(val(e))), \eta)$	4.1, 4.7
	$= a(ac, val(e), \eta)$	4.1
	$= a(ac, value(e, \xi), \eta)$	1.1
	$=_t a(ac, value(e, \xi), \eta)$.	5.3, 5.4
II. $isvar(e)$.		

$$
\begin{aligned}
outcome(compile(e, t), \eta) &= outcome(mkload(loc(e)), \eta) && 5.2\\
&= a(ac, c(adr(mkload(loc(e))), \eta, \eta) && 4.8,\ 4.2,\ 4.7\\
&= a(ac, c(loc(e), \eta), \eta) && 4.2\\
&= a(ac, c(e, \xi), \eta) && 5.1\\
&= a(ac, value(e, \xi), \eta) && 1.1\\
&=_t a(ac, value(e, \xi), \eta). && 5.3,\ 5.4
\end{aligned}
$$

III. $issum(e)$.

$$
\begin{aligned}
&outcome(compile(e, t), \eta) = outcome(compile(s1(e), t) * mksto(t) && 5.2\\
&\qquad * compile(s2(e), t + 1) * mkadd(t), \eta)\\
6.1)\quad &= outcome(mkadd(t), && 4.9\\
&\qquad outcome(compile(s2(e), t + 1),\\
&\qquad\quad outcome(mksto(t),\\
&\qquad\qquad outcome(compile(s1(e), t), \eta)))
\end{aligned}
$$

using the relation between concatenating programs and composing the functions they represent. Now we introduce some notation. Let

$$
\begin{aligned}
v &= value(e, \xi),\\
6.2)\quad v_1 &= value(s1(e), \xi),\\
v_2 &= value(s2(e), \xi),\\
&\vdots
\end{aligned}
$$

so that $v = v_1 + v_2$. Further let

$$
\begin{aligned}
\zeta_1 &= outcome(compile(s1(e), t), \eta),\\
6.3)\quad \zeta_2 &= outcome(mksto(t), \zeta_1),\\
\zeta_3 &= outcome(compile(s2(e), t + 1), \zeta_2),\\
\zeta_4 &= outcome(mkadd(t), \zeta_3)
\end{aligned}
$$

so that by (6.1) $\zeta_4 = outcome(compile(e, t), \eta)$, and the statement to be proved becomes

$$\zeta_4 =_t a(ac, v, \eta).$$

In order to apply the induction hypothesis to show that $s2(e)$ compiles correctly we need to know that for each variable v the following equation holds:

$$6.4)\qquad c(loc(v), \zeta_2) = c(v, \xi)$$

This is proved as follows:

$$
\begin{aligned}
c(loc(v), \zeta_2) &= c(loc(v), a(t, v_1, \eta)) && \text{since } loc(v) < t\\
&= c(loc(v), \eta) && \text{for the same reason}\\
&= c(v, \xi) && 5.1
\end{aligned}
$$

thus the induction hypothesis together with (5.1) and (6.4) and the definitions (6.3) give

6.5) $\zeta_1 = {}_t\, a(ac, v_1, \eta)$

6.6) $\zeta_3 = {}_{t+1}\, a(ac, v_2, \zeta_2)$

Now we resume the main proof

6.7)	$c(ac, \zeta_1) = v_1$	4.6, 6.5
	$\zeta_2 = outcome(mksto(t), \zeta_1)$	6.3
	$= a(t, c(ac, \zeta_1), \zeta_1)$	4.8, 4.3, 4.7
	$= a(t, v_1, \zeta_1)$	6.7
	$= {}_{t+1}\, a(t, v_1, a(ac, v_1, \eta))$	5.5, 6.5
6.8)	$= {}_{t+1,ac}\, a(t, v_1, \eta)$	5.6, 4.6
6.9)	$\zeta_3 = {}_{t+1}\, a(ac, v_2, a(t, v_1, \eta))$	6.6, 4.6, 5.5, 6.8
6.10)	$c(ac, \zeta_3) = v_2$ and $c(t, \zeta_3) = v_1$	4.6, 6.9
	$\zeta_4 = outcome(mkadd(t), \zeta_3)$	6.3
	$= a(ac, c(t, \zeta_3) + c(ac, \zeta_3), \zeta_3)$	4.8, 4.4, 4.7
	$= a(ac, v, \zeta_3)$	6.2, 6.10, 1.1
	$= {}_{t+1}\, a(ac, v, a(ac, v_2, a(t, v_1, \eta)))$	5.5, 6.9
	$= {}_{t+1}\, a(ac, v, a(t, v_1, \eta))$	4.6
	$= {}_t\, a(ac, v, \eta).$	4.6, 5.6, 5.7

This concludes the proof.

7. Remarks.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in McCarthy [1962a,1962b,1963,1964] is to make it possible to use a computer to check proofs that compilers are correct. The concepts of abstract syntax, state vector, the use of an interpreter for defining the semantics of a programming language, and the definition of correctness of a compiler were introduced in [McCarthy 1962b]. [McCarthy and Painter 1967], however, was the first in which the correctness of a compiler was proved.

The problem of the relations between source language and object language arithmetic is dealt with here by assuming that the + signs in the definitions of *value* (1.1) and *step* (4.7) which define the semantics of the source and object languages represent the same operation. Theorem 1 does not depend on any properties of this operation, not even commutativity or associativity.

The proof is entirely straightforward once the necessary machinery has been created. Additional operations such as subtraction, multiplication and division could be added without essential change in the proof. For example, to put multiplication into the system the following changes would be required.

1. Use the analytic syntax, semantics, and induction principle given in §1 for the language including products.

2. Add an instruction *mul x* and the three syntactical functions *ismul(s)*, *adr(s)*, *mkmul(x)* to the abstract syntax of the object language together with the necessary relations among them.

3. Add to the definition (4.7) of *step* the clause

$$\textbf{else if } ismul(s) \textbf{ then } a(ac, c(adr(s), \eta) \times c(ac, \eta), \eta).$$

4. Add to the compiler (5.2) the clause

$$\textbf{else if } isprod(e) \textbf{ then } compile(p1(e), t) * mksto(t) * compile(p2(e), t + 1) * mkmul(t).$$

5. Add to the proof a case *isprod(e)* which parallels the case *issum(e)* exactly.

Many extensions of this compiling problem are treated in Painter [1967].

Exercise

1. Modify the arithmetic expression language to allow variable length sums. Describe how to modify the abstract syntax and how to fix up the interpreter and compiler, in order to give the semantics of the new language an compile it. Are any modifications of the object language syntax or semantics needed. How can the proof of correctness of the original compiler be turned into a proof of correctness of this new compiler?

2. What problems are encountered when conditional expressions are added to the source language? In what way is the current object language inadequate for this case? How would you solve this problem?

8. Some substantial exercises.

The following are some problems in programming and proving that can be solved using the techniques described in this chapter.

1. Consider the class of boolean conditional expressions (bces). A bce is either a literal (*lit*) or conditional (*if*) composed from a triple of bces with component parts *premise*, *consequent* and *alternate*. Make a table giving the abstract syntax of bces. Define an evaluator *bval* for bces given that *lval* determines the truth value of literals. Your evaluator should satisfy the usual rules for conditional expressions as given in Chapter III (III.3.1) - (III.3.6).

Work out the relations between the analytic and synthetic syntaxes, and give an induction principle for bces.

We say that a bce is in normal form if the premise of every subexpression is a literal. Define a function that converts a bce to normal form.

[Hint: Observe that in our external language the equivalence
if *p* **then** [**if** *q* **then** *s* **else** *t*] **else** [**if** *r* **then** *s* **else** *t*] ≡ **if** [**if** *p* **then** *q* **else** *r*] **then** *s* **else** *t*
holds.]

Prove that your function is total and that the result is in normal form. Proving totality will take some thought to get the correct induction predicate. Proving correctness, e.g. that the result is normal, will require making a formal statement of the notion of normality. This can be done by defining a predicate *normal* on bces. Show that the normalizing transformation is value preserving.

