DOCUMENT RESUME

ED 133 989 FL 007 168

AUTHOR	King, Margaret
TITLE	Introduction to Programming in LISP.
PUB DATE	76
NOTE	31p.; Course notes for a Tutorial on Computational Semantics given at the Institute for Semantic and Cognitive Studies in Castagnola, Switzerland
AVAILABLE FROM	North Holland Press, 335 Jan van Galenstraat, P.O. Box 1270, Amsterdam, The Netherlands (as part of conference proceeding, "Computational Semantics)
EDRS PRICE	MF-$0.83 HC-$2.06 Plus Postage.
DESCRIPTORS	Algorithms; Artificial Languages; *Computational Linguistics; Computer Programs; Computer Science; *Computer Science Education; Course Content; Courses; *Instructional Materials; *Programing; *Programing Languages

ABSTRACT

The first section of this course on programming introduces LISP as a programming language designed to do symbol manipulation, with consequent prevalence in auto-instructional (AI) work. A programming language specifies in its description a number of primitive operations which the computer knows how to perform. An algorithm is the set of instructions which constitutes a description of the method by which the task can be performed. This algorithm must be written in the programming language which the computer can deal with. The constructs (elements of LISP) that work in algorithms are described and an example task provided. LISP is a functional language, in that its instructions consist of directives to apply some function to some argument. Part 1 of the course concludes with a description of some of LISP's functions, and Part 2 deals with its symbol manipulation functions, specifically in manipulating LISP constructs known as lists. Part 3 gives instruction in writing inherently recursive functions non-recursively. Practice exercises follow each section of the course. (CLK)

((INTRODUCTION TO PROGRAMMING) IN LISP)

PART 1

Introduction

This course is a course about programming: that is about how to get a computer to do what you want it to do. To get a computer to perform a specified task, you must give it a set of instructions which constitute a description of the method by which the task can be performed (this description is technically known as an algorithm). The instructions have to be written in a programming language, in our case LISP. Each programming language specifies in its description a number of primitive operations which the computer already knows how to perform and also specifies what you must write so as to make it perform these primitive operations. Your description must be built up entirely of primitive operations - you cannot assume that the computer itself will be able to break down a complicated task into the primitive operations which make it up, even though to you the resolution may seem obvious.*

An analogy

In order to clarify this point, let us forget about LISP for a moment and imagine a human-like robot, which only knows how to perform the following simple tasks:

1. It can read a line of writing and distinguish between items on a line, i.e. it assumes that one or more spaces appearing after a sequence of characters delimits an item. Thus a line

   THIS IS A LINE OF PRINT

   contains six items.

---

* Exactly what primitive operations are available varies from one computer language to another. You will begin to become acquainted with LISP's repertoire as we procede with the course.

2. It can count.

3. It can maintain a number of lists, and can manipulate lists to the extent of adding or deleting an item.

4. It knows what 'if' means, and can transfer its attention from one task to another when commanded to do so.

5. When told to stop work it will do so.

Now let us assume that, for some obscure reason, we want to discover how many people in a specific collection of people have only one forename, and we want our robot to perform the machanical task of collecting into a list the names of all such people.

First we must get our initial information (our data) into a form which can be manipulated by the robot. So, in this case, we might write each person's full name on a separate line:

e.g. JOHN HIGGINBOTTOM SMITH
MARY JONES
DIETMAR HANS KRIEGER
COLLETTE MEUNIER
-etc-

Now we must specify an algorithm for the robot:

0. Set up a list to contain the names which have only one forename. This list will initially be empty (i.e. will contain no names).

1. Check if there are lines to be read. If there are not, give back the list of names which have only one forename and stop work. (Note that if there are no lines at all to be read, this means give back a list with nothing in it).

2. Count how many items there are on the next line of data.

3. If there are only two items, add this line of writing to the list of names with only one forename. If there are more than two, go on to the next instruction.

4. Delete this line of data and return to instruction 1.

There are several points worth noticing in this. First, the robot does not know, and does not need to know, that he is dealing with the names of people, or even names at all. We have arranged the data in a form whereby it conforms to a construct ("a line of writing") which the robot knows about. What this line represents is irrelevant to the robot. The same situation obtains with the computer. Via a programming language it knows about certain sorts of constructs, and what these constructs relate to in the world outside is irrelevant to it.

Secondly, deciding on the set of instructions is an arduous task. It is difficult to be sure of the logic, to be sure that everything which has to be done is done, that nothing which should not be done is done, that the robot knows when to stop. This is true also of programming. Finding a correct algorithm is the most difficult part of the job, and the part that merits most attention. Once an algorithm is found, describing it in the correct syntactic form for a particular language is a secondary, although sometimes frustrating, task.

