The Use of BCL in a List Processing Environment.

By

Richard John William Housden.

University of London

Institute of Computer Science.

Thesis submitted for the Degree of Doctor of Philosophy at the University of London.

October, 1969.

ABSTRACT

This thesis is primarilly concerned with the design and implementation of machine independent systems for teaching generalised list processing techniques. Two systems, LSIX and BCL, are considered in detail. LSIX is a London version of Knowlton's L6, and BCL is a general purpose data processing language with special emphasis on the input and output of structured data. The use of BCL here is as a compiler compiler for LSIX and, in an extended form, as a list processor in its own right.

Part I, which is largely expository, gives a brief introduction to list processing, outlines those features of the classical list processing systems which are pertinent to this report and describes LSIX and BCL.

Part II deals with the implementation of LSIX in BCL, and storage allocation and collection. In general, students have favoured the direct use of BCL as a list processor. The author's extensions to BCL provide a system in which all levels of list processing are possible. It is particularly suitable for teaching, as the student is able to define his own structures and list processing functions. The facilities available are illustrated by a number of BCL list processing programs. This thesis has itself been edited using a BCL list processing program.

Further extensions to BCL are proposed in Part III. These allow the user to define, within his program, new types of

structured objects, and operations to be performed upon them. The result is a general purpose language which is capable of handling data structures of any complexity, is suitable for teaching, and whose implementation is largely machine independent.

The Appendices give details of programs and computer output. A preliminary account of some of the work described in this thesis has already been published in the Computer Journal. The first paper containing the gist of sections 2.1 and 2.2 is included as Appendix 7 and a further paper, on List Processing in BCL, has been accepted for publication.

Acknowledgements

I should like to express my thanks to my supervisor, Prof.B. Higman, for his valuable advice and encouragement throughout this work.

I also wish to thank Prof.R.A. Buckingham and the staff of the University of London Institute of Computer Science for the facilities which have been made available to me. In particular I am grateful to the members of the BCL group for their help during the early stages of this work, and to Miss.M. Mozetich for her careful typing of the thesis.

To Susan, Gillian, Jeremy,

Andrew, Clare, Simon,

Sarah and Jonathan.

CONTENTS

APPENDICES

PROLOGUE

The original intention in the work described in this thesis was to investigate the system requirements (operational and linguistic) of a medium for teaching generalised list processing techniques- i.e. techniques which transcend the uniform node structure imposed by the classical list processing languages. At the time when it was begun, Knowlton's paper on L6 had recently appeared, and a first working version of BCL had just become available. As L6 promised all the flexibility required and a field test of BCL as a compiler compiler was in order, the programme appeared to involve implementation of L6 using BCL as a compiler compiler as a first phase, followed by use of L6 in a year's teaching, and possibly some iteration on these two steps in the light of student reaction. In the event, student reaction (and indeed my own) was to favour direct use of BCL as a list processing language in its own right. However, this did not become apparent until the middle of the second year's work, and as a result there is a sort of non sequitur in the work reported here, in that after describing the implementation of L6 in §2.1 to §2.3, in §2.4, where one would expect a discussion of the pros and cons of L6 in the light of experience, instead one finds a discussion of list processing in BCL. Had the results been known before it was begun, doubtless the present section §2.4 would have been §2.1; however, the time spent in implementing L6 has had its value in other directions, and no further apology seems necessary for the space devoted to it in what follows.

PART I

List Processing.

List-Processing

§1.1 Introduction

List-processing is a method of storage organisation which bridges the gap between the one dimensional store of a digital computer and the multi-dimensional problems of the real world. Often tables or rectangular arrays of information to be operated upon by a program are not just amorphous masses of numbers but involve important structural relationships between the data elements. During the processing of this information the actual structure of the data may be changed as well as the values contained in the structures. Techniques for manipulating such structures were introduced in 1956 when Newell, Shaw and Simon designed the first information processing language (IPL II) for use in their investigations of heuristic problem solving by machine. Research in this and other areas such as mechanical translation, information retrieval and operational research generated problems involving a form of information processing which could not be handled conveniently in any of the conventional languages. Often the precise form of the data was not known in advance and complex data structures evolved and were modified dynamically during the execution of the program. The IPL II system made use of linked data elements, which were not necessarilly stored in consecutive locations.

The idea of simple linked lists of information originated in connection with the design of computers with drum memories. After executing the instruction in location n, such a computer is usually not ready to get its next instruction from location n+1 because the drum has already rotated past this point. The most favourable position for the next instruction depends upon the time taken to

execute the current instruction and the machine can operate many times faster if its instructions are optimally located rather than consecutively. The machine design allows for an extra address field in each instruction to store a link to the next instruction to be obeyed. Programs for 'plus one' address machines, as they are called, are the earliest examples of linked lists although there is no hardware provision for dynamic insertion or deletion of instructions.

Linked memory techniques are now recognised as basic computer programming tools which can be used in ordinary programming languages without requiring sophisticated subroutines or interpretive routines. Much of the work described in this report has arisen in the course of teaching these techniques to M.Sc. students and the systems which have been implemented for this purpose allow the students to get near to the innermost workings of list-processing programs.

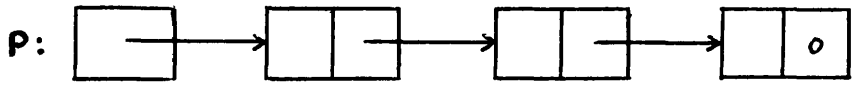
Definition of terms

It is necessary at this point to define several terms and notations which will be used frequently in this report. The information stored in a list or a table consists of sets of nodes or data blocks (called beads, records or list elements by some authors). Each node consists of one or more consecutive computer words divided into named parts called fields. In the simplest case a node is just one word of computer memory and has just one field comprising the whole word. A more interesting example is a node which represents an element of a sparse matrix. Such a node might be divided into five fields:

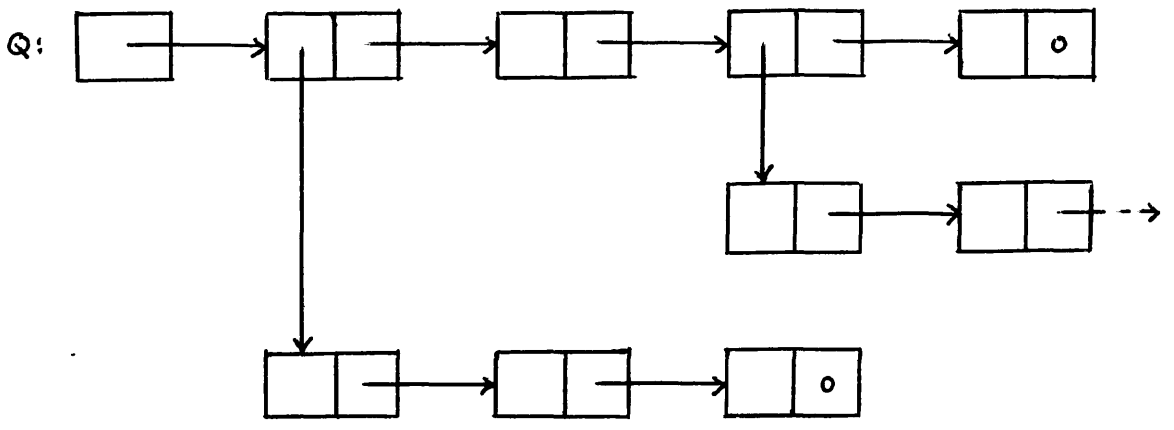
ROW INDEX	COLUMN INDEX	VALUE	COLUMN LINK	ROW LINK

Thus each element is a member of both a row list and a column list. The contents of a field may be an address, a number, alphabetic information or anything else the programmer may desire. The address of a node (a link, pointer or reference to that node) is usually the address of the first word of the node. Linked structures are built by acquiring new nodes and planting their addresses in link fields of other nodes in the structure.

A field within a node may be referenced by writing its name followed by the address of the node enclosed in parentheses. For example suppose that each node of a simple linked list consists of two fields named SYMBOL and LINK and that a link variable P contains the address of the first node of the list:



Then the symbol field of the first node is referred to as SYMBOL(P) and the link field as LINK(P). Fields in nodes other than the first may be accessed via the link fields of preceding nodes. Thus SYMBOL(LINK(P)) refers to the symbol field of the node to which LINK(P) points, i.e. the second node in the list. Note that the last node in the list has an easily recognisable symbol in its link field to indicate that this is the end of the list. More complex structures may be built by storing in the symbol fields pointers to sublists as shown below.



In this example SYMBOL(LINK(LINK(Q))) contains a pointer to the second sublist of the list Q (the list to which Q points).

Note that names have been used for two quite different things: as variables and as fields. It is meaningless to use a field name on its own, it should always be followed by the address of the node of which the field is a part. The notation used here for referring to fields is that used in BCL and is similar to the notation of LISP. An alternative notation is that used in LSIX in which the address of the node precedes the name of the field. Thus if A is a field name and W a link to a node or data block we refer to field A of that block as WA.

A simple (or linear) list is defined as a list without sublists, that is a list of nodes whose only structural relationship is essentially a linear one. Some important examples of simple lists are those in which all insertions and deletions take place at the ends:

A stack is a simple list in which all insertions and deletions take place at one end.

Stacks are particularly useful where a nested structure is involved, for example in arithmetic expressions. They occur frequently in connection with recursive algorithms.

A queue is a simple list in which insertions are made at one end and deletions at the other.

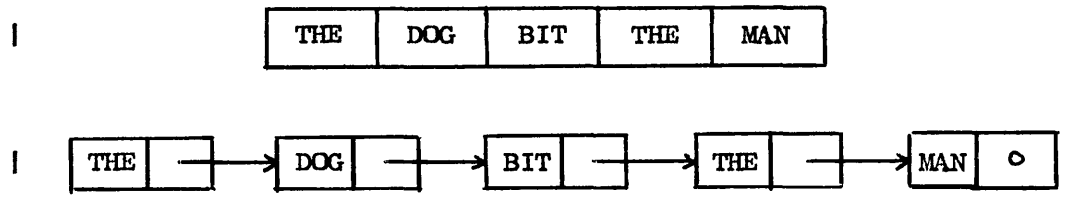
A deque (or double ended queue) is a simple list in which insertions and deletions are made at the two ends.

Queues and deques occur in simulation models in which objects are delayed and awaiting service.

The advantages of linked storage compared with sequential allocation are clear when we need to insert a new node or delete a node in the middle of a list. For example consider the sentence

THE DOG BIT THE MAN

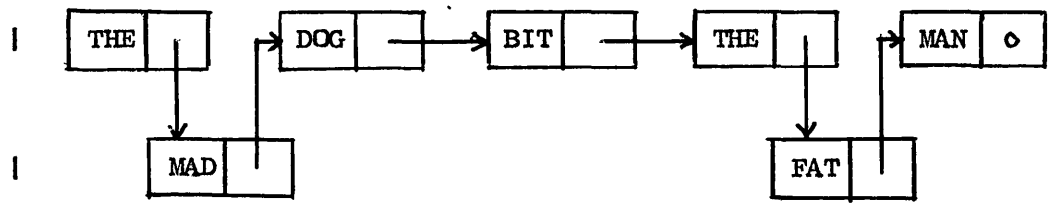
These words could be stored in sequential locations or in a linked form:



Suppose now that we wish to insert extra words to give

THE MAD DOG BIT THE FAT MAN

In the sequential case, some words already in the list must be moved to make room for the insertions. In a long list this is very inefficient. If linked locations are used then additional items may be stored in any locations that happen to be available, and insertions are effected simply by changing a few links.

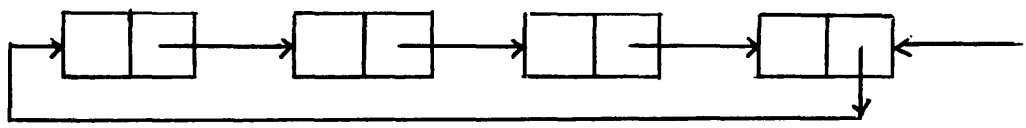


Clearly it is just as easy to delete a node from a linked list.

One serious disadvantage of linked storage is the time taken to access nodes other than the first in the list. Access to a random node is gained by linking down from the beginning of the list. When data is to be accessed at random, sequential allocation of storage is preferred and the address L[k] of the kth node is L[0]+(k-1)c where c is the number of words in a node.

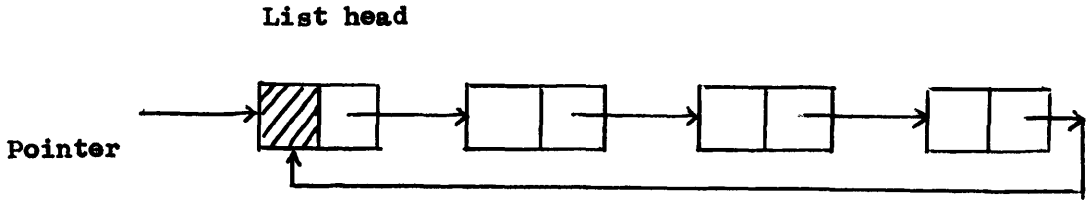
Circular Lists

A circular list has the property that its last node links back to the first instead of storing a terminator in its link field. It is then possible to access any node in the list, starting at any point. The following situation is typical:



A circular list can be used conveniently as a stack or as a queue, since a circular list with one pointer to the rear node is equivalent to a linear list with a pointer to each end.