2. Write the syntax for a language in which programs are sequences of statements (possibly labeled), where a statement is either an assignment of a variable to the value of an expression, a conditional branch where the only tests are $e=0$ and $e>0$, or a return statement which returns the value of an expression. For simplicity assume expressions are of the form described in §3. Extend the machine language of §4 to allow conditional branches. Now write interpreters and a compiler for your new language and machine. Outline a proof of correctness. What new difficulties are encountered when branches are allowed in the source language?

Chapter X

COMPUTABILITY

The theoretical topics discussed in this chapter are not required for practical LISP programming. Neither are they required for proving correctness of programs - except where they touch on the theory itself. However, they are interesting, because they concern the fundamental limitations on what can be done with computers and the connection of LISP with computational formalisms that have been studied in mathematical logic since before computers were available. Moreover, the programming constructions required for computability theory are particularly easy and concrete in LISP, and this makes parts of the theory more accessible than if other computation formalisms are used.

The mathematical theory of computable functions is thoroughly treated in (Kleene 1952) and (Davis 1958). This theory is not concerned with proving facts about particular programs but rather with the existence of programs for solving various classes of problems. Once a problem has been shown to be algorithmically solvable, the theory of computable functions has little more to say.

1. A Call-by-name LISP interpreter.

Except for speed and memory size all present day stored program computers are equivalent in what computations they can do in that any such computer can be programmed to simulate any other.

This is well understood intuitively, and the first mathematical theorem of this kind was proved by A.M. Turing (1936), who defined a primitive kind of computer now called a Turing machine, and showed how to make a universal machine that could do any computation done by any Turing machine when given a description of the machine to be simulated and the initial tape of the computation to be imitated. There is no formal definition of *computable* apart from computable by Turing machine or a similar definition based on some other computing formalism. Therefore, the best that can be done is to prove different formalisms equivalent. (McCarthy 1960) shows how to write a LISP program for simulating any particular Turing machine which shows that any Turing computable function is LISP computable. The converse is an exercise in Turing machine programming.

In LISP the function *eval*, defined in Chapter I (I.13.1) is a universal LISP function in the sense that any computation done by any LISP function can be done by *eval* applied to suitable arguments.

The way *eval* handles arguments corresponds to the call-by-value method of parameter passing in ALGOL and similar languages. There is also a form of *eval* that corresponds to call-by-name. The practical difference between call-by-value and call-by-name is that sometimes a function doesn't need all its arguments, because the values of some arguments may make the evaluation of others unnecessary. A trivial example is that if x is 0, then the value of the product

xy can be determined without evaluating y. Other examples involve conditional expressions. In call-by-value, a subfunction is given the values of its arguments, while in call-by-name, it is given the location of program for computing them and can decide which it wants to compute. If this were all there is to say, call-by-name would clearly be superior. However, in straightforward call-by-name, it can happen that an argument that is used in two different places is computed twice, and there are other problems. Most LISPs use call-by-value, mainly for the historical reason that call-by-name was invented later in connection with Algol.

Here is a call-by-name *eval* called *neval*:

```
        neval[e, a] ←
        if at e then
              [if e = T then T
              else if e = NIL then NIL
              else if numberp e then e
              else neval[ad assoc[e, a], dd assoc[e, a]]]
        else if at a e then
              [if a e = CAR then a neval[ad e, a]
              else if a e = CDR then d neval[ad e, a]
1.1)          else if a e = CONS then neval[ad e, a] . neval[add e, a]
              else if a e = ATOM then at neval[ad e, a]
              else if a e = EQ then neval[ad e, a] = neval[add e, a]
              else if a e = QUOTE then ad e
              else if a e = COND then nevcon[d e, a]
              else if a e = LIST then mapcar[d e, λx: neval[x, a]]
              else neval[d assoc[a e, a] . d e, a]]
        else if aa e = LAMBDA then neval[adda e, nprup[ada e, d e, a]]
        else if aa e = LABEL then neval[adda e . d e, [ada e . a e] . a],
```

where the auxiliary function *nevcon* is given by

1.2) $nevcon[u, a] \leftarrow$ if $neval[aa\,u, a]$ then $neval[ada\,u, a]$ else $nevcon[d\,u, a]$.

and *nprup* is

1.3) $nprup[u, v, a] \leftarrow$ if $n\,u$ then a else $[a\,u . [a\,v . a]] . nprup[d\,u, d\,v, a]$.

eval and *neval* differ only in two terms. *eval* evaluates a variable by looking it up on the association list whereas *neval* looks it up on the association list and evaluates the result in the context in which it was put on the association list. Thus constants on the *neval* association list must be quoted. Correspondingly, when a λ-expression is encountered, *eval* forms a new association list by pairing the values of the arguments with the variables bound by the λ and putting the new pairs in front of the old association list, whereas *neval* pairs the arguments themselves with the variables and puts them on the front of the association list. The function *neval* also saves the current association list with each variable on the association list, so that the variables can be evaluated in the correct context. In most cases they give the same result with the same work, but *neval* gives a result for some functions of several arguments for which *eval* loops. An example is obtained by evaluating $F[1, 0]$ where F is defined by

$$F[m, n] \leftarrow \text{if } m=0 \text{ then } 0 \text{ else } F[m-1, F[m, n]].$$

eval always loops, because in order to evaluate $F[m, n]$ it must when m isn't 0, it must evaluate both arguments of $F[m-1, F[m, n]]$, and one of them is $F[m, n]$ again. When *neval* evaluates $F[1, 0]$, it finds itself evaluating $F[1-1, F[1, 0]]$, which expands to if 0=0 then 0 else $F[0-1, F[1, 0]]$, and this is evaluated without further recursion.

Vuillemin [1973] showed that if a function is *strict*, i.e. never has a defined value unless each argument has a defined value, then call-by-name and call-by-value always give the same result. When the functions have free variables as parameters, the situation is more complicated.

Exercises

1. Write *neval* and the necessary auxiliary functions in list form, and try them out on some examples.

2. Suppose a variable has several occurrences in a λ-expression, and this variable is bound to a complicated expression by *neval*. Each time the variable is reached by *neval*, the complicated expression is re-evaluated. Make a version of *neval* that uses *rplacd* to replace the expression by quoting the result of evaluating it the first time the expression is evaluated, so that subsequent evaluations of the variable find a quoted constant.

3. If you know about Turing machines, write a LISP function to simulate an arbitrary Turing machine given a description of the machine in some convenient notation.

4. Write a LISP function that will translate a Turing machine description into a LISP function that will do the same computation.

5. If you really like Turing machines, write a description of a Turing machine that will interpret LISP internal notation.

2. Non-computability.

Some of the most surprising results in mathematics are proofs that something cannot be done, especially when many people have spent years trying. While it is supposed that mathematicians are merely elevating their failure to solve a problem into a general principle, completely convincing impossibility proofs often exist. The first such result of was the discovery by the Pythagoreans that the square root of 2 is irrational, i.e. that no-one can find a square with integer sides and an integer diagonal. The nineteenth century brought the unsolvability of the three ancient straight-edge-and-compass construction problems - trisecting angles, duplicating the cube, and squaring the circle.

Perhaps the most spectacular and philosophically significant results of this kind were obtained in the 1930s when it was shown that no general methods exist for deciding certain well-known problems. LISP provides a convenient way of showing the existence of a problem undecidable by LISP computation - whether a LISP computation terminates or goes on indefinitely.

Some LISP calculations run on indefinitely. The most trivial case occurs if we make the recursive definition

2.1) $$loop\ x \leftarrow loop\ x$$

and attempt to compute *loop*[*x*] for any *x* whatsoever. Don't dismiss this example just because no-one would write such an obviously useless function definition. It is the "zero" of a certain class function definitions, and, as the Romans experienced but never learned, leaving zero out of the number system is a mistake.