A LISP program

Just so that you can see what LISP looks like, what follows is a translation into LISP of the steps detailed above. Although you obviously cannot, and should not try, to understand in detail how this program works, some parts of it are intuitively clear. It consists of the definition in LISP of a function called NAMES, which will return as its value the list of people with only one forename. This list is built up in the next to the last line of the function.

```
(DE NAMES (NAMESLIST)
          (COND ((NULL NAMESLIST) NIL)
                ((EQUAL (LENGTH (CAR NAMESLIST)) 2)
                 (CONS (CAR NAMESLIST) (NAMES (CDR NAMESLIST))))
                (T (NAMES (CDR NAMESLIST)))))
```

To try out this program a specific list of names must be set up and the computer instructed to apply to that list the function, NAMES, as defined above. (We shall return to this notion of a function and its application later in the course). This is done by using the function name to call the function, and giving it a specific list as its argument:

```
e.g.   (NAMES (QUOTE ((JAMES SMITH)                    }
                      (MARY ANN ROGERS)                } argument
          function name (PETER WILLIAMS)               }
                      (JOHN ALBERT GOOD))))            }
```

We shall return to this example in later lectures to illustrate various points.

Lists

One point is worth mentioning now. Earlier it was said that information had to be put into a form which the machine could deal with. One of LISP's primitive constructs is the list. LISP lists have to be given in a linear form. So the LISP version of the original list, when it appears in the argument of the function call above, is one long list enclosing a number of shorter lists, where each of the shorter lists is the name of one person.

The structure of a list is shown by bracketing. A list is always enclosed in brackets, so all the following are lists:

(A B C)	is a list of three elements
(A)	is a list of one element
( )	is a list of no elements (the empty list). It can also be written as NIL.
((A B) (C D))	is a list of two elements, each of which is also a list.

You should distinguish carefully between the list (A) which is a list containing just one element A, and A (without brackets) which is an atom, discussed in the following section.

Atoms

The other chief primitive of LISP is the atom, which may be either a literal atom or a numeric atom.

A literal atom is a string of capital letters and decimal digits.

e.g. A
APPLE
A2

are all atoms. Literal atoms are used as names: in our example program one, NAMESLIST, was used as the name of a list, and another, NAMES, as the name of a function. Sometimes the value assigned to a particular name is changed during the course of execution of the program, as happens with NAMESLIST in the example program. Names whose value changes are technically known as variables.

When a literal atom is evaluated, the result is the value assigned to that atom.

A numeric atom is simply a number, and means what it intuitively ought to mean. In this course we shall restrict ourselves entirely to integer numbers.

e.g. 2
-10
-357

are all numeric atoms. When a numeric atom is evaluated the result is the number itself.

Some LISP functions

With these two primitives defined we can make a start in learning LISP.

LISP is often said to be a functional language, in that its instructions consist of instructions to apply some function to some argument(s). Most people are familiar with the function notation of mathematics, where, for example, f(x) means that f (not here specifically defined) is a function which is to be applied to some argument x. LISP works the

same way, the functions being defined either within the system itself or by the user. We shall first consider some system defined functions and return to user defined functions in part 2.

The arithmetic functions

The arithmetic functions are intuitively obvious.

1. (TIMES 2 3)

fairly obviously requires the computer to multiply 2 and 3 together to get the answer 6. The result of typing this expression into the system will be that 6 is printed out as the result of the function.

One thing that may seem a little odd to some people is that the function name TIMES appears before the arguments rather than between them. This follows the function notation of mathematics and is modelled on it, except that the function name appears inside rather than outside the opening bracket.

Before we continue with the arithmetic functions it is worth pointing out that when the user is communicating with the LISP system, the system always expects him to type in an expression to be evaluated. This expression may be a list or an atom. As we noted earlier, the result of evaluating a literal atom is the value assigned to that atom, and the result of evaluating a numeric atom is the number itself. When the expression typed in is a list, the system expects that list to consist of the name of the function to be applied, TIMES in example (1) above, followed by the arguments to which the function is to be applied, 2 3 in example (1). Sometimes the arguments themselves are calls on functions so that they too consist of a list structured in the same way

(<function name> <argument(s)>).

In this way quite complicated instructions can be built up.

To return to the arithmetic functions:

TIMES has already been mentioned. It may have any number of arguments and will return their product.

2. (TIMES 3 3 3) will return 3x3x3 i.e. 27.

PLUS also may have any number of arguments and will return their sum.

3. (PLUS 3 3 3) will return 3+3+3 i.e. 9.

To illustrate the point about arguments sometimes being function calls themselves, consider what happens if the expression

4. (TIMES (PLUS 2 2) 3)

is given to the LISP system for evaluation. The first element of the list which forms the expression is the name of the function to be applied, TIMES. The second element is itself a list, so the system expects to find an instruction to evaluate a function here, and then use the result as an argument to the first function. So it will evaluate

(PLUS 2 2)

PLUS, which is a call of the function which performs addition, together with its arguments, 2 2, to which the function is to be applied. This intermediate result, of course, is 4, although, since it is an intermediate result, the 4 will not be printed out by the system. This result can now be used as an argument of TIMES, so TIMES has the arguments 4 3. The result of evaluating TIMES then is 12, and this final value of the expression will be printed out. The process of including function calls as arguments may go on as far as the user wishes: the following is valid LISP, and the reader may find it instructive to evaluate it himself:

5. (TIMES (PLUS (TIMES (PLUS 2 2) 3) (TIMES 2 (PLUS 3 4)) 5)
(TIMES (PLUS (TIMES 2 4) (TIMES 3 6) 6) (PLUS 2 3)))

Here, briefly is a list of some other arithmetic functions and what they do:

(ADD1 arg)	adds 1 to the argument, which may be either a number or an expression which evaluates to a number.
(SUB1 arg)	subtracts 1 from the argument, with the same constraints on the argument as in ADD1.

