J. A. Campbell

Imperial College of Science and Technology

Computer Unit

LISP on the Imperial College 7090 Computer

The LISP version available on this machine is LISP 1.55, the Stanford (California) Export Version of LISP 1.5. Details regarding the structure of the system and preparation of program decks may be found in the LISP 1.5 PROGRAMMER's MANUAL, published by the M.I.T. Press. Any program prepared for LISP 1.5 will run without change in the present system. The extra features available in LISP 1.55 are listed in notes available from Programs Records Office, Room 405, Electrical Engineering Dept., Imperial College, London SW7.

The system uses the FMS monitor and a small FAP calling program to gain control, and returns control to the FMS monitor at the end of a job. A complete job, with calling program, will have the form:

```
*ID   7
*          PLEASE LOAD CURRENT LISP TAPES
*      PAUSE
*      XEQ
*      PACK
*      FAP
*      RTBB  7
       RCHB  10C
       LCHB  Ø
       TRA   1
10C    IOCT  Ø,,3
       END
*      DATA
```

(transfer card: 7, 9 in column 1: 2,3,4,5,6,8,9 in column 3)
(blank card)
(LISP program)
(End-of-file card)

Assembled copies of cards 2 through 13 (i.e. including transfer card) are available on request from the Reception Room 404, address as above.

SYSTAP is B7 and SYSTMP A5. Tape B7 should therefore be specified as "LISP" and A5 as scratch on the job slip. A5 could, of course, be reserved and later used as SYSTAP. SYSPOT and SYSPPT are A3; punched output is processed with the standard FMS output without any further action from the programmer.

IMPERIAL COLLEGE LISP PROGRAMMING GUIDE

J.A. Campbell

CONTENTS

1. Plan of the Guide

It is reported in section 3.1.4 of the User's Manual at Imperial College that the programming langugage LISP is available at Imperial College but is 'difficult to learn'. This view may have been induced by a reading of the LISP 1.5 Programmer's Manual (reference (3.1)), which is a work far more suited to the regular user of LISP than to the beginner. The first part of the present Guide, up to Section 33, has the purpose of disproving the quotation in the first sentence above. With the help of this part it should be possible to write and run simple LISP programmes without reference to the LISP 1.5 Manual.

Following the first Pause at section 33 come nine sections that presuppose knowledge of the LISP 1.5 Manual, (which is readily available from efficient British booksellers), and that contain comments or reports of tricks which may be helpful for the advanced programmer.

The second Pause (section 43) is followed by some information of use to programmers who wish to construct their own LISP systems from the set-up tape at Imperial College. The set-up tape also contains card images for LISP programmes that have applications in the physics of sub-atomic particles, and in section 45 there is a description of thse programmes.

Apart from a postscript, the Guide ends with a few addresses of people who may be able to provide advice if the beginning programmer in LISP encounters what appear to be inexplicable bugs.

2. Some History and Propaganda

Most non-specialised literature about computers, in the classical (pre-1950?) period of the subject, spent much time in discussing their possible uses for logical problems (e.g. chess-playing) that could not properly have been reduced to problems only of numerical manipulation. At that stage it would have seemed that numerical problems represented a very small part of the total field of scientific applications of computers. Nevertheless, the most popular computing language continues to be FORTRAN, which is most effective only for programmes that consist mainly of arithmetic operations.

The earliest successful attempts to construct a programming language suitable for the description of general manipulations of data not limited merely to numerical material led to the IPL series of languages. The best-known member of this series is IPL-V. While still used in some centres, IPL-V is now rather less popular than LISP, because its superficially close resemblance to a machine code makes it more difficult to employ either as a programming language or as a medium for the teaching of principles of non-numerical computing.

LISP (LISP 1) dates from 1959, although the first publications (notably a LISP 1 Programmer's Manual) to deal with it did not appear until 1960. It was developed first for an IBM 7090 at MIT by J. McCarthy and others. Its flavour of mathematical logic (well demonstrated by the reference (3.4)), which was carried over into the LISP 1.5 Programmer's Manual (3.1), may have been responsible for the legends about the difficulty of LISP.

The MIT LISP 1.5 evolved into LISP 1.55 in Boston, and in 1963 and 1964 some further and more utilitarian additions (see Appendix I of the reference (3.1)) by courtesy of the Artificial Intelligence Project at Stanford University turned it into LISP 1.56. The LISP 1.56 system was imported from Stanford to Imperial College in February 1965, and at various times up to August 1966 it was altered somewhat to provide a 'LISP 1.6' which was of greater use for several specialised problems in physics. However, to avoid alienation of non-physicists, it has since been 'de-tuned' to approximately the specification of LISP 1.56.

Some of the examples in this Guide have a bias towards theoretical physics, but that does not imply that the uses of LISP are narrowly specialised. The LISP 1.5 manual lists as uses game-playing, electrical circuit theory and symbolic calculations in differential and integral calculus. To this list, from experience at Imperial College alone, we can add modal logic, the automatic proof of theorems in the first-order predicate calculus, teaching, mechanical anthropology (including the editing of text), spectroscopy (applied to physical chemistry), group theory, neurophysiology, automatic writing and debugging of FORTRAN programmes and translations between different machine codes. In all of these problems, LISP programmes have primarily manipulated non-numerical data and performed non-arithmetic operations. Therefore it is not necessary to consider the IBM 7090 at Imperial College as being confined to the very limited range of scientific problems that can be solved only by numerical methods and FORTRAN programmes. Any procedure that can be described completely in unambiguous English can be described (and presumably executed) in LISP.

3. References

This Guide is not completely self-contained, and ideally it should be read in conjunction with the LISP 1.5 Programmer's Manual, which is in any case an essential reference for the serious user of LISP. The references (3.2) and (3.3) are in effect commentaries on the Manual, with reports of original work on the uses of LISP and its preparation for computers other than the IBM 7090. (3.4) is now largely of historical interest. (3.5) is a Manual for LISP on the CDC 3600 computer, and its points of difference with (3.1) demonstrate what are the most difficult parts of the job of setting up LISP for a new computer.

3.1. - 'LISP 1.5 Programmer's Manual' (J. McCarthy et al.) MIT Press, Cambridge 42, Massachusetts 02142, U.S.A. The original edition was dated 17 August 1962, but probably the only version now obtainable is the revised edition from February 1965. The price is $3.00, or 23/- from British booksellers.

3.2.- 'The Programming Language LISP; Its Operation and Applications' (editors E.C. Berkeley and D. Bobrow), Information International Inc., 200 Sixth Street, Cambridge, Mass. (1964). Copies may still be available from the Defense Documentation Center, Arlington, Virginia, U.S.A. for $7.50.

3.3.- 'The Nature, Uses and Implementation of the Computer Programming Language LISP' (editors E.C. Berkeley and D. Edwards), Information International Inc., address as in (3.2). The expected date of publication is March 1967.

3.4.- J. McCarthy, contribution to 'Computer Programming and Formal Systems' (editors P. Braffort and D. Hirschberg), North-Holland Publishing Company, Amsterdam, Holland (1963).

3.5.-'LISP 3600: User's Manual' (J.G. Kent), Teknisk notat E-98, Forsvarets Forskningsinstitutt, Box 25, Kjeller, Norway (1966).

4. Availability

At 25 October 1966, LISP programming systems were available for four large commercial computers: IBM 7090/94, D.E.C. PDP-6, CDC 3600 and ICT Atlas (formerly Ferranti Atlas). Information about PDP-6 LISP is rather vague, except that research efforts by the System Development Corporation (2500 Colorado Avenue, Santa Monica, California 90406, U.S.A) on varieties of LISP 1.5 (NOT to be confused with LISP 2) are due to be transferred from the military AN/FSQ-32 machine to the PDP-6 shortly, so that enquiries either to S.D.C. or to the Digital Equipment Corporation Users' Service, Maynard, Massachusetts, U.S.A., should produce whatever news is available.

Details of CDC 3600 LISP can be derived from (3.5), or from CO-OP, the CDC equivalent of the IBM SHARE users' association. In Europe, this LISP system is available in Kjeller and Paris.

Atlas LISP has been developed by Dr. D. Russell of the Atlas Computer Laboratory, Chilton, Didcot, Berkshire, and is so far available only on the Chilton Atlas. The Atlas laboratory produces occasional duplicated sheets of information about current developments of its LISP system.

A version of LISP suitable for CDC computers in the 3600 series exists but has not yet been tested. It will be debugged on the CDC 6400 of the Computing Centre, University of Adelaide, Adelaide, South Australia, during 1967.

5. The Scope of LISP

Let us begin an examination of the advantages of LISP by a short comparison with the well-known language FORTRAN. Firstly, FORTRAN programmes only described operations on numbers, whereas LISP allows operations on numbers, letters, algebraic symbols and text. Secondly, FORTRAN programmmes have rigid requirements of storage, which means effectively that any variable in a programme, or any member of a dimensioned array can only stand for a single number. By contrast, a variable or a member of an array in a LISP programme can be made to stand not only for a single number,

like 2.7, but for a list of numbers, like (2.7 4.5 9 288.31), a letter like A, a word or combination of letters like LISP, a list of these entities in any order (e.g. (A LISP 2.7 4.5 9 288.31)), a list of lists, like (A (LISP 2.7) 4.5 ((9)) 288.31), and so on. Therefore quite general operations may be programmed in LISP, and the conventional arithmetic operations make up a subset of LISP.

In theoretical physics, LISP programmes have been written to take traces of Dirac gamma matrices and their matrix products with four-vectors, producing as output the algebraic result of the trace operation, and to perform the analytic procedures of integration both over internal four-momenta in matrix elements derived from Feynman diagrams and over phase space in the final state. Such programmes have derived results which are sums of up to 35000 algebraic terms in 20 minutes of computing time on an IBM 7090. Other programmes have generated Feynman diagrams, specified the asymptotic behaviour of scattering amplitudes at high energy, and reproduced the calculation for the magnetic moment of the electron, correct to fourth order in perturbation theory.

There are two other advantages of LISP in comparison with FORTRAN. It is obvious to any user where a FORTRAN programme stops and its data begins, because the rules for writing FORTRAN statements are laid down in the programming manuals, while the (different) rules for reading and writing data are in effect contained in the FORMAT statements of the programme. By contrast, units of LISP programmes and data are both written in the same basic form - the S-expression. (We shall examine the definition of 'S-expression' below.) It follows that programmes can be treated as data, and in particular they can debug and correct each other to a limited extent.

The final advantage of LISP programmes over FORTRAN, since they can operate on combinations of letters, numbers and text, is that they can write other LISP programmes - or even FORTRAN programmes. In practice, it is quite easy to design LISP programmes to do this.

In conclusion, a disadvantage. For large quantities of arithmetic, LISP is significantly slower than FORTRAN. Therefore, if a calculation is primarily numerical, as much of it as possible should be programmed in FORTRAN.

6. Definitions of the Basic Entities in LISP

The most basic object that can occur in a LISP programme (or data) is an atom.

An atom is either a number (fixed-point, floating-point or octal, i.e. any number permissible in a FORTRAN programme, except that the first character of the number must not be the decimal point) smaller in magnitude than 2^{128}, or a string of letters, certain special characters and decimal integers not separated by blank spaces (provided that the first character in the string is a letter, and that the total length of the string does not exceed 29 characters in IBM 7090 LISP or 82 characters in CDC 3600 LISP). Some possible atoms are:

0 1 1.Ø 3257Q (this is an octal number, equal to 1711)
2.986E+12 A AB. A2B A EXTRALONGSTRINGOF29CHARACTERS
SIMP+6 SIMP+6++-

LISP programmes always treat these atoms as a whole, and do not split them up into their component characters.

The next object that we meet is the dotted pair. This is most simply a pair of atoms which is separated by a dot (which is the same thing as a decimal point) and enclosed by a pair of brackets. A trivial dotted pair is:

(A· B)

and another possibility is:

(2.4 ·Ø.3).

Now we can see why a number cannot begin with a decimal point. Also, to avoid ambiguity, it is always good to leave a blank space between the dot and each of the elements of a dotted pair - otherwise we run the risk of writing nonsense like (2.4.Ø.3).

Next, a most important statement: all atoms are S-expressions.

This starting point is sufficient for us to make a complete definition of 'S-expression'. Any S-expression consists of the following:

A left-hand bracket	'('
An S-expression	
Optional blank spaces	
A dot	
Optional blank spaces	
An S-expression	
A right-hand bracket	')'

in that order.

Notice that 'S-expression' is defined in terms of itself; the definition still makes sense, though, because we know that an atom is an S-expression. We call this property of 'definition in terms of itself' recursion. Later we shall use recursion extensively to define functions in terms of themselves.

Obviously (A . B) obeys all the demands of the S-expression definition, so that it must be an S-expression. Therefore all dotted pairs are S-expressions. Therefore, using the definition again, we find that ((A . B) . (C . D)) is an S-expression. So is ((AB .(CD..EF)).. (GH . IJK2)). In this way, we can build up S-expressions of arbitrary complexity.

7. The Atom NIL, and Equivalence between 'Dot' Notation and 'List' Notation

The greatest difficulty in the writing of long S-expression in the 'dot' notation above is that it involves much hard work to get all of the dots and brackets in the right places. It seems reasonable to expect some complication with brackets in long lists, like:

((1 1)(2 1.414)(3 1.732)(4 2) etc)

(whose meaning should be evident!) but the dots merely provide unnecessary detail. Therefore, in LISP, the programmer may use an alternative 'list' notation and forget most of the dots.

There is a special atom named NIL in the LISP system, to assist in the establishment of an equivalence between list notation and dot notation. Let the lower-case letters x and y stand for any S-expressions. Then we have two important LISP identities:

(x. NIL) = (x)
(x. (y)) = (x y).

By repeated use of these identities, we can write (for example):

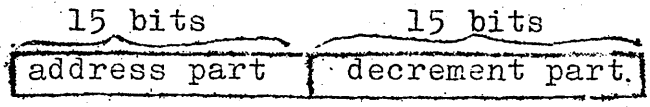
(A B C D) = (A. (B. (C. (D. NIL))))
((A B) (C D)) = ((A. (B. NIL)). ((C. (D. NIL)) . NIL)) .

However, there is no guarantee that all dots can be removed with the help of the identities. As an example ((A. B). ((C. D). NIL)) can be reduced only to ((A. B)(C. D)). Therefore it must be possible to mix the use of list and dot notations in any S-expression.

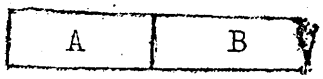
Although we need to write a minimum of dots when we use list notation, we must nevertheless remember the equivalences with the dot notation, in order to understand how the most basic functions in the LISP system obtain their results.

8. The Functions CAR, CDR and CONS, and the Computer Representation of List Structure

The word-length in the core memory of the IBM 7090 is 36 bits. The LISP system uses the positions of 6 of these bits for special purposes (whose significance is outside the range of this discussion), leaving 30 bits per word for storage of information. We divide the word into two 15-bit parts, the 'address' part and the 'decrement' part:



Because the IBM 7090 has three special 15-bit registers, we can look at the two parts of a word separately, and use the two parts to store separate pieces of information. Recall the dotted pair (A. B). We store this in an obvious manner, by putting A in the address part of one computer word and B in the decrement part of the same word:

(A. B) = 

On a very basic level, then, we need LISP functions which operate on a compound structure like this and give us back the simpler parts of the structure, e.g. A or B.