Nevertheless, in most programming, non-terminating calculations are the results of errors in defining functions. Therefore, it would be useful to be able to tell whether a function definition gives a result for all arguments. In fact, it would be useful to be able to tell whether a function will terminate for a single argument. Let us make this goal more precise.

Suppose that *f* is a LISP function and *a* is an S-expression, and we would like to know whether the computation of *f*[*a*] terminates. Suppose *f* is represented by the S-expression *f** in internal notation. Then the S-expression (*f** (QUOTE *a*)) represents *f*[*a*]. Define the function *term* by giving *term*[*e*] the value true if *e* is an S-expression of the form (*f** (QUOTE *a*)) for which *f*[*a*] terminates and false otherwise. *term* is a perfectly well-defined function, because we have just defined it, but we can ask whether it is a LISP function. Can it be constructed from *car*, *cdr*, *cons*, *atom*, and *eq* using λ, *label*, and conditional expressions? Well, it can't, as we shall shortly prove, and this means that it is not *computable* whether a LISP calculation terminates, since if *term* were computable by any computer or in any recognized sense, it could be represented as a LISP function. Here is the proof:

Consider the function *terma* defined from *term* by

2.2) $$terma\ u \leftarrow \text{if } term\ list[u,\ list[\text{QUOTE},\ u]] \text{ then } loop\ u \text{ else true},$$

and suppose that *f* is a LISP function and that *f** is its S-expression representation.

What is *terma f**? Like *term f**, it tells us whether the computation of *f*[*f**] terminates, but it tells us this by going into a loop if *f*[*f**] terminates and giving true otherwise. Now if *term* were a LISP function, then *terma* would also be a LISP function. Indeed if *term* were represented by the S-expression *term**, then *terma* would be represented by the S-expression

2.3) *terma** = (LAMBDA (U) (COND ((*term** (LIST U (LIST (QUOTE QUOTE) U))) (LOOP U))
(T T))).

Now consider *terma*[*terma**]. According to the definition of *terma*, this will tell us whether *terma*[*terma**] is defined, i.e. it tells about itself. However, it gives this answer in a contradictory way; namely *terma*[*terma**] looping tells us that *terma*[*terma**] terminates, and *terma*[*terma**] being true tells us that *terma*[*terma**] doesn't terminate. Our assumption that *term* was a LISP function is thereby refuted, and there is no general procedure for telling whether a LISP calculation terminates. It is not logically excluded that there might be some more powerful means of effectively determining whether LISP computations terminate, because the notion of "effectively determining" is not definite. However, all the formal notions of "effectively determining" that have been proposed, including LISP computability, turn out to be mutually equivalent.

The above proof is the same in principle as Turing's proof that there is no Turing machine that tells whether Turing machine computations terminate. However, the technical details are simpler, because LISP is more facile than the Turing machine formalism at such constructions as the one that tells how to construct *terma* from *term*. Other problems are proved unsolvable by showing that a method for solving them would lead to a method for determining whether a computation terminates. (Davis 1958) is a good reference.

The above result does not exclude LISP functions that tell whether some LISP calculations terminate. It just excludes methods that work on all LISP calculations. This result can be strengthened.

Suppose a LISP function *term* sometimes says a calculation terminates, sometimes says a calculation doesn't terminate, and sometimes runs on indefinitely. Further suppose that when *term* gives an answer it is always right. We can further assume that *term*[*e*] = *T* whenever the calculation represented by *e* terminates. This can be assured by mixing the computation of *term*[*e*] with a computation of *eval*[*e*, NIL] and doing the computations alternately. If the *eval*[*e*, NIL] computation ever terminates, then the new function asserts termination.

Given such a *term*, we can always find a calculation *e* that does not terminate but *term* doesn't say so. The construction is just like that used in the previous proof. We construct

2.4) *terma u* ← if *term list*[*u*, *list*[QUOTE, *u*]] then *loop u* else true,

and then we consider *terma*[*terma**]. If this had the value true, then it wouldn't terminate so therefore it doesn't terminate but is not one of those expressions which *term* decides. Thus for any partial decider we can find a LISP calculation which doesn't terminate but which the decider doesn't decide. This can in turn be used to get a slightly better decider, namely

2.5) $term_1[e]$ ← if *e* = *terma** then DOESN'T-TERMINATE else *term*[*e*].

Of course, $term_1$ isn't much better than *term*, since it can decide only one more computation, but we can form $term_2$ by applying the same process, and so forth. In fact, we can even form $term_\omega$ which decides all the cases decided by any $term_n$. This can be further improved by the same process, etc. How far can we go? The answer is technical; namely, the improvement process can be carried out any recursive ordinal number of times.

Unfortunately, this kind of improvement is superficial, since none of the new computations proved not to terminate are likely to be of practical interest.

Exercises

1. Write a function that gives $term_{n+1}$ in terms of $term_n$.

2. Write a function that gives $term_\omega$ in terms of *term*.

3. A LISP predicate *term* is called a termination tester if for any S-expression *e*, *term*[*e*] = T if and only if the LISP evaluation of *e* terminates; otherwise *term*[*e*] = F or is undefined. Thus we must have *term*[(CAR (QUOTE (A)))] = T, and *term*[((LABEL FOO (LAMBDA (X) (FOO X))) NIL)] may have the value F or may be undefined. Let *term** be the S-expression representation of *term*.

Write a LISP function *bad* such that for any termination tester *term*, *bad*[*term**] is defined, but *term*[*bad*[*term**]] is undefined. Show that your *bad* works.

Hint: A two line answer is possible using *subst* and the function *quine*[*x*] = *subst*[*x*, X, (X (QUOTE X))].

3. Lambda calculus

The lambda notation used in LISP was taken from the lambda calculus developed by Alonzo Church in the late 1920s and described in his 1940 book, *The Calculi of Lambda Conversion*. This section covers some aspects of the lambda calculus that go beyond what is used in LISP.

Each LISP expression either evaluates to some object (like a x) or it represents a function (like $\lambda x.[a\,x\,.\,y]$). [In the terminology of logic, it has a definite type, but the word "type" is often used differently in computer science where numbers and list structure are considered of different types. Logicians use the word "sort" for the computer scientist's "type"]. The lambda calculus doesn't make this distinction, and any expression can be taken as a function.

Lambda calculus is much simpler than LISP in its basic structure, and its well-formed formulas (abbreviated wffs) are constructed as follows:

1. A variable is a wff.

2. If f and e are wffs, then so is $f(e)$.

3. If v is a variable and e is a wff, then $(\lambda v.e)$ is a wff.

4. An expression is a wff only if its being a wff follows from the above rules.

Here are some examples.

1. x and y are wffs by rule 1.

2. $x(x)$, $x(y)$, $y(x)$ and $(x(y))(x)$ are wffs. The same symbol can and often does appear in both function and argument positions.

3. $(\lambda x.(\lambda y.x))$, $(\lambda x.(\lambda y.y))$ and $(\lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x))))$ are all wffs that will be important in subsequent developments.

The occurrences of variables in a λ-expression may be divided into free and bound occurrences. Any occurrence of x within an expression $(\lambda x.exp)$ is a bound occurrence. The expression represented by exp is called the scope of this occurrence of λx. Other occurrences are called free, and sometimes the variables themselves are called free or bound. This is a misnomer, because a given variable may have both free and bound occurrences in a particular wff. An occurrence of a variable x is called bound by λx in $\lambda x.exp$ if it is free in exp. In the last λ-expression in example 3 above, each λx binds two occurrences of x.

Just as in LISP or in logic or in definite integrals in calculus, bound variables may be changed without changing the meaning of the expression provided all occurrences in the scope of a given variable binder are changed and no variable that is free in a subexpression becomes bound. Thus we may change $(\lambda x.(\lambda y.x))$ to $(\lambda u.(\lambda y.u))$ without changing the meaning of the expression. However, we could not change the x in $(\lambda x.(\lambda y.y))$ to y without changing the y to something else first, because this would give $(\lambda y.(\lambda y.y))$ which replaces the second occurrence of x by a variable which is now bound by the second λ. This kind of change of variables is called an alphabetic change and is also referred to as α-conversion.