(DIFFERENCE arg1 arg2) subtracts the second argument from the first. (Same constraints on arguments as above)

(QUOTIENT arg1 arg2) divides the first argument by the second, giving as its result the integer part of the actual answer, i.e. (QUOTIENT 6 2) is 3, but (QUOTIENT 6 4) is 1, since the fractional part of the answer is discarded. (Same constraints on arguments as above)

PART ONE

EXERCISES

Use the computer as a calculating machine to calculate the following:

1. 3 + 7
2. 7 - 3
3. 8/3
4. 8^2
5. 10 + 1
6. 10 - 1
7. (3 + 7) x 10
8. (5 + 1)/2
9. ((3 + 7)/5) x 10
10. ((5 + 1)/(4 - 1)) x ((7 + 3)/(7 - 2))
11. 3^7
12. The remainder when 538 is divided by 3.

PART ONE SOLUTIONS TO EXERCISES

1. (PLUS 3 7)

2. (DIFFERENCE 7 3)

3. (QUOTIENT 8 3)

4. (TIMES 8 8)

5. (ADD1 10)

6. (SUB1 10)

7. (TIMES (PLUS 3 7) 10)

8. (QUOTIENT (ADD1 5) 2)

9. (TIMES (QUOTIENT (PLUS 3 7) 5) 10)

10. (TIMES (QUOTIENT (ADD1 5) (SUB1 4)) (QUOTIENT (PLUS 7 3) (DIFFERENCE 7 2)))

11. (TIMES 3 3 3 3 3 3 3)

12. (DIFFERENCE 538 (TIMES (QUOTIENT 538 3) 3))

PART 2

Introduction

In Part 1 we introduced the idea of LISP as a functional language, saying that all instructions in LISP, except when an atom is being evaluated, were instructions to apply a function to one or more arguments. We then looked briefly at the arithmetic functions as an illustration of this. But LISP is not designed as a programming language in which to do arithmetic: it is designed as a language to do symbol manipulation. Hence its prevalence in AI work. In this lecture then we shall begin to investigate LISP's symbol manipulation facilities by considering first its ability to manipulate lists.

The reader should remember that a list may have other lists among its elements, and that these other lists may themselves have sublists, and that this sort of nesting may go to any depth.

Primitive list manipulating functions

1. CAR

CAR is a function which takes one argument. This argument is always evaluated and should evaluate to a list. (An attempt to find CAR of an atom will cause a failure). The value CAR returns will be the first, i.e. leftmost, element of the list to which its argument evaluates. Thus

(i) (CAR (QUOTE (A B C D)))

will return A as its value.

A further function QUOTE has crept into example (i). It is needed because, as we said in the paragraph introducing CAR, the argument of CAR is always evaluated. Now, the reader will remember from Part 1 that when the system evaluates a list, it expects to find a function name as the first element of the list and to be able to interpret the remaining elements of the list as arguments

of that function. Thus the value of (A B C D) is not (A B C D). In fact, unless a function called A has been defined, it hasn't got a value, and an attempt to evaluate it will cause an error message to be printed out. If what we want to do is to find the first element of (A B C D) we must therefore find some way of suppressing evaluation of the list. This is precisely what the function QUOTE does. Its value is simply its argument, unevaluated.

(ii) (QUOTE (A B C D))

has the value (A B C D), and example (i) will work correctly.

If the list (A B C D) was to be used frequently, it would soon become tiresome to have to write (QUOTE (A B C D)) every time we wanted to perform an operation on the list. It would be easier to assign the value (A B C D) to a variable, and use the variable instead. This can be done by using the SETQ function. SETQ takes two arguments: the first is the variable to which the value is to be assigned, the second the value itself. Only the second argument is evaluated, i.e. SETQ quotes its first argument - the Q is there to remind you of this. Thus

(iii) (SETQ D1 (QUOTE (A B C D)))

gives the value (A B C D) to the variable D1, and a later function call

(iv) (CAR D1)

will have the value A, since the result of evaluation of a literal atom is the value which has been assigned to that atom.

QUOTE need not be used with numeric atoms, since they evaluate to the number itself.

(v) (SETQ B1 4)

will assign the value 4 to the literal atom (variable) B1.

Calls to other functions can of course appear in the second argument of SETQ.

(vi) (SETQ NUMBER (PLUS 2 3))

assigns the value 5 to the literal atom (variable) NUMBER.

To recapitulate: CAR returns as its value the first element of the list to which its argument evaluates. This first element may of course be either an atom or a list.

2. CDR

CDR also takes one argument, which again should evaluate to a list. It returns as its value a list of the items remaining when the first element of the list has been deleted. Thus

(vii) (CDR (QUOTE (A B C D)))

returns as its value (B C D) and

(viii) ((CDR (QUOTE ((A B) C D)))

returns as its value (C D).

The relationship between CAR and CDR should be quite clear.

3. Combining CAR and CDR

CAR's and CDR's may be combined, as you would expect. For example, say we have a list (THIS IS A LIST) and we want to get at the A. In order to achieve this we can evaluate

(ix) (CAR (CDR (CDR (QUOTE (THIS IS A LIST))))).