These two functions are:

CAR ('contents of the address register')
CDR ('contents of the decrement register')

In the simplest case, the CAR of (A. B) is A, and the CDR of (A. B) is B. More generally, when x and y again stand for any S-expression, we find that

CAR of (x) = x
CAR of (x y) = x
but CDR of (x y) is not y, it is (y)
and likewise CDR of (x) = NIL

These results are easy to understand if we remember the equivalences between dot and list notation from section 5.

How are structures more complicated than (A. B) represented in the computer ? For example, the list (A B) ?

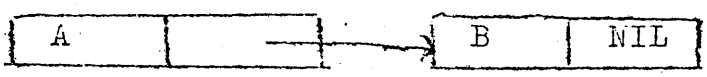
To do this, we must consider the manner in which the core memory of the computer is organised. Each word of the memory is an unique location, and this location is specified by an octal number between Ø (ØQ) and 32767 (77777Q) in the IBM 7090. Suppose that the parts of (A B) = (A. (B. NIL)) are stored in the computer like this:

LOCATION	WORD	
45332Q	A	???
47214Q	B	NIL

It is necessary to associate the two words in some way in order to set up the list (A B). This is done by replacing '???' in the decrement part of the word at location 45332Q by a pointer to location 47214Q. The LISP convention is that the pointer to any location L is the octal number equal to the difference of 1ØØØØØQ and L. Therefore, in the present case, '???' must be replaced by the octal number 30564 (=1ØØØØØQ- 47214Q).

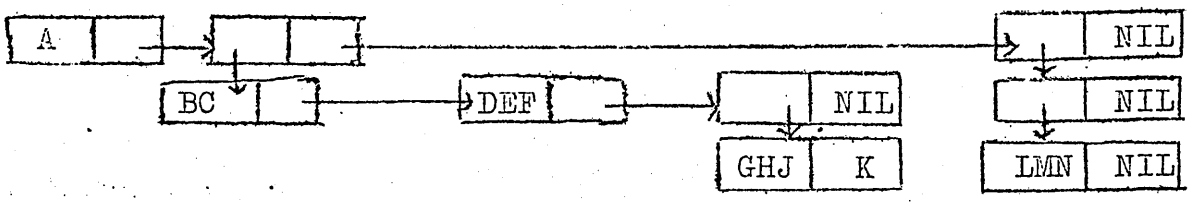
Now here is an important piece of information which makes life easy for the programmer. We need not worry about the actual storage locations of information, or about the need to calculate the pointer explicitly. This is done automatically by the LISP system, which inserts new information, as it is calculated or read in, into the first convenient unused word which it finds in the core memory. Because of the concept of the pointer, LISP (unlike FORTRAN) is not limited to a sequential type of storage or the storage of information which has a fixed length.

In view of what has been said above, we can represent (A B) in a simple diagrammatic form:



Both the address part and the decrement part of any word used to store information in LISP can contain either information or a pointer to another location where information is stored. The functions CAR and CDR, as their names imply, each detect the contents of a half-word in the computer but, if what they find in the half-word is a pointer, they follow up the pointer to the place where information is stored, and return the information as their value.

Let us examine this in more detail by looking at a complicated example - (A (BC DEF (GHJ . K))((LMN))). From the principles that we have learned already, it is evident that this structure can be represented diagrammatically as:



It is also evident that repeated applications of the functions CAR and CDR are necessary to recover most of the atoms in this list structure. Below, we write 'equations' in which the operations that occur to the left of any = sign produce the results on the right-hand side when they are applied to the structure in the last diagram:

CAR = A
CAR of CDR = (BC DEF (GHJ . K))
CAR of CAR of CDR = BC
CAR of CDR of CAR of CDR = DEF

This tends to become rather tedious, so the LISP system contains as names of functions any combination of As and Ds between the 'C' and the 'R', provided that that combination has a maximum length of four characters. Therefore we can write CAADR = BC and CADADR = DEF. Let us now complete the analysis of the structure in the diagram.

CAR of CDDADR = (GHJ. K)
CAAR of CDDADR = GHJ
CDAR of CDDADR = K
CDDR = (((LMN)))
CDDDR = NIL
CDDDDR = undefined result (error condition in programme)
CADDR = ((LMN))
CAADDR = (LMN)
CDADDR = NIL
CAR of CAADDR = LMN
CDR of CAADDR = NIL

CAR, CDR and their more complicated compounds are all functions of one argument, an S-expression which is to be analysed or split up in some way.

The inverse of this type of operation is performed by a function CONS, which takes two arguments (both S-expressions) and makes them into a new S-expression by creating a dotted pair.

For example, CONS applied to A and B as arguments gives the result (A. B), CONS applied to (A B) and (C (D. E)) gives ((A B) . (C (D. E))) = ((A B) C (D. E)) and so on.

We have now seen the uses of the functions CONS, CAR, CDR, CAAR, CADR, CDAR, CDDR, CAAAR, CAADR, CADAR, CADDR, CDAAR, CDADR, CDDAR, CDDDR and the 16 functions whose names take the form 'C(4 letters, each being either A or D)R'. All of these functions exist in the LISP system, and are ready for use by the programmer.

9. Prefix Notation

Suppose that we wish to carry out a simple operation, such as the addition of 2, 3 and 4. We write ? + 3 + 4 or perhaps 2 PLUS 3 PLUS 4. The operator is located between pairs of operands. This is characteristic of infix notation, which we use in ordinary numerical manipulations and in FORTRAN. In LISP, however, we always use prefix notation, in which the operator occurs once, to the left of all its arguments. Thus 2 + 3 + 4 becomes + 2 3 4 or PLUS 2 3 4, and a more complicated example like, say, 2 + (3 x 4) + (5 x 6 x 7), becomes PLUS 2 (TIMES 3 4)(TIMES 5 6 7).

This usage is consistent with our conventional notation for functions, like F(X,Y) or G(X,Y,Z), where the function-name occurs once, to the left of all its arguments.

Neither PLUS 2 3 4 nor G(X,Y,Z) is a valid S-expression. Can we find a consistent prefix-notation method of writing both as S-expressions ? The answer is simple. PLUS 2 3 4 becomes (PLUS 2 3 4), and G(X,Y,Z) becomes (G X Y Z).

10. How to Write Simple Definitions of Functions

The greatest part of the specification of any LISP programme is taken up with definitions (written as S-expressions) of functions which the programmer designs to perform various tasks, and to give various values.

Imagine that we want to define a LISP function to which we give the name NIENTE. The function takes two arguments, but its value is always NIL. The S-expression definition which we punch on an IBM card for inclusion into a LISP programme is:

(NIENTE (LAMBDA (U V) NIL))

i.e. a list which has the two elements on the top level. The first element is the LISP atom which is the name of the function, and the second is a specification of the definition of the function. This second element is itself quite complicated: we always find that it has three elements on its own top level.

The first of these elements is the atom LAMBDA. (This is a LISP convention, which is justified in the reference (1.1). However, to begin writing simple programmes, we do not need to know the justification).

The second element is a list of the (dummy) arguments for the function. (By 'dummy', we mean that the programme can later substitute any S-expression in their place.) The number of elements of the list must be equal to the number of arguments for the function. If the function has no arguments, the list becomes ( ) or simply NIL, since the LISP system regards ( ) and NIL as being equivalent.

The third element, which is usually the longest and most complicated S-expression, is either the value of the function or an S-expression which, on evaluation by the LISP system, reduces to a final value for the function.

How do we make the definition of NIENTE available to the LISP system ? By itself, the card containing the S-expression (NIENTE (LAMBDA (U V) NIL)) is insufficient. We must precede it in the LISP programme by a card containing

DEFINE ((

and follow it by

))

on another card. In fact, any number of complete function definitions can be inserted between any one pair of these key cards, to make up a valid statement of definition which is part of a LISP programme. The effect of the statement is to add the function-definitions to the basic LISP system.

In particular, after NIENTE has been defined, any subsequent card in the programme which contains

NIENTE (x y)

for any two S-expressions x and y will cause the LISP system to return the value NIL and print it in the output. An example of this is presented in section 17.

11. The Use of QUOTE, and some Special Atoms

Consider the atoms U and V in the definition of NIENTE in section 10. Obviously they cannot stand for themselves in a programme, because then a card containing NIENTE(A B) would cause the LISP system to return a meaningless or undefined result. They must be dummy variables, for which actual S-expressions used in calculation may later be substituted.

As an example, let us define a function of these arguments which makes use of CONS.

```
DEFINE((   (JOIN (LAMBDA (U V W)(CONS U (CONS V W)) ))      ))
```

The result of JOIN(A B C) is (A. (B. C)), JOIN (A B NIL) gives (A. (B. NIL)) = (A B), JOIN(ABC (DEF GHJ)(KLM)) = (ABC (DEF GHJ) KLM), and so on. Suppose that we now want to use JOIN to build up many three-element lists which have the common property that their last element is always KLM. We should be able to do this with a new function JOIN2:

(JOIN2 (LAMBDA (X Y)(JOIN X Y (KLM))))

But this is wrong ! The LISP system examines X and Y in the definition in order to replace them by the current arguments of the function JOIN2, before performing the functional operation JOIN on these arguments, so it will try to do the same thing with the S-expression (KLM). We must have some means of distinguishing between dummy variables, which are to be evaluated (i.e. replaced by the S-expressions for which they stand), and S-expressions which stand for themselves.

The distinction is made possible by the 'function' QUOTE, which has one argument. We may understand its behaviour from the definition (QUOTE (LAMBDA (X) X)). The correct structure of JOIN2 is:

(JOIN2 (LAMBDA (X Y)(JOIN X Y (QUOTE (KLM)))))

Now you may object that the definition of NIENTE in section 10 is wrong - NIL in that definition should actually be written (QUOTE NIL). The form (QUOTE NIL) is certainly correct, but there are three non-numerical atoms in the LISP system that do not need to be QUOTED, because they have special properties which make it illegal for them to be used as dummy variables. These three atoms are:

T, F and NIL

T stands for 'true' and F for 'False'.

All numbers share with T, F and N. the qualification that they never need to be preceded by QUOTE: they always stand for themselves.

12. Predicate Functions and COND

A predicate is a function which can take only two possible values, T or F, according to whether the proposition which it expresses is true or false. For example, NULL is a commonly-used predicate function of one argument. If this argument has the current value of NIL, the value of the function is T; if the argument has any other S-expression as its current value, the result is F.

Now we make an important definition, which is not part of the LISP programming language, but which we use frequently for clarity in later sections of this Introduction. Let r(...) be an abbreviation for 'the S-expression which is the current value of ... , i.e. the S-expression which the LISP system causes to be substituted for ... in a programme'.

Our first use of this definition occurs below, in a description of some of the other useful predicate functions that are part of the basis LISP system.

(ATOM X)	:True if r(X) is an atom, false otherwise
(NOT X)	:True if r(X) is the atom NIL or the atom F, false otherwise

(EQUAL X Y) :True if r(X) and r(Y) are identical S-expressions, false otherwise

(EQ X Y) :A faster version of EQUAL which, however, it is only safe to use in situations where it is known that r(X) and r(Y) are both atoms

(NUMBERP X) :True if r(X) is a number, false otherwise

(MEMBER X Y) :True if r(X) is a member of r(Y) (which should be a list) on the top level of r(Y), false otherwise.

The three following predicates all generate error conditions if r(X) is not a numerical atom, and should therefore be used with care.

(ZEROP X) :True if r(X) is zero, false if r(X) is any other number

(ONEP X) :True if r(X) = 1, 1.Ø, 1.ØE+ØØ, 1.ØE-ØØ or 1Q, false if r(X) is any other number

(MINUSP X) :True if r(X) is a negative number (including any representation of zero which begins with a minus sign), false if r(X) is any other number.

The function COND provides a fast means of testing up to 20 predicates in order to find one which has the value T. This function can have any number of arguments between 1 and 20, and each argument is a list which is made up of two S-expressions, so that it has the general form (P E).r(P) is almost always T, F or NIL but, if r(P) is any S-expression other than F or NIL, its effect on the function COND is the same as if r(P) = T.

The value of:

(COND (P1 E1) (P2 E2) (P3 E3) (P4 E4) .....)

is the value of the ALGOL-like expression 'if r(P) is T then r(E1), else if r(P2) is T then r(E2), else if r(P3) is T then r(E3), else if r(P4) is T then r(E4) ..... '. Note that, in general, if no r(P) in the COND is effectively T, an error condition ('error A3') will occur; for the only exception to this rule see section 17. Therefore it is wise to make the last r(P) of the COND expression equal to T, which is most simply done if we put T in place of P itself.

In the evaluation of the function COND, the LISP system scans the arguments of COND from left to right until it finds some r(P) that is effectively T. The value of the entire function COND is then the value of the corresponding r(E). As an example, let us define in LISP the step-function, which has one argument and which is defined as the integral from minus infinity up to the value of that argument for the one-dimensional Dirac delta function. This step function is zero if the argument is negative, Ø.5 by convention when the argument is zero, and 1 when the argument is positive. If we give the function the name STEP in LISP, we can combine some of the results of sections 8 and 10 to write:

```
(STEP (LAMBDA (N) (COND ((ZEROP N) Ø.5)
                        ((MINUSP N) Ø)
                        (T 1    )))
```

Probably about half of the functions used in any programme are defined by consideration of a finite number of cases satisfied by their arguments. Therefore COND is of great use in the definition of functions. We shall see our first detailed example in section 13, and others in section 17.

13. Recursive Definitions of Functions

In sections 11 and 12 we have met examples of how a programmer can build up definitions of complicated functions by using both the simpler functions defined in the LISP system and simpler functions that he has previously defined for himself. However, it is often true that the best definition of a function is a definition (by induction) of the function in terms of the function itself (e.g. the factorial function for positive integers).

Consider one such function: PAIR, whose two arguments are lists with the same number of elements, and whose value is a list in which corresponding elements from the two arguments are made into dotted pairs. For illustration, PAIR acting on (A1 A2 A3) and (B1 B2 B3) gives ((A1. B1) (A2. B2) (A3. B3)). The appropriate definition is:

```
(PAIR (LAMBDA (X Y) (COND ((NULL X) NIL)
      (T (CONS (CONS (CAR X) (CAR Y)) (PAIR (CDR X)(CDR Y)) )) )))
```

Firstly, it is strongly recommended that any intending LISP programmer should spend some time analysing on paper the manner in which this function works, for the arguments (A1 A2 A3) and (B1 B2 B3). The structure of the definition of PAIR is a structure that is highly characteristic in LISP, and it will be encountered many times. If a simpler example is needed to demonstrate only why the first of the arguments of COND above is ((NULL X) NIL), then consider the function REMAKE, which acts on its one (list) argument like the unit operator:

```
(REMAKE (LAMBDA (X) (COND ((NULL X) NIL)
        (T(CONS (CAR X) (REMAKE (CDR X)) )) )))
```

Secondly, the LISP system accepts all recursive definitions of functions, and it handles computations with these recursive functions very easily by making automatic use of an internal device that is called push-down list.