Some programmers insert a special easily recognisable node into each circular list to mark the beginning and end of the list. This also has the advantage that the list is never empty. The special node used for this purpose is known as a listhead. References to a circular list are usually made via the list head. The circular list now becomes

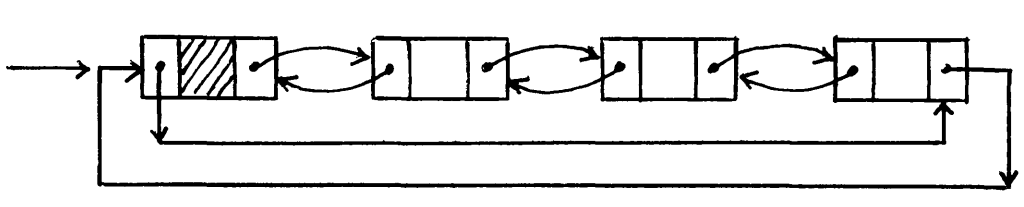


List heads are not confined to circular lists and can be used as an 'anchor' in any linked list. It is sometimes found useful to store information about a list in its head e.g. the number of nodes on the list.

Doubly linked lists

For even greater flexibility we can include two links in each node, to the preceding and the following nodes:

As in the case of circular lists, it is often convenient to include a list head giving the following typical representation:

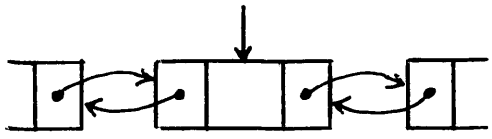


If the link fields are named LLINK and RLINK it is clear that if X is the address of any node in the list then

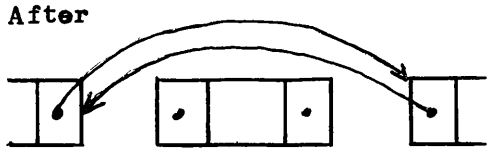
$$RLINK(LLINK(X)) = LLINK(RLINK(X)) = X$$

A doubly linked list permits movement backwards and forwards along the list. Another of its many advantages is the ease with which a node can be deleted from the list. For example the node with location X may be deleted as follows:

Before X



After



RLINK(LLINK(X)):=RLINK(X)

LLINK(RLINK(X)):=LLINK(X)

and the node X is returned to the pool of free space.

List structures

A list structure is a list in which several fields in a node may contain cross links to other nodes in the structure.

§1.2 List processing languages

Many list processing systems have been developed, both as independent computer languages and as extensions to existing languages, to deal with the manipulation of complex data structures. The most widely used systems have been IPL-V (Newell, Shaw and Simon, 1959), and LISP (McCarthy, 1959), and more recently SLIP (Weizenbaum, 1963). Several books have been written about these systems and we give here only a brief description of those features which are pertinent to this report.

Storage allocation and collection

An important common feature of list-processing languages is that storage for data structures is not preassigned but is allocated dynamically when it is needed. As the pattern of the data, both structure and contents, evolves, new nodes are acquired and added to the structure by creating links from the structure to each new node. This implies some mechanism for allocating nodes as they are required. Usually this is accomplished by means of a list of available space (a linked stack) which contains all those blocks which are not being used. Initially this list contains all storage locations not occupied by the program. Blocks of store (nodes) are detached from this for use in building data structures during the execution of the program.

Eventually the available space list may be exhausted and the problem arises of reclaiming any blocks which, after being used, have become free again. In some systems, notably IPL-V and SLIP, it is the responsibility of the programmer to return data-blocks to the available space list when they become free. To do this the programmer must keep track of the status of all lists and sublists. Part of a list may be shared with several other lists and the

structures involved may be so complex that it is difficult to keep track of them. SLIP deals with this problem by keeping a reference count in the head of each list. In other systems such as LISP it is impossible to keep track of all list-cells and no blocks are returned to the available space list until the latter has been exhausted. Then a 'garbage collection' procedure is initiated which scans all active list structures marking those blocks which are in use. Blocks which are no longer attached to the active list-structure will not be marked. When the lists have been scanned, all data blocks are examined and those which are free are returned to the available space list. At the same time marks are erased from the blocks which are still in use, as the garbage collector may be entered several times during the execution of a program. Details of some algorithms for automatic garbage collection are given in §2.4.

IPL-V

IPL-V is the assembly code of a hypothetical machine, and, like most list-processing languages, it is interpreted, not translated. A node or list-word in IPL-V consists of two address-fields called SYMB and LINK and two 3-bit fields P and Q. The fields P and Q contain information about the contents of the fields SYMB and LINK. Usually SYMB contains an IPL symbol and LINK points to the next node in the list. The SYMB field may point to a sublist. If the LINK field of a node is zero then that node is taken as the last node in the list.

LISP

Programs in LISP are expressed in mathematical functional notation combined with conditional expressions. The internal representation of data structures is similar to that in IPL-V in that each node (or pointer word as it is called in LISP) consists of two address-size fields called car and cdr. Usually cdr points to the next node in the list and car points either to a node or to numbers or strings of characters called atomic symbols. An atomic symbol is distinguished from other nodes by a special symbol in its first field.

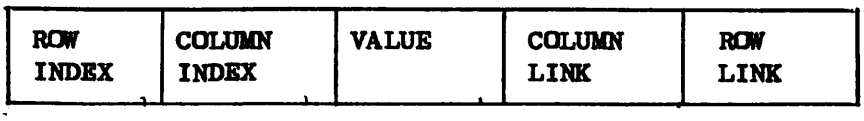
SLIP

Unlike IPL-V and LISP which are autonomous systems, SLIP consists of a set of subroutines which can be embedded within a FORTRAN-type language. This therefore has the advantage that the usual arithmetic facilities of algebraic languages are readily available. The internal representation also differs in that a list structure in SLIP is both circular and doubly linked. Each node consists of (a) two link fields called the left and right links, (b) a 2-bit field which identifies the type of the list item and (c) a full word field which contains the actual item. This item may be a full data word or a pointer to a sublist.

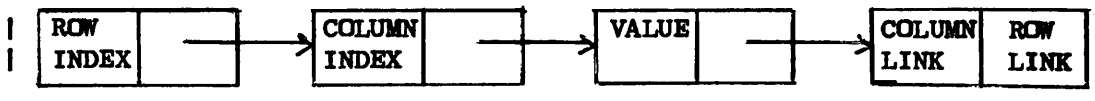
Need for flexibility

Often the most natural data form for a particular problem representation is not the same as the basic form used in the list processing system which has been selected. The three systems mentioned above allow no flexibility in the type of node set up. A programmer may wish to build linked structures in which the nodes

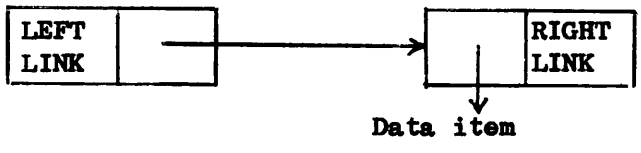
are not all of the same size, to partition nodes into fields in different ways or even to allow some fields to overlap. It is always possible to represent a data form using the basic nodes provided by a system. For example in a LISP-like system the five field node



could be represented by



and a doubly linked list could be constructed using pairs of nodes:



However, this is unnecessarilly complicated and pedagogically it is better to allow a student to define the structure of nodes required in his program. This thesis is concerned mainly with the design and implementation of list processing systems for teaching purposes. Such a system must be sufficiently flexible to demonstrate the techniques used in list programming in both high level functional languages such as LISP and low level languages which allow the user to get close to the innermost workings of his program. The two systems considered are LSIX, the author's version of the Bell Telephone Laboratories low-level list processing language L6, and a system based on BCL, a general purpose data processing language which is currently being developed under the direction of D.F.Hendry

at the University of London Institute of Computer Science. Both are more flexible than most of the popular list processing systems and allow the programmer to specify nodes of several different sizes and structures, which can be used simultaneously in any program. LSIX is essentially a low-level system whilst BCL is a high-level language which allows the use anywhere in the program of symbolic assembly language instructions for the machine concerned.

A number of extensions have been made, by the author, to the BCL compiler to provide a list processing system with most of the advantages of other high and low-level list processors. In the extended BCL a programmer is able to define and build his own list processing system. Examples of demonstration programs used in an M.Sc. course are given in §2.4. Students have been able to practise list programming without the restrictions imposed by better known systems. By simulation of LISP, IPL-V and SLIP in LSIX or BCL the basic operations which underlie their implementation can be understood.

§1.3 LSIX

The most important features which distinguish LSIX from other list processors are the availability of several sizes of storage blocks and a flexible means of specifying within them fields, containing data or pointers to other blocks. Data structures are built by appropriating blocks of various sizes, defining fields (simultaneously in all blocks) and filling these fields with data and pointers to other blocks. Available blocks are of lengths 2**n machine words where n is an integer in the range 0-7. The user may define up to 36 fields, which have as names single letters or digits. The fields may overlap and may be redefined several times during the execution of a program. For example the field named D may be defined as bits 5 through 17 of word number 2 of any block. Any field which is long enough to store an address may contain a pointer to another block. The contents of a field are interpreted according to the context in which they are used.

The LSIX system contains 26 basefields called bugs. The contents of a bug are referred to by naming the bug (a single letter). If the bug contains a pointer to a block, a particular field in that block is referred to by concatenating the names of the bug and the field. For example, WD refers to the D field of the block to which W points. A field more remotely positioned from the bug is referred to by concatenating the names of the bug, the sequence of pointers and the field. Thus if bug X points to a block whose B field points to a block whose A field points to a block whose D field is to be referenced, the latter is called XBAD.

LSIX Instruction format

In general an LSIX instruction consists of an optional label followed in order by optional tests, optional operations and an optional transfer of control. An example given by Knowlton is

```
| L2   IFNONE (XD,E,Y)(XA,E,0) THEN (XD,E,1)(X,P,XA) L2
```

which says that

```
| IFNONE       of the following is true: that the contents of XD
|              equals the contents of Y or that the contents of
|              XA equals 0,
| THEN         perform the following operations: set the contents
|              of XD equal to 1, make X point where the current
|              contents of XA point and then go to the instruction
|              labelled L2 (the same instruction in this case).
| OTHERWISE    no operations are to be performed and control
|              goes to the next line of coding.
```

Other conditions are

```
| IFALL        satisfied IF ALL of the elementary tests are
|              satisfied,
| IFNALL       satisfied IF NOT ALL of the elementary tests
|              are satisfied
| IFANY        satisfied IF ANY of the elementary tests are
|              satisfied.
| IF and NOT   are synonymous with IFALL and IFNONE.
```

The other instruction type is the unconditional instruction consisting of a sequence of operations to be performed. A complete LSIX program and computer output illustrating the diagnostic aids available is given in Appendix 1, and Appendix 7 includes a complete list of LSIX tests and operations.

Three pushdown stores are available in the system for saving field contents, field definitions and for subroutine calls.

The author's main extensions to the original LSIX are the generalisation which allows blocks of any size and the provision of an automatic garbage collector.

§1.4 BCL

BCL is a general purpose data processing language with special emphasis on the input and output of structured data (Hendry, 1966). The structure of the data to be transferred is defined by means of a group or ordered set of objects (elements).

Consider the sequence

```
| FIELD IS (OSP.,(EITHER 'T.', TIMEFIELD
|                 OR     BUG,(EITHER FLDNAMES OR NIL.)
|                 OR     INTEGER,'.', IF INTEGER LE 128, READFIELD),
|          OSP., OCT:=0, PLANT)
```

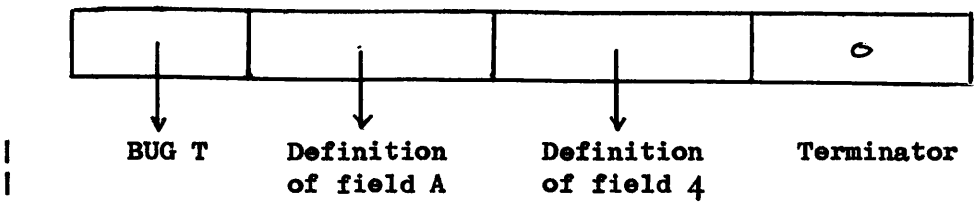
which appears in Section 2.1. The names TIMEFIELD,BUG,FLDNAMES and READFIELD are the names of groups of objects defined elsewhere in the program; INTEGER and OCT are previously defined variables of type A (i.e. they can store an address or an integer). The first two words, FIELD IS, indicate that the above sequence is a definition of the name FIELD. That the rest of it is a parenthesised structure with commas indicates that FIELD is the name of a group. The commas between the objects denote juxtaposition and for alternatives the notation EITHER...OR...OR... is used. The objects within a group may be literals or names. Character literals are enclosed with primes, numeric literals are obvious, also literal commands such as x := z, and group denotations, i.e. unnamed groups, which are enclosed with parentheses. Names which must of course be defined somewhere, but can be defined passim, may be names of variables or groups. Group definitions may be recursive, i.e. the name of a group may appear in its own list of objects.