One of LISP's chief characteristics is function composition (i.e. allowing function calls to appear as arguments to functions, as in the examples of Part 1 and as in example (ix) here). Since the composition of CAR's and CDR's is very frequent in most LISP programs, special functions have been defined which serve as a shorthand. C always begins the function name, R always ends it. In between the C and the R may come up to 8 (in this system) A's and/or D's: each A stands for a CAR, each D for a CDR. The order of A's or D's follows the order in which the expressions would come if written out in full. Thus the expressions in example (ix) may be re-written:

(x) (CADDR (QUOTE (THIS IS A LIST)))

4. CONS

CAR and CDR take lists apart. CONS provides a way of building lists up. CONS is a function which takes two arguments; the first may be any expression (i.e. a literal atom, a numeric atom, or a list), but the second must evaluate to a list. Both arguments are evaluated. The value of CONS is the new list formed by adding the value of first argument on to the beginning of the value of the second. Thus

(xi) (CONS (QUOTE THE) (QUOTE (GIRL IS GERMAN)))

will return as its value (THE GIRL IS GERMAN).

If the first argument of CONS is a list, the new list formed will have the first argument as its first element - the lists will not be joined together. Thus

(xii) (CONS (QUOTE (THE PRETTY)) (QUOTE (GIRL IS GERMAN)))

will return as its value ((THE PRETTY) GIRL IS GERMAN).

Using CONS with an atom as its first argument and the empty list, NIL, as its second will, as you would expect, form a new list which has the atom as its only member. Thus

(xiii) (CONS (QUOTE THE) NIL)

will return (THE).

If the first argument is a list and the second the empty list the new list will, similarly, be a list of only one member, but that one member will itself be a list. Thus

(xiv) (CONS (QUOTE (A LIST)) NIL)

will return ((A LIST)).

CAR, CONS and CDR are logically related. The reader should convince himself of the relationship by working out the values of the following, assuming that A has the value (FIRST PART) and B has the value (SECOND PART).

a. (CAR (CONS A B))

b. (CDR (CONS A B))

c. (CONS (CAR A) (CDR A))

CAR, CDR and CONS are the three primitive list manipulation functions.

5. The COND function

A program rarely consists of instructions which must all be obeyed under all circumstances. An instruction is needed which allows some instructions sometimes not to be obeyed (as in the example in Part 1 where we didn't want the robot to add the new name to the list of names with only one forename if in fact that name had more than one forename). LISP provides this facility in the COND function:

Its general form is this:

```
(COND ((if this expression is true) (evaluate these expressions))
      ((otherwise, if this expression is true) (evaluate these
                                                expressions))
                          .
                          .
                          .
      ((otherwise, if this expression is true) (evaluate these
                                                expressions)) )
```

As you can see, COND may have any number of arguments, each argument consisting of a list, the first element of which is a test condition the result of whose evaluation is either true or false, the second a list of one or more expressions which are to be evaluated if the value of the test condition is true. COND procedes through its argument list until it finds a test condition which is true. Then it evaluates the instructions associated with that test condition and stops. It does not carry on through the remaining arguments.

The most obvious question raised by this account of COND is how a test condition can be formulated. A number of LISP functions act specifically as predicate functions, i.e. they return the values true or false. A list of some of the most useful predicate expressions follows:

Predicate Expressions

1. Functions which compare atoms or expressions.

   a. (EQUAL arg1 arg2) compares two expressions. It returns the value true (*T* on our system) if they evaluate to the same expression, false (NIL) otherwise.

2. Functions associated with arithmetic predicates

   a. (NUMBERP arg) returns true if its argument is a number, false otherwise.

   b. (ZEROP arg) returns true if its argument is zero, false otherwise.

   c. (LESSP arg1 arg2) returns true if its first argument is greater than the second, false otherwise.

   d. (GREATERP arg1 arg2) returns true if its first argument is greater than the second, false otherwise.

3. Functions associated with logical predicates

   a. (AND arg1 arg2 arg3.....argn) AND evaluates each of its arguments in turn starting with the first. If it finds an argument which evaluates to false, it returns false as its own value and does not evaluate the remaining arguments. If all its arguments evaluate to true it returns true.

   b. (OR arg1 arg2 arg3.....argn) OR evaluates each of its arguments in turn. If it finds an argument which evaluates to true, it returns true as its own value and does not evaluate the remaining arguments. If all its arguments evaluate to false it returns false as its own value.

   c. (NULL arg1)

   (NOT arg1). These are equivalent ways of writing the same function. It returns true if its argument is false, i.e. NIL. Otherwise if its argument is true it returns false. It is the logical negation operator. (Remember that an empty list is re-

presented by NIL - this makes this particular predicate very useful).

Example program

If we now look again at the example program from lecture one, we find that it has become a great deal more comprehensible. It is reproduced for convenience here.

```
(DE NAMES (NAMESLIST)
          (COND ((NULL NAMESLIST) NIL)
                ((EQUAL (LENGTH (CAR NAMESLIST)) 2)
                 (CONS (CAR NAMESLIST) (NAMES (CDR NAMESLIST))))
                (T (NAMES (CDR NAMESLIST)))))
```

A function LENGTH is used which has not so far been mentioned. It takes one argument, a list, and returns the number of items in the list. The last argument of the COND looks a little odd, since the test condition is simply T. This is an atom which always evaluates to true and so is used, usually in the last argument of a COND, to make certain that if all the other tests have failed, this one won't and the instructions associated with it will be executed.