14. The Function ERROR

The functions STEP in section 12 and PAIR in section 13 will work correctly when they are given arguments of the correct form, but an error in computation will occur if the form is unacceptable - e.g. a non-numerical argument for STEP or two lists of unequal lengths for PAIR. But the question of acceptability of the argument(s) is merely a question that can be answered if an extra argument to test for acceptability is given to COND in the definition of the original function. If it is the case that the form of the argument(s) is wrong, we may make use of the function ERROR, which has one argument, to bring the computation to an orderly halt. The choice of this one argument is completely open to us, and we use it to gain the greatest possible amount of information about the source of the error.

In particular, if PAIR works correctly, r(X) should become NIL exactly when r(Y) becomes NIL, and not at any other point of the computation. Therefore we can amend the definition in section 11 to:

```
(PAIR (LAMBDA (X Y) (COND ((NULL X) (COND ((NULL Y) NIL)
          (T(ERROR (QUOTE PAIR-IS-WRONG)))  ))
   (T (CONS (CONS (CAR X) (CAR Y)) (PAIR (CDR X)) (CDRY)) )) )))
```

The modification of the definition of STEP, to cause termination if the function is given a non-numerical argument, is left until section 17.

15. The Functions LIST and APPEND

LIST provides a quick means of bypassing excessive use of CONS and NIL when we are making up a list from a variable number of X-expressions (between 1 and 20). Some simple equivalences are:

```
   (LIST X) = (CONS X NIL)
 (LIST X Y) = (CONS X (CONS Y NIL)
(LIST X Y Z) = (CONS X (CONS Y (CONS Z NIL)))
```

Therefore LIST is a function of between 1 and 20 arguments, and its value is a list of those arguments.

APPEND is a function of two arguments, that provides the means for joining two lists into a single list without multiple uses of CONS, CAR and CDR. For example, APPEND applied to the arguments (A (BC DE) FG) and ((HIJ KLM) NOP (QRS. TUV)) gives the result (A (BC DE) FG (HIJ KLM) NOP (QRS. TUV)).

It is useful also to bear in mind that APPEND (y NIL) = APPEND (NIL y) = y.

16. Constants

Until now, we have only seen functions that generate results and cause them to be printed in the output from the computer. Once these results are printed, they are not accessible to subsequent steps in a programme. We may therefore ask: 'What is the procedure for putting a result somewhere in storage from which it can be recovered for later use?'

The answer is that we can assign an unique name (a LISP atom) to the result, just as each function-definition that we make is referenced by an unique name which is the name of that function. We may later refer to this name and thereby recover the result.

For function definitions, the assignment is performed by the function DEFINE, which we have seen already in section 10. The analogous function for storing results (and for storing many items of input information that are not function-definitions), and which makes the name of any result into a constant of the system, is CSET. In some contexts we use CSETQ, another function which has the same effect.

Practical examples of the uses of CSET and CSETQ will be given in section 17.

17. Example - Part of a LISP Programme

A LISP programme is not organised sequentially, with a well-defined beginning and end (which may be observed in any numerical programming language like FORTRAN). It consists first of a collection of programmer-defined functions to do certain parts of a given job, and then a command to the LISP system to use those functions to do the job. The functions call each other, and there is one function which initiates the entire sequence of calls that occurs during the performance of the job. The command that is mentioned two sentences above consists of the occurrence of the name of that function on one card of the programme, together with a list of the function's arguments.

As an example, we write a set of functions to perform the algebraic expansion of $(1+x)^n$ for arbitrary x and (positive integer) n. EXPAND is the initiating function, and it calls EXPAND2, which calls itself and BINOM, a function that determines binomial coefficients and calls the function FAC, the factorial function, in order to do so. We also cause the functions STEP and NIENTE (which we have seen before) to be defined at the same time, for convenience, because this does not interfere in any way with the set of functions associated with EXPAND.

To do this, we punch on IBM cards:

```
DEFINE((
          (NIENTE (LAMBDA (U V) NIL))
    (STEP (LAMBDA (Y) (COND ((NOT (NUMBERP Y)) (ERROR (LIST
                                (QUOTE STEP) Y)))
      ((ZEROP Y) 5) (((MINUSP Y) 0) (T 1) )))
  (EXPAND (LAMBDA (X N) (LIST (QUOTE PLUS) 1 (EXPAND2 X N 1) )))
(EXPAND2 (LAMBDA (FORM N NA) (COND
             ((EQ N NA) CONS (LIST (QUOTE EXPT) FORM N) NIL))
   (T (CONS (LIST (QUOTE TIMES) (BINOM N NA)
      (COND ((ONEP NA) FORM) (T (LIST (QUOTE EXPT) FORM NA))))
                (EXPAND2 FORM N (ADD1 NA)) ))  )))
(BINOM (LAMBDA (A B) (QUOTIENT (FAC A) (TIMES (FAC B)
                          (FAC (DIFFERENCE A B))) )))
(FAC (LAMBDA (N) (COND ((ZEROP N) 1) (T (TIMES N (FAC (SUB1 N))))
                    ))) ))
```

By way of additional explanation, (ADD1 X) is a function whose value is the sum of 1 and r(X), and SUB1 X) subtracts 1 from r(X).

Now suppose that we want to apply NIENTE to the arguments 1 and FRED, STEP to 52, STEP to FRED, EXPAND to (TIMES ALPHA X) and 4, FAC to 4 and BINOM to 9 and 5. We punch these cards:

```
NIENTE(1 FRED)
STEP(52)
STEP(FRED)
EXPAND( (TIMES ALPHA X)  4)
FAC(4)
BINOM(9 5)
```

and add them to our card-deck after the function definitions.

When all the cards are run on the computer as part of a LISP job, the printed output looks approximately like this:

```
FUNCTION ...
 DEFINE
ARG(S)...
      (( (NIENTE (LAMBDA (U V) NIL)) (STEP (LAMBDA (Y)
                and so on, down to
                                  (FAC (SUB1 N)))) ))) ))

VALUE OF RESULT IS ..
 (NIENTE STEP EXPAND EXPAND2 BINOM FAC)

FUNCTION...
 NIENTE
ARG(S)...
 (1 FRED)
VALUE OF RESULT IS..
 NIL

FUNCTION...
 STEP
ARG(S)...
 (52)
VALUE OF RESULT IS..
 1

FUNCTION...
 STEP
ARG(S)...
 (FRED)
------ERROR NUMBER A 1 -----
(STEP FRED)
(STEP)

FUNCTION...
 EXPAND
ARG(S)...
 ((TIMES ALPHA X) 4)
VALUE OF RESULT IS..
 (PLUS 1 (TIMES 4 (TIMES ALPHA X)) (TIMES 6 (EXPT (TIMES ALPHA X)
2)) (TIMES 4 (EXPT (TIMES ALPHA X) 3)) (EXPT (TIMES ALPHA X) 4))

FUNCTION...
FAC
ARG(S)...
 (4)
VALUE OF RESULT IS..
 24

FUNCTION...
 BINOM
ARG(S)...
 (9 5)
VALUE OF RESULT IS..
 126
```

In the programme above, the result of the binomial expansion for EXPAND is printed and then lost from the core memory of the computer. However, if we had wished to preserve it for future use, we could have decided on the name EXPANSION for it, and replaced the card with the command EXPAND ((TIMES ALPHA X) 4) by:

(LAMBDA (A B) (CSET (QUOTE EXPANSION)(EXPAND A B))) ((TIMES
ALPHA X) 4)
or(LAMBDA (A B) (CSETQ EXPANSION (EXPAND A B))) ((TIMES ALPHA X) 4)

The equivalence of these last two commands indicates the nature of the relationship between the functions CSET and CSETQ.

18. The Function EVAL

We have previously used the phrase 'the LISP system' quite freely. The system is a collection of basic functions (like CAR, CDR, etc) written in machine code (FAP, in the case of the IBM 7090, ABL in the case of the Atlas), but there are several key functions in the system whose task it is to interpret automatically all of the operations and commands that occur in any programme on cards. The principal function, which is named EVALQUOTE, is discussed in detail in the reference (3.1). EVALQUOTE makes use of EVAL, a function that is explicitly available for the purposes of the programmer.

EVAL is a function of two arguments. For (EVAL X A), r(X) is the quantity to be evaluated, and r(A) is the association-list or a-list, which is also described in detail in (3.1). However, for many practical uses of EVAL, we can replace A or r(A) by NIL. Let us look at a case where EVAL is needed. Consider a part of a programme made up of the cards

```
 CSET(TRICKLIST  (PLUS 2 3 4))
DEFINE((
 (TESTFN (LAMBDA (N) (COND ((ONEP N) TRICKLIST)
    (T (EVAL TRICKLIST NIL)) )))    ))
TESTFN(1)
TESTFN(2)
EVAL(TRICKLIST NIL)
```

The last three steps in the printed output from this programme will be:

```
FUNCTION...
 TESTFN
ARG(S)..
 (1)
VALUE OF RESULT IS ..
 (PLUS 2 3 4)

FUNCTION...
 TESTFN
ARG(S)...
 (2)
VALUE OF RESULT IS..
 9

FUNCTION...
 EVAL
ARG(S)...
 (TRICKLIST NIL)
VALUE OF RESULT IS ..
 (PLUS 2 3 4)
```

This example demonstrates the most important basic uses of CSET and EVAL.

19. The PROG Feature, and the Functions GO, SETQ and RETURN

Although a complete LISP programme is not organised like a complete FORTRAN programme, there are occasions on which it may be desirable to write single LISP functions that are organised like a FORTRAN programme. An example of this method which uses the so-called PROG feature, is the function LENGTH, which has as its value the number of S-expressions on the top level of its argument if this argument is a list, and is zero if the argument is NIL (and is meaningless otherwise!). We write:

```
(LENGTH (LAMBDA (N)  (PROG (A B)
        (SETQ A Ø)
        (SETQ B N)
  GX    (COND ((NULL B)(RETURN A)))
        (SETQ A (ADD1 A))
        (SETQ B (CDR B))
        (GO GX)    )))
```

The four new functional operations which we find here are PROG, GO, SETQ and RETURN.

PROG occurs always in the position shown, in a function-definition which make use of the PROG feature. It is a special conventional LISP marker like LAMBDA, and must be followed immediately by a list of programme variables. If there are no programme variables, it should be followed by ( ) or NIL. (This is unlikely in the case of FORTRAN , in which programme variables do not have to be declared as such).

GO is effectively a transfer statement. It can be understood immediately in terms of FORTRAN, except that the labels of statements in a PROG feature in LISP are LISP atoms, not numbers.

SETQ is analogous to CSETQ, except that its effect is confined to the PROG feature within which it occurs. The function-name SETQ is an abbreviation of 'set and quote'. (SETQ U V) has the effect of establishing that the name U (not r(U)!) is associated temporarily with r(V). In the PROG feature for the function LENGTH, (SETQ A Ø) is the LISP equivalent of the FORTRAN statement A = Ø, and (SETQ A (ADD1 A)) is equivalent to A = A +1.Ø.

RETURN causes computation within the PROG feature to end. (RETURN V) ensures that r(V) is the value of that computation. RETURN may occur any number of times within the one PROG.

If a PROG is not caught up in a non-terminating loop, if its last statement is reached and that statement does not contain RETURN, the value of the computation determined by that PROG will always be NIL.

If no r(P) within the general form of a function COND (see section 12) on the top level inside a PROG is effectively equal to T, an error condition will not occur. The computation will proceed normally to the next statement of the PROG.

Functions can also be defined recursively with PROG. An artificial but correct example of this is an alternative definition of LENGTH:

```
(LENGTH (LAMBDA (N) (LGTH N Ø)))
(LGTH   (LAMBDA (N M) (PROG (A)
              (COND ((NULL N) (RETURN M)))
              (SETQ A (ADD1 M))
              (RETURN (LGTH (CDR N) A)) )))
```

Beware of a common mistake, though - if RETURN is omitted and the last line of LGTH is simply (LGTH (CDR N) A), the result will always be NIL.

20. Preparation of the Card Deck for an IBM 7090 LISP programme

We have already seen some examples of the contents of cards which make up part of a LISP programme. The only important additional item of information about these cards is that their contents can be punched anywhere in columns 1 through 72 of each 80-column card, and they can overflow from any one card to the next. No continuation markers are necessary.

Let us now look at the components of the card deck for a LISP job.

20.1 - the deck is headed by the Imperial College ID card. The correct entry for the section beginning at column 31 is FMS, since LISP uses the FMS monitor. LISP jobs will not run under the IBSYS monitor.

20.2 - A comment card comes next, with an asterisk (*) in column 1, and a message to the computer operators to mount a current LISP system tape (you have the choice of X51 or X247, but please specify one of these), to be saved and file-protected, on tape drive B7. The message can be punched in columns 7 through 72.

20.3 - the LISP loader, a short binary programme. Copies of the loader may be found in the 'LISP' library drawer in Room 405 (and may be taken away for permanent use). Contents of the cards of this loader are as follows:-

Card 1

* in column 1, and the word PAUSE beginning in column 7.

Card 2

* in column 1, and the word XEQ beginning in column 7.

Card 3

12, 7,9 punches in column 1
1 punch in column 2
12, 7,9 punches in column 4
1 punch in column 5
7,9 punches in column 6
7,9 punches in column 9

Card 4

11, 7, 9 punches in column 1
1, 3 punches in column 2
0, 1, 2, 3, 6, 8 9 punches in column 4
0 punch in column 5
11, 2, 5, 6, 7, 8 punches in column 6
1 punch in column 7
1, 2, 3, 4, 5, 8 punches in column 13
11, 2, 5, 7, 8, 9 punches in column 15
12, 1, 3, 4 punches in column 16
7 punch in column 18
12, 1, 3, 4, 7 punches in column 19
5 punch in column 22
9 punch in column 24
12, 0 punches in column 25
2, 3 punches in column 26

Card 5

* in column 1, and word DATA beginning in column 7

Card 6

7, 9 punches in column 1
2, 3, 4, 5, 6, 8, 9 punches in column 3

Card 7

Blank card (optional but recommended)

20.4 - The LISP programme itself

20.5 - A card containing FIN in columns 8 through 10, and blank elsewhere

20.6 - An end-of-file card - punches in rows 7 and 8 of columns 1 and 80, and blank elsewhere

The programme in (20.4) may be divided into any number of parts (including one). We may choose to allow these parts to communicate with each other, or to be self-contained. Each part must be headed by a special control card which is not written in LISP, and concluded by the statement STOP))))) on the last card. The word STOP should be followed by several right-hand brackets, although the actual number of these brackets is unimportant.

Each such part of a LISP programme is called a packet.

It is important for the discussion below to note that, as soon as the LISP loader is processed by the computer (and before the processing of any packets), the machine puts a copy of the system tape's contents (from tape drive B7) onto a tape which (for the FMS monitor) is mounted on tape drive B3 in the case of Oxford LISP 1.6.

The control card at the head of each packet should have a control word beginning in column 8, and whatever comments you like (including no comments at all) in columns 16 through 72.

For basic programming, only three control words should be known.