The second kind of conversion is λ-conversion. It can be done when a wff consists of a λ-expression applied to another wff. An example is $(\lambda x.x(y))(u)$, and the general form is $(\lambda x.e1)(e2)$. The conversion consists essentially in replacing the given expression by the result of substituting $e2$ for x in $e1$. Thus the above-mentioned $(\lambda x.x(y))(u)$ converts to $u(y)$.

There is a catch. Suppose we want to convert $(\lambda x.(\lambda y.x(y)))(y(y))$. If we simply subsitute $y(y)$ for x in $(\lambda y.x(y))$, the free occurrences of y in $y(y)$ will become bound giving $(\lambda y.y(y)(y))$, and this isn't allowed, because it would change the intended meaning of the λ-expression. When a λ-conversion will result in the "capture" of a free occurrence of a variable, it must be preceded by a change of bound variables. In the above example, we can replace the bound occurrences of y before doing the λ-conversion and get first $(\lambda x.(\lambda u.x(u)))(y(y))$ and then $(\lambda u.y(y)(u))$. In thinking about programs to do λ-conversion, it is usually convenient to regard the preliminary α-conversion that may be needed as part of the λ-conversion rather than as a precondition.

We write $e1 \rightarrow e2$ as a way of saying that $e1$ converts to $e2$. We include conversions of subexpressions as λ-conversions. Here are some examples of λ-conversion.

$$(\lambda x.(\lambda y.x))(a)(b) \rightarrow (\lambda y.a)(b) \rightarrow a$$

$$(\lambda x.(\lambda y.y))(a)(b) \rightarrow (\lambda y.y)(b) \rightarrow b$$

$$(\lambda x.f(x(x)))(\lambda x.f(x(x))) \rightarrow f(\lambda x.f(x(x)))(\lambda x.f(x(x)))).$$

Each of them will be important.

A wff which is not subject to λ-conversion is said to be in normal form. Some λ-expressions can be converted successively 0 or more times and will eventually reach a normal form. The first two conversions above are reductions to a normal form. The third example is not reduced to normal form, and it is is clear that it can't be, because the argument of f on the right is just the whole expression on the left. We have

$$e \rightarrow f(e) \rightarrow f(f(e)) \rightarrow f(f(f(e))) \rightarrow \text{etc.}$$

Sometimes a wff may be convertible in several ways, because several of its subexpressions may be applications of a λ-expression to another expression. Whether a normal form is reached may depend on the order in which reductions are done. However, the Church-Rosser theorem of 1936 asserts that if a normal form is reached by some sequence of conversions, it is unique; different conversion paths cannot give rise to different normal forms. It can happen, however, that some paths give a normal form and others don't.

Church originally proposed the λ-calculus as a way of formalizing the foundations of

mathematics, and the original version included logical operations and quantifiers. Unfortunately, this system proved inconsistent (any formula was provable) and had to be abandoned. However, Church was able to show that arithmetic computations could be encoded as reductions of λ-expressions to normal form, and it was shown in the 1930s that λ-computations and Turing machine computations were equally powerful.

When we compare reduction of λ-expressions to normal form to LISP computations, it seems surprising that any significant computing can be done with only λ-conversion. There are apparently no lists, no a , d or *cons*, no conditional expressions and no recursion. However, as we shall now show all these things can be done in the λ-calculus. While Church first showed how to do computation in λ-calculus, the most straightforward and instructive course for us is to use constructions due essentially to Dana Scott that imitate pure LISP.

First we need a slightly abbreviated notation. All our functions have had just one argument, but the value of a function can be another function, and this is enough to imitate functions of several arguments. The idea is to regard $f(x, y)$ as just an abbreviation of $f(x)(y)$, i.e. to regard a function of two arguments as a function of one argument (the first argument) whose value is a function of one argument (the second argument). We then also regard $(\lambda x\, y.e)$ as an abbreviation of $(\lambda x.(\lambda y.e))$. We now proceed by identifying certain objects and functions of LISP as certain λ-expressions.

We begin with truth values and write

$$true = (\lambda x\, y.x) = (\lambda x.(\lambda y.x))$$

and

$$false = (\lambda x\, y.y) = (\lambda x.(\lambda y.x)).$$

So far this is quite arbitrary, and the motivation for these identifications is not apparent. Now we do conditional expressions and write

$$\textbf{if } p \textbf{ then } a \textbf{ else } b = p(a)(b) = p(a, b),$$

and we already get a return from our investment. Namely,

$$\textbf{if } true \textbf{ then } a \textbf{ else } b = true(a)(b) = (\lambda x\, y.x)(a, b) \to a,$$

which is just the property this conditional expression should have. Likewise

$$\textbf{if } false \textbf{ then } a \textbf{ else } b = false(a)(b) = (\lambda x\, y.y)(a, b) \to b.$$

Using this conditional expression notation, we can define the propositional connectives as

$$p \wedge q = \textbf{if } p \textbf{ then } q \textbf{ else } false = p(q, false)$$

$$p \vee q = \textbf{if } p \textbf{ then } true \textbf{ else } q = p(true, q)$$

$$\neg p = \textbf{if } p \textbf{ then } false \textbf{ else } true = p(false, true),$$

and all the usual truth table reductions immediately follow by reducing the λ-expressions to normal form. For example

$$true \wedge false = true(false, false) = (\lambda x\, y.x)(false, false) \rightarrow false.$$

It is important to notice that this only works if the propositional argument p of "if p then a else b" converts to *true* or *false*. In LISP anything but *false* behaves like *true*, but there is no way to make this happen in λ-calculus.

Next we need *car*, *cdr* and *cons*. We begin with versions that meet part of our requirements and will use them to construct the final versions. We write

$$cons1 = (\lambda\, x\, y\, z.z(x, y))$$

$$car1 = (\lambda x.x(true))$$

$$cdr1 = (\lambda x.x(false)),$$

which seem mysterious, but it is comforting to observe that

$$\begin{aligned} car1(cons1(e1, e2)) &= (\lambda x.x(true))(cons1(e1, e2)) \\ &\rightarrow cons1(e1, e2)(true) \\ &= (\lambda x\, y\, z.z(x, y))(e1, e2, true) \\ &\rightarrow true(e1, e2) \\ &\rightarrow e1 \end{aligned}$$

Likewise

$$cdr1(cons1(e1, e2)) \twoheadrightarrow e2$$

where $\twoheadrightarrow$ refers to eventual reduction rather than a single step. Thus *cons*1 has one main property of a pairing operation in that *car*1 and *cdr*1 enable us to get the operands back. However, we lack atoms and a way of identifying them. We get them by defining an atom NIL and new λ-expressions *cons*, *car* and *cdr*.

Our example system will have a single atom NIL, and we define

$$NIL = cons1(true, true)$$

Now we can define a *cons* that produces imitation S-expressions that can be distinguished from our imitation atom NIL.

$$cons = (\lambda x\, y.cons1(false, (cons1(x, y))))$$

$$car = (\lambda x.car1(cdr1(x)))$$

$$cdr = (\lambda x.cdr1(cdr1(x)))$$

$null = (\lambda x.car1(x))$.

The idea is that the atom NIL has its *car*1 part *true*, while every expression produced by *cons* has its *car*1 part *false*. Therefore, we use the *car*1 part in conditional expressions as a test for *null*.

Finally, to get recursion, we write

$recur = (\lambda f.(\lambda x.f(x(x))(\lambda x.f(x(x))))$,

which is Church's famous Y combinator. The property that makes it useful for defining recursion, as mentioned above, is that

$recur(f) \to f(recur(f)) \to f(f(recur(f))) \to$ etc.

Notice that *recur* is not in normal form, since it contains an application of a λ-expression inside it. Moreover, the above "fixed point" property shows that it has no normal form.