Defining functions

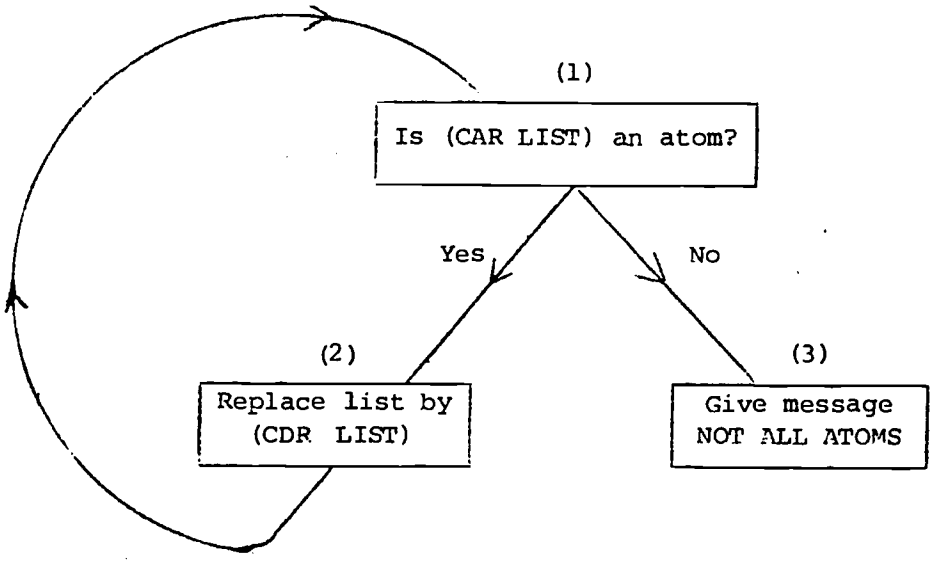
To understand this program completely now, it is only necessary to understand the DE function which starts it off, and which is indeed the function of which all the rest are arguments. It is the function which allows the user to define functions for himself, in addition to the predefined system functions. Its first argument is the name by which the function being defined is to be known - in the example NAMES. Its second argument is a list of variable names. These variables are essentially dummies. When the function is actually used, the arguments given it in the call are evaluated. Their (quoted) values are then substituted for the dummy variables in the remaining argument. This remaining argument is the expression (which may be quite complicated) which is to be evaluated when the function is called.

An example

To elucidate this a little further let us define a function ATOMIC which will return the message (ATOMS ONLY) if the list which is its argument contains only atoms, and the message (NOT ALL ATOMS) if any of the elements in the list are not atoms. Thus if LISTA has the value (A B C D) then (ATOMIC LISTA) should produce the message (ATOMS ONLY), whilst if LISTB has the value ((A B) C D) (ATOMIC LISTB) should produce the message (NOT ALL ATOMS).

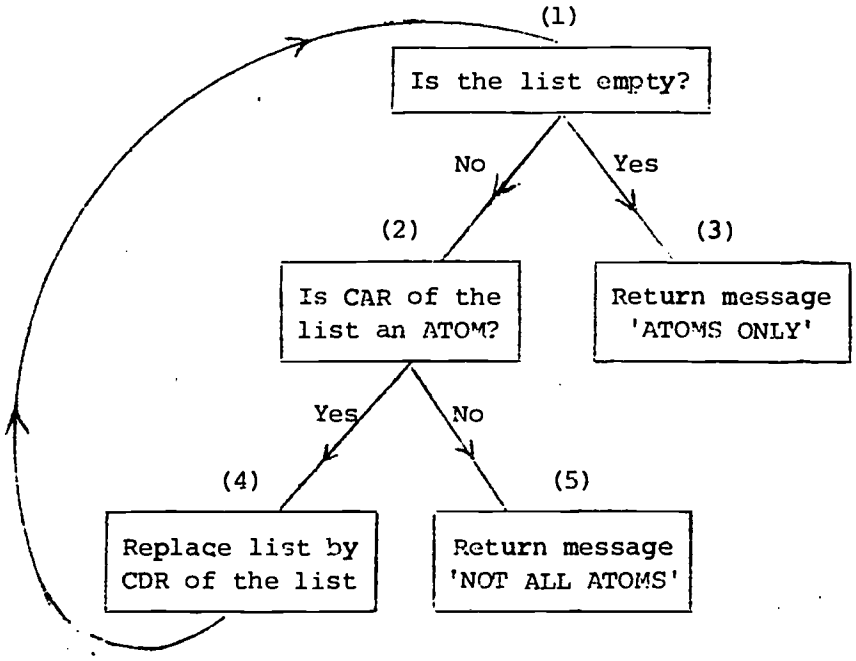
First we must define an algorithm for the task, that is we must find a general recipe by which it can always be decided whether a list contains only atoms.