A packet controlled by the word SET will be processed conventionally, but, if it is free from computational errors, the results of all new function-definitions (via DEFINE) and establishment of constants (via CSET or CSETQ) will be added to the basic LISP system. At the conclusion of the packet, the computer copies this 'updated' LISP system onto the tape on the drive B3, and uses that tape thereafter as the source of the system for all further packets. However, if an error occurs within the packet, nothing is copied onto B3.

A packet controlled by the word SETSET behaves in all respects like a SET packet, except that, if errors occur within it, an updated system (complete with the effects of the errors) is copied onto B3. Therefore SETSET should be used with care (or low cunning), or it will affect the behaviour of all subsequent packets adversely.

A packet controlled by the word TEST can make use of definitions and constants established in all previous SETSET or error-free SET packets. It is self-contained because, even if it is free from errors, its definitions and constants are lost at the end of the packet and not copied onto B3.

Finally, if we wish to preserve all the successful results of SETSET and SET packets from any one job on tape (rather than on cards) for use with other jobs at later times, it is evident that all we have to do is to ask the operators at the computer installation to remove the tape on the drive B3 at the end of the job, and save it. This tape should then be mounted on the drive B7 for the later jobs, for which it becomes the LISP system tape.

The job request slip for an ordinary LISP job should specify the use of the FMS monitor, and indicate that either tape X51 or tape X247 - whichever you prefer - should be placed (file-protected and saved) on the tape drive B7. Additionally, in connection with the paragraph above, if the tape on B3 is to be saved at the end of a job, presumably it will be a numbered tape which the programmer has previously asked the tape librarian to reserve. In that case, the job slip should also carry the request that this tape be mounted (saved but not file-protected) on the tape drive B3.

If the tape on B3 is to be unloaded and saved at the end of the job, one method of ensuring that it is not partly over-written before it can be unloaded is the addition of a second trick 'job' to the original card deck following its end-of-file card. This trick job should have a maximum running time of Ø.1 minutes recorded on its ID card, and (in addition to the ID card) should consist of:

i) a comment card advising the operators that the job is merely to allow time to unload B3 and B7
ii) a PAUSE card identical to the PAUSE card of the LISP loader (20.3)
iii) another end-of-file card

There is no need to make any reference to this trick 'job' on the job request slip.

Next, let us consider a short example that illustrates the different uses of SET and TEST packets, by looking at the card deck:

```
        SET    ERROR-FREE EXAMPLE
CSET(EXAMPLE (A B C D E))                STOP))))))))
        SET    EXAMPLE WITH AN ERROR (BRACKET MISSING)
CSET(EXAMPLE (A B C       STOP))))))))
        TEST   THE EFFECT OF THE PREVIOUS 2 PACKETS
(LAMBDA (M) (PLUS M (LENGTH EXAMPLE))) (4)      STOP)))))))
```

In the second SET packet, we have tried to make EXAMPLE the name of a list (A B C) of length 3. However, that packet contains an error, so our efforts are not recorded on the tape on the drive B3. Instead, on B3 we have the earlier result that EXAMPLE is the name of a the list (A B C D E) of length 5. The calculated result in the TEST packet is therefore 4 + 5 = 9.

Finally, a small point about the punching of a card deck. Suppose that we have a function FN2 of one argument, and that the atoms T and F are likely to occur as possible values of the argument. The print-names of T and F are respectively ⌘T⌘ and ⌘F⌘, and the effect of this information is to make the choice of which of the forms we punch dependent on how the function FN2 is called. Below, we have three correct examples (in the first column) and three incorrect examples:

```
    FN2(⌘T⌘)                          FN2(T)
(LAMBDA (X) (FN2 X)) (⌘T⌘)      (LAMBDA (X) (FN2 X)) (T)
(LAMBDA NIL (FN2 T)) ( )        (LAMBDA NIL (FN2 ⌘T⌘)) ( )
```

It is also a point worth remembering that, inside the LISP system, NIL can always replace F or ⌘F⌘ without any undesirable results. Therefore we need never punch F or ⌘F⌘ at all, provided that we always put NIL in their places.

21. Some Common Programming Errors

With the use of the information in the first 20 sections, it should be possible to write quite detailed LISP programmes. Most of the sections which follow (up to section 32) contain information about more advanced concepts and functions which, in any case, are described in the reference (3.1). Therefore now is the time for a practical review of the most common programming errors.

21.1 - By far the most important error is failure to count brackets correctly. In any function or command, in fact in any non-atomic S-expression, the number of left-hand brackets should equal the number of right-hand brackets. In any packet, the refusal to obey this rule means that part or all of the packet will not be processed. It is a peculiar psychological effect that right-hand brackets are omitted between 8 and 10 times as frequently as left-hand brackets. In either case, the lesson is: COUNT YOUR BRACKETS CAREFULLY! This is initially the most difficult part of LISP programming, but it may be some consolation that any LISP programmer can usually count brackets subconsciously (and correctly) after two or three months of experience with the language.

21.2 - Quite frequently some quantity q is written where (QUOTE q) is intended, or needed. Thus the programme tries to evaluate q, and usually produces the error messages A8 or A9. In such a case, q is printed out on the line below the error message in the output, so that the mistake can be identified quickly.

21.3 - A function may be given the wrong number of arguments, which leads to the error messages F2 or F3. The most common cause of the error is that one argument is omitted at the end of a recursive definition of a function, e.g.

```
(FN (LAMBDA (X Y) (COND ((NULL X) Y)
    ((ATOM (CAR X))(CONS (CAR X)(FN (CAR X) Y) ))
    (T (CONS (CAR Y) (FN (CDR X)) )) )))
```

This error may also have a psychological origin - it always seems to happen in the same place as it is displayed in FN.

21.4 - Physicists and other binomial-series expanders may have trouble with peculiarities of the numerical functions (EXPT X Y) and (RECIP X), which are equivalent to X^Y and $1/X$ respectively. For the first argument of EXPT, r(X) must not be negative, otherwise an error message I2 will occur. A similar LISP deficiency, which is harder to identify because it produces no error message directly, is that (RECIP X) is zero if r(X) is a fixed-point or octal number. A cure for this inconvenience is that fixed-point numbers can be turned into floating-point numbers by the following trick:

```
(FLOAT (LAMBDA (N) (PLUS 2.3 N - 2.3)))
```

Octal numbers are susceptible to the same trick.

21.5 - A numerical predicate like ZEROP or ONEP may be given an argument which is not a numerical atom. This produces the error message I3.

21.6 - Suppose that we define a function ABCF as a SET packet, and that the definition is correct, but that some other error occurs in the packet. Then ABCF is not available for use in any subsequent packet. If we try to use it by making reference to the name ABCF, we get the error message A2, and the name of the function is printed out on the line below the error message for rapid identification.

21.7 - CAR, CDR and their compound functions CADR, CDDAAR etc. produce undefined results when they are given atoms as arguments. You may get away with the application of CDR to an atom without generating an error message, but the result will be a long and irrelevant list which will probably stop your computation at some later stage because your own functions expect to be given sensible lists as arguments. On the other hand, CAR applied to an atom generates between Ø.5 and 2 pages of junk, in which ✱T✱ TXI, NIL and left-hand brackets are prominent, followed by an error message GC2. Once seen, the unique type of junk that CAR of an atom produces in IBM 7090 LISP is not forgotten.

22. Aids to Debugging

Once we have overcome the methodological errors (mainly those mentioned in section 21) in a LISP programme, we may find that some sequence of functions is not working correctly because

of faults in the concepts of the function-definitions. We may have particular difficulty with recursive functions, because from the computer output it may not be obvious at what stage of the recursion the trouble occurs.

When this happens, and in any case in a TEST or SET packet which is not known to work correctly, it is a good idea to make the first card of that packet:

TRACECOUNT(Ø) (where Ø is a zero)

Then, if an error occurs, the error message will include a statement about NUMBER OF FUNCTION ENTRANCES, together with a number. Subtract about 450 from that number, and call the result n. If this n is negative, call it zero. Now suppose that we also have an idea of the sequence of functions in which the error must have occurred. Write down a list of the names of these functions, and let it be denoted by m. In the next [illegible] computer job which contains the relevant packet, make the first two cards of the packet

TRACECOUNT (n)
and TRACE(m)

The computer will print out the name of any function in m, together with its arguments, near the point of error, every time that the function is called (recursively or otherwise), and will also state its value.

If, at any later point in the packet, we wish to cease tracing all the functions whose names we can write in a list denoted by mm, we only need to insert the card UNTRACE(mm) at that point.

In a PROG feature, we may require more detailed tracing: it may be necessary to print out the value of every programme variable every time it is set or reset by SETQ. If we denote by mmm a list of names of all the functions for which this facility is desired, we insert the command TRACESET (mmm). The inverse operation, UNTRACESET, is not included in the system.

A final (advanced) aid to debugging is the function ERRORSET, which is described in (3.1). Errors that occur under the control of ERRORSET, if they are the only errors in a SET packet, do not prevent the results of that packet from being added to an 'updated' LISP system tape on the tape drive B3.

23. The Most Common Error Messages and their Interpretation

A1: This occurs whenever the function ERROR is called. ERROR, which is used by the programmer and not automatically by the LISP system, is described in section 12.

A2: An atom has been used as the name of a function, but that atom has not been successfully associated with any function-definition by DEFINE. See (21.6).

A3: Out of all of the arguments of the form (P E) for COND, each r(P) is either F or NIL, and none is T. This fault will not occur, however, for COND used on the tope level within a PROG feature.

A8 or A9: The second most popular cause of these messages is given in (21.2). They occur most commonly, though , while the LISP system is trying to process a packet in which the brackets are not matched correctly (see 21.1). If the bracketing error (type R1, or occasionally F2) is corrected, these errors may also disappear.

F2 or F3: Either some left-hand brackets are missing from a packet, or the cause of the messages is to be found in (21.3).

G1: You have probably tried to divide by zero.

GC2: If this message is preceded by junk in the output, you have probably attempted to take the CAR of an atom. See (21.7). However, if the characteristic junk is missing and there is no obvious source of error, it is most probable that the computation is too large for the core memory of the computer.

I 2: The first argument of the function EXPT is a negative number.

I3: A function which should have a numerical atom as an argument has been given some different type of S-expression.

Ø5: The control card of the packet has been wrongly prepared. Check that the control word begins in column 8, and that no comments are punched to the left of column 16.

R1: Some right-hand brackets are missing from a packet. Below the error message, the whole packet will be printed out, ending with STOP and one or more right-hand brackets. The number of these brackets is the total number of brackets missing from within the packet.

For the interpretation of other error messages, see the reference (3.1).

24. The Functions SUBST and SUBLIS, and a Comment about Standard Tables of Integrals

(SUBST X Y Z) examines r(Z), substitutes r(X) for any occurrence of r(Y) on any level (not merely the top level) therein, and returns as its value the modified value of r(Z). For example,

SUBST(A 1.732 (ABC D (3 1.732) 1.732 3))=(ABC D (3 A) A 3)

r(Y) can be any S-expression, not only an atom.

(SUBLIS U V) is a generalisation of SUBST. r(U) must be of the form ((G1 . H1) (G2 . H2) etc.). SUBLIS then causes r(H1) to be substituted for any occurrence of r(G1) in r(V), r(H2) for any occurrence of r(G2) in r(V), and so on. For example:

SUBLIS(((A.1)(B.2)(C.3)) (A (B C) D (A B))) = (1 (2 3) D (1 2))

SUBST and SUBLIS are obviously two very useful functions, which occur in many places in most large LISP programmes. In particular, in some calculations in theoretical physics that involve extensive use of integrals, SUBST and SUBLIS insert particular values (the limits of integration) in standard forms

for indefinite integrals, a simple means of making up a table of integrals already exists - the function CSET. If we denote by x a list of the form ((A1 B1)(A2 B2) etc.), where A1, A2 ... are integrands and B1, B2 are the corresponding integrals, we can insert:

```
CSET(TABLE x)
```

into our programme and thereby set up a table of integrals that can later be addressed by its name, TABLE.

A simple table of one-dimensional integrals may begin:

```
(((TIMES A (EXPT X N))(TIMES A (RECIP (ADD1 N)) (EXPT X (ADD1 N))))
                                ((COS X)(SIN    etc. ).
```

It has already proved practical to extend this method to four-dimensional integrals, and to build up tables of integrals by LISP functional operations (e.g. integration by parts) on the contents of the original table.

25. The Function MAPLIST - An Introduction to Functional Arguments

Consider a common type of problem in LISP programming. We have a list, say (1 2 3 4), and we wish to perform a computation which will have as its value a list in which all elements of the original list have been subjected to the same operation. If this operation is, say, the addition of 2, our answer will be (3 4 5 6). It is most convenient to have a single function that can map any given operation onto any given list. This function is MAPLIST:

```
(MAPLIST (LAMBDA (X FN) (COND ((NULL X) NIL)
   (T (CONS (FN X)(MAPLIST (CDR X) FN) )). )))
```

Two precautions must be observed in connection with this definition:

25.1 - Note that we have (FN X) in the last line of the COND, and not (FN (CAR X)). The most common programming mistake with MAPLIST is the assumption that the argument of FN is (CAR X). The fact that we have (FN X) explains why (CAR J) occurs in the function ADD2LIST below.

25.2 - MAPLIST is not defined in the LISP system according to the S-expression form above, but it is written in machine code. The definition above is only intended to show roughly how MAPLIST works. Programmers cannot use DEFINE to define their own new functions that take functional argument; such functions will not work!

An example of the use of MAPLIST to add 2 to each element of a list of numbers is given below:

```
(ADD2LIST (LAMBDA (U) (MAPLIST U (FUNCTION (LAMBDA (J)
                        (PLUS 2 (CAR J)) ))) ))
```

It may seem at first that the final part of the definition of ADD2LIST should be (PLUS 2 J). But this is not so; it is an illustration of the mistake (25.1).

The correct specification of the functional argument of MAPLIST must begin with the atom FUNCTION, which the LISP system uses temporarily as the name of the function whose definition is a part of the argument.

Other functions of the LISP system that use functional arguments are MAP, MAPCON, SEARCH and PROP. These are discussed fully in Appendix A of(3.1).

26. Boolean Logic

The three basic Boolean functions are AND, OR and NOT. We have already encountered NOT in section 12. AND and OR also exist in the LISP system. Like PLUS and TIMES, they are allowed to have any number of arguments between 2 and 20. The LISP system evaluates the arguments of AND and OR from left to right.

If each result of the evaluation of arguments for AND is T, the value of AND is T. If any one evaluation gives the result F or NIL, evaluation of all arguments to the right of that argument is not carried out, and the value of AND is F.

OR behaves in the same manner. If all evaluations of arguments give F or NIL, the value of OR is F. However, if any one argument has the value T on evaluation, all arguments to the right of that argument are not processed, and the value of OR is T.

27. The Garbage Collector

In section 8 we have discussed the way in which new information produced by the LISP system during reading operations or computations is inserted into the most convenient unused words in core memory. Eventually, though, this procedure must fill all of the space available for storage. We must have a means of removing information from storage when it is no longer needed. The garbage collector is a function, written in machine code, which does this.