A group may be encountered either in input mode or in output mode. When a group is encountered in input mode the next characters

in the input stream are matched with objects in the group. If the data in the input stream matches the structure defined it is input. During the matching process any literal commands are obeyed and groups whose names, appear in the list of objects are called. If at any point the data fails to match the structure defined in the current branch then control backtracks to the nearest branchpoint and attempts to match the next alternative. With the exception that the input stream pointer is reset, any side effects resulting from the execution of commands in branches which eventually fail to match are not undone.

Suppose the object 'FIELD' is encountered in input mode and the next characters in the input stream are TA4 , a remote field. These characters are matched with the objects in FIELD. OSP. is a built-in group which matches any number (including zero) of spaces. Next we have a literal group consisting of three alternatives which are tried in order. The next two characters are compared with 'T.'; matching is successful on T but fails on the period so the first alternative fails and the second is tried. The group named BUG is entered. It recognises T as the name of a bug or basefield and plants its address in the object area, The second object in this branch is itself a literal group consisting of a pair of alternatives, FLDNAMES and NIL. FLDNAMES matches any number of field names (A and 4 in this case) and plants in the object area the addresses of the corresponding field definitions. NIL. is the system defined null element. After successfully matching A and 4 with the second branch of the literal group the input process continues with OSP. which again reads any spaces. The variable OCT is assigned the value zero and the group PLANT is called to plant the value of OCT in the object area. Thus as a side effect of the

recognition of the remote field TA4 the following sequence of pointers is planted in the object area.



A second example is the LSIX read-only field called 64. (an integral power of two terminated by period). Attempts to match this with 'T.' and BUG fail. In the third alternative, INTEGER being a variable of type A, the integer 64 is assigned to it, then the period is matched. The condition INTEGER $\leq$ 128 is satisfied so READFIELD is called to check that the input integer is an integral power of 2. The final objects are processed as before and the group READ is completely matched.

When a group is encountered in output mode the process is that of assembling characters for output instead of matching characters for input. By means of conditions alternative objects may be selected for output. Programs are entered in input mode and are switched to output mode on encountering the special group name O/P which is followed by a literal group of objects to be output. On completion of this literal group the system reverts to input mode.

That a BCL program is driven by the structure of the data in the input stream was the main reason for its use as a compiler compiler for LSIX. Through the experience gained in this work and as BCL developed after the first version was produced in 1966 it became clear that BCL itself is suitable for teaching list processing techniques. Many of the basic operations required were already built into the language. It is possible to define nodes consisting of any number of fields which may store numbers,

addresses, or character strings of any length. A student can define the basic list-processing functions and define his own storage allocation and collection mechanism including an automatic garbage collector. A number of facilities such as returns from the middle of a group and the provision of functions with parameters have been implemented by the author to provide a teaching system within which all levels of list-processing are possible. Details of these extensions to the BCL compiler are given in Appendix 5. The use of BCL as a list-processor is described in section 2.4 in which BCL programs are given for the solution of a number of problems including differentiation of a polynomial expression stored as a binary tree, a solution of the classical transportation problem using orthogonal lists and updating a tree structured file directory. Groups of commands for automatic garbage collection are also described.

PART II

The Implementation of List Processing Systems

§2.1 The Definition of LSIX in BCL

In this section the syntax of LSIX is defined in BCL. Embedded in the syntax definitions are commands, including calls to routines, which are obeyed during the matching of input LSIX source statements. For the reader who has difficulty in following the complete definition of LSIX and who would prefer at the first reading to separate the syntax from the semantics, a definition of the syntax only is given in Appendix 2. The semantic commands generate a linked list of object code in the object area of the store. In LSIX the newline character is the instruction terminator and programs are analysed line by line. Each record in the linked list corresponds to a single LSIX instruction.

Format of object code

The general form of the data to be analysed and the format of the object code is illustrated by the complete LSIX program given in Appendix 1. It will be seen from the object listing for that program that each record consists of four halfwords of links and descriptive information, followed by information of variable length representing elementary tests and operations. A typical record is that representing line 15 of the program i.e.

```
BACK IF (XB,L,XDB) THEN (XB,IC,XDB) (X,D) BACK
```

The object code generated for this instruction is described in detail below.

Word	Contents	Remarks
0	3 01 02 01 7	a description word consisting of five fields: (1) the condition type, 3 represents IF; (2) the number of elementary tests, 1 in this case; (3) the number of elementary operations, 2 in this case; (4) the type of transfer of control, 1 represents normal transfer; (5) a flag to distinguish description words from other information.
1	00112244	Address of next instruction in sequence, line 16 in this case.
2	00112140	Address of first elementary operation in current record. This field is not used for unconditional instructions.
3	00000170	The line number (15).

Words 4-11 contain information representing the test (XB,L,XDB).

4	00000040	Test code, 4 represents 'less than'.
5	00101234	Address of bug X.
6	00100260	Address of definition of field B.
7	00000000	Zero terminator of the sequence of addresses representing XB.
8	00101234	Address of bug X.
9	00100320	Address of definition of field D.
10	00100260	Address of definition of field B.
11	00000000	Terminator of the sequence representing XDB.

The representation of elementary operations is similar to that for tests but in addition to the operation code the number of operands also is stored. Thus (XB,IC,XDB) is represented by words 12 to 20 inclusive (see Appendix 1) and (X,D) , which is an abbreviation for (X,P,XD), is represented in its full form by words 21 to 27. The final word of this record, word 28, contains the address to which control is transferred after execution of the elementary operations, i.e. the address of line 15.

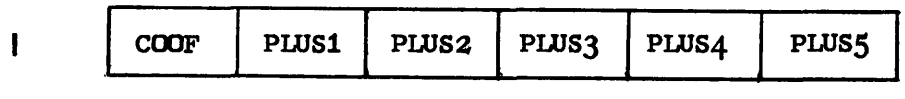
Definition of LSIX

The following program compiles object code, the format of which was described in the preceding paragraphs. Declarations of variables and the details of some routines are not included here but with the aid of the detailed comments on most of the groups, the reader should have no difficulty in following the program.

It is convenient to define first an 'indefinite' group in BCL, namely

```
MISC(?) IS (A COOF,A PLUS1,A PLUS2,A PLUS3,A PLUS4,A PLUS5)
```

The variables COOF,PLUS1,PLUS2, etc, are defined as type A i.e. they can store either an address or an integer. Together they constitute the group named MISC. The query indicates that MISC is an indefinite group i.e. that any variables declared within it are allocated relative addresses, or offsets, and not absolute addresses. Thus COOF is an offset of zero, PLUS1 an offset of one halfword, PLUS2 two halfwords etc. We can think of them as defining a node consisting of six fields



As described in Part I, a field name is meaningless on its own and must always be used as an offset from some specified base. Thus if PTR points to the first word of a node COOF(PTR) refers to the first (half)word of the node, PLUS1(PTR) to the second and so on. The offsets defined in the group MISC are used frequently in both the definition which follows, and in routines associated with the execution of LSIX.

When LSIX was first defined in BCL, labels and GO TO commands were not available in groups and several operations such as the

dictionary search have been implemented recursively to overcome this problem.

Main program structure

```
| LSIX IS (INITIALISE, LSIXSTATS)
| LSIXSTATS IS (LINE := LINE+1,
|               (EITHER INSTR
|                OR     DIRECTIVE
|                OR     NONMATCH,GARBAGE),LSIXSTATS)
```

The routine INITIALISE initialises certain variables, such as the object area pointer, the dictionary area pointer and storage locations for bugs and field definitions, before compilation commences. An LSIX program is defined as LSIX statements which in turn are defined recursively as instructions or directives followed by other statements. In the event of failure to recognise an instruction or a directive, NONMATCH outputs a suitable message and GARBAGE skips all characters up to the end of the line. Compilation of an LSIX program is terminated by the directive '*ENTER'. The variable LINE contains the line number. STARTP, which is used below, contains the object area address of the start of the current line.

```
| DIRECTIVE IS ( OSP., EITHER '*LSIX', OPTIONS,
|                             OBJECTP:= STARTP
|                    OR     '*ENTER',LSIXEND)
```

```
| OPTIONS IS ( OSP.,(EITHER 'SOURCE',TRACEDATA
|                     OR     'LIST',LIST:=1
|                     OR      UNTRACEDATA),
|                    (EITHER  SEP,OPTIONS
|                     OR      NLS))
| SEP IS ( OSP., ',',OSP.)
```

OSP. is a built-in group for matching optional spaces. Other built-in groups include SP. for a single space, NL. for a newline and NIL. the null group. The options SOURCE and LIST following the directive *LSIX ask for source and object listings respectively. *LSIX without any options inhibits source listing. TRACEDATA switches on the data trace so giving source listing commencing with the next line of input and if LIST=1, when LSIXEND is entered at the end of compilation, then an object listing is given in octal before entering the program at the first instruction.

```
| INSTR IS (INSTRSTRT, OSP.,
|           EITHER CONDNL
|           OR     UNCONDNL
|           OR     LABEL,(EITHER CONDNL
|                          OR     UNCONDNL
|                          OR     EOL))
```

Any instruction may be labelled. The label setting is dealt with by the group LABEL which is defined below. In LSIX the GO TO <label> command is specified by the occurence of a label name at the end of an instruction. Thus if an LSIX instruction consists only of a label name that name is to be translated as a label reference and

not a label setting. The group LABEL deals with this case also. EOL matches the instruction terminator and plants descriptive information.

```
| INSTRSTRT IS (STARTP:=OBJECTP,OBJECTP:=OBJECTP+2,
|               COND:=0,NT:=0,NO:=0,GOTOFLAG:=0,
|               PLUS2(STARTP):=0)
```

INSTRSTRT assigns to STARTP the address of the first word of the next record to be constructed in the object area, it advances the object pointer by four (half)words and initialises the condition type, number of tests, number of operations, the transfer flag and word number 2 of the new record.

Types of instruction

```
| CONDNL IS (CONDITION,TESTS,
|            (EITHER 'THEN',OPERATNS OR NIL.),
|             TRANSFER,EOL)

| UNCONDNL IS ((EITHER 'THEN' OR NIL.), OPERATIONS,
|              (EITHER TRANSFER OR NIL.),EOL)
```

The literal 'THEN' which may precede the list of elementary operations in an unconditional instruction is included for compatibility with the original L6.

Types of condition

When a condition is found its type is noted in the variable COND.

```
| CONDITION IS (EITHER 'IFANY',  COND := 1
|               OR    'IFNALL', COND := 2
|               OR    'IFALL',  COND := 3
|               OR    'IFNONE', COND := 4
|               OR    'IF',     COND := 3
|               OR    'NOT',    COND := 4)
```

Analysis of tests

TESTS is a series of elementary TESTs defined in the usual manner. During execution, the conditions IFANY and IFNALL may be satisfied before all of the elementary tests have been performed, in which case control is transferred immediately to the first operation. As each test takes space in the object area and the number of tests is unknown, when no more tests are found, the address of the first operation is planted, by TESTSEND, in PLUS2(STARTP) the field reserved for this purpose by INSTRSTRT. Test codes and operand types are assigned to K and J respectively and at the end of each test, TESTEND plants the value of K in the object area in a location reserved by TESTSTRT. TESTEND also keeps a count of the elementary tests in NT, (note the difference between TESTSEND and TESTEND).

```
| TESTS IS (OSP.,TEST,EITHER TESTS OR OSP.,TESTSEND)
```

```
| TEST IS  ('(',TESTSTRT, FIELD,SEP,
|           (EITHER(EITHER 'E',K:=1
|                   OR     'N',K:=2
|                   OR     'G',K:=3
|                   OR     'L',K:=4),
|                  (EITHER 'O',J:=2
|                   OR     'H',J:=3
|                   OR         J:=0)
|            OR    (EITHER 'O',K:=5
|                   OR     'Z',K:=6),
|                  (EITHER 'D',J:=1
|                   OR     'H',J:=3
|                   OR         J:=4)
|            OR     'P',K:=7  ,J:=0),
```

This completes the first argument and the predicate. Matching continues with the separator and second argument. SEP is not used for the next separator as any spaces following the comma are significant in a 'hollerith' literal.

```
|         OSP.,',',(EITHER IF J=0,(EITHER FIELD
|                                      OR     DLITERAL)
|                   OR     IF J=1, DLITERAL
|                   OR     IF J=2, OLITERAL
|                   OR     IF J=3, HLITERAL
|                   OR     IF J=4, (EITHER FIELD
|                                         OR     OLITERAL)),
|         ')',TESTEND)
```

Literal operands and FIELD are defined below.

```
| TESTSTRT IS ( TESTP:=OBJECTP,OBJECTP:=OBJECTP+ONE)
| TESTEND IS ( NT := NT+1, COOF(TESTP):=K)
| TESTSEND IS ( PLUS2(STARTP) := OBJECTP)
```

Analysis of operations

In general, operations have either three or four arguments the second of which is the mnemonic function code but there are two special cases, (DO, symbol), the subroutine call, and (a,Δ), an abbreviation for (a,P,aΔ), with only two arguments. The analysis of an operation is performed in two passes during the first of which no object code is planted. On the first scan a shallow analysis determines the operation code (K) and the number of operands (NA). The matching process is then deliberately failed by using the group REJECT. This technique of deliberately failing an alternative is commonly used in BCL programming as a means of scanning the same data several times. Information picked up during the shallow analysis is used in the deep analysis on the second pass. OPSTART sets K and NA to zero and reserves locations for their final values which are planted by OPEND when the operation has been matched. OPEND also keeps a count of the operations in the variable NO.

```
| REJECT IS (IF 1=0)
| OPERATNS IS (OSP.,OPERATN,EITHER OPERATNS OR OSP.)
```

OPERATNS is defined in the usual way as a series of elementary operations. In the next group, OPERATN, the first alternative is a shallow analysis which attempts to match a two argument operation,

if it succeeds NA is set to 2 and a deep analysis performed. The group ARG skips all characters except comma and right bracket.

```
| OPERATN IS ('(', OPSTART, OSP.,
| (EITHER ARG, SEP, ARG, ')', NA:=2, REJECT
```

Shallow analysis for two argument operations completed, we go on to deep analysis of two argument operations if NA=2.

```
| OR     IF NA=2, (EITHER 'DO',SEP,(EITHER 'STATE',K:=41,NA:=0
|                                  OR     'DUMP',K:=42,NA:=0
|                                  OR     SYMBOL,K:=35,NA:=1
|                OR     (EITHER FIELD,SEP,REJECT
|                         OR     FIELD,SEP,
|                         OBJECTP := OBJECTP - ONE,
|                         FLDNAMES,K:=12)),OSP.,')'
```

This completes the analysis of two argument operations. 'STATE' and 'DUMP' are system subroutines. SYMBOL is defined below. Note the special technique for dealing with the operation (a,Δ) which must be expanded to its full form (a,P,aΔ) in the object area. For example (AB,CDE) is an abbreviation for (AB,P,ABCDE). Thus the field defined by AB must be matched first as the first operand and then as the first part of a sequence of addresses specifying the second operand. This is achieved by first matching FIELD, planting object code for the first operand as a side effect, and then failing the match using REJECT, so that in the next alternative FIELD matches the same sequence of names again and is followed by SEP. FIELD plants a zero terminator. This is undone by

stepping back the object pointer by one (half)word (OBJECTP:=OBJECTP-ONE). Finally the remaining sequence of names is matched by FLDNAMES and the operation code set to 12 which corresponds to the function P.