This is relatively easy, since it is clear that we must look at each element in turn and test to see whether it is an atom or not, and LISP provides a predicate function (ATOM arg) which returns true if its argument is an atom (literal or numeric) and false otherwise. But how do we look at each member of a list? CAR will get us the first element of the list, CDR will get what remains of it when the first element has been deleted. So if we test CAR of the list to see if it is an atom, we can stop with the appropriate message if it is not (there is no point in continuing to look at the remaining elements of the list) and, if it is an atom, we can replace the list by CDR of the list and repeat this operation. We can summarize the decisions taken so far thus:



This looks satisfactorily tidy, except that there is no way of getting out of the loop which results from answering 'yes' to the question in box (1), i.e. we have no way of knowing when every element of the list has been checked. But successive CDR's of the list will eventually, after all the elements have been eliminated, produce the empty list. If we get to the empty list without having found a non-atomic element, then obviously all the elements are atoms and we should produce the message 'ATOMS ONLY'. So a test for the empty list must be included somewhere in the loop. It would be sensible to put it before the test to see if CAR of the list is an atom for two reasons: first, someone may call the function with an empty list as its argument - so this should be tested first. Secondly the empty list is simply the atom NIL, and an attempt to find CAR of an atom will cause a failure, as we noted earlier.

With these modifications in mind the algorithm can be completed.



Note that both these diagrams are only an approximation of the process involved. The process described as 'replace list by CDR of list' in box (4) could more accurately be described as 're-enter the whole process which this diagram defines with the list used this time through the process

replaced by its CDR. When this is translated into LISP terms, since the process will be defined as a function and will therefore have a name by which it can be called, it comes out as 'call this same function with CDR of the list as argument'. Any reader who is interested in a fuller account of this technique of calling a function from within its own definition (technically known as recursion) will find such an account in Hayes' notes for the advanced programming course.

The function ATOMIC can now be defined as follows:

```
(DE ATOMIC (L)
           (COND ((NULL L) (QUOTE (ATOMS ONLY)))
                 ((ATOM (CAR L)) (ATOMIC (CDR L)))
                 (T (QUOTE (NOT ALL ATOMS)))))
```

Once the programmer knows how to define functions himself it becomes possible to produce quite complicated programs.

EXERCISES PART 2

1. Use the machine to find the result of the following sequences of instructions.

   a. (CAR (QUOTE (FIND THE FIRST CHARACTER)))

   b. (SETQ A (QUOTE A B C D))

      (CAR A)

      (CDR A)

      (CADR A)

      (CDDR A)

   c. (SETQ B (QUOTE ((A B C) (1 2 3))))

      (CAR B)

      (CDR B)

      (CAAR B)

      (CADR B)

      (CDDR B)

      (CADDR B)

2. Write instructions which will pick out the word 'THE' in each of the following lists:

   a. (THE CAR WAS RED)

   b. (IT WAS IN THE GARAGE)

   c. (ON THE ROAD WAS A CAT)

   d. (IT HAD DRUNK ALL THE WHISKY)

3. Write a COND expression which will set a variable B to the list (VALUE IS ZERO) if another variable A is equal to zero, and to the list (VALUE IS NOT ZERO) if it is not. Give A a value before typing in the COND expression. (You will be able to tell whether your COND expression has worked correctly by the value it prints out. COND always returns as its value the value of the last expression evaluated within the COND).

4. For each of the following conditions define a function of one argument L which has value true if the condition is satisfied and NIL otherwise.

a. The first element of L is a literal atom.

b. The first element of L is 12.

c. L has at most four elements (either atoms or lists).

d. The second element of L is greater than the fourth. (Assume that L is a four element list where each element is a numeric atom).

5. The function factorial (n) can be defined by:

factorial (0) = 1

factorial (n) = nx(n - 1)

Define a corresponding LISP function and test it with various values of n.

6. Define a function of two arguments (the first may be any expression, the second a list) called MEM which will return a value true if the first of its arguments is identical with any top level member of its second argument, NIL otherwise.

e.g. (MEM (QUOTE A) (QUOTE (BED IS A WORD))

should return true.

(MEM (QUOTE A) (QUOTE ((BED IS A) WORD)))

should return NIL.

SOLUTIONS TO EXERCISES PART 2

2. a. (CAR (QUOTE (THE CAR WAS RED)))

b. (CADDDR (QUOTE (IT WAS IN THE GARAGE)))

c. (CADR (QUOTE (ON THE ROAD WAS A CAT)))

d. (CADDDDR (QUOTE (IT HAD DRUNK ALL THE WHISKY)))

3. (SETQ A.....)

```
(COND ((ZEROP A) (SETQ B (QUOTE (VALUE IS ZERO))))
      (T (SETQ B (QUOTE (VALUE IS NOT ZERO)))))
```

4. a.
```
(DE F1 (L)
       (AND (ATOM (CAR L)) (NOT (NUMBERP (CAR L)))))
```

b.
```
(DE F2 (L)
       (AND (NUMBERP (CAR L)) (EQUAL (CAR L) 12)))
```

c.
```
(DE F3 (L)
       (NOT (GREATERP 4 (LENGTH L))))
```

d.
```
(DE F4 (L)
       (GREATERP (CADR L) (CADDDR L)))
```

5.
```
(DE FACTORIAL (N)
              (COND ((ZEROP N) 1)
                    (T (TIMES N (FACTORIAL (SUBI N))))))
```

6.
```
(DE MEM (S L)
        (COND ((NULL L) NIL)
              ((EQUAL S (CAR L)) T)
              (T (MEM S (CDR L)))))
```