During ordinary operation of the computer, the LISP system uses the garbage collector automatically whenever the storage available for computation is full. However, the programmer may start a garbage collection himself by using the function RECLAIM, which is a function of no arguments. Because of this latter fact, we put it onto an IBM card as:

RECLAIM ( )

For short programmes, however, this option is not necessary.

The value of RECLAIM (which is unimportant by comparison with its effect) is NIL.

28. How to write Functions of No Arguments

The list structure ( ) is equivalent to NIL. Either structure is a specification of no arguments for a function. Therefore RECLAIM ( ) and RECLAIM NIL are equivalent.

Now suppose that we must make RECLAIM into part of a function-definition or a PROG feature. We must use prefix notation, as with any other function. Remember that G(X,Y) in infix notation becomes (G X Y) in prefix notation. Therefore G( ) becomes (G). Similarly, RECLAIM( ) becomes (RECLAIM). For example, we may write a function-definition:

```
(TESTFN (LAMBDA (X) (PROG (A B)
    (SETQ A (CAR X))
    (SETQ B (CDR X))
    (RECLAIM)
    (RETURN (CONS B A)) )))
```

and another definition, in which RECLAIM occurs inside a COND, is:

```
(ANOTHER (LAMBDA (X) (COND ((NULL X) (RECLAIM)) (T X) )))
```

29. Functions for Printing and Similar Operations

The LISP system will normally print out only the final result of a computation, as we can see from the examples in section 17. But we may want to have some intermediate results of a computation printed (or punched on cards) also. We have five functions available for operations of this type.

(PRINT X) will print r(X), space up the carriage of the printer by one line, and also return r(X) as its value.

(PUNCH X) will punch r(X) in S-expression form on IBM cards, and also return r(X) as its value.

(PRIN1 X) will work only if r(X) is an atom, In that case, its value is r(X), but it will also print r(X) followed by a blank space, and leave the carriage of the printer on the same line.

(TERPRI), a function of no arguments, spaces up the carriage of the printer by one line. It has the value NIL.

(EJECT) spaces up the carriage to the top of a new page. Its value is NIL.

Here is an example which uses all of the printing and output functions:

```
(PRINTTEST (LAMBDA (N) (PROG (A B)
    (PRIN1(QUOTE CALCULATION))
    (PRIN1(QUOTE BEGINS))
    (TERPRI)
    (SETQ A (CADR N))
    (SETQ B (CDDR N))
    (SETQ C (PRINT (PUNCH (LIST B A (CAR N)) )))
    (PRIN1(QUOTE CALCULATION))
    (PRIN1(QUOTE FINISHED))
    (EJECT)
    (RETURN C) )))
```

30. Some More Useful Functions

30.1-In addition to the functions that have been mentioned as functions included in the LISP system, the following non-arithmetic functions are members of the system:

(LENGTH X), which has as its value the number of elements on the top level of the list r(X). If r(X) = ( ) = NIL, the value is zero.

(REVERSE X) has the value of the list obtained from r(X) by the reversal of the order in which its elements occur on the top level. For example:

REVERSE((A (B C)(D E) F)) = (F (D E)(B C) A)

(DELETE X Y) returns r(Y) from which the first occurrence of r(X) on the top level has been removed. For example:

DELETE(A (A B A C)) = (B A C)
DELETE(A (B C D) = (B C D)
DELETE(A (B (C A) D)) = (B (C A) D)

(SETDIFF X Y) is a 'set difference' of its arguments. i.e. it treats r(X) and r(Y) as sets and returns r(X) - r(Y). For example:

SETDIFF((A B C) (B C D)) = (A)
SETDIFF((B C D) (A B C)) = (D)
SETDIFF((A B C) (D)) = (A B C)

30.2 - The following arithmetic functions (in addition to those mentioned in previous sections) are available in the LISP system:

MAX takes any number of arguments between two and twenty (but all of them must evaluate to numerical atoms), and returns as its value the largest of them. MIN does the same for the smallest of them.

(GREATERP X Y) is T if r(X) is greater than r(Y), and F otherwise. A predicate function LESSP, whose meaning is obvious, also exists in the system.

EXPT, RECIP (for information about these two functions and their shortcomings, see (21.4)), PLUS and TIMES are available.

(MINUS X) has the value -r(X)
(DIFFERENCE X Y) has the value r(X) - r(Y)
(QUOTIENT X Y) has the value r(X)/r(Y)

31. Additional Functions in the LISP System at Imperial College

Several new functions not listed in the reference (3.1) are present on the LISP system tapes. They are described below by name, property (EXPR or SUBR), S-expression definition, effect, value and (if necessary) examples and comments.

FNL (SUBR)
(FNL (LAMBDA NIL NIL))

Value: NIL

DELETE (SUBR)

```
(DELETE (LAMBDA (X Y) (COND ((NULL Y) NIL)
     ((EQUAL X (CAR Y)) (CDR Y))
     (T (CONS (CAR Y) (DELETE X (CDR Y)) )) )))
```

Effect, value and examples: See section 30.

Comment: superficially it may seem that the function EFFACE, which is also in the LISP system, does exactly the same thing as DELETE. This is not strictly true. A good rule of thumb is that DELETE should always be used in preference to EFFACE until one has understood fully i) the difference between MAP and MAPLIST, and ii) the use of RPLACA and RPLACD.

PLUSP (SUBR)

```
(PLUSP (LAMBDA (N) (COND ((ZEROP N) F) (T (NOT (MINUSP N))) )))
```

Value : T if r(N) is a positive number, F if r(N) is a negative number or any representation of zero, and undefined ('error I3') otherwise.

Comment: in practice, PLUSP seems to be needed more frequently than MINUSP.

SETDIFF (SUBR)

```
(SETDIFF (LAMBDA (X Y) (COND ((NULL Y) X)
     (T (SETDIFF (DELETE (CAR Y) X) (CDR Y))) )))
```

Value and examples: See section 30.

TRIM (SUBR)

```
(TRIM (LAMBDA (X) (COND ((NULL X) NIL)
    ((MEMBER (CAR X) (CDR X)) (TRIM (CDR X)))
    (T (CONS (CAR X) (TRIM (CDR X)) )) )))
```

Value: the list obtained from the list r(X) by the deletion from r(X) on the top level of all but the last occurrences of any of its elements which occur more than once.

Example: TRIM((A B A C (D A) A E A)) = (B C (D A) E A)

FLATTEN (SUBR)

```
(FLATTEN (LAMBDA (X) (COND ((NULL X) NIL)
   ((ATOM X) (LIST X))
   ((ATOM (CAR X)) (CONS (CAR X) (FLATTEN (CDR X)) ))
   (T(APPEND (FLATTEN (CAR X)) (FLATTEN (CDR X)) )) )))
```

Value: (r(X)) if r(X) is an atom, otherwise the list obtained from r(X) by the removal of all brackets on all levels and all dots in all explicit dotted pairs.

Example: FLATTEN((A ((B C)) (D . E)(((2 3)) 4))) =
(A B C D E 2 3 4)
FLATTEN((A . B)) = (A B)
FLATTEN(A) = (A)

ALLOUT (SUBR)

```
(ALLOUT (LAMBDA NIL (PROG2 (REMPROP (QUOTE *) (QUOTE SYM)) (
                                EXCISE T) )))
```

Effect: ALLOUT removes LAP and the compiler from storage, thus making more space available for computation. It is probably not necessary to insert the doublet EXCISE(NIL) into a programme until the same programme (without it) has run out of space and produced the error message GC2 as a result. (This doublet removes only the compiler). Further, it is unnecessary to use ALLOUT( ) until a programme containing EXCISE(NIL) has run out of space in the same manner.

Value: NIL.

Comment: After LAP has been removed, functions in whose definitions the programmer has made use of LAP (see section 39) will not work, and will cause the computer to stop processing the entire LISP job.

FLAGP (SUBR)

```
(FLAGP (LAMBDA (A PR) (COND ((NULL A) F)
  ((EQ (CAR A) PR) T)
  (T (FLAGP (CDR A) PR)) )))
```

Value: T if the property-list of r(A) contains the flag r(PR), and F otherwise, provided that r(A) is an atom. It is not intended for use with any other form of r(A).

Comment: The function FLAG, which is mentioned on p.60 of the reference (3.1), can be used to place flags on the property-lists of atoms. FLAGP tests forthe presence of such flags.

ATOMLIS (SUBR)

```
(ATOMLIS (LAMBDA (X) (OR (NULL X) (AND (ATOM (CAR X)) (ATOMLIS
                        (CDR X)) )) ))
```

Value: T if r(X) is a list whose members on the top level are all atoms, and F if r(X) is any other kind of list. r(X) should not be an atom, or an explicit dotted pair like (A . B).

NUMLIS (SUBR)

```
(NUMLIS (LAMBDA (X) (OR (NULL X) (AND (NUMBERP (CAR X)) (NUMLIS
                        (CDR X)) )) ))
```

Value: T if r(X) is a list whose members on the top level are all numbers, and F if r(X) is any other kind of list. The restrictions on r(X) are the same as those for ATOMLIS.

TRACESET (EXPR)

```
(TRACESET (LAMBDA (U) (MAP U (FUNCTION (LAMBDA (J)
  (MAP (CADDAR (PROP (CAR J) (QUOTE EXPR) (FUNCTION (LAMBDA ( )
  (PROG NIL (PRIN1 (CAR J)) (PRINT (QUOTE ØØØ HAS NO EXPRØ))
  (RETURN (QUOTE ((NIL NIL (NIL)))))  )))))
  (FUNCTION (LAMBDA (K) (COND ((OR (NULL K) (ATOM (CAR K))
  (NOT (EQUAL (CAAR K) (QUOTE SETQ)))) NIL)
  (T (RPLACD K (APPEND (SUBST (CADAR K) (QUOTE VBL) (QUOTE ((
                        PRIN1 (QUOTE VBL))
   (PRINT (QUOTE ØØØ =Ø)) (PRINT VBL) (TERPRI) ))) (CDR K)
                                                ))))))  ))))))
```

Value: NIL

Effect: Let r(U) be a list of names of functions. Any function in this list which is defined as an EXPR via the PROG feature will have its definition in storage altered so that, whenever the function is used subsequently, all programme variables that are bound by SETQ on the top level of the PROG will be printed out automatically, together with their current values, whenever these values are changed or re-established.

Example: Suppose we have defined a function (FN (LAMBDA (Y) (PROG (A B) (SETQ A (CAR Y)) (SETQ B (CDR Y)) (RETURN (CONS B A))))), and have also applied TRACESET to it. Then, if FN(((K L) (M N))) is presented for execution by EVALQUOTE, the appearance of the printed output is:

```
FUNCTION...
FN
ARG(S)...
 (((K L) (M N)))
A =
(K L)
B =
((M N))
VALUE OF RESULT IS..
(((M N)) K L)
```

Comment: Section 40 contains an improved definition of TRACESET. The function UNTRACESET in section 40, while being the inverse of the function TRACESET there, is obviously not the inverse of TRACESET as defined above.

SAMEP (SUBR)

```
(SAMEP (LAMBDA (U V) (COND ((NULL U) (NULL V))
   (T (AND (MEMBER (CAR U) V) (SAMEP (CDR U) (DELETE (CAR U)
                                  V)) )) )))
```

Value: T either if r(U) and r(V) are identical lists or if r(U) can be obtained from r(V) by permutations of the members of r(?) on the top level only, and F otherwise.

UNION (SUBR)

Definition: See p.15 of the reference (3.1). UNION has two arguments.

Value: A single list equivalent to the set-theoretic union of the two arguments.

Examples: UNION((A B C) (D EF G)) = (A B C D EF G)
UNION((A B C) (B C DE)) = (A B C DE)
UNION((A (B C)) ((B D) E (B C))) = (A (B C) (B D) E)
UNION(NIL (A)) = UNION((A) NIL) = (A)

XN (SUBR)

Definition: See p. 15 of the reference (3.1). XN has two arguments.

Value: A single list equivalent to the set-theoretic intersection of the two arguments.

Examples: XN((A B C) (D EF G)) = NIL
XN((A B C) (B C DL)) = (B C)
XN((A (B C)) ((B D) E (B C))) = (A (B C) (B D) E)
XN(NIL (A)) = NIL

EXPEL (SUBR)

Effect: EXPEL takes one argument, a list of atoms for which all properties (including print-names) are to be removed from the system - most frequently, to make more space available for computation. EXPEL is an improved version of the function REMOB, which is mentioned on p.67 of the reference (3.1). Note that, although the argument of EXPEL is a list of atoms, REMOB can only be applied to one atom at a time.

Value: NIL.

POST (SUBR)

```
(POST (LAMBDA (NAME NR LINE)  (PROG (PA  )
   (SETQ PA (GET NAME (QUOTE EXPR)))
   (COND (PA (GO GX)))  (TERPRI) (PRIN1 NAME) (PRINT (QUOTE $$$
                                          HAS NO EXPR$))
   (TERPRI)
GX (COND ((EQ -2 NR) (DEFINE (CONS (LIST LINE PA) NIL)))
         ((MINUSP NR) (RPLACA(CDR PA) LINE))
         (T (GO (GA)))
   (GO GB)
GA (PSUB (CADDR PA) LINE NR)
GB (RETURN (GET NAME (QUOTE EXPR))) )))
(PSUB (LAMBDA (A B N)  (COND ((MINUSP N) (ERROR N))
   ((ZEROP N) (RPLACD A B))
   (T (PSUB (CDR A) B (SUB1 N))) )))
```

Comment: The function PSUB cannot be called directly by name by the programmer.
Further information. See section 32.

32. The Postal Function POST

Suppose that we have followed the option (see section 20) of updating the basic LISP system for our own specific use by adding some of our own function-definitions to it and reserving a numbered tape which contains this updated system. Suppose also that, at some later time, we find that we would like to make changes in the definitions of some of the functions that we have added to the system. The simplest method of repair is to make the changes in the original cards containing the function definitions, and add these cards as a whole to

the card decks for all later jobs. (In such cases, the definition on cards overrides the definition on tape). However, if there are many repairs, card decks can grow excessively large. This is a particular problem for the user who posts card decks to Imperial College and receives cards and output by post also. If a card deck has more than about 40 cards, it must go by parcel post and cannot conveniently be packed into the same envelope as printed output, for transmission by letter post. Therefore one tries to express repairs by punches on as few cards as possible.

The function POST is designed for this purpose. Its three distinct uses can be summarised by example. Imagine that our system contains as function-definitions:

```
(FRED (LAMBDA (X Y Z) (PROG (A B) (TERPRI)H(COND ((NULL Y) (
                  RETURN A))) (SETQ A (CONS (CAR Y) A))
   (SETQ B (CONS (ACR X) B)) (COND ((SAMEP A B) (GO H)) (T
                   (RETURN (FRED2 A B)))) )))
(FRED2 (LAMBDA (U V) (COND ((NUMBERP (CAR U)) V) ((NUMBERP
                   (CAR V)) U) (T (LIST U V)) )))
```

and that we want to do the following things:

i) Make (X Y) the list of dummy arguments for FRED, because Z is superfluous.

ii) Replace the mistake ACR by CAR in FRED. Note that, if we start counting on the top level the members of the list which begins with the atom PROG, but exclude PROG itself from the counting, the mistake occurs in the sixth element.

iii) Define a completely new function TOM having the same definition as FRED2 above.

iv) Finally change the definition of FRED2 to replace LIST by APPEND. (LIST occurs in the third argument of COND).