Now we can do pure LISP. We can write

$append = recur(\lambda f.(\lambda u\ v.$**if** $null(u)$ **then** v **else** $cons(car(u), f(cdr(u), v))))$

and

$equal = recur(\lambda f.(\lambda x\ y.(null(x) \wedge null(y) \vee (\neg null(x) \wedge \neg null(y)) \wedge (f(car(x), car(y)) \wedge f(cdr(x), cdr(y))))))$

Using the above abbreviations we can show such facts as

$append$(NIL, NIL) $\to$ NIL

$append(cons$(NIL, NIL), NIL) $\to cons$(NIL, NIL)

$equal$(NIL, NIL) $\to$ *true*

$equal(cons$(NIL, NIL), NIL) $\to$ *false*.

The expressions on the right are all abbreviations of normal forms.

There are two catches. First, while the Church-Rosser theorem assures us that if we get an answer it will be unique, it doesn't assure us that all sequences of reduction get answers, and some don't. Suppose we are computing $append$(NIL, NIL). We make the abbreviation

$app1 =$ **if** $null(u)$ **then** v **else** $cons(car(u), f(cdr(u), v))$

and have

$append$(NIL, NIL) $= recur(\lambda f\ u\ v.app1)$(NIL, NIL)

$\to (\lambda f\ u\ v.app1)(recur(\lambda f\ u\ v.app1)$(NIL, NIL).

At this point two conversions are possible. We could convert the new *recur*(λ*f u v*.*appl*) or we can convert the application of λ*f u v*.*appl* to its arguments and get

→ if *null*(NIL) then NIL else *cons*(*car*(NIL), *recur*(λ*f u v*.*appl*)(*cdr*(NIL), NIL)).

The former could go on indefinitely, while the latter is what imitates the usual LISP computation and promptly leads to a normal form. The general rule is to do first the outermost possible reduction. It can be shown that this rule leads to a normal form whenever a normal form exists. It corresponds to a call-by-name evaluation rule.

The preceding shows how to translate any pure LISP function as a wff of λ-calculus. If we also encode its arguments and form the wff consisting of the application of the function to its arguments and then normalize *e*, the process will terminate provided the LISP computation taken in the call-by-name sense terminates, and the resulting normal form will be an encoding of the result of the LISP computation.

The second catch is that the λ-calculus provides no way of making or proving general statements. While every individual case of *append*(*append*(*u*, *v*), *w*) will reduce to the same normal form as *append*(*u*, *append*(*v*, *w*)), the λ-calculus provides no way of stating or proving it within the formalism. Of course, it isn't hard to show by informal reasoning *about* the λ-calculus. Church's original 1920s inconsistent formalism did provide for general statements.

Here are some remarks.

1. Reducing to normal form even simple imitation LISP expressions like *append*(*cons*(NIL, NIL), *cons*(NIL, NIL)), i.e. imitating the computation of (NIL)*(NIL) = (NIL NIL), is a long computation best done by a λ-calculus reduction program written in LISP.

2. Further definitions could give an improved language for LISP in λ-calculus. One way to get more atoms is to regard a certain class of composite expressions formed with *cons*l as atoms just as we did with NIL.

3. The LISP in λ-calculus formalism works only if all expressions that occur in the computation are constructed in the manner provided for. Including general λ-expressions even as list elements can cause strange results when an attempt is made to normalize them.

4. While encoding LISP in λ-calculus shows that λ-calculus is universal in its computing capabilities, this doesn't seem to be a good way of taking advantage of the ability of the λ-calculus to express functions of higher type.

BIBLIOGRAPHY

Allen, John R. [1979]: *The Anatomy of LISP*, McGraw-Hill.

Baker, Henry G. [1978]: "List Processing in Real Time on a Serial Computer", *Communications of the ACM* 21, pp. 280-294.

Description and analysis of a list processing system that continuously reclaims garbage while linearizing and compacting space in use.

Boyer, Robert S. and J. Strother Moore [1978]: *A Computational Logic*, Academic Press, New York, xiv + 397 pp.

Brudno, A. L. [1963]: "Bounds and Valuations for Shortening the Scanning of Variations", [in Russian], *Problemy Kibernetiki* 10 pp.141-150.

First published account of the αβ algorithm.

Cartwright, Robert [1977]: *Practical Formal Semantic Defintion and Verification Systems*, Ph.D. thesis, Computer Science Department, Stanford University, Stanford, Ca.

Davis, Martin [1958]: *Computability and Unsolvability*, McGraw-Hill.

This book contains many unsolvability theorems.

Filman, R. E. and R. W. Weyhrauch [1976]: *A FOL Primer*, Stanford Artificial Intelligence Laboratory Memo AIM-288.

An excellent introduction to using the automatic first order logic proof checker FOL.

Gordon, M. [1975]: *Operational Reasoning and Denotational Semantics*, Stanford Artificial Intelligence Laboratory Memo, AIM-264.

Kleene, Stephen C. [1952]: *Introduction to Metamathematics*, Van Nostrand.

This is the standard work on recursive function theory.

Knuth, D. E. [1968a]: *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Addison-Wesley.

Section §2.3.5 contains a discussion of list structures and garbage collection algorithms.

Knuth, D. E. [1968b]: *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley.

Section §6.4 contains a discussion of hashing.

Knuth, D. E. and R. Moore [1975]: "An Analysis of Alpha-Beta Pruning", *Artificial Intelligence*, 6.

A history, description, and analysis of the $\alpha\beta$ algorithm.

Manna, Zohar [1974]: *Mathematical Theory of Computation*, McGraw-Hill.

Chapter 2 contains a discussion of first order logic and of the *natural deduction* method of proof.

Chapter 5 contains a discussion of functionals and fixedpoints.

The book also contains a discussion of structural induction and other methods of proving properties of programs.

McCarthy, John [1962a]: "Computer Programs for Checking Mathematical Proofs", in *Proc. Symp. Pure Math.* Vol.5 , Amer. Amth. Soc., Providence, R. I., pp219-227.

McCarthy, John [1962b]: "Towards a Mathematical Science of Computation," in C. M. Popplewell (ed.), *Information Processing, Proceedings of IFIP Congress* 62, pp21-28, North Holland Publishing Company, Amsterdam.

Contains discussions of transforming Flow Chart programs into recursive programs and of the Abstract Syntax of programming languages.

McCarthy, John [1963]: "A Basis for a Mathematical Theory of Computation", in P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, pp. 33–70. North-Holland Publishing Company, Amsterdam.

Contains a complete discussion of properties of conditional forms, including canonical forms. Also discusses computable functionals and recursion induction.

McCarthy, John [1964]: "A Formal Description of a Subset of Algol", *Proc. Conf. on Formal Language Description Languages*, Vienna.

McCarthy, John [1978a]: "History of LISP", in *Proceedings of the ACM conference on History of Programming Languages*, 1978.

McCarthy, John [1978b]: "Representation of Recursive Programs in First Order Logic", in *Proceedings the International Conference on Mathematical Studies of Information Processing*, Kyoto Japan, 1978.

McCarthy, John and James Painter [1967]: "Correctness of a Compiler for Arithmetic Expressions", *Proceeding of Symposia in Applied Mathematics*, Vol. 19, J. T. Schwartz (ed.), American Mathematical Society, pp 33-41.

Moszkowski, B. [1978]: informal memo

Nilsson, N. J. [1971]: *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill.

Painter, James [1967]: *Semantic Correctness of a Compiler for an Algol-like Language*, Ph.D. thesis, Computer Science Department, Stanford University, Stanford, Ca..

Steele, G.L. [1978]: *RABBIT: A Compiler for SCHEME* M.S. thesis, Department of Electrical Engineering and Computer Science, M.I.T.

Sussman, G.J. and G.L. Steele[1978]: *The revised report on SCHEME, a Dialect of LISP, MIT-AI Memo* 452.

Vuillemin, J. [1973]: *Proof Techniques for Recursive Programs*, Ph.D. thesis, Computer Science Department, Stanford University, Stanford, Ca..