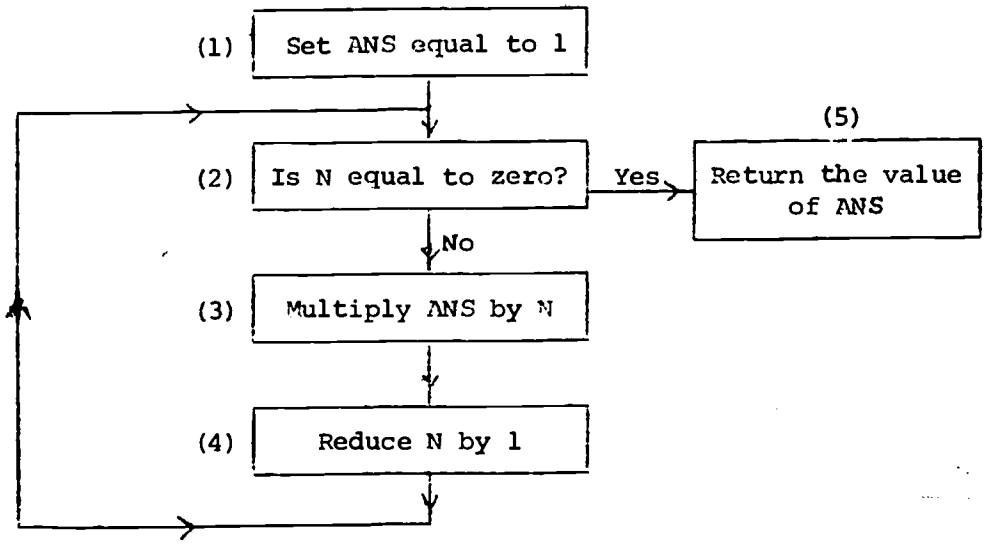
PART 3

Introduction

At the end of the last lecture we discussed how to define a function. The basic pattern of the function defined there was: 'If the list is not empty, do something to the first element of the list. Substitute the remaining elements of the list for the original list and repeat the whole operation on the list thus obtained. It was convenient to be able to define ATOMIC as a function and then to be able to re-call the same function from within the function definition. Many operations upon lists follow the same basic pattern, so the recursion technique is very useful. But some operations are not recursive in character, and it sometimes happens that for efficiency's sake it is preferable to write even inherently recursive functions non-recursively. In this part we shall discuss how that can be done.

Iteration

If we want to be able to write non-recursive functions we need to be able to execute instructions sequentially, so that we can write loops. Many of you will have written a factorial function recursively in doing the exercises attached to part 2. A non-recursive version needs a loop, as in the diagram below.



(Actually this could be made slightly more efficient by testing to see if N is 1 rather than zero. But we will leave it like this so that it matches the earlier definition of factorial).

An inspection of the diagram should convince the reader that he knows how to deal with boxes (1) to (4): their LISP equivalents can easily be stated in terms of SETQ, COND, TIMES and SUB1. But as yet we have no way of dealing with (5) or of constructing a LISP equivalent of the line which shows what must be executed after the instruction equivalent to box (4) has been executed.

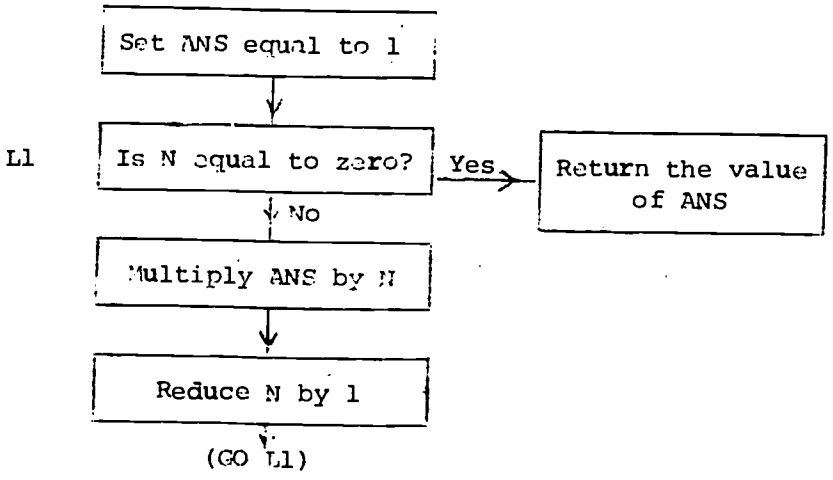
Both these difficulties are overcome by use of the function PROG. A call of PROG has a general form thus:

```
(PROG (list of variables)
      (1st instruction)
      (2nd instruction)
            .
            .
            .
      (nth instruction)
```

It allows iteration ('jumping back') because the sequence of instructions may contain GO and/or RETURN instructions.