We achieve them by punching on cards only the following four specifications:

```
POST(FRED -1  (X Y) )
POST(FRED 6  (SETQ B (CONS (CAR X) B)) )
POST(FRED2 -2  TOM)
POST(FRED2 3  (T (APPEND U V)) )
```

With luck, these corrections should fit onto two cards, and therefore avoid the necessity of having to include all of the original function-definitions on cards as part of the job.

Without the LISP 1.5 Manual, it should be possible to understand exactly what POST does when its second argument is either -2 or -1 from a study of the definition of POST in section 31. An understanding of what happens when the argument is a positive integer requires knowledge of the functions RPLACA and RPLACD, but the actual use of POST for corrections when the argument is positive is simply a matter of careful counting of S-expressions inside the definition of the function to be corrected.

The value of POST is in every case the corrected definition of the function named as the first argument.

For programmers familiar with the concept of EXPR and the use of COMPILE, it is worth mentioning that POST will work correctly only if the function named as the first argument has its definition in the form of an EXPR in the system.

33. Pause 1

All of the preceding material is designed to be useful to the intending LISP programmer who has no access to the LISP 1.5 Manual. That Manual is an essential reference for serious programming, but it is not an expository manual in the usual sense, so that this Guide makes profitable reading in parallel to clear up points that may be obscure.

The ten following sections assume a knowledge of the Manual (3.1), and a certain amount of programming experience. Sections 34 and 35 are for the IBM 7090 LISP user who may have to run card decks temporarily on CDC 3600 or Ferranti Atlas computers elsewhere in Europe, sections 36, 37 and 38 are especially recommended as companion pieces to the Manual, and the remaining sections record various specialised programming tricks that have been effective in filling some obvious gaps in LISP procedure.

34. Peculiarities of the CDC 3600 LISP

CDC 3600 LISP is available to any CDC 3600 installation through CO-OP, the CDC equivalent of SHARE. The relevant manual for its general operation is the reference (3.5). However, let us consider the more specific problem of getting a card deck of a successful IBM 7090 LISP job to run on a 3600, for which LISP is implemented. The following points should be noted:

34.1 - The division of the card deck into packets is not permitted. It must be in effect a single packet, without STOP)))))) cards or Overlord directive cards (including FIN). Therefore all of these cards must be removed.

34.2 - The FMS identifying card (20.1), comment cards (20.2) and the LISP loader must also be removed and replaced by:

the ?JOB card of the 3600 installation
?EQUIP,6=(BINARY LISP3600),SV
?EQUIP,10=60
?EQUIP,11=61
?LOAD,6
card containing ?RUN, in the first five columns, and a specification of the maximum job time and lines output according to the direction of the installation

Above, ?stands for 7-9 punch. Punching on all of these cards begins in column 1.

34.3 - Remove the end-of-file card from the deck, and be sure that the FIN card is also removed. The CDC 3600 LISP deck ends with two standard cards, firstly a card with a 4-8 punch in column 1, and secondly an 'end-of-file' card with 7-8 punches in columns 1 and 2.

34.4 - Of the functions present in the LISP 1.5 Programmer's Manual (reference 3.1)), subject to the corrections in Section 37, at least the following are not available: LAP, COMPILE, EXCISE, CLEARBUFF, INTERN, MKNAM, ARRAY, ERRORSET, RECLAIM, COUNT, UNCOUNT, SPEAK, SYMTAB, LISTDEF, TRACECOUNT, MREAD, MPRINT, REMIND, BACKSPACE, TEMPUS-FUGIT, TIME, TIME1, BACKTRACE, SPACE, EJECT, PLB, VERBOS. None of the functions in Section 31 is included. However, the popular character-reading procedure (LAMBDA NIL (INTERN (MKNAM))) ( ) is performed by MKATOM ( ).

34.5 - If TRACE is used at all in the job, its first occurrence should be preceded by the doublet SETBIT(7).

34.6 - In the reference (3.1), it is stated that ONEP, ZEROP, EQUAL and EQ (for numerical arguments) operate with a tolerance of 3×10^{-6}. In CDC 3600 LISP, this figure is 10^{-8}.

34.7 - CAR of an atom is not junk, but the print-name or PNAME of that atom.

34.8 - + and - should not be used as characters of an atom.

34.9 - If PRIN1 is used anywhere (e.g. in a PROG), it is not advisable for PRINT to be the next function to be executed by EVAL QUOTE. For safety, (TERPRI) should be inserted between them in such a case.

35. Peculiarities of Atlas LISP

The Atlas LISP system is still in a state of flux, but the material presented in this section will probably not become obsolete for some time. With respect to the current state of Atlas LISP, documentation and/or folklore are available from the Atlas Computer Laboratory, Chilton, Didcot, Berkshire.

35.1 - There are no packets in Atlas LISP. Therefore all control cards and STOP)))))) cards should be removed from a 7090 LISP deck. If any STOP card is left in the deck, the Atlas system will take it at its face value and read no more input beyond it.

35.2 - Remove the FIN control card and FMS end-of-file card at the end of the deck. Substitute an Atlas 'end-of-file' card with a 7-8 punch in column 1 and Z in column 80.

35.3 - Remove the LISP loader, and substitute the prescribed Atlas job request cards (ID, maximum number of lines output, maximum running time etc.). The last three job cards, which are specific to LISP jobs, should be:

```
TAPE 1 SRC COMPILERS FP*
COMPILER,SPECIAL
MKØ4
```

(the '0' is a zero, not a letter)

35.4 - The following regularly-used functions are not implemented in Atlas LISP: ARRAY, TIME, TIME1, ERRORSET, all CAR-CDR-type functions with four As and Ds between the 'C' and the 'R', all character-reading and character-classifying functions. COMPILE exists, but it is fairly rudimentary.

35.5 - LAP is defined in Atlas LISP, but it uses ABL instructions. Therefore it is completely incompatible with IBM 7090 LAP.

35.6 - Do not use the atoms SYSTEM, INDEX and USE.

35.7 - CAR of an atom will not produce 7090-type junk, but the message SV OPERAND, which is an error message. On occasions, though, many pages of ordered junk will occur because the backtrace following an error is not as effectively inhibited in Atlas LISP as in 7090 LISP. On these occasions, your chances that EVALQUOTE will process any doublets after the one which caused the error are extremely small.

35.8 - The tolerance of 3×10^{-6} between numbers compared by EQ and EQUAL in 7090 LISP is exactly zero in Atlas LISP. For this reason, it is dangerous to have EQUAL and EQ compare numbers of different types. For example, EQUAL (1 1.0) may be false because of changes in the floating-point number (round-off effects) during reading-in. Therefore it is safe to say that EQ and EQUAL will almost always have the value NIL if either or both of their arguments are floating-point (or octal with negative exponent).

35.9 - TRACESET and UNTRACESET, working on the top level of any PROG, are already defined in the Atlas LISP system. There is an additional valuable debugging aid: the function (DEPTH N), which sets up a push-down-list trap in a manner similar to that in which (COUNT N) sets up a CONS counter trap. The inverse of DEPTH is (UNDEPTH), which is to be compared with (UNCOUNT NIL). (Note the difference in number of arguments!) In this connection, the function (HOWDEEP) is the analogue of (SPEAK NIL).

35.10 -The application of CDR to an atom, to get the property list of that atom, is not allowed; it leads to the error described in (35.7). Atlas LISP has a special function (PLIST X) whose value is the property list of r(X).

There is a final practical point to observe. I.C.T. card readers digest I.B.M. cards without complaint, but the converse is not true. Therefore, if an I.B.M. deck is developed by several runs at an Atlas installation, during which time the developments are added to the deck on I.C.T. cards, those cards should be replaced by duplicated I.B.M. cards if the deck is to be run again on a 7090 or 7094.

36. An Index for the LISP 1.5 Programmer's Manual

The LISP 1.5 Programmer's Manual (3.1) is supplied without a general index. To improve its readability, E.C. Berkeley and D.C. Bobrow have supplied such an index, on page 377 of the reference (3. 2). For convenience, an updated version of the index, relevant to the 1965 reprinting of the Manual, is reproduced here. References to functions not mentioned below may be found in the index on Page 100 of the Manual.

37. List of Errata for the LISP 1.5 Programmer's Manual

The following list is intended for use only with the February 1965 reprinting of the Manual. The 1962 edition contains many more defects which have been corrected in the later version.

Page	Line	Correction
11	17	For atomic symbol read S-expression
18	-9	For from 10 to 100 read from 3 to 40
59	-8	For is the value is read is the value of
63	2	For from the list l read from the top level of the list l
63	6	Note that this line refers to the text below it, not above it
67	3	For MIT users only read MIT and Atlas users only

Page	Line	Correction
67	18	Underline excise
67	19	For pair read doublet
67	19	Underline remprop
67	19	Add However, in the system at Imperial College, remprop (*; SYM) and excise (*T*) need never executed. Their combined effect is produced by the new function allout( )
68	2	Add this LISP set-up tape, which is distributed by SHARE, is NOT equivalent to the setup tape (X 248) at Imperial College, so that the descriptions of the library functions (with the exception of traceset) do not apply. traceset is included automatically in the setting-up procedure from X 248.
77	5	Add If such a free variable is y, the typical diagnostic reads (y UNDECLARED). There is no error message
80	12	For A4 read A3
93	-1	Add This facility exists at Imperial College.
95	-1	For A4, A5, A6, A7, A8, B2, B3, B4, B5, B6 read A4, A6, B6, A7, B4, B2, A5, B1, B5, B6.
97	1	Insert The paragraphs headed Obkeep and Reserved on this page are inapplicable to the system at Imperial College.
98	1	Insert The paragraphs headed If, For and Sublis on this page are inapplicable to the system at Imperial College.
99	18	Add However, the present system at Imperial College differs from this specification of the SHARE LISP system in the following respects: System Temporary Tape (SYSTMP) B3 SW2 and SW4 nominally have no effect but it is wise to leave them both off if unpredictable effects are to be avoided.

38. Commentary on Appendix 1 of the LISP 1.5 Programmer's Manual

Verbos and the Garbage Collector: This form of non-talkative garbage collector is now the standard for all current LISP systems except Atlas LISP.

Flap Trap: In brief, an underflow during an arithmetic operation produces a result of zero for that operation. No error message occurs.

Time: The timing print-out has the format

a MIN b MS. c MIN d MS

(MS = milliseconds). It occurs automatically at the beginning (directly after the print-out of the control-card) and end of each packet. a = b = Ø at the beginning of any packet, but a and b at the end of that packet measure the time taken for the processing of the packet. The 'age' of the system is measured by c and d. Without affecting any of these numbers, we can use TIME and TIME1 inside any packet. TIME1( ) causes a print-out in which a = b = c = d = Ø. Thereafter, TIME( ) may be used any number of times. Its effect is as described in the Appendix. The 'age' of the system, mentioned above, is the time that the LISP job has run since all of the relevant tapes have been mounted and 'START' pressed on the 7090 console.

Lap and Symtab; Non-Printing Compiler: Unless there is a special advantage in having the LAP code for compiled functions printed out, it is wise to execute SYMTAB(NIL) and LISTING(NIL) before any compilation.

Tracecount: see section 22

Space and Eject: see section 29

Tape: For the Imperial College LISP system, note the appropriate correction in section 37.

Backtrace: Ideally BACKTRACE( ) should be executed by EVALQUOTE (i.e. it should never occur on cards as (BACKTRACE)), but frequently one can get away with its use in other contexts. This loophole has been exploited, for example, in some self-debugging programmes.

Obkeep; Reserved: Not in the Imperial College LISP system.

If; For: Not in the Imperial College LISP system.

Sublis: This note is not applicable to the Imperial College LISP system.

Characteristics of the System: Use Appendix E in preference to this paragraph, and note the corrections in section 37.

39. Some Useful Tricks in LAP

39.1 - INSET: Occasionally it may be necessary to insert a given expression r(V) into a specific location r(U) in core storage. r(U) is obviously an octal number and, since the contents of any word in core storage can be written as an octal number of up to 12 digits, so is r(V). The definition of the function which performs the insertion is:

```
(INSET (LAMBDA (U V) (LAP (CONS U (CONS (CONS V NIL)
                                        NIL)) NIL)))
```

39.2 - TMP: Here we find the first practical instance of the use of INSET. Suppose that we have a programme which is so long that it cannot fit onto one LISP system tape, but it can be divided between two such tapes. If the processing of data by the first 'half' of this programme is complete before the application of any operations from the second 'half', we can use MPRINT in the first half to put the intermediate results onto a scratch tape, from which MREAD in the second half can recover them for the rest of the processing - provided that we have some means of transferring control of the LISP job from one system tape to another.

Because we have seen on p.80 of the reference (3.1) that, after any TEST packet, a copy of the system for use with the next packet is read in from the system temporary tape (SYSTMP), it follows that the core memory must contain one location with information on where the SYSTMP is to be found. In the Imperial College system, this location is 367Q and, for the conventional SYSTMP allocation of tape drive B3, it contains the octal number 2002223Q. The last digit is the tape-drive number, the first and fourth digits are 2 if the channel is B and 1 if it is A, and the others are always as shown. Therefore, if we put the tape containing the first half of our programme on B7 and the tape containing the second half on (say) A5, and change 2002223Q at location 367Q to 1001225Q during the last TEST packet that makes use of the first half of the programme, the second half of the programme will be read in from A5 for the next packet. The function that performs the trick is:

```
TMP(1001225Q)
```

where we have

```
(TMP (LAMBDA (N) (INSET 367Q N)))
```

Generalisations of the trick for longer and more complex programmes are obvious.

39.3 - GETCEL: It is desirable to have an inverse of a function that is as useful as INSET. This function, GETCEL, returns as its value the number stored at the octal location r(U). GETCEL is defined in terms of the LAP function GETT. 12321Q is a location reserved for special use by GETCEL. Definitions are as follows:

```
LAP(( (GETT SUBR 0) (XEC EXEV) (LDQ OCTD) (TRA MKNO))
          ((OCTD . 5401) (EXEV . 12321Q) (MKNO 13645Q)))
(GETCEL (LAMBDA (U) (PROG2 (INSET 12321Q (PLUS U 5Q10))
                           (GETT) )))
```

Prior to the definition of GETT, it is necessary to make the FAP instruction XEC available to LAP by the execution of CPDEFINE(((XEC 522Q8))).