The next alternative deals with the shallow analysis of operations having more than two arguments. Note that in the following analysis NA is set to the number of operands.

```
| OR     ARG,SEP,OPCODE,SEP,ARG,NA:=2,SEP,ARG,NA:=3,
|        (EITHER  IF K=10,K:=36
|         OR      IF K=23,K:=36
|         OR      IF K=35,K:=36),REJECT
```

This completes the shallow analysis. Certain ambiguities arising in OPCODE, which deals with the function code and assigns values to K (operation code) and J (operand type), are removed once the number of operands is known. For example, DB with two operands means 'convert from decimal to binary' but with three operands it is 'define field B '. Values of K are then corrected, if necessary, before going on to the deep analysis. In the deep analysis which follows, OPCD reads the function code, OCT is a working variable and PLANT plants the value of OCT in the object area. A table of operations with the corresponding values of K is given in Appendix 7.

```
| OR     IF K LE 29,FIELD,SEP,OPCD,OSP.,',',
|        ,(EITHER IF K LE 27,
|                 (EITHER IF J=0, (EITHER FIELD
|                                         OR     DLITERAL)
|                  OR     IF J=1, DLITERAL
```

```
|                OR      IF J=2, OLITERAL
|                OR      IF J=3, HLITERAL
|                OR      IF J=4, (EITHER FIELD
|                                        OR      OLITERAL)
|                OR      IF J=5, FIELD), ')'
|          OR      IF K GT 27,
|                (EITHER FIELD OR DLITERAL),
|                (EITHER IF NA=3, ',' ,
|                       (EITHER IF J=1, DLITERAL
|                        OR      IF J=3, HLITERAL
|                        OR      IF J=4, (EITHER FIELD
|                                                OR      OLITERAL))
|                 OR      NIL.), ')'
| OR       IF K LE 31, IF K GT 29,
|                (EITHER FIELD OR DLITERAL),
|                ',',OSP.,OPCD,OSP., ',' ,
|                (EITHER IF J=3, PRPUHLIT
|                 OR      FIELD
|                 OR      OLITERAL), ')'
| OR     IF K=32,FIELD,SEP,OPCD,SEP,FLDNAME,OCT:=0,PLANT,
|              (EITHER IF NA=3, ',', (EITHER FIELD
|                                           OR      DLITERAL)
|               OR      OSP.), ')'
| OR     IF K=33, (EITHER 'S' OR 'R', K:=43),SEP,'FC',
|               SEP,FIELD, ')', NA:=1
| OR     IF K=34, (EITHER 'S' OR 'R',K:=44),SEP,'FD',
|               SEP,FLDNAME,OCT:=0,PLANT,OSP.,')',NA:=1
| OR     IF K=35, SYMBOL,SEP,'DO',SEP,SYMBOL,OSP.,')'
```

```
| OR      IF K=36, (EITHER FIELD OR DLITERAL),
|                  SEP,'D',FLDNAME,OCT:=0,PLANT,
|                  SEP,(EITHER FIELD OR DLITERAL),
|                  SEP,(EITHER FIELD OR DLITERAL),')'
| OR      IF K=37,('*',OLITERAL,SEP,OPCD,SEP,DLITERAL,
|                  SEP,'*',OLITERAL),')'
```

'*' followed by octal digits is the LSIX representation of an octal address in the Atlas computer.

```
| OR      IF K=38,FIELD,SEP,OPCD,SEP,
|                  (EITHER FIELD OR DLITERAL),
|                  (EITHER IF NA=3,SEP,FIELD OR NIL.),')',),
| OPEND)
```

The efficiency of the group OPERATN could be improved in the deep analysis (when K is known) by using a switch, with K as control variable, to select the appropriate branch, so avoiding the tests IF K=32,IF K=33, etc. However, this facility was not available at the time of this first implementation.

```
| OPCODE IS (EITHER OPCD,OPSEARCH,IF K NE 100
|            OR     'D',(EITHER LETTER OR DIGIT),
|                    K:=36, J:=6)
```

The second argument of an operation is usually the mnemonic function code. During the shallow analysis this is read into the variable OPCD and then looked up in a table of operations by OPSEARCH (a binary search) which returns values of K and J. In the

event of failure to find the code in the table K is set to 100. The code D for field definitions is dealt with separately. The next section is partly in Atlas Machine Code.

```
| OPSEARCH IS (WS1:=32, POINTER:=POINTER+WS1, OPSRCH)
| OPSRCH IS (163,WS1,0,0,                  :: Halve WS1.
|            127,WS1,0,*00000770,         :: Clear octal fraction.
|           (EITHER IF OPCD=COOF(POINTER),
|                   OCT:=PLUS1(POINTER),
|                   165,J,OCT,*7,
|                   125,J,0,0,             :: Get J.
|                   165,K,OCT,*00000770,:: Get K.
|            OR     IF WS1 NE 0,
|                  (EITHER IF OPCD GT COOF(POINTER),
|                          POINTER:=POINTER+WS1
|                   OR     POINTER:=POINTER-WS1),
|                   OPSRCH
|            OR     K:=100))
```

OPSEARCH starts by setting a POINTER to the middle of an ordered table of operation codes and the corresponding values of K and J and then calls OPSRCH which is a recursive binary search. Comments in BCL are preceded by double colon and terminated by a newline.

Types of field

```
| FIELD IS (OSP.,(EITHER 'T', TIMEFIELD
|                 OR     BUG,(EITHER FLDNAMES OR NIL.)
|                 OR     INTEGER,'.',
|                        IF INTEGER LE 128,READFIELD),
|          OSP., OCT:=0, PLANT)
```

The group FIELD was described in detail in section §1.4. TIMEFIELD plants the address of the system defined field 'T' in which time is stored. READFIELD checks that the integer read into the variable INTEGER is an integral power of 2 and plants the field address in the object area. The LSIX read-only fields are called 'T.', '1.', '2.', '4.', ..., '128.'.

```
| BUG IS (LETTER, BUGADDR)
```

BUGADDR computes and plants the address of the specified bug.

```
| FLDNAMES IS (FLDNAME, EITHER FLDNAMES OR NIL.)
| FLDNAME IS ((EITHER LETTER OR DIGIT), FLDADDR)
```

FLDADDR computes and plants the address of the definition of the specified field.

```
| LETTER IS (LTRTEST,LTR)
| DIGIT IS (DGTEST,DGT)
```

LTRTEST and DGTEST look ahead at the next character in the input

stream and test if it is a letter or a digit respectively. If it is, then the character is input to either LTR or DGT each of which is defined as a one character variable. These tests leave in OCT a character value which is used by FLDADDR to determine the address of the appropriate field definition.

Types of literal

```
| DLITERAL IS (OSP.,WS1,STCONST,PLANT,
|              OCT:=0,PLANT,OSP.)
```

Decimal literals (positive integers in the range 0,2**24 -1) are assigned to the variable WS1. STCONST enters the constant in the constants table and returns with its address in OCT which is then planted in the object area by PLANT.

```
| STCONST IS (CONSTP:=CONSTP+ONE,
|             COOF(CONSTP):=WS1,
|             OCT:=CONSTP)
```

```
| OLITERAL IS (OSP.,WS1:=0,COUNT:=8,ODIGITS,STCONST,
|              PLANT,OCT:=0,PLANT,OSP.)
```

Octal integers of not more than eight digits may be assembled and stored in the constants area. WS1 is a work space.

```
| ODIGITS IS (DIGIT, IF DGT LT 8,
|             COUNT:=COUNT-1,ASMBLODGT,
|             EITHER IF COUNT GT 0, ODIGITS
|             OR     NIL.)
```

DIGIT inputs a single decimal digit in integer form to the variable DGT. ASMBLODGT is a group of machine orders which is functionally equivalent to WS1:=8*WS1+DGT. The group ODIGITS is terminated either on finding a non octal digit or after reading eight octal digits.

```
| HLITERAL IS (WS1:=0,COUNT:=4,HCHARS,STCONST,PLANT,
|              OCT:=0,PLANT)
```

HCHARS reads up to four characters (Atlas inner set) not including newline, comma and right bracket, packs them (right justified) in the work space WS1 whence they are picked up by STCONST and stored in the constants area. The characters comma and right bracket are acceptable if written as (,) and ()) respectively, otherwise they may be written in the equivalent octal form and read by OLITERAL. The restriction to four characters is removed in the case of the output operations PRH and PUH using the group PRPUHLIT which deals with literals to be printed or punched, the length of a character string for output is limited only by the length of a line. Allowable characters are stored, one per (half) word in the constant area. A typical record for the PRH operation is shown in the object listing (LINE 2) in Appendix 1. The three operands set up are

| (a) the number of characters to be output,

| (b) the address of the first character,

| and (c) the length of the stored character string.

PRPUHLIT begins by switching to a character set which allows all characters except comma, close bracket and newline; it then counts and stores the allowable characters which are input one at a time by the group CHARS.

```
| PRPUHLIT IS (CHSET:=HCHSET,
|              OCT:=CONSTP+ONE, PLANT
|              OCT:=0, PLANT,
|              CHARS)
| CHARS IS (EITHER CHAR, STCONST,
|                      COUNT:=COUNT+1,
|                      CHARS
|           OR     WS1:=COUNT,STCONST,PLANT,
|                  OCT:=0, PLANT,
|                  NA:=NA+1 )
```

CHAR inputs a single character and stores it right justified in WS1. STCONST stores it in the constants area. When the last allowable character has been read the character count is stored and its address planted in the object area. Finally the number of operands (NA) is increased by one.

Labels, label references

LABREC(?) IS (A DICLINK, &C NAME, A ADDR, A REFADDR)

LABREC defines a label record which consists of four fields:

| (a) a link to the next label record in the name list,
| (b) the label name,
| (c) the object area address of the label,
| and (d) a link to any forward references that occur before
| the label is set.

Here again we use the concept of an indefinite group in BCL to define the structure of a record which consists of several fields. The field names DICLINK, NAME, ADDR and REFADDR, associated with LABREC, are used as selector functions.