A GO instruction has the form

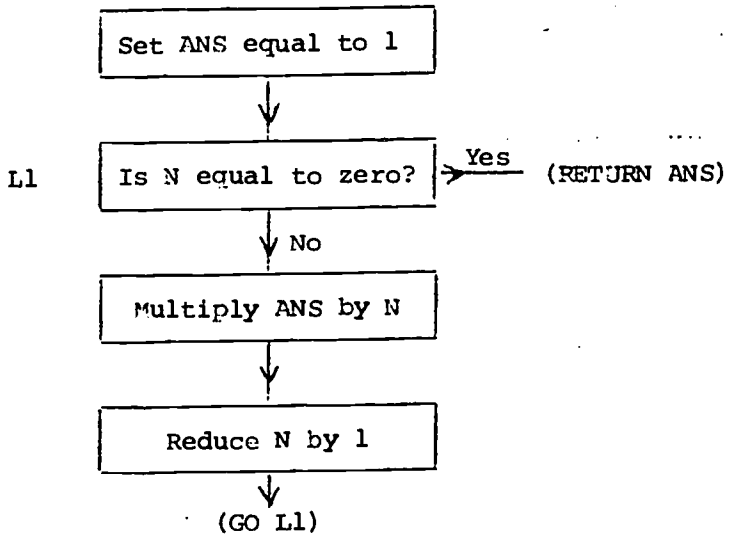
(GO label) where the label is a literal atom which must appear before some instruction within the PROG sequence. When the GO is executed, control is transferred to the instruction preceded by the label specified. This can be shown by altering our earlier diagram slightly:



The RETURN instruction allows execution to leave the PROG sequence at any point. It has the form

(RETURN expression).

When the RETURN instruction is reached, the expression which forms its argument is evaluated and becomes the value of the whole PROG. So we can incorporate it into our diagram:



If the last instruction in a PROG sequence is executed and it is not a call on either of the functions GO or RETURN, PROG returns NIL. Execution has "dropped out of the bottom" as it were.

The only other new feature of the PROG is the list of variables which appears as its first argument. These variables serve as temporary variables within the PROG. They are initially set to NIL and cannot be accessed from outside the PROG. Even if no temporary variables are used, this list must still appear. In the case where there are no temporary variables it will be the empty list, ( ). In the factorial example ANS is a variable which is only used within the PROG: it can therefore be a temporary variable. N is different, since it carries a value into the PROG sequence. It must therefore be a dummy variable (an argument) in the function definition of which the PROG sequence forms the body.

We can now translate the whole of this sequence into LISP.

```
(DE FACTORIAL (N)
              (PROG (ANS)
                    (SETQ ANS 1)
                 L1 (COND ((ZEROP N) (RETURN ANS)))
                    (SETQ ANS (TIMES ANS N))
                    (SETQ N (SUB1 N))
                    (GO L1)))
```

It is worth comparing this with the recursive definition of the same function:

```
(DE FACTORIAL (N)
              (COND ((ZEROP N) 1)
                    (T (TIMES (FACTORIAL (SUB1 N)) N))))
```

In this case it is clear that the recursive version is simpler and easier to follow. Although the exercises attached to this lecture ask that the same function should be defined both iteratively and recursively, the programmer should in real life take some care about which version he chooses, using clarity and simplicity as guides in making his choice.

In these three lectures we have by no means covered the whole of the LISP language. You will probably hear other features referred to in other courses. All I have attempted to do is to show you what LISP is like and to give enough of a foothold to allow anyone interested to continue on their own.

Margaret King

PART 3 EXERCISES

Define each of the following functions first recursively and then iteratively:

1. A function LONG which counts how many top level elements there are in a list.

   e.g. (LONG (QUOTE ((A B) C))) is 2
   (LONG (QUOTE (A B C))) is 3

   This is the function LENGTH which was used in the example program in the first lecture. But don't use that name for it or you will re-define the system function.

2. A function ZEROES which returns the number of zeroes in a list.

3. A function LAST which returns the last element in a list.

PART 3 SOLUTIONS TO EXERCISES

1. a)
```
(DE LONG (L)
         (COND ((NULL L) 0)
               (T (ADD1 (LONG (CDR L))))))
```

b)
```
(DE LONG (L)
   (PROG (X)
         (SETQ X 0)
    BACK (COND ((NULL L) (RETURN X)))
         (SETQ X (ADD1 X))
         (SETQ L (CDR L))
         (GO BACK)))
```

2. a)
```
(DE ZEROS (L)
          (COND ((NULL L) 0)
                ((AND (NUMBERP (CAR L)) (ZEROP (CAR L)))
                      (ADD1 (ZEROS (CDR L))))
                (T (ZEROS (CDR L)))))
```

b)
```
(DE ZEROES (L)
   (PROG   (SUM)
           (SETQ SUM 0)
    BACK   (COND ((NULL L) (RETURN SUM))
           ((AND (NUMBERP (CAR L) (ZEROP (CAR L)))
           (SETQ SUM (ADD1 SUM))))
           (SETQ L (CDR L))
           (GO BACK)
                 ))
```

3. a)
```
(DE LAST (L)
        (COND ((NULL (CDR L)) (CAR L))
              (T (LAST (CDR L)))))
```

b)
```
(DE LAST (L)
   (PROG ( )
    BACK (COND ((NULL (CDR L)) (RETURN (CAR L))))
         (SETQ L (CDR L))
         (GO BACK)))
```

n.b. This solution will not work if LAST is called with the empty list as its argument. Why not? Can you alter it so that it will work even under this condition?