39.4 - ALIST: The a-list or association list inside a PROG is not normally accessible to the programmer. However, if we make explicit use of functions like EVAL and APPLY for which one argument must be the a-list, we may need a function whose value is the current a-list. A function of no arguments which has this property is (ALIST). Its LAP definition is:

```
LAP(( (ALIST SUBR 0) (CLA $ALIST) (TRA 1 4)) NIL)
```

39.5 - ORDERP: Suppose that we wish to establish a specific ordering of certain atoms which occur as data. (This ordering may be of use in fast factoring functions for manipulation of algebraic polynomials). An ordering is established by the order in which the atoms are read into the LISP system for the first time. Therefore, if we want AA to precede AB, AB to precede AC, and so on, we can head the first SET or SETSET packet by a harmless doublet like:

```
CAR((AA AB AC AD AE etc.))
```

whose value is unimportant but whose effect is to make the desired ordering. We now need a predicate (ORDER U V) which takes two atomic arguments, and which is true if r(U) precedes r(V) in the order and false otherwise. In LAP, the relevant definition is.

```
LAP(( (ORDERP SUBR 2) (40Q8 A) (CLA (QUOTE [illegible] ))
        (TRA 1 4) A (PXD) (TRA 1 4)) NIL)
```

39.6 - POL: If we are present when a LISP job is being run on the 7090, it may be convenient to have certain results printed on-line to avoid the delay involved in the usual off-line printing of the output tape. We can obtain on-line printing in bulk by depressing sense switch 3 on the console, but the slow on-line printer effectively increases the running time of the job and usually prints out much inessential material besides the information that is needed. A more efficient method of printing is provided by the function (POL U), which prints r(U) both on-line and off-line. This function temporarily alters the instruction that tests the position of sense switch 3 to an unconditional transfer of control to the on-line printing routine. In the Imperial College system, the sense-switch test occurs at location 1562Q and the printing routine begins at 1564Q

```
(POL (LAMBDA (U)    (PROG (A B)
     (SETQ A (QUOTE (1562Q (522000017140)))))
     (SETQ B (QUOTE (1562Q (200000156400)))))
     (LAP B NIL)
     (PRINT U)
     (LAP A NIL)
     (RETURN U) )))
```

40. TRACESET and UNTRACESET for all Levels of a PROG

The function TRACESET presently in the system (see section 31) has the disadvantage that it detects occurrences of

SETQ only on the top level of a PROG, and not inside a COND or in any other well-hidden places. There is the further disadvantage that the inverse function, named UNTRACESET in the LISP 1.5 Programmer's Manual, is not defined in the system because of the amount of storage which it takes up.

Since TRACESET has an EXPR in the system, its old defintion can be removed by the execution of REMPROP(TRACESET EXPR), or simply overwritten by a programmer's new definition if DEFINE is used. Below, we have the S-expression definitions of a good version of TRACESET and UNTRACESET that works on SETQ at any place inside a PROG. The subsidiary functions PNTSET, NBLKJ and NRP2S are common to TRACESET and UNTRACESET, as can be seen from the definitions.

```
(TRACESET (LAMBDA (V) (PROG2 (CSETQ TRACECT T) (MAP V (QUOTE
                                                    NBLKJ)))))
(UNTRACESET (LAMBDA (V) (PROG2 (CSETQ TRACECT NIL) (MAP V
                                                (QUOTE NBLKJ)))))
(NBLKJ (LAMBDA (U) (PROG (PA)
  (SETQ PA (GET U (QUOTE EXPR)))
  (COND ((NULL PA) (PROG NIL (PRIN1 (CAR U)) (PRINT (QUOTE ØØØ
                                HAS NO EXPRØ)) (TERPRI)))
        ((OR (ATOM (SETQ PA (CADDR PA))) (NULL (CDR PA)) (NULL
                                (CDDR PA))) NIL)
        (T (GO PB)))                (RETURN NIL)
PB (MAP PA (QUOTE NRP2S)) )))
(NRP2S (LAMBDA (X) (COND
  ((ATOM (CAR X)) NIL)
   ((NOT (MEMBER (CAAR X) (QUOTE (SETQ COND PROG2)))) (MAP
                                (CAR X) (QUOTE NRP2S)))
   ((EQ (CAAR X) (QUOTE SETQ)) (RPLACA X (CONS (CAAR X) (COND
                                (TRACECT (LIST (CADAR X)
      (LIST (QUOTE PNTSET) (LIST (QUOTE QUOTE) (CADAR X))
                                (CADDAR X)))) (T (CONS
      (CADAR X) (COND ((OR (ATOM (CADDAR X)) (NOT (EQ (QUOTE
                                PNTSET) (CAR (CADDAR X)))))
      (CADDAR X)) (T (CDDR (CADDAR X)))) )) ))))
   ((EQ (CAAR X) (QUOTE PROG2)) (MAP (CDAR X) (QUOTE NRP2S)))
   (T (MAP (CDAR X) (FUNCTION (LAMBDA (J) (MAP (CAR J) (QUOTE
                                NRP2S)) )) )) )))
(PNTSET (LAMBDA (X Y) (PROG NIL (TERPRI) (PRIN1 X) (PRINT
                                (QUOTE ØØØ =Ø)))
  (RETURN (PRINT Y)) )))
```

41. Readable Displays of Function-Definitions

When functions are defined by DEFINE, their EXPRS are printed in the output over the full width of the page, in a single-spaced format that is difficult to read. To improve the readability of this output, functions are available to print EXPRS in a decorative and spacious format (at the rate of one per page).

If we have EXPRS on cards which we wish to introduce into a packet via DEFINE and print in this new format at the same time, we punch the function-name SANDEF in the place where we would have put DEFINE previously. If we wish to display, define and compile the EXPRS in the one operation, the relevant function-name is CANDEF.

If, on the other hand, we have previously used DEFINE to put some EXPRS into a LISP job (or onto a tape) and we want only to have these EXPRS printed readably, we use the function SHOW, which must be given one argument, a list of the names of the functions to be displayed.

Finally, if we wish to display and compile some functions that have previously been defined via DEFINE, the relevant function is CHOW, whose argument is the same as for SHOW.

All of the four display-functions are included in a single SET packet which is punched on cards in the drawer marked 'LISP library' in Room 405. Please duplicate your own copy from this packet and return the original cards to the drawer.

42. FORTRAN Input-Output Format

The prefix notation (see section 9) characteristic of LISP is occasionally difficult to handle. In particular, long algebraic results of LISP computations may be more easily read by a greater number of people if converted to the infix notation of FORTRAN (with * for mulitplication, ** for exponentiation, and so on). The 'LISP library' function (MATHPRINT X) prints r(X) in FORTRAN format in the output and returns the value NIL.

The reading of FORTRAN-like input by the computer is a more difficult proposition, but it may be necessary under some circumstances. Obviously such input cannot occur in the body of a packet in LISP, as it is not in the form of valid S-expressions. However, we can legally put it in the 'tail' of a packet, i.e. between the word STOP and the large number of right-hand brackets which normally follow STOP. We can then make use of the function (MATHREAD), analogous to (READ), within the packet, to ingest the FORTRAN-like input and convert it automatically into S-expressions. Consider a simple example:

```
MATHREAD( )
(LAMBDA (N) (TIMES N (EVAL (MATHREAD) NIL))) (5)
STOP
(A*B/SQRTF(C))
(2**6)    )))))))
```

As part of a LISP packet, these card-images produce the output

```
FUNCTION...
 MATHREAD
ARG(S)...
 NIL
VALUE OF RESULT IS..
 (TIMES A (QUOTIENT B (SQRTF C)))
FUNCTION...
 (LAMBDA (N) (TIMES N (EVAL (MATHREAD) NIL)))
ARG(S)...
 (5)
VALUE OF RESULT IS..
 320
```

When MATHREAD is used, the following rules for card-punching must be observed.

42.1 - The word STOP at the end of a packet should be punched by itself on a single card.

42.2 - Each FORTRAN-like expression to be read in should be surrounded by a pair of brackets, additional to any pairs that may be needed in the expression itself.

42.3 - Each FORTRAN-like expression to be read in should begin on a new card.

42.4 - The last of the expressions to be read in at the end of any packet should be followed by a large number of right-hand brackets.

The functions which make up the complete specification of MATHPRINT are available in a single SET packet. Another SET packet contains MATHREAD. The cards for each of these packets are present in the 'LISP library' drawer. If you wish to use either MATHREAD or MATHPRINT, please follow the instructions in the last sentence of section 41.

43. Pause 2

The last nine sections may be read in parallel with the LISP 1.5 Programmer's Manual, as they contain information which has been learned over about 2.5 man-years of LISP programming.

The remaining sections deal with material which does not have parallel references in the Manual. Section 44 sets out the procedure for the establishment of a LISP system tape from the source tape, and section 45 gives an account of some programmes for use in theoretical physics that are also stored on the source tape.

44. Setting-up an IBM 7090 LISP System

The establishment of a LISP system tape is a two-pass process. The source tape, containing the card images for both passes, is X 248 at Imperial College. If the system is to be set up elsewhere, any copy of one file of X 248 onto a blank tape constitutes an acceptable source tape.

The first pass produces a binary deck of about 400 cards (which is part of the input for the second pass) and printed output of about 17700 lines (300 pages). The output is a listing of the part of the LISP system that fills about the first 201000 locations of the computer, plus LAP, permanent list structure and entry points to the compiler which occupy a region downwards from location 77777Q. Check that the serial number (columns 75-80) of the first card of the binary deck is LISP0000 and that the card is labelled TRA00371.

The programme which produces the output is:

```
*ID      usual Imperial College FMS ID card
*          PLEASE MOUNT TAPE X 248 (SAVE + F.P.) ON CHANNEL A 5
*        PAUSE
*        PACK
*        FAP
         UPDATE  9
         SKIPTO                                                    LSØØØØØØ
            ::if you want to update the source tape, update cards
                                  in FAP format go here::
ENDEND END       CONTIN                                            LS122Ø7ØØ
```

On the job slip, request tape X 248 (saved + file-protected) on channel A5, maximum output 18000 lines, maximum running time 8.0 minutes.

Important: The programme above, and all subsequent setting-up programmes in this Guide, must be punched according to the rules for FAP (FORTRAN II Assembly Programme)coding. The location field (containing the asterisks and the first "E" of ENDEND above) begins in column 1, the operation field (PAUSE, PACK etc.) begins in column 8, and the variable field ("9" in UPDATE 9) begins in column 16. All serial numbers (e.g. LS122070) for cards begin in column 73. If these rules are ignored, the programmes will not work. Also, be sure that no programme deck is loaded on-line, because the on-line card reader does not scan columns 73 to 80.

All programme decks should end with an FMS end-of-file card.

The second pass of the setting-up procedure produces the LISP system tape (SYSTAP). The binary programme from the previous pass is built upon from cards with serial numbers prefixed by LU on the source tape. We reserve from the tape librarian in Room 405 the tape to be used as a SYSTAP, and this is mounted on channel B3 (to be saved but not file-protected). X 248 is mounted on A5 as before. The second-pass programme is as follows:

```
*ID       usual Imperial College FMS ID card
*         PLEASE MOUNT TAPE X 248 (SAVE + F.P.) ON CHANNEL A 5
*         PLEASE MOUNT TAPE      (AND SAVE IT) ON CHANNEL B 3   ::(The number of
                                                                    your reserved
                                                                    tape will go in
                                                                    the blank space)::
*       PAUSE
*       XEQ
*       PACK
*       FAP
        UPDATE  9,,,NO
        SKIPTO                                                    LUØØØØØØ
        ENDUP
```

```
*       PACK
*       FAP
        RTBA    2
        RCHA    IOC
        LCHA    Ø
        TRA     1
IOC     IOCT    Ø,,3
        END
*       DATA
            ::the LISP binary deck from the first pass goes here::
        TAPE    SYSPIT,A5
        SET
RECLAIM NIL STOP))))))
        FIN
```

However, suppose that we want to alter some of the LU cards and produce an updated LISP system. In that case, we run a slightly different programme, which takes account of the FMS UPDATE option. X 248 goes on A5 and the intended SYSTAP on B3, as before, but in addition we mount a good scratch tape on B5. This tape is not file-protected, and not saved after the job. The programme is:

```
*ID     Imperial College FMS ID card
*         PLEASE MOUNT TAPE X 248 (SAVE + F.P.) ON CHANNEL A5
*         PLEASE MOUNT (AND SAVE) TAPE       ON CHANNEL B3
*         PLEASE HAVE A GOOD SCRATCH TAPE ON CHANNEL B5
*       PAUSE
*       XEQ
*       PACK
*       FAP
        UPDATE  9,10,U,NO
        SKIPTO                                              LUØØØØØØ
            :update cards for the LU section go here::
        TAPE    SYSPIT,A2                                   LUØ1392Ø
        UNLOAD  9
        REWIND
        ENDUP
*       PACK
*       FAP
        RTBA    2
        RCHA    IOC
        LCHA    Ø
        TRA     1
IOC     IOCT    0,,3
```

```
        END
*       DATA
              ::the LISP binary deck from the first pass goes here::
        TAPE    SYSPIT,B5
        SET
RECLAIM NIL STOP)))))
        FIN
```

On the job slips for either of these jobs, request a maximum running time of 11.0 minutes, and a maximum output of 3000 lines.

The result is that the tape on B3 at the end of either job is a LISP system tape, which should be saved and mounted on channel B7 in the usual way for all future standard LISP jobs.

To be sure that the tape on B3 is not overwritten at the end of the job, please note the trick (see section 20) of putting a small 0.1-minute "pseudo-job" at the end of the regular card deck.

When punching the TAPE control for either of the two jobs above, be sure that there is no blank on either side of the comma in the variable field.

The SET packet at the end of the jobs is essential.

Before updating can be decided upon for either of the two passes of the LISP system assembly, it is necessary to know what is on the appropriate card images on the source tape. Card images for the first pass run from LSØØØØØØ to LS122Ø7Ø, and for the second pass from LUØØØØØØ to LUØ1357Ø. Selective printing can be carried out by the following programme:

```
*ID     usual I.C. FMS ID card
*
*         PLEASE MOUNT TAPE X 248 (SAVE + F.P.) ON CHANNEL A5
*         PLEASE MOUNT A GOOD SCRATCH TAPE ON B5, AND PRINT ONE FILE OF IT
                                                        AFTER THE JOB
*       PAUSE
*       PACK
*       FAP
        UPDATE  9,10,U,NO
        SKIPTO                  ::columns 73-80 contain serial nr. of first card
                                                        to be printed::
         END OF PRINTING        ::columns 73-80 contain serial nr. of last card
                                                        to be printed::
        UNLOAD  9
        UNLOAD
        ENDUP
```

The job slip should be filled in accordingly.

The programme on the previous page can be used in connection with the next section of this Guide, to investigate the contents of the Physics Programmes LW and LX.

45. Programmes for Calculations in Theoretical Physics

The set-up tape (X 248) contains material for three programmes in theoretical physics. The first, designed largely by A.C. Hearn, occupies card-images with serial numbers LWØØØØØØ through LWØ1482Ø, and is called programme LW below. It may be used to perform trace operations on products of terms containing Dirac gamma matrices, four-vectors and scalars. The second, running from LWØ15Ø4Ø to LWØ1726Ø, is a short programme to square matrix elements of the type derived from the last programme on the tape. This last programme (Programe LX) is accommodated on cards LXØØØØØØ through LXØØ567Ø. It takes as input a specification of a Feynman diagram or diagrams and produces as output an expression for the corresponding matrix element(s).

Commentaries on each of the three programmes (under the headings Programme LW, LWØ15Ø4Ø or LX) are given below.