FWDREF(?) IS (A LINK, A ADDRESS)

FWDREF defines a forward reference record of two fields the first of which is a link to the next forward reference for this label and the second is the object address at which the address is to be planted when the label is set.

```
| LABEL IS (LBL, JUNK, SP., OSP.,
|           EITHER NLS, GOTOFLAG:=1, REJECT
|           OR     IF GOTOFLAG=0, LABELSET
|           OR     IF LBL = 'DONE', GOTOFLAG:=2
|           OR     IF LBL = 'FAIL', GOTOFLAG:=3
|           OR     IF LBL = 'END' , GOTOFLAG:=4
|           OR     LABELREF)
```

Any combination of alphanumeric characters terminated by a space is accepted as a label. Only the first eight characters are significant, these are assigned to the character variable LBL. Insignificant characters are skipped by JUNK which is defined below. A label name followed by newline is interpreted as a reference to a label, i.e. it represents a transfer of control, and GOTOFLAG is set. System transfers DONE and FAIL are returns from subroutines and END is a logical end of the program.

```
| JUNK IS (EITHER JNK, JUNK OR NIL.)
```

JNK is a character variable to which insignificant characters are assigned.

```
| MATCH IS (EITHER IF CURRENT=0,
|                  SETUP(LABREC,CURRENT,DICTP),
|                  DICLINK(CURRENT) := DICP,
|                  DICP:=CURRENT,
|                  NAME(CURRENT) := LBL,
|                  ADDR(CURRENT) := 0,
|                  REFADDR(CURRENT) := 0
|           OR     IF LBL = NAME(CURRENT)
|           OR     CURRENT := DICLINK(CURRENT), MATCH)
```

MATCH compares LBL with entries in the labels dictionary. Before entry the pointer variable CURRENT points to the last entry in the dictionary. If CURRENT is zero the name in LBL is not in the dictionary so a new label record is set up by the system defined group SETUP which allocates space from an area pointed to by DICTP and assigns the address of the new record to CURRENT. The record is linked on to the labels list, in which DICP points to the last entry, and the label name recorded. Eventually MATCH is terminated with CURRENT pointing to the record required.

```
| LABELSET IS (CURRENT := DICP, MATCH,
|              EITHER IF ADDR(CURRENT) = 0,
|                     ADDR(CURRENT) := STARTP,
|                    (EITHER IF REFADDR(CURRENT) = 0
|                     OR     NCURRENT := REFADDR(CURRENT),
|                            REFADDR(CURRENT) := 0,
|                            PLUGLIST)
|              OR     O/P('LABEL ', LBL, 'SET TWICE'))
```

STARTP is the address of the current object code record. When a label is set any forward references are plugged by PLUGLIST. NCURRENT is a pointer to a forward reference record.

```
| PLUGLIST IS (COOF(ADDRESS(NCURRENT)):=STARTP,
|              NCURRENT:=LINK(NCURRENT),
|              EITHER IF NCURRENT = 0
|              OR     PLUGLIST)
```

The following groups deal with label references.

```
| TRANSFER IS (OSP., LBL, JUNK,
|              EITHER IF LBL = 'DONE', GOTOFLAG:=2
|              OR     IF LBL = 'FAIL', GOTOFLAG:=3
|              OR     IF LBL = 'END' , GOTOFLAG:=4
|              OR     LABELREF, GOTOFLAG:=1)

| LABELREF IS (CURRENT := DICP, MATCH,
|              OCT := ADDR(CURRENT), PLANT,
|              EITHER IF ADDR(CURRENT) = 0,
|                     SETUP(FWDREF,NCURRENT,DICTP),
|                     LINK(NCURRENT):=REFADDR(CURRENT),
|                     ADDRESS(NCURRENT):=OBJECTP,
|                     REFADDR(CURRENT):=NCURRENT
|              OR     NIL.)
```

If a label has not been set, a record of the forward reference is 'SETUP' and inserted in the pluglist.

Subroutines are entered by means of the (DO, symbol) operation where the symbol is the label, or name, of the entry point. The two system subroutines 'STATE' and 'DUMP' do not use the normal subroutine entry and return.

```
| SYMBOL IS (LBL, JUNK,
|            EITHER IF LBL = 'STATE', K:=41, NA:=0
|            OR     IF LBL = 'DUMP' , K:=42, NA:=0
|            OR     LABELREF, OCT:=0, PLANT)
```

Some miscellaneous groups

```
| PLANT IS (COOF(OBJECTP) := OCT,
|           OBJECTP := OBJECTP + ONE)
```

Information stored in OCT is planted in the object area by PLANT and the object pointer is advanced one word.

```
| EOL IS (OSP., NL., EITHER EOL OR INSTREND)
```

An LSIX instruction is terminated by one or more newlines. INSTREND packs descriptive information - COND, NT, NO and GOTOFLAG, into a description word which is stored in the first word of the current object record.

```
| NLS IS (OSP., NL., EITHER NLS OR NIL.)
```

NLS is similar to EOL but no information is planted.

```
| LSIXEND IS (FINISH:=OBJECTP, OBJECTP:=START,
|             (EITHER IF LIST=1, OBJECTPRINT
|              OR     NIL.),     INTERPRET)
```

Compilation is completed and, if requested, an object listing is output in octal, before the interpreter is entered and execution commenced.

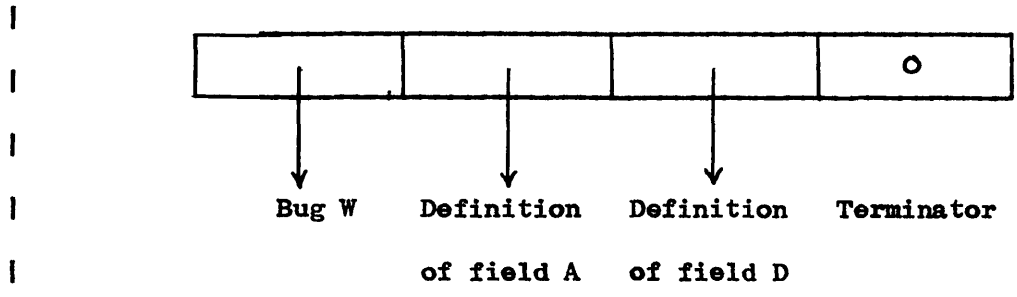
§2.2 The execution of an LSIX program

During the analysis and recognition of LSIX source instructions descriptive information is planted in the object area. For each source instruction this information includes a description word, the address of the next description word in sequence, i.e. the start of the next instruction, the address of the first operation in the current instruction and the line number. The description word gives the type of instruction (conditional or unconditional), the number of tests, the number of operations and the type of transfer of control (normal transfer, subroutine return) if any. This information is packed as follows

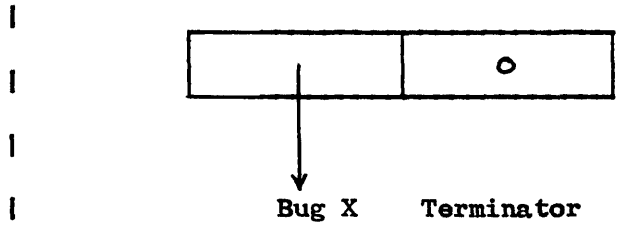
Bits 0-2	Type of instruction	0 for unconditional
		1 - IFANY
		2 - IFNALL
		3 - IFALL, IF
		4 - IFNONE, NOT
Bits 3-8	Number of Tests	0 - 63
Bits 9-14	Number of Operations	0 - 63
Bits 15-20	Type of Transfer	0 for no transfer
		1 - normal transfer
		2 - normal subroutine exit (DONE)
		3 - fail subroutine exit (FAIL)
		4 - logical end of program (END)

For each test and operation is stored the test or operation code (values of K in Appendix 7) and the addresses of operands. The outline flow diagram in Fig.II.2.1 describes the operation of the main interpreter routine which is written in BCL.

Each operand, whether a base field (bug), remote field or a constant (decimal, octal or hollerith literal), is specified by a sequence of one or more addresses terminated by a zero. For example the remote field WAD is represented by the sequence of pointers



and the basefield X by



Constants are stored in a constants area and referred to by their addresses.

A field is defined at run time by its word number, left-most bit and right-most bit. For example the operation (2,D6,3,17) defines field 6 of any data block as bits 3 through 17 of word number 2. The execution of such an operation results in the setting up of a field definition, including a 24 bit mask, which is used by the routines for fetching and storing the contents of fields. Because of the complete generality of field definitions no attempt is made to use the few special hardware facilities for handling special cases. The only special case which might have been worth detecting is the field which spans the full 24 bits of the word.

Three general field handling subroutines FINDFIELD, GETFIELD and STOREFIELD are used during the execution of fetch and store operands.

Any field in the data structures may be specified by two pointers one to the first word of the block containing the field and the other to the definition of the field concerned. Basefields and constants are referred to by the first of these pointers and the second is conventionally zero. The routine FINDFIELD given below determines, from a sequence of addresses in the object area, the values of the two pointers specifying an operand.

Subroutine to find a field

On entry OBJECTP points to the first of a sequence of addresses in the object area. The routine is terminated when the location to which OBJECTP points contains a zero i.e. when COOF(OBJECTP) = 0. On exit, for a remote field WREG1 points to the block containing the specified field and WREG2 to the definition of the field; for basefields and constants WREG1 points to the basefield or constant and WREG2 is zero.

```
| DEFINE  R  FINDFIELD
| DO
| WREG2     := 0
| WREG1     := COOF(OBJECTP)          Fetch first address.
| OBJECTP   := OBJECTP + ONE          Advance object pointer by one word.
| IF COOF(OBJECTP) = 0 GO TO END      If next address is zero go to end.
| WREG1     := COOF(WREG1)            Get address of block to which bug
                                      points.
| MORE) WREG2 := COOF(OBJECTP)        Get address of next field definition.
| OBJECTP   := OBJECTP + ONE          Advance object pointer.
| IF COOF(OBJECTP) = 0 GO TO END      If next address is zero go to end.
| GETFIELD                           Routine to get contents of field
| GO TO MORE                         specified by WREG1 and WREG2.
| END) OBJECTP := OBJECTP + TWO       Advance object pointer to next item
| RETURN                             of information (i.e. step over
| END                                zero).
```

Subroutine to fetch the contents of a field

The subroutine FINDFIELD calls GETFIELD which is defined below. GETFIELD fetches the contents of the field which is specified by the two pointers WREG1 and WREG2 in the usual way. On exit WREG1 contains the contents of the specified field right justified (not all fields are 24 bits in width) and WREG2 is unchanged.

```
| DEFINE   R   GETFIELD
| DO
| IF WREG2 NE 0 GO TO REMOTE  If not basefield or constant go
| WREG1 := COOF(WREG1)        to remote, otherwise get contents
| RETURN                      and return.
| REMOTE) WREG1 := WREG1 + WORD(WREG2)
|                             Address of word containing field.
| WREG1 := COOF(WREG1)        Get word including field.
| WREG3 := MASK(WREG2)        Get mask from field definition.
| 127,WREG1,WREG3,0           Machine order to get field from word.
| SHIFT := 23 - RBIT(WREG2)   Determine any right shift required.
| IF SHIFT = 0 GO TO END      If right justified go to end.
| 1342,WREG1,SHIFT, 0         Right justify.
| END) RETURN                 Return with field in WREG1.
| END
```

Subroutine to store a field

The item to be stored is held in the variable OCT. The field in which the item is to be stored is specified in the usual way by WREG1 and WREG2.

```
| DEFINE   R   STOREFIELD
| DO
| IF WREG2 NE 0 GO TO REMOTE        WREG2 is zero for basefields.
| IF WREG1 LT BUGBASE GO TO ERROR   Protect read only fields.
| COOF(WREG1)  :=  OCT              Store item in basefield.
| RETURN
| REMOTE) SHIFT := 23-RBIT(WREG2)   Determine necessary shift.
| 1343,OCT,SHIFT,0                  Shift operand into position.
| WREG3  :=  MASK(WREG2)            Get mask from field definition.
| 127,OCT,WREG3,0                   Clear most significant bits if
|                                   item too long.
| WREG1  :=  WREG1 + WORD(WREG2)    Address of word containing field.
| WREG2  :=  COOF(WREG1)            Fetch present contents of word.
| 126,WREG3,0,*77777777             Complement mask.
| 127,WREG2,WREG3,0                 Clear field to receive new item.
| 167,WREG2,OCT,0                   Write item into specified field.
| COOF(WREG1)  :=  WREG2            Store field.
| END) RETURN
| ERROR) O/P ('ATTEMPTING TO WRITE TO READ FIELD')    Error message.
| RETURN
| END
```

The efficiency of LSIX depends largely upon the efficiency of these three field handling routines which are used for all operands. It is important to make use of any special hardware facilities which exist for shifting operands and the LSIX user is encouraged to define fields which may be handled efficiently by the hardware of the machine on which the program is to be run. In his original L6 compiler for the IBM 7090 computer, Knowlton recompiles at run time the routines to fetch and store operands each time that a field is defined or redefined, making use of special hardware facilities where possible. On the Atlas computer the only shift instructions, apart from the inefficient extra-codes, are the circular shift one bit right and circular shift six bits left. For this reason it was decided to use the same three general field handling routines for all operands. The facility for defining and redefining general fields in L6 is the main justification for the interpretive nature of the Atlas LSIX compiler.

Compilation of an LSIX program is terminated when the directive *ENTER is recognised. If the LIST option has been specified by means of an earlier directive, *LSIX LIST, then the object program is listed in octal by calling the routine STACKPRINT (see Appendix 3). A typical object listing is shown in the complete example of an LSIX program in Appendix 1. On entry to the routine INTERPRET the first operations throw away those parts of the compiler which are no longer required, the object pointer is initialised and execution commences.

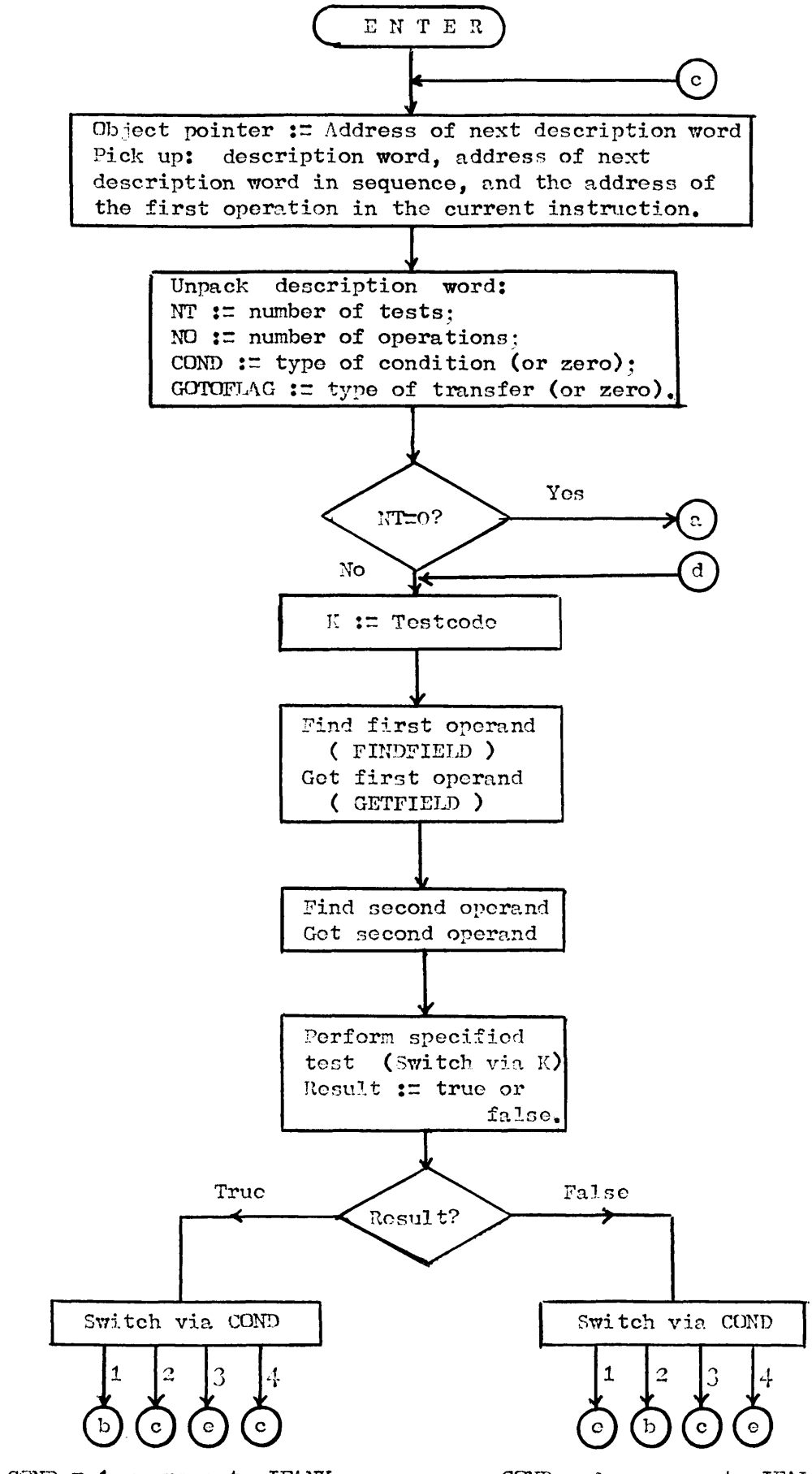
The interpreter and associated routines

The operation of the interpreter routine is described in outline by the flow diagram in Fig.II.2.1. The following anotated BCL program is a more detailed specification. In general, tests and operations involve two operands but some involve more than two. The routines FINDFIELD and GETFIELD described in the previous section are used to locate and fetch the operands. The address and value of the first operand are assigned to the variables

BP1	-	pointer to block
FP1	-	pointer to field definition (zero for basefields and constants)
and OP1	-	contains field (fetched only if required)

and the second and third operands are assigned to BP2, FP2, OP2 and BP3, FP3, OP3 respectively. These values are then ready for use in the various sets of BCL instructions corresponding to the LSIX tests and operations.

In addition to the field handling routines already described several other basic routines are called from the main interpreter routine. These are described in Appendix 3. The reader is reminded that routines with parameters were not available in BCL when this compiler was written.



COND = 1 represents IFANY
COND = 2 represents IFNALL
COND = 3 represents IFALL, IF
COND = 4 represents IFNONE, NOT

Fig. II.2.1(a). The operation of the Interpreter Routine

(Continued on Page 63)

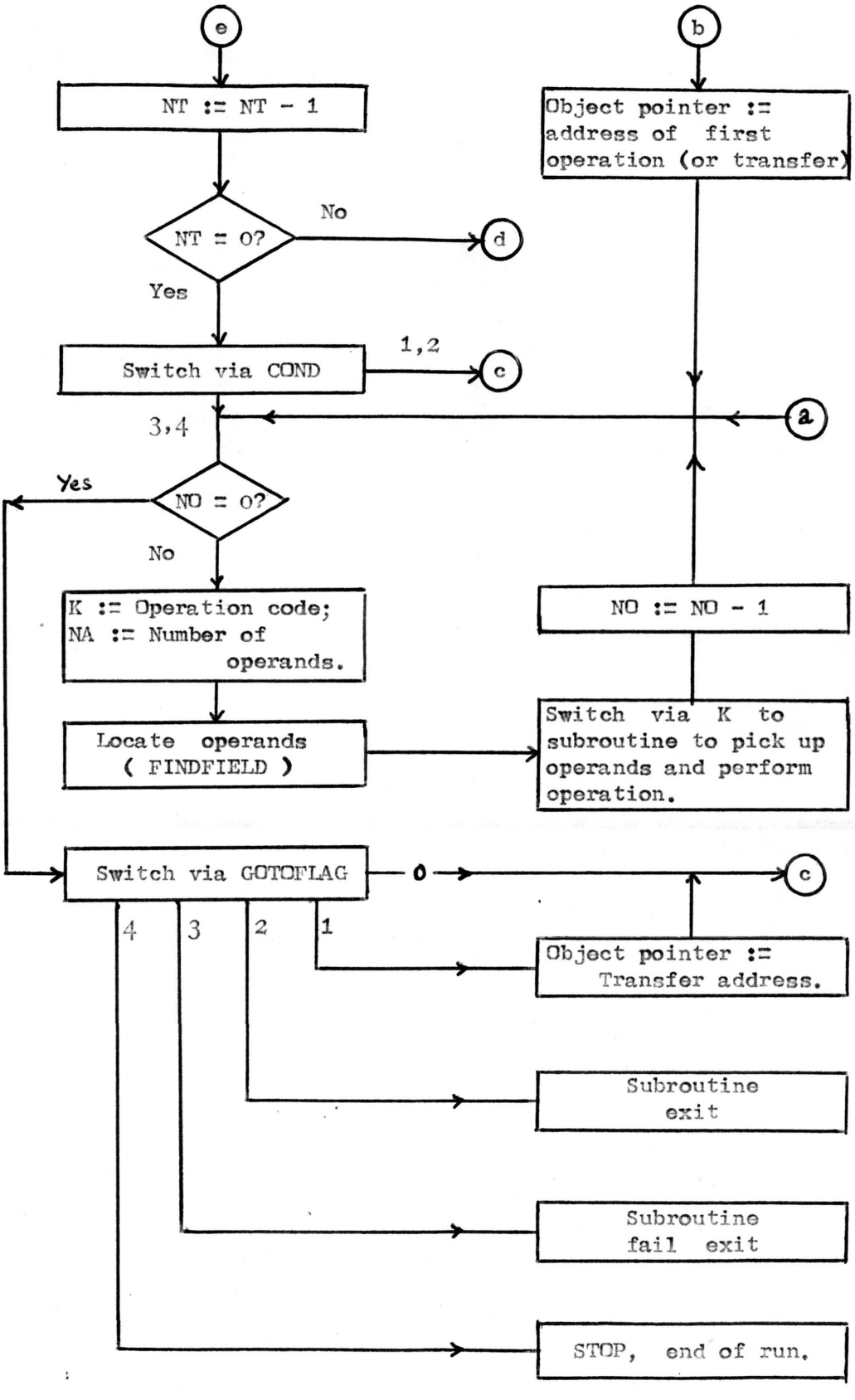


Fig. II.2.1(b). The operation of the Interpreter Routine.

(Continued from Page 62)

```
DEFINE  R  INTERPRET

DO

121,WR1,0,*1001

1142,WR1,0,(NEXT)                    :: Lose routines no longer required.

NEXT) 1065,0,0,5,0                   :: Space to top of next page.

O/P (NL.(2), 'LSIX PROGRAM ENTERED', NL.(2))

NDESCRWD := START                    :: Initialise pointer to next
                                     :: instruction.
GAMMA) OBJECTP := NDESCRWD           :: Get pointer to next instruction.

DESCRWD := COOF(OBJECTP)             :: Get description word.

NDESCRWD := PLUS1(OBJECTP)           :: Get address of next description
                                     :: word in sequence.
FIRSTOP := PLUS2(OBJECTP)            :: Get address of first operation in
                                     :: current instruction (zero for
                                     :: unconditional instruction).
LINE := PLUS3(OBJECTP)               :: Get line number.

OBJECTP := OBJECTP + 2               :: Advance object pointer to next
                                     :: item.


:: The following instructions deal with LSIX tests.


165, NT, DESCRWD, *077               :: Get number of tests.

IF NT = 0 GO TO ALPHA                :: If unconditional go to operations.

125, NT, 0,0                         :: Convert to 21 bit integer.

125,NT, 0,0

165, COND, DESCRWD, *7               :: Get condition (1,2,3 OR 4).

125, COND, 0,0                       :: Covert to 21 bit integer.

BETA1) K := COOF(OBJECTP)            :: Get test code.

OBJECTP := OBJECTP + ONE             :: Increment object pointer.

FINDFIELD                            :: Locate first operand.

GETFIELD                             :: Get first operand.

OP1 := WREG1                         :: Save operand in OP1.

FINDFIELD                            :: Locate second operand.
```

```
GETFIELD                          :: Get second operand.
OP2 := WREG1                      :: Save second operand in OP2.

GO TO E, N, G, L, O, Z, P VIA K   :: Switch via testcode.
E)    IF OP1 = OP2 GO TO TRUE     :: If operands equal go to true
      GO TO FALSE                 :: otherwise false.
N)    IF OP1  = OP2 GO TO FALSE   :: If equal go to false
      GO TO TRUE                  :: otherwise true.
G)    IF OP1 GT OP2 GO TO TRUE    :: If greater than, go to true
      GO TO FALSE                 :: otherwise false.
L)    IF OP1 LT OP2 GO TO TRUE    :: If less than, go to true
      GO TO FALSE                 :: otherwise false.
O)    127, OP1, OP2, 0            :: OP1 := OP1 and OP2 and test
      GO TO E                     :: equality for test 'one' bits.
Z)    167, OP1, OP2, 0            :: OP1 := OP1 or OP2 and test
      GO TO E                     :: equality for test 'zero' bits.
P)    GO TO E                     :: Test equality of pointers.
TRUE) RESULT:=1
      GO TO CONDSPLT              :: Go to switch via condition.
FALSE)RESULT:=0
CONDSPLT) GO TO ANY,NALL,ALL,NONE VIA COND     :: Switch.
ANY)  IF RESULT =0 GO TO BETA     :: If false go on to next test.
      OBJECTP := FIRSTOP          :: If true, get address of first
      GO TO ALPHA                 :: operation and go to it.
NALL) IF RESULT NE 0 GO TO BETA   :: If true go on to next test.
      OBJECTP := FIRSTOP          :: if false, go to operations.
      GO TO ALPHA
ALL)  IF RESULT NE 0 GO TO BETA   :: If true go to next test
      GO TO GAMMA                 :: otherwise go to next instruction
                                  :: in sequence.
```

```
NONE) IF RESULT NE 0 GO TO GAMMA :: If true go to next instruction.

BETA) NT := NT-1                 :: Decrease number of tests.

      IF NT NE 0 GO TO BETA1     :: If more tests go to next test.

      IF COND LE 2 GO TO GAMMA   :: if ANY or NALL go to next
                                 :: instruction.


:: The following instructions deal with LSIX operations.

ALPHA)  165,NO,DESCRWD,*00077    :: Get number of operations
                                 :: from description word.
        125,NO,0,0               :: Convert to 21-bit integer.

        125,NO,0,0

        125,NO,0,0

MOREOPS) IF NO = 0 GO TO NOOPS   :: If no more operations.

        K := COOF(OBJECTP)       :: Get operation code K.

        NA := PLUS1(OBJECTP)     :: Get number of operands.

        OBJECTP := OBJECTP + 1   :: Advance object pointer.

        IF NA = 0 GO TO OPSPLIT  :: If no operands

        FINDFIELD                :: Find first operand.

        BP1 := WREG1             :: Save address pointers..

        FP1 := WREG2

        IF K GE 25 GO TO GETOP1  :: Get first operand only if

        IF K LT 9 GO TO GETOP1   :: necessary.

        GO TO SKIPOP1

GETOP1) GETFIELD                 :: Fetch first operand

        OP1 := WREG1             :: Save first operand.

SKIPOP1) NA := NA-1              :: Decrease number of operands.

        IF NA = 0 GO TO OPSPLIT  :: If no more operands.

        FINDFIELD                :: Find second operand.

        BP2 := WREG1             :: Save address.

        FP2 := WREG2
```

```
        GETFIELD                    :: Get second operand.
        OP2 := WREG1                :: Save second operand.
        NA := NA-1                  :: Decrease number of operands.
        IF NA = 0 GO TO OPSPLIT     :: If no more operands.
        IF K = 38 GO TO GT          :: Third operand of GT operation
                                    :: is special.
        FINDFIELD                   :: Find third operand.
        BP3 := WREG1                :: Save address.
        FP3 := WREG2
        GETFIELD                    :: Get third operand.
        OP3 := WREG1

:: The operands are now ready for use in the various sets of
:: instructions corresponding to the LSIX operations.  The
:: appropriate instructions are entered by means of a switch
:: using the operation code K as control variable.

OPSPLIT)IF K LE 44 GO TO OPSPLIT1
OPERR)  O/P (NL.,'ILLEGAL FUNCTION IN LSIX OPERATION' ,NL.)
        STATEPRINT                  :: Output state of system.
        O/P (NL.(2), 'JOB TERMINATED')
        STOP                        :: Error halt.

OPSPLIT1) GO TO IC,ADD,SUB,MPY,DIV,OR,AND,XOR,C,DP,
                EQ,OPP,LO,LZ,RO,RZ,OS,ZS,BZ,ZB,
                BD,BD, DB,DB,OP,FR,IN,L,R,PR,
                PU,PL,SFC,SFD,DO,D,SS,GT,OPERR,OPERR,
                STATE,DUMP,RFC,RFD    VIA K
```

```
:: Control has now been transferred to the set of instructions
:: for the operation specified by K.  Details of these instructions
:: are given in Appendix 3.  After execution of these instructions
:: control is returned to OPRTN.

OPRTN)  NO := NO -1                 :: Decrease number of operations.
        OP1 := 0                    :: Reset working variables to zero.
        OP2 := 0
        OP3 := 0
        BP1 := 0
        BP2 := 0
        BP3 := 0
        FP1 := 0
        FP2 := 0
        FP3 := 0
        GO TO MOREOPS               :: Go back to execute any further
                                    :: operations.
:: When all operations have been obeyed control is transferred
:: in one of several ways according to the transfer code stored
:: in the description word.  The following instructions deal
:: with the transfer of control.

NOOPS)165,GOTOFLAG,DESCRWD,7        :: Get transfer code (0,1,2,3 or 4).
      IF GOTOFLAG = 0 GO TO GAMMA   :: If no transfer go to next
                                    :: instruction in sequence
                                    :: otherwise switch via GOTOFLAG
      GO TO FLAG1,DONE,FAIL,FLAG4 VIA GOTOFLAG
FLAG1)NDESCRWD := COOF(OBJECTP)     :: Get transfer address.
      GO TO GAMMA                   :: Continue with instruction
                                    :: specified.
```

```
:: The following instructions deal with subroutine returns.

DONE)   SUBP := SUBP-3              :: Pop-up subroutine stack.
        SUBL := SUBL-1              :: Decrease subroutine level
                                    :: number.
        OBJECTP := COOF(SUBP)       :: Restore object pointer.
        NDESCRWD := PLUS1(SUBP)     :: Restore address of next
                                    :: description word.
        DESCRWD := PLUS2(SUBP)      :: Restore description word.
        NO := PLUS3(SUBP)           :: Restore number of remaining
                                    :: operations.
        LINE := PLUS5(SUBP)         :: Restore line number.
        GO TO OPRTN                 :: Normal operation return.

FAIL)   SUBP := SUBP-3              :: Pop-up subroutine stack.
        SUBL := SUBL-1              :: Decrease subroutine level
                                    :: number.
        NDESCRWD := PLUS4(SUBP)     :: Get 'fail' transfer address.
        GO TO GAMMA                 :: Continue with specified
                                    :: instruction.

:: End of subroutine return instructions.

FLAG4)  O/P(NL.,'END OF PROGRAM')
        STOP                        :: Logical end of program.
        END                         :: End of interpreter routine.
```

The storage allocation routines which are called by the interpreter routine are described in section §2.3 and routines for other LSIX operations are given in Appendix 3 .

§2.3 Storage allocation and collection in LSIX

In this section we describe several variations on a method for organising the free space allocator in LSIX. Routines for setting up the free space lists, getting new blocks, returning blocks which are no longer required and automatic garbage collection are described.

An important feature of LSIX is the availability of several different sizes of blocks which may be linked together by pointers stored in fields which the programmer himself defines. Any field which is of address length may contain a pointer and the contents of a field are interpreted according to the context in which they are used. Consequently it is difficult to collect garbage automatically as the system does not know which fields contain pointers and the responsibility for freeing blocks which are no longer in use is usually left to the programmer. A garbage collector which has been written for the Atlas LSIX is described in this section.

The available blocks in LSIX are in general of size 2**n words where n is an integer in the range 0-7. The choice of block size being an integral power of two, blocks are easily halved to form smaller blocks (called mates) and when two consecutive blocks are free simultaneously they could be recombined to form a larger block; better, when two mates are free they may be recombined. The free space is organised as a number of separate simple lists, one for each size of block. On being freed, a block is returned to the appropriate list. When a block is asked for there are three possibilities:

(a) the appropriate list is not empty in which case a block is immediately available;

(b) the list is empty but a larger block exists on another list and this can be split to provide a block of the required size;

and (c) the list is empty and there are no larger blocks available for splitting.

In the latter case all is not lost. It is possible that smaller blocks in contiguous parts of the store are free simultaneously and can be recombined to provide a block of the required size.

In the Atlas LSIX implementation, the list head of each free space list consists of four fields, each being 24 bits long, containing the following information:

field 1: Pointer to the first block in the list (0 if empty),

field 2: The size of the data blocks on this list.

field 3: log (size) i.e. a 3 bit integer in the range 0-7.

field 4: The potential number of blocks of this size.

The potential number of blocks of any size is the number of blocks on the list plus the number of blocks that could be obtained by splitting all larger blocks down to this size. The fields containing the potential numbers of blocks are the read-only fields called 1. ,2. ,4.,- - -, 128. in LSIX. The size of any block which is in use must be available to the system for use in the free block and duplicate block operations and log2(size) provides a very compact form of storage.

The list heads of the eight free space lists occupy consecutive storage locations and may be accessed (by the system) via the link variable FREEHDR, a constant pointer to the list head of 1-blocks. Each free space list is a simple linked list terminated by a zero link.

Fig. II.3.1 shows how the free space lists are stored.

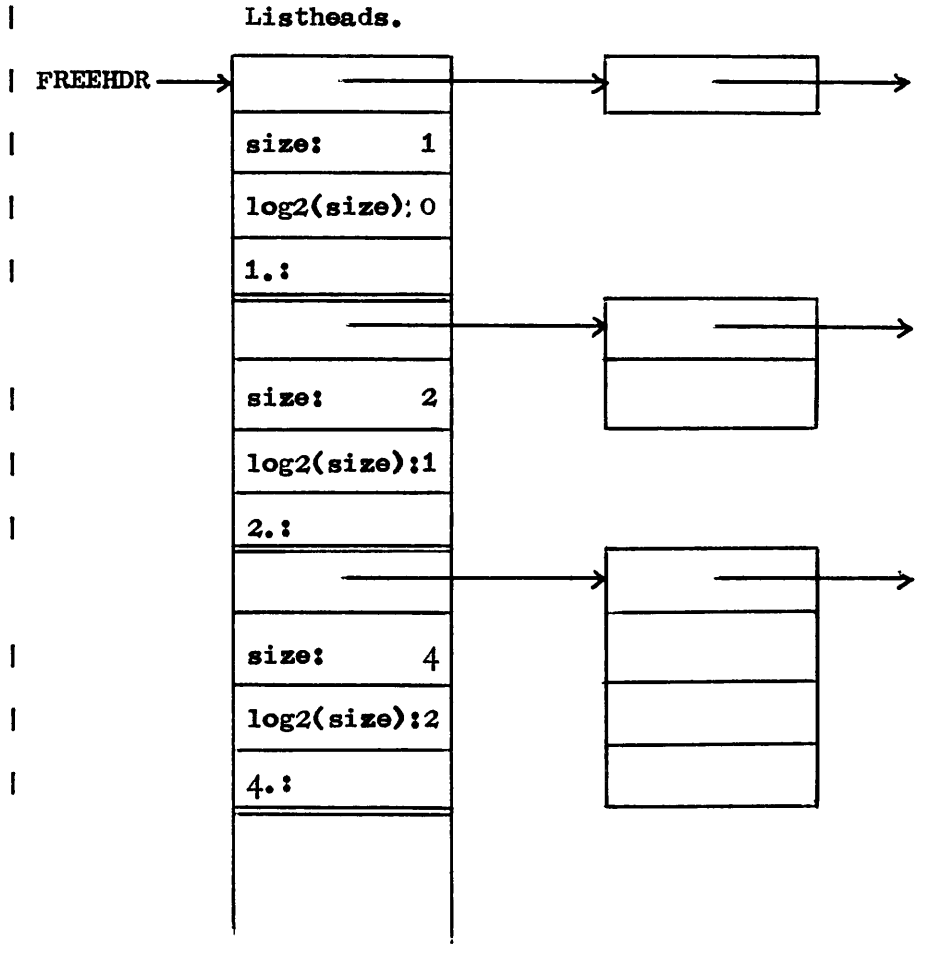


Fig. II.3.1

The four LSIX operations which affect the state of the free space lists are: Set up Storage (SS), Get a block (GT), Free a block (FR) and Duplicate a block (DP).

Set up Storage

One of the first operations in any LSIX program is to set up a list of free space using the set up store operation which has the form

(s1,SS,d,s2)

where d is the size of block to be set up and s1 and s2 are the addresses of the beginning and end of the storage area to be linked in blocks of the specified size. For example the operation (*20000000,SS,8,*20040000) causes 2048 words of Atlas store (i.e. 4096 LSIX words) to be linked together in blocks of size 8 (24 bit) words. The specified value of d need not be an integral power of 2 but the size of block actually set up is the smallest integral power of 2 which is greater than or equal to d. It is intended that in a future version of the compiler the user may specify and actually get any size of block between 1 and 128 (24 bit)words.

When setting up storage the appropriate free space listhead is located by a routine FINDLIST which, starting with the listhead for 1-blocks, searches for the first listhead for which the block size is not less than d and returns with a pointer to that listhead and with the value of d corrected. The addresses s1 and s2 specified by the user are interpreted as relative addresses and the list storage area always starts at the octal address *20000000. This implementation differs from the original L6 in that set up store operations may be executed several times during the running of a program, so setting up additional linked space, and the end of the

appropriate free space list must be found before new blocks are linked on. Details of the SS and other storage organisation routines are given at the end of this section.

Get Block

There are two forms of the get block operation:

| (a1,GT,cd) and (a1,GT,cd,a2)

- Get a block of the size specified by cd and plant its address in the field a1. After getting the block, assign to the field a2 (if this is specified) the previous contents of the field a1. The GET routine uses FINDLIST to locate the appropriate free space list. Thus the size specified may be any integer in the range 1-128 but the size of block actually allocated is at present an integral power of 2. If the free space list located is not empty, then a block of the required size is detached, otherwise the potential number of blocks of this size is examined. If the potential number is not zero then a larger block exists and the routine SUBDIVIDE is called to split the first available larger block. In the event of the potential number being zero the routine RECOMBINE is called to search for smaller blocks which being free simultaneously may be recombined to give a block of the required size. Whenever RECOMBINE is entered it searches all free space lists from the 1-blocks upwards performing all of the recombinations possible up to the maximum size that has been set up by the program. After this, if the potential number of blocks of the required size is still zero then the program is abandoned. It is not possible to relocate smaller blocks in an attempt to recombine.

Duplicate Block

The duplicate block operation (a,DP,c) gets a new block (using GET) of the same size as that to which field c points and assigns its address to field a. It then copies the given block word for word into the new block.

Free Block

The operations (a1,FR,0) and (a1,FR,a2) are used to free the block to which field a1 points, i.e. to return the block to the appropriate free space list. The contents of field a2 are assigned to field a1 but if a2 is not specified a1 is set to zero.

Neither the Duplicate block nor the Free block operations specify the size of block involved. For this reason the system must keep a note of the sizes of all data blocks which are in use. A convenient place to store the size is in the data-block itself. Bits 21-23 of word 0 of every block are reserved for the system. In these three bits the size of the block is stored in the form log2(size). Later in this section we describe briefly another Atlas implementation of LSIX in which the whole of word 0 is made available to the user and the size is stored in a separate part of the store. When a block is freed or duplicated the actual size is found from log2(size) using a routine FINDSIZE which searches the free space list heads for log2(size) starting with the 1-block list head and locates the required free space list at the same time.

Whenever the state of the free space lists is changed by one of the LSIX operations described above the potential numbers of blocks must be updated. This is performed by the routine UPDTNDOT. Details of this and other storage organisation routines are given below.

```
      DEFINE R FINDLIST

:: The input to this routine is the size of a block.  The output
:: is a POINTER to the appropriate list and the corrected size.

      DO

      POINTER := FREEHDR - 2                  :: Initialise pointer.

FL1)  POINTER := POINTER + 2                  :: Point to next list.

      IF PLUS1(POINTER) LT SIZE GO TO FL      :: If list not yet found.

      SIZE := PLUS1(POINTER)                  :: Correct the size.

      RETURN

      END

      DEFINE R FINDSIZE

:: The input to this routine is LOGSIZE, the output is SIZE and a
:: POINTER to the appropriate list.

      DO

      POINTER := FREEHDR - 2                  :: Initialise.

FS)   POINTER := POINTER + 2                  :: Point to next list.

      IF PLUS2(POINTER) LT LOGSIZE GO TO FS   :: If not found.

      SIZE := PLUS1(POINTER)                  :: Get size.

      RETURN

      END

      DEFINE R UPDTNDOT

:: The value of WR2, set before entry, is the change in potential
:: number.

      DO

      WR1 := POINTER                          :: Copy pointer.

UPDT) PLUS3(WR1) := PLUS3(WR1) + WR2          :: Update potential number.

      IF WR1 = FREEHDR GO TO END              :: If finished.

      124,WR2,WR2,0                           :: Double WR2.

      WR1 := WR1 - 2                          :: Point to next list.

      GO TO UPDT

END)  RETURN

      END
```

```
        DEFINE R SETUPSTORE
        DO
        SIZE := OP2                          :: Get size of block.
        FINDLIST                             :: Locate list for given
        IF MAXSIZE GE SIZE GO TO SKIP        :: blocksize.
        MAXSIZE := SIZE                      :: Note maximum size of block
                                             :: setup.
SKIP)   124,SIZE,SIZE,0                      :: Convert size to 22 bit
                                             :: integer, i.e. to unit of
        124,SIZE,SIZE,0                      :: halfwords of Atlas store.
        WS1 := POINTER                       :: Point to appropriate free
                                             :: space list. POINTER was set
                                             :: by FINDLIST.
LINKBL1) IF COOF(WS1) = 0 GO TO LINKBL2      :: If end of list
        WS1 := COOF(WS1)                     :: Step down list.
        GO TO LINKBL1
LINKBL2) COOF(WS1) := ENDLIST                :: ENDLIST points to next block
                                             :: to be set up, initialised
                                             :: before entry to program.
        TOBELNKD := OP3 - OP1                :: s2 - s1 gives amount to be
                                             :: linked.
        ENDLIST := ENDLIST + TOBELNKD        :: Advance ENDLIST.
        COUNT := 0                           :: Initialise count.
LINKNEXT) WS1 := COOF(WS1)                   :: Point to next new data block.
        COUNT := COUNT + 1                   :: Count new block.
        COOF(WS1) := WS1 + SIZE              :: Plant link to next new block.
        IF COOF(WS1) LT ENDLIST GO TO LINKNEXT
        COOF(WS1) := 0                       :: If finished, terminate list.
        WR2 := COUNT                         :: Set parameter for UPDTNDOT.
        UPDTNDOT                             :: Update potential numbers of
        RETURN                               :: blocks.
        END
```

```
        DEFINE R GET
        DO
        SIZE := OP2                     :: Get block size.
        FINDLIST                        :: Locate list, correct size.
        OP3 := OP1                      :: Save present contents of a
        LOGSIZE := PLUS2(POINTER)       :: Get log2(size).
        GETBLOCK                        :: Get block.
CLEAR)  COOF(WS1) := 0                  :: Clear block to zero.
        WS1 := WS1 + ONE
        122,SIZE,0,0.1
        IF SIZE GT 0 GO TO CLEAR
        COOF(OCT) := LOGSIZE            :: Record size in block.
        IF NA = 0 GO TO END             :: If previous pointer not
                                        :: to be saved.
        FINDFIELD                       :: Find field in which to
                                        :: store previous pointer.
        OCT := OP3
        STOREFIELD                      :: Store previous contents
                                        :: of pointer to new block.
END)    RETURN
        END
```

```
        DEFINE R GETBLOCK
:: This routine is called from GET and from DUPLICATE.
:: On entry POINTER points to the appropriate free space list and SIZE
:: contains the size.
        DO
        IF COOF(POINTER) NE 0 GO TO GTBLOCK :: If list not empty, get
                                            :: block.
        IF PLUS3(POINTER) GT 0 GO TO SPLIT  :: If potential number > 0.
        RECOMBINE                           :: Attempt recombination.
        IF COOF(POINTER) NE 0 GO TO GTBLOCK :: Try again.
        IF PLUS3(POINTER) GT 0 GO TO SPLIT
        O/P(NL.,'FREE SPACE EXHAUSTED   JOB TERMINATED')
        STOP                                :: Stop if space exhausted.
SPLIT)  SUBDIVIDE                           :: Call routine to split
                                            :: larger blocks.
GTBLOCK) WS1 := COOF(POINTER)               :: Point to block.
        COOF(POINTER) := COOF(WS1)          :: Detach block from list.
        OCT := WS1                          :: Address in OCT.
        STOREIN1                            :: Store address of block.
        WR2 := 0-1                          :: Set WR2 for UPDTNDOT.
        UPDTNDOT                            :: Update potential
                                            :: numbers.
        RETURN
        END
```

```
        DEFINE R SUBDIVIDE
        DO
NEXT)   POINTER := POINTER + 2              :: Examine list of next
                                            :: larger blocks.
        IF COOF(POINTER) = 0 GO TO NEXT     :: If empty, try next.
        WS1 := COOF(POINTER)
        COOF(POINTER) := COOF(WS1)          :: Detach first block.
HALVE)  PLUS3(POINTER) := PLUS3(POINTER)-1  :: Adjust potential number.
        IF PLUS1(POINTER) = SIZE GO TO END  :: If required size.
        POINTER := POINTER - 2              :: Step back to list of next
                                            :: smaller block size.
        COOF(POINTER) := WS1
        COOF(WS1) := 0                      :: Link first half block to
                                            :: current list.
        WS2 := PLUS1(POINTER)               :: Get current block size.
        124,WS2,WS2,0
        124,WS2,WS2,0                       :: Convert to address units.
        WS1 := WS1 + WS2                    :: Point to second half of
                                            :: block.
        GO TO HALVE
END)    COOF(WS1) := COOF(POINTER)
        COOF(POINTER) := WS1                :: Link second half on to
                                            :: list.
        RETURN
        END
```

```
        DEFINE R RECOMBINE
:: This routine starts with the list of free 1-blocks and for each
:: block searches the list for its mate. If its mate is found the
:: two blocks are recombined and transferred to the list of next
:: larger blocks. The process continues for each list in turn until
:: the maximum size of block is reached.
        DO
        WS3 := FREEHDR                          :: Start with 1-blocks.
NEXTLIST) WS1 := WS3                            :: Point to next listhead.
        WS3 := WS3 + 2
        IF PLUS1(WS1) GE MAXSIZE GO TO END      :: If finished.
        CSIZE := PLUS1(WS1)                     :: Get current size.
        124,CSIZE,CSIZE,0                       :: Convert size to address
                                                :: units.
        124,CSIZE,CSIZE,0
        WS2 := WS1                              :: Initialise working
                                                :: pointer.
NEXTBLOCK) IF COOF(WS2) = 0 GO TO NEXTLIST      :: If list finished.
        WR1 := COOF(WS2)                        :: Address of next block.
        126,WR1,CSIZE,0                         :: Address of mate.
        WR2 := COOF(WS2)                        :: Initialise working
                                                :: pointer.
TESTMATE) IF COOF(WR2)=0 GO TO NOMATE           :: If no mate in list.
        IF WR1 = COOF(WR2) GO TO MATE           :: If mate found.
        WR2 := COOF(WR2)                        :: Step down list.
        GO TO TESTMATE
```

```
NOMATE) WS2 := COOF(WS2)                    :: Point to next block.
        GO TO NEXTBLOCK
MATE)   167,WR1,CSIZE,0
        126,WR1,CSIZE,0                     :: Address of recombined
                                            :: block.
        COOF(WR2) := COOF(COOF(WR2))
        COOF(WS2) := COOF(COOF(WS2))        :: Detach two halves from
                                            :: current list.
        COOF(WR1) := COOF(WS3)
        COOF(WS3) := WR1                    :: Link recombined block
                                            :: into next list.
        PLUS3(WS3) := PLUS3(WS3) + 1        :: Increase potential
                                            :: number.
        GO TO NEXTBLOCK
END)    RETURN
        END


        DEFINE R DUPLICATE
        DO
        LOGSIZE := COOF(OP2)
        127,LOGSIZE,0,0.7                   :: Get log2(size) of block.
        FINDSIZE                            :: Locate list, get size.
        GETBLOCK                            :: Get block, WS1 points to it.
COPY)   COOF(WS1) := COOF(OP2)              :: COPY block.
        WS1 := WS1 + ONE
        OP2 := OP2 + ONE
        122,SIZE,0,0.1
        IF SIZE GT 0 GO TO COPY             :: If not finished.
        RETURN
        END
```

```
DEFINE R FRBLOCK
DO
LOGSIZE := COOF(OP1)
127,LOGSIZE,0,0.7              :: Get log2(size).
FINDSIZE                       :: Locate free space list.
COOF(OP1) := COOF(POINTER)
COOF(POINTER) := OP1           :: Link block on to free
                               :: space list.
OCT := OP2
STOREIN1                       :: Assign specified value to
                               :: field a1.
RETURN
END
```

Improvements to the LSIX storage allocator

Suppose that a 2-block is requested when the smallest blocks available are 16-blocks then the routine SUBDIVIDE splits the first available 16-block into two 8-blocks the first of which is added to the 8-block free space list and the second is halved again to give two 4-blocks the second of these is then halved giving a block of the required size. Fig.II.3.2 shows the result of the subdivision.

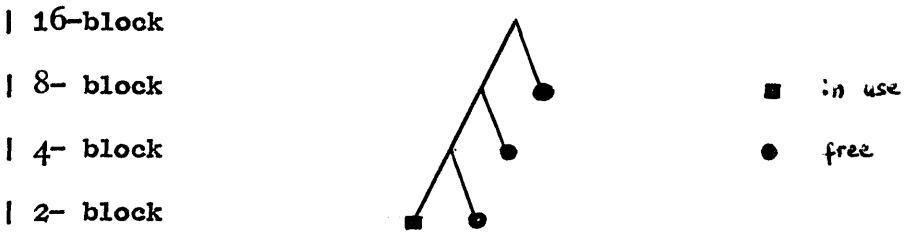


Fig. II·3·2

The two halves formed when a block is subdivided are called mates. When a block and its mate are free simultaneously they can either be recombined immediately or recombination can be deferred for as long as possible, that is until a block is required and the potential number of blocks of the required size is zero. Whenever the recombination is attempted, the process as defined by the routine RECOMBINE is very inefficient. This is mainly because it is necessary to search a free space list for the mate. Deferring recombination of blocks leads to fragmentation of the available store; larger blocks may be split unnecessarily when smaller blocks could have been recombined. On the other hand immediate recombination may result in several otherwise unnecessary calls on the routine SUBDIVIDE.

Given the address and size of any block it is a trivial matter to determine the address of its mate, the address of the mate of a block of size n is obtained by complementing the n-bit of the given address. Once the mate is located there are two problems: how do we know if it is free and if free, how can it be detached quickly from the free space list. Only 1 bit is required to indicate that a block is free or in use. A convenient choice is the sign bit of word 0 of any block.

Part of word 0 is already reserved for the system to store log2(size), the sign bit is easily tested and perhaps most important of all, word 0 of any free block contains a link to the next block on the free space list and as no address is negative we set the sign bit to 1 for blocks which are in use and to 0 for free blocks . The used/free bit also provides an additional safeguard in that we can now check that the user does not attempt to free a block which is already free. The second problem, efficient deletion of random blocks from a free space list, can be solved only by keeping doubly linked free space lists. An Atlas address occupies 24 bits (i.e. one LSIX word) therefore it is not possible to store both a forward and a backward link in a 1-block. A second version of the storage allocator has been written in which the free space lists are doubly linked and no 1-blocks are allowed. As 4 bits of word 0 have already been reserved for the system there is little use for 1-blocks anyway. In this version recombination takes place as soon as two mates are free simultaneously. Only minor modifications are necessary for the SETUPSTORE and GET routines. The revised FRBLOCK routine is described below. Fig. II.3.3 shows the linkages in the free space lists for the revised storage allocator.

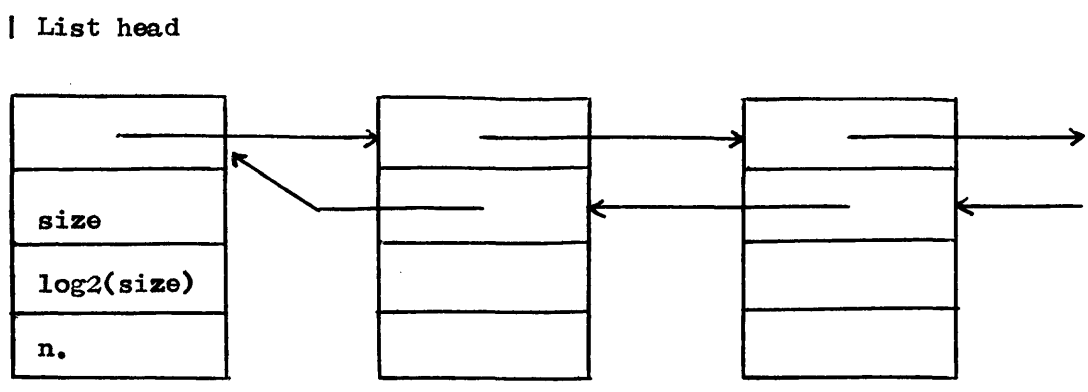


Fig. II.3.3

If WR1 points to a block on a free space list, that block can be deleted by means of the following commands:

```
|        COOF(PLUS1(WR1)) := COOF(WR1)
|        IF COOF(WR1) = 0 GO TO LASTBL    :: If last block on list.
|        PLUS1(COOF(WR1)) := PLUS1(WR1)
|                  . . . .