Weyhrauch, R. W. [1977]: *A users manual for FOL*, Stanford Artificial Intelligence Laboratory Memo, AIM-235.1.

A manual for the automatic proof checker FOL.

Winston, P. H. [1977]: *Artificial Intelligence*, Addison-Wesley.

Appendix A

LISP Compilers

1. LCOM0: listing of MACLISP version.

```
(DECLARE (SETQ NO-DISK-HACKS T))
(DECLARE (READ))

(DEFPROP LCOFNS
 (LCOFNS COMPL COMP PRUP MKPUSH COMPEXP COMPLIS LOADAC COMCOND COMBOOL COMPANDOR)
VALUE)

(DEFPROP COMPL
 (LAMBDA(FILE)
  (UWRITE)
  (APPLY (QUOTE EREAD) FILE)
  (SELECT-DISK-INPUT
   (READ-UNTIL-EOF
    WITH
    Z
    DO
    (COND ((OR (EQ (CAR Z) (QUOTE DEFUN)) (AND (EQ (CAR Z) (QUOTE DEFPROP)) (EQ (CADDDR Z) (QUOTE EXPR))))
           (PROG (PROG)
                 (SETQ PROG
                       (COND ((EQ (CAR Z) (QUOTE DEFUN)) (COMP (CADR Z) (CADDR Z) (CADDDR Z)))
                             (T (COMP (CADR Z) (CADR (CADDR Z)) (CADDR (CADDR Z))))))
                 (UNSELECT-TTY (SELECT-DISK-OUTPUT (MAPC (FUNCTION PRINT) PROG)))
                 (PRINT (LIST (CADR Z) (LENGTH PROG)))))
          (T (UNSELECT-TTY (SELECT-DISK-OUTPUT (PRINT Z))))))
   (APPLY (QUOTE UFILE) (LIST (CAR FILE) (QUOTE LAP)))
   (QUOTE ENDCOMP)))
FEXPR)

(DEFPROP COMP
 (LAMBDA(FN VARS EXP)
  ((LAMBDA(N)
    (APPEND (LIST (LIST (QUOTE LAP) FN (QUOTE SUBR)))
            (MKPUSH N 1)
            (COMPEXP EXP (MINUS N) (PRUP VARS 1))
            (LIST (LIST (QUOTE SUB) (QUOTE P) (LIST (QUOTE Z) 0 0 N N)))
            (QUOTE ((POPJ P) NIL))))
   (LENGTH VARS)))
EXPR)

(DEFPROP PRUP
 (LAMBDA (VARS N) (COND ((NULL VARS) NIL) (T (CONS (CONS (CAR VARS) N) (PRUP (CDR VARS) (ADD1 N))))))
EXPR)

(DEFPROP MKPUSH
 (LAMBDA (N M) (COND ((LESSP N M) NIL) (T (CONS (LIST (QUOTE PUSH) (QUOTE P) M) (MKPUSH N (ADD1 M))))))
EXPR)

(DEFPROP COMPEXP
 (LAMBDA(EXP M VPR)
  (COND ((NULL EXP) (QUOTE ((MOVEI 1 0))))
        ((EQ EXP T) (QUOTE ((MOVEI 1 (QUOTE T)))))
        ((NUMBERP EXP) (LIST (LIST (QUOTE MOVEI) 1 (LIST (QUOTE QUOTE) EXP))))
        ((ATOM EXP) (LIST (LIST (QUOTE MOVE) 1 (PLUS M (CDR (ASSOC EXP VPR))) (QUOTE P))))
        ((OR (EQ (CAR EXP) (QUOTE AND)) (EQ (CAR EXP) (QUOTE OR)) (EQ (CAR EXP) (QUOTE NOT)))
         ((LAMBDA(L1 L2)
           (APPEND (COMBOOL EXP M L1 NIL VPR)
```

```
                (LIST (QUOTE (MOVEI 1 (QUOTE T))) (LIST (QUOTE JRST) 0 L2) L1 (QUOTE (MOVEI 1 0)) L2)))
          (GENSYM)
          (GENSYM)))
        ((EQ (CAR EXP) (QUOTE COND)) (COMCOND (CDR EXP) M (GENSYM) VPR))
        ((EQ (CAR EXP) (QUOTE QUOTE)) (LIST (LIST (QUOTE MOVEI) 1 EXP)))
        ((ATOM (CAR EXP))
         ((LAMBDA(N)
           (APPEND (COMPLIS (CDR EXP) M VPR)
                   (LOADAC (DIFFERENCE 1 N) 1)
                   (LIST (LIST (QUOTE SUB) (QUOTE P) (LIST (QUOTE Z) 0 0 N N)))
                   (LIST (LIST (QUOTE CALL) N (LIST (QUOTE QUOTE) (CAR EXP))))))
          (LENGTH (CDR EXP))))
        ((EQ (CAAR EXP) (QUOTE LAMBDA))
         ((LAMBDA(N)
           (APPEND (COMPLIS (CDR EXP) M VPR)
                   (COMPEXP (CADDAR EXP)
                            (DIFFERENCE M N)
                            (APPEND (PRUP (CADAR EXP) (DIFFERENCE 1 M)) VPR))
                   (LIST (LIST (QUOTE SUB) (QUOTE P) (LIST (QUOTE Z) 0 0 N N)))))
          (LENGTH (CDR EXP))))
        (T NIL)))
EXPR)

(DEFPROP COMPLIS
 (LAMBDA(U M VPR)
  (COND ((NULL U) NIL)
        (T (APPEND (COMPEXP (CAR U) M VPR) (QUOTE ((PUSH P 1))) (COMPLIS (CDR U) (SUB1 M) VPR)))))
EXPR)

(DEFPROP LOADAC
 (LAMBDA(N K)
  (COND ((GREATERP N 0) NIL) (T (CONS (LIST (QUOTE MOVE) K N (QUOTE P)) (LOADAC (ADD1 N) (ADD1 K))))))
EXPR)

(DEFPROP COMCOND
 (LAMBDA(U M L VPR)
  (COND ((NULL U) (LIST L))
        (T
         ((LAMBDA(L1)
           (APPEND (COMBOOL (CAAR U) M L1 NIL VPR)
                   (COMPEXP (CADAR U) M VPR)
                   (LIST (LIST (QUOTE JRST) 0 L) L1)
                   (COMCOND (CDR U) M L VPR)))
          (GENSYM)))))
EXPR)

(DEFPROP COMBOOL
 (LAMBDA(P M L FLG VPR)
  (COND ((ATOM P)
         (APPEND (COMPEXP P M VPR) (LIST (LIST (COND (FLG (QUOTE JUMPN)) (T (QUOTE JUMPE))) 1 L))))
        ((EQ (CAR P) (QUOTE AND))
         (COND ((NOT FLG) (COMPANDOR (CDR P) M L NIL VPR))
               (T
                ((LAMBDA(L1)
                  (APPEND (COMPANDOR (CDR P) M L1 NIL VPR) (LIST (LIST (QUOTE JRST) 0 L)) (LIST L1)))
                 (GENSYM)))))
        ((EQ (CAR P) (QUOTE OR))
         (COND (FLG (COMPANDOR (CDR P) M L T VPR))
               (T
                ((LAMBDA (L1)
                  (APPEND (COMPANDOR (CDR P) M L1 T VPR) (LIST (LIST (QUOTE JRST) 0 L)) (LIST L1)))
                 (GENSYM)))))
        ((EQ (CAR P) (QUOTE NOT)) (COMBOOL (CADR P) M L (NOT FLG) VPR))
        (T (APPEND (COMPEXP P M VPR) (LIST (LIST (COND (FLG (QUOTE JUMPN)) (T (QUOTE JUMPE))) 1 L))))))
EXPR)

(DEFPROP COMPANDOR
 (LAMBDA(U M L FLG VPR)
  (COND ((NULL U) NIL) (T (APPEND (COMBOOL (CAR U) M L FLG VPR) (COMPANDOR (CDR U) M L FLG VPR))))))
```

EXPR)

2. LCOM4: listing of MACLISP version.

```
(DECLARE (SETQ NO-DISK-HACKS T))
(DECLARE (READ))

(DEFPROP COMPFCNS
 (COMPFCNS COMPL
           COMP
           SUBSTACK
           PRUP
           MKPUSH
           COMPEXP
           STACKUP
           CCCHAIN
           COMPC
           COMCOND
           COMPLISA
           CCOUNT
           LOADAC
           COMPLIS
           CLASSIFY
           CLASS1
           CLASS2
           MKJRST
           COMBOOL
           COMPANDOR
           COMPANDOR1
           FLAT)
VALUE)

(DEFPROP COMPL
 (LAMBDA(FILE)
  (UWRITE)
  (APPLY (QUOTE EREAD) FILE)
  (SELECT-DISK-INPUT
   (READ-UNTIL-EOF
    WITH
    Z
    DO
    (COND ((OR (EQ (CAR Z) (QUOTE DEFUN)) (AND (EQ (CAR Z) (QUOTE DEFPROP)) (EQ (CADDDR Z) (QUOTE EXPR))))
           (PROG (PROG)
                 (SETQ PROG
                       (COND ((EQ (CAR Z) (QUOTE DEFUN)) (COMP (CADR Z) (CADDR Z) (CADDDR Z)))
                             (T (COMP (CADR Z) (CADR (CADDR Z)) (CADDR (CADDR Z))))))
                 (UNSELECT-TTY (SELECT-DISK-OUTPUT (MAPC (FUNCTION PRINT) PROG)))
                 (PRINT (LIST (CADR Z) (LENGTH PROG)))))
          (T (UNSELECT-TTY (SELECT-DISK-OUTPUT (PRINT Z))))))
   (APPLY (QUOTE UFILE) (LIST (CAR FILE) (QUOTE LAP)))
   (QUOTE ENDCOMP)))
FEXPR)

(DEFPROP COMP
 (LAMBDA(FN VARS EXP)
  ((LAMBDA(VPR N)
    (FLAT (LIST (LIST (LIST (QUOTE LAP) FN (QUOTE SUBR)))
                (MKPUSH N 1)
                (COMPEXP EXP (MINUS N) VPR)
                (SUBSTACK N)
                (QUOTE ((POPJ P) (LABEL NIL))))
          NIL))
   (PRUP VARS 1)
   (LENGTH VARS)))
EXPR)

(DEFPROP SUBSTACK
 (LAMBDA (N) (COND ((= N 0) NIL) (T (LIST (LIST (QUOTE SUB) (QUOTE P) (LIST (QUOTE Z) 0 0 N N))))))
EXPR)
```

```
(DEFPROP PRUP
 (LAMBDA (VARS N) (COND ((NULL VARS) NIL) (T (CONS (CONS (CAR VARS) N) (PRUP (CDR VARS) (ADD1 N))))))
EXPR)

(DEFPROP MKPUSH
 (LAMBDA (N M) (COND ((LESSP N M) NIL) (T (CONS (LIST (QUOTE PUSH) (QUOTE P) M) (MKPUSH N (ADD1 M))))))
EXPR)

(DEFPROP COMPEXP
 (LAMBDA(EXP M VPR)
  (COND ((NULL EXP) (QUOTE ((MOVEI 1 0))))
        ((OR (EQ EXP T) (NUMBERP EXP)) (LIST (LIST (QUOTE MOVEI) 1 (LIST (QUOTE QUOTE) EXP))))
        ((ATOM EXP) (LIST (LIST (QUOTE MOVE) 1 (PLUS M (CDR (ASSOC EXP VPR))) (QUOTE P))))
        ((EQ (CAR EXP) (QUOTE CAR))
         (COND ((ATOM (CADR EXP))
                (LIST (LIST (QUOTE HLRZ) 1 (QUOTE @)(PLUS M (CDR (ASSOC (CADR EXP) VPR))) (QUOTE P))))
               (T (LIST (COMPEXP (CADR EXP) M VPR) (QUOTE ((HLRZ 1 @ 1)))))))
        ((EQ (CAR EXP) (QUOTE CDR))
         (COND ((ATOM (CADR EXP))
                (LIST (LIST (QUOTE HRRZ) 1 (QUOTE @)(PLUS M (CDR (ASSOC (CADR EXP) VPR))) (QUOTE P))))
               (T (LIST (COMPEXP (CADR EXP) M VPR) (QUOTE ((HRRZ 1 @ 1)))))))
        ((OR (EQ (CAR EXP) (QUOTE AND))
             (EQ (CAR EXP) (QUOTE OR))
             (EQ (CAR EXP) (QUOTE NOT))
             (EQ (CAR EXP) (QUOTE EQ)))
         ((LAMBDA(L1 L2)
           (LIST (COMBOOL EXP M L1 NIL VPR)
                 (LIST (QUOTE (MOVEI 1 (QUOTE T)))
                       (LIST (QUOTE JRST) 0 L2)
                       (LIST (QUOTE LABEL) L1)
                       (QUOTE (MOVEI 1 0))
                       (LIST (QUOTE LABEL) L2))))
          (GENSYM)
          (GENSYM)))
        ((EQ (CAR EXP) (QUOTE COND)) (COMCOND (CDR EXP) M (GENSYM) VPR))
        ((EQ (CAR EXP) (QUOTE QUOTE)) (LIST (LIST (QUOTE MOVEI) 1 EXP)))
        ((ATOM (CAR EXP))
         (LIST (COMPLISA (CDR EXP) M VPR)
               (LIST (LIST (QUOTE CALL) (LENGTH (CDR EXP)) (LIST (QUOTE QUOTE) (CAR EXP))))))
        ((EQ (CAAR EXP) (QUOTE LAMBDA))
         ((LAMBDA(N)
           (LIST (STACKUP (CDR EXP) M VPR)
                 (COMPEXP (CADDAR EXP) (DIFFERENCE M N) (APPEND (PRUP (CADAR EXP) (DIFFERENCE 1 M)) VPR))
                 (SUBSTACK N)))
          (LENGTH (CDR EXP))))
        (T NIL)))
EXPR)

(DEFPROP STACKUP
 (LAMBDA(U M VPR)
  (COND ((NULL U) NIL)
        (T (LIST (COMPEXP (CAR U) M VPR) (QUOTE ((PUSH P 1))) (STACKUP (CDR U) (SUB1 M) VPR)))))
EXPR)

(DEFPROP CCCHAIN
 (LAMBDA(EXP)
  (AND (OR (EQ (CAR EXP) (QUOTE CAR)) (EQ (CAR EXP) (QUOTE CDR)))
       (OR (ATOM (CADR EXP)) (CCCHAIN (CADR EXP)))))
EXPR)

(DEFPROP COMPC
 (LAMBDA(EXP N2 M VPR)
  (COND ((ATOM EXP) (ERROR (QUOTE COMPC)))
        ((EQ (CAR EXP) (QUOTE CAR))
         (COND ((ATOM (CADR EXP))
                (LIST (LIST (QUOTE HLRZ) N2 (QUOTE @)(PLUS M (CDR (ASSOC (CADR EXP) VPR))) (QUOTE P))))
               (T (CONS (LIST (QUOTE HLRZ) N2 (QUOTE @) N2) (COMPC (CADR EXP) N2 M VPR)))))
        ((ATOM (CADR EXP))
         (LIST (LIST (QUOTE HRRZ) (QUOTE @) N2 (PLUS M (CDR (ASSOC (CADR EXP) VPR))) (QUOTE P))))
```

```
        (T (CONS (LIST (QUOTE MRRZ) N2 (QUOTE @) N2) (COMPC (CADR EXP) N2 M VPR)))))
EXPR)

(DEFPROP COMCOND
 (LAMBDA(U M L VPR)
  (COND ((NULL U) (LIST (LIST (QUOTE LABEL) L)))
        ((AND (NOT (ATOM (CAAR U))) (EQ (CAAAR U) (QUOTE NULL)) (NULL (CADAR U)))
         (LIST (COMPEXP (CADAAR U) M VPR) (LIST (LIST (QUOTE JUMPE) 1 L)) (COMCOND (CDR U) M L VPR)))
        ((EQ (CAAR U) T) (LIST (COMPEXP (CADAR U) M VPR) (LIST (LIST (QUOTE LABEL) L))))
        (T
         ((LAMBDA(L1)
           (LIST (COMBOOL (CAAR U) M L1 NIL VPR)
                 (COMPEXP (CADAR U) M VPR)
                 (LIST (LIST (QUOTE JRST) @ L) (LIST (QUOTE LABEL) L1))
                 (COMCOND (CDR U) M L VPR)))
          (GENSYM)))))
EXPR)

(DEFPROP COMPLISA
 (LAMBDA(U M VPR)
  ((LAMBDA(Z)
    (LIST (COMPLIS Z M 1 VPR)
          (LOADAC Z (DIFFERENCE 1 (CCOUNT Z)) 1 (DIFFERENCE M (CCOUNT Z)) VPR)
          (SUBSTACK (CCOUNT Z))))
   (CLASSIFY U)))
EXPR)

(DEFPROP CCOUNT
 (LAMBDA (Z) (COND ((NULL Z) @) ((= (CAAR Z) 4) (ADD1 (CCOUNT (CDR Z)))) (T (CCOUNT (CDR Z)))))
EXPR)

(DEFPROP LOADAC
 (LAMBDA(Z M2 N2 M VPR)
  (COND ((NULL Z) NIL)
        ((= (CAAR Z) 1)
         (CONS (LIST (QUOTE MOVE) N2 (PLUS M (CDR (ASSOC (CDAR Z) VPR))) (QUOTE P))
               (LOADAC (CDR Z) M2 (ADD1 N2) M VPR)))
        ((= (CAAR Z) @)
         (CONS (LIST (QUOTE MOVEI) N2 (LIST (QUOTE QUOTE) (CDAR Z))) (LOADAC (CDR Z) M2 (ADD1 N2) M VPR)))
        ((= (CAAR Z) 2) (CONS (LIST (QUOTE MOVEI) N2 (CDAR Z)) (LOADAC (CDR Z) M2 (ADD1 N2) M VPR)))
        ((= (CAAR Z) 3) (LIST (REVERSE (COMPC (CDAR Z) N2 M VPR)) (LOADAC (CDR Z) M2 (ADD1 N2) M VPR)))
        ((= (CAAR Z) 5) (LOADAC (CDR Z) 1 (ADD1 N2) M VPR))
        (T (CONS (LIST (QUOTE MOVE) N2 M2 (QUOTE P)) (LOADAC (CDR Z) (ADD1 M2) (ADD1 N2) M VPR)))))
EXPR)

(DEFPROP COMPLIS
 (LAMBDA(Z M K VPR)
  (COND ((NULL Z) NIL)
        ((= (CAAR Z) 4)
         (LIST (COMPEXP (CDAR Z) M VPR) (QUOTE ((PUSH P 1))) (COMPLIS (CDR Z) (SUB1 M) (ADD1 K) VPR)))
        ((= (CAAR Z) 5)
         (LIST (COMPEXP (CDAR Z) M VPR) (COND ((= K 1) NIL) (T (LIST (LIST (QUOTE MOVE) K 1))))))
        (T (COMPLIS (CDR Z) M (ADD1 K) VPR))))
EXPR)

(DEFPROP CLASSIFY
 (LAMBDA (U) (CLASS2 (CLASS1 U NIL) NIL T))
EXPR)

(DEFPROP CLASS1
 (LAMBDA(U V)
  (COND ((NULL U) V)
        ((ATOM (CAR U))
         (COND ((OR (EQUAL (CAR U) NIL) (EQUAL (CAR U) T) (NUMBERP (CAR U)))
                (CLASS1 (CDR U) (CONS (CONS @ (CAR U)) V)))
               (T (CLASS1 (CDR U) (CONS (CONS 1 (CAR U)) V)))))
        ((EQUAL (CAAR U) (QUOTE QUOTE)) (CLASS1 (CDR U) (CONS (CONS 2 (CAR U)) V)))
        ((CCCHAIN (CAR U)) (CLASS1 (CDR U) (CONS (CONS 3 (CAR U)) V)))
        (T (CLASS1 (CDR U) (CONS (CONS 4 (CAR U)) V)))))
```

```
EXPR)

(DEFPROP CLASS2
 (LAMBDA(U V FLG)
  (COND ((NULL U) V)
        ((AND FLG (= (CAAR U) 4)) (CLASS2 (CDR U) (CONS (CONS 5 (CDAR U)) V) NIL))
        (T (CLASS2 (CDR U) (CONS (CAR U) V) FLG))))
EXPR)

(DEFPROP MKJRST
 (LAMBDA (L) (LIST (LIST (QUOTE JRST) 0 L)))
EXPR)

(DEFPROP COMBOOL
 (LAMBDA(P M L FLG VPR)
  (COND ((EQ P T) (COND (FLG (MKJRST L)) (T NIL)))
        ((ATOM P) (LIST (COMPEXP P M VPR) (LIST (LIST (COND (FLG (QUOTE JUMPN)) (T (QUOTE JUMPE))) 1 L))))
        ((EQ (CAR P) (QUOTE EQ))
         (LIST (COMPLISA (CDR P) M VPR)
               (COND (FLG (QUOTE ((CAMN 1 2)))) (T (QUOTE ((CAME 1 2)))))
               (MKJRST L)))
        ((EQ (CAR P) (QUOTE AND))
         (COND ((NOT FLG) (COMPANDOR (CDR P) M L NIL VPR))
               (T
                ((LAMBDA (L1) (LIST (COMPANDOR1 (CDR P) M L1 L NIL VPR) (LIST (LIST (QUOTE LABEL) L1))))
                 (GENSYM)))))
        ((EQ (CAR P) (QUOTE OR))
         (COND (FLG (COMPANDOR (CDR P) M L T VPR))
               (T
                ((LAMBDA (L1) (LIST (COMPANDOR1 (CDR P) M L1 L T VPR) (LIST (LIST (QUOTE LABEL) L1))))
                 (GENSYM)))))
        ((EQ (CAR P) (QUOTE NOT)) (COMBOOL (CADR P) M L (NOT FLG) VPR))
        ((EQ (CAR P) (QUOTE NULL))
         (LIST (COMPEXP (CADR P) M VPR) (LIST (LIST (COND (FLG (QUOTE JUMPE)) (T (QUOTE JUMPN))) 1 L))))
        (T (LIST (COMPEXP P M VPR) (LIST (LIST (COND (FLG (QUOTE JUMPN)) (T (QUOTE JUMPE))) 1 L))))))
EXPR)

(DEFPROP COMPANDOR
 (LAMBDA(U M L FLG VPR)
  (COND ((NULL U) NIL) (T (LIST (COMBOOL (CAR U) M L FLG VPR) (COMPANDOR (CDR U) M L FLG VPR)))))
EXPR)

(DEFPROP COMPANDOR1
 (LAMBDA(U M L L2 FLG VPR)
  (COND ((NULL U) (MKJRST L2))
        ((NULL (CDR U)) (COMBOOL (CAR U) M L2 (NOT FLG) VPR))
        (T (LIST (COMBOOL (CAR U) M L FLG VPR) (COMPANDOR1 (CDR U) M L L2 FLG VPR)))))
EXPR)

(DEFPROP FLAT
 (LAMBDA(U S)
  (COND ((NULL U) S)
        ((NULL (CAR U)) (FLAT (CDR U) S))
        ((EQ (CAR U) (QUOTE LABEL)) (CONS (CADR U) S))
        ((ATOM (CAR U)) (CONS U S))
        (T (FLAT (CAR U) (FLAT (CDR U) S)))))
EXPR)
```

FUNCTION INDEX