In section 44, we have already seen that two passes are required to make up a basic LISP system tape (SYSTAP) from the set-up tape (X 248). We use a third pass to construct a system tape containing basic LISP plus any of the three physics programmes. In the job that constitutes the third pass, we ask for tape X 248 to be placed on A5 and the SYSTAP resulting from the second pass of the LISP assembly to be placed on B7 (both tapes saved and file-protected) We reserve from the tape librarian in Room 405 a tape to carry our physics programme, and have this placed (saved but not file-protected) on B3. Finally, we request a good scratch tape on A7. All of these requests must be entered on the job slip and punched on comment cards to go after the ID card that is the first card of the job deck. After the last of the comment cards, we put the following programme:

```
*       PAUSE
*       XEQ
*       PACK
*       FAP

        UPDATE  9,13,U,NO
        SKIPTO                                          ::a::
          ::any cards to update the programme go here, if needed::
        TAPE    SYSPIT,A2                               ::b::
          ::for Programme LX only, a card with blanks up to column 72
                and serial number LXØØ567Ø beginning in column 73::
        UNLOAD  9
        REWIND
        ENDUP
*       PACK
*       FAP
        RTBB    7
        RCHB    IOC
        LCHB    Ø
        TRA     1
```

```
IOC     IOCT      0,,3
        END
*       DATA
            ::octal correction cards for the basic LISP system, if needed::
           ::transfer card - 7,9 punches in column 1, and 2,3,4,5,6,8,9
             punches in column3, otherwise blank::
        TAPE      SYSPIT,A7
        SET
RECLAIM NIL STOP)))))))
        FIN
        ::FMS end-of-file card::
```

Follow the same rules for placement of columns that have been stated in section 44 (e.g. all FAP instructions like UPDATE begin in column 8, arguments for the instructions begin in column 16, and the commas which separate arguments must not be preceded or followed by a blank.

The third-pass programme should run for a maximum of 10.0 minutes and produce a maximum of 3000 lines of output.

The letters a and b in the listing above stand for the following serial numbers (to begin in column 73 of the relevant cards):

Programme LW	a = LWØØØØØØØ	b = LWØ1482Ø
Programme LWØ15Ø4Ø	a = LWØ15Ø4Ø	b = LWØ1726Ø
Programme LX	a = LXØØØØØØØ	b = LXØØ566Ø

The result of the third-pass programme is the production of a tape on B3 containing the basic LISP system plus the specified physics programme. This tape should be saved and placed on B7 as a SYSTAP for future jobs which make use of that physics programme.

Programme LW

This programme occupies so much space in core storage that we cannot process it with a conventional LISP system tape of the type produced in the no-updating scheme in section 44. During the second pass of the assembly, we must use updating to insert a new SIZE control card in place of the standard card with the serial number LUØØØØ4Ø. The old card contains:

```
SIZE    14ØØØQ,5ØØØQ,422ØQ,224ØØQ                                  LUØØØØ4Ø
```

where the "S" occurs in column 8 and the numbers begin in column 16, but the new card should contain:

```
SIZE    156ØØQ,26ØØQ,4ØØØQ,232ØØQ                                  LUØØØØ4Ø
```

The following two cards should also be included in the appropriate place in the update section:

```
TAPE((12Ø4Q 12Ø6Q 22Ø6Q 12Ø7Q 22Ø4Q 22Ø2Q 12Ø5Q 22Ø1Q ))      LUØ1145Ø
   DELETE   THRU                                                LUØ121ØØ
```

On the last card, the two words should begin in columns 8 and 16 respectively. The serial numbers on all cards should begin in column 73.

All errors of the type A2 or R1 in the output accompanying the second pass of the assembly of the system tape should be ignored. The tape produced on B3 during this pass can be mounted on tape drive B7 for subsequent assembly of Programme LW by the methods mentioned in the earlier part of this section.

Each trace calculation is the subject of a separate TEST packet. Following the TEST control card, we place a card containing the words START NIL STOP and nothing else. Data for the calculation begin on the next card, and are surrounded by a pair of brackets. All operations within the data are punched in a FORTRAN-like notation and separated by commas. They may have any number of arguments, also separated by commas. The available popular operations are:

45.1 - FACTOR, an operation which causes its arguments to be factored out where they occur in the result of the calculation, to make the reading of the result easier.

45.2 - ORDER, which establishes an order of precedence for its arguments in the printing of the result.

45.3 - INDEX, whose atomic arguments stand in the places where Greek indices are required.

45.4 - LET, whose arguments represent replacements to be made in a calculation. For example, if the scalar product of the four-momentum P1 with itself is the square of the electron mass EM and the four-momentum P2 refers to a proton, we can punch:

```
LET((P1 . P1) = EM**2, (P2 . P2) = PM**2)
```

Note the convention used to denote scalar products.

45.5 - MSHELL, which specifies the four-momenta of particles which are supposed to be on their mass shells.

45.6 - FMAKE, each of whose arguments represents the substitution for a functional form used as shorthand in the expression to be simplified by trace or other operations. For example, if we have to write $d + bx + cx^2$ frequently in the calculation, and wish to abbreviate it by $f(x)$, we use:

```
FMAKE(F(X) = D +B*X +C*X**2)
```

45.7 - TITLE, whose argument is the atom we want to use as a heading for the printing of the result of a calculation. Only letters and digits can occur in this atom.

45.8 - Following the use of any combination of the seven operations listed above, we represent the expression to be simplified as the argument of the functional operation SM.

Dirac matrices and scalar products of the matrices with four-vectors are specified by the operator G, which has two arguments. The first argument is always a (non-numerical) atom labelling the fermion line on which the entity occurs. The second atomic argument, if previously

declared as one of the arguments of INDEX, causes the expression beginning with G to be interpreted as a Dirac matrix; otherwise the second argument is assumed to be the name of a four-vector, so that the entire expression is interpreted as the inner product of Dirac matrices and a four-vector. A further operator EPS, having four arguments, represents the totally antisymmetric unit fourth-rank tensor if all of the arguments have been declared via INDEX, and is otherwise taken to be the partial contraction of this tensor with the (four-vector) arguments that are not so declared. Finally, the occurrence of the Dirac matrix with subscript 5 on a fermion line labelled by, say, LB, is written as G(LB,A). A is a special symbol, so that we cannot use it elsewhere in the argument of SM.

As a simple example, suppose that we wish to calculate the sum of two terms. The first is the product of the four-vector p (with index mu), q (with index nu) and the unit second-rank tensor (indices mu and nu). The second is the product of the inner product of Dirac matrices with p and the same matrices with q. We punch:

```
          TEST
START NIL STOP
(INDEX(U,V), SM((P . U)*(U . V)*(Q . V) + G(L,P)*G(L,Q)))
```

and the answer produced by the programme is 2*(P . Q).

In any expression of the form G(L,P) + EM, where P is a four-vector and EM is a scalar, the term EM is assumed to be the product of this scalar and the unit 4x4 matrix.

In addition to the operation described up to (45.8) on the previous page, any function named in the object list (we can look for its presence by executing EVAL(OBLIST NIL)) can be used as an operator.

Conventional organisation of LISP packets can be mixed with the type of organisation quoted in the example above, where SM actually uses the function MATHREAD to read any FORTRAN-like material following the atom STOP.

Programme LWØ15Ø4Ø

This programme is intended to extend slightly the usefulness of Programme LX. The function RESPROC is ideally used after the function PROCESS from Programme LX has printed its results on the output tape and written them onto scratch tape 1 via MRPINT. Therefore it is necessary to see the results of PROCESS and to understand broadly how PROCESS works before RESPROC and the other functions of Programme LWØ15Ø4Ø are used. To sum up, its effect is to square the representation of a matrix element and obtain a quantity proportional to a cross-section.

Programmes LWØ15Ø4Ø and LX can be set up together onto one tape.

Programme LX

The structure of a Feynman diagram can be specified by a list of sublists which described individual vertices. Any sublist is so arranged that CAR applied to it gives a list of the four-momenta entering the vertex and CDR gives a list of the four-momenta leaving it. Therefore we can write:

```
(((P1) K P2) ((P3 K) P4))                                    (45.9)
```

as a specification of the diagram shaped like an upper-case H.

The principal function in Programme LX is PROCESS. It has one argument, a list of lists like (45.9). Because of this, we can give an arbitrary number of Feynman diagrams to PROCESS at any one time, and the corresponding matrix elements are calculated one after the other. In practice, though, the most common use of PROCESS deals with only one diagram at a time. If we want to apply PROCESS to (45.9), we punch:

```
PROCESS((((P1) K P2) ((P3 K) P4))))                    (45.10)
```

Before using PROCESS, be sure that the reasons for the presence of each pair of brackets in (45.10) is understood.

Each four-momentum in an expression like (45.9) has properties like spin and mass. These properties must be assigned to the four-momenta prior to the use of PROCESS, by DEFLIST with the appropriate indicator as the second argument. This requirement can be explained best by example. Suppose that P1 and P2 refer to electrons, P3 and P4 to protons and K to a photon. With some obvious mnemonics for masses, we can punch:

```
DEFLIST(((P1 EM) (P2 EM) (P3 PM) (P4 PM) (K LA)) MASS)
```

The conventional spin assignments for fermions (P1 to P1Ø), bosons of spin K 1 (K1 to K6, then K1Ø) and bosons of spin Ø (Q1 to Q1Ø) are already in the system. K1 is given a polarisation four-vector E1, K2 is given E2 and so on. The current assignments of mass give odd-numbered fermions masses of ME, even-numbered fermions masses of MP, all spin-1 bosons except K1Ø masses of LA (K1Ø has zero mass) and all spin-bosons masse of MU.

If there are any four-momenta which must be integrated over during the determination of a matrix element, we punch:

```
LOOPVAR(x)
```

where x is a list of these four-momenta.

If any four-momenta are to be associated with anti-particles, we punch:

```
FLAG(x ANTIPTL)
```

where x is a list of these four-momenta.

If we introduce any new four-momenta (other than P1 to P1Ø) which stand for fermions, we punch:

```
FLAG (x FMN)
```

where x has the usual meaning.

All uses of FLAG, DEFLIST, LOOPVAR and similar initialising functions that we may wish to define must precede the use of PROCESS, which is the instruction to the programme to determine matrix elements.

Recognition of integrals is carried out through the standard table ITYP near card LXØØ494Ø in the listing of Programme LX. If we wish to add new types of basic integral to the table, we use the function ITYPES in the manner demonstrated near card LXØØ494Ø. Further comment on this point occurs below. If we wish to introduce new types of vertex or particle, we use the function RULE near card LXØØ1Ø5Ø. Cards LXØØ1Ø6Ø to LXØØ131Ø contain examples of RULE.

The section of Programme LX devoted to four-dimensional integration is presently prepared only to recognise denominator terms of the form $k^2 - \lambda^2$ (to which it gives the code number 1) and $k^2 - 2p.k$ (code number 2). In the table ITYP, for example, (1 1 1) is a shorthand notation for $(k^2 - \lambda^2)^{-3}$, and the list following (1 1 1) is a representation of the result of the four-dimensional integral of this term over k. (1 2 2) represents $(k^2 - \lambda 2)^{-3}(k^2 - 2p.k)^{-1}(k^2-2q.k)^{-1}$. We can add different types of term to

ITYP by making our own assignment of the codes 3, 4 etc. and modifying the LISP functions which perform recognition of integrals to take these new codes into account. Hopefully, the integral-recognition functions are so arranged that the necessary modification will be confined to changes of the definitions of the two functions IRULE and CODES, to be found after card LXØØ54ØØ. Therefore, unlike the case of Programme LW, whose abilities and coding are fixed, Programme LX requires modification under some circumstances. This is so because the methods of taking traces are fixed and well-defined, whereas different strategies are appropriate to different integrals where integration must be carried out. In that sense, Programme LX is incomplete and will always be incomplete, but an improved version of it should be available at the Stanford Linear Accelerator Centre and on the time-shared PDP-6 computer of the Stanford Artificial Intelligence Project from April 1967. Potential users of Programme LX who wish to include integrations in their calculatinns are cordially invited to read the previous 44 sections of this Guide and the LISP 1.5 Programmer's Manual (3.1), and learn enough LISP to be able to do their own programming research and development.

In connection with the questions of use and development, all users should obtain listings of the programmes they wish to use, in advance of setting up the programmes themselves, by executing the selective-printing programme given in section 44. A study of any listing, even by someone who knows very little LISP, is often quite useful for an understanding of what is going on inside the programme while it is running.

If Programme LX is used independently of Programme LWØ15Ø4Ø, PROCESS will appear to be very taciturn and give the value NIL instead of the expected matrix element. We recapture the result of the calculation by following the PROCESS instruction of the general form (45.10) in our card deck with:

```
(LAMBDA NIL (PROG NIL (PRINT (MREAD 1)) (PRINT (MREAD 1)) (PRINT (MREAD 1))
      (PRINT (MREAD 1)) (REWIND 1))) ( )
```

Users who have read all of the preceding sections of this Guide will doubtless be able to effect this procedure more concisely and understand why it is necessary.

46. Help !

In case of trouble with LISP programmes that cannot be explained adequately either here or in the reference (3.1), it may be a good thing to consult people who have had some experience in LISP programming. Such people are known to exist in the following places:

*Atlas Computer Laboratory, Chilton, Didcot, Berkshire
AERE Computing Centre, Harwell, Berkshire
Physics Department, Royal Holloway College, Englefield Green, Surrey
*Mathematics Department, University of Manchester
*departments and units concerned with computing, University of Edinburgh, Edinburgh 8, Scotland
Philosophy Department, University of Oxford
Project MAC, Massachusetts Institute of Technology, Cambridge 39, Mass. 02139, USA
Forsvarets Forskningsinstitutt, Kjeller, Norway
*Artificial Intelligence Project, Stanford University, Stanford, Calif. 94305, USA
National Bureau of Standards, Boulder, Colorado, USA
*School of Physical Sciences, Flinders University, Bedford Park, South Australia

Mathematics Department, King's College, London
Institut für theoretische Physik, Johannes-Gutenberg-Universität
Mainz, Germany.
Laboratori Nazionali di Frascati, Frascati, (Roma), Italy

Information about apparent misbehaviour of the LISP system (as distinct from functions defined by the programmer) is best obtained at the places marked with an asterisk. Good luck !

47. Postscript

For Imperial College users, copies of the LISP system exist on tapes X 51 and X 247, and the set-up tape containing card images for LISP and programmes in physics is X 248. At 11 December 1966, the address for mailing of jobs was Computer Unit, Centre for Computing and Automation, P.O. Box 346, Imperial College, Prince Consort Road, London, S.W.7.

For other readers of this Guide or intending users of LISP within range of an IBM 7090/94 installation, copies of the Guide can be obtained from Computer Reception, Room 404, Electrical Engineering Department, Imperial College, Exhibition Road, London, S.W.7. To set up a LISP system for the first time on a 7090 or 7094, send a blank tape to Imperial College with a request to copy one file of X 248 onto that tape. The copy may be used to set up LISP according to the directions in the Guide section 44. The physics programmes may be set up from the same tape if the instructions in section 45 are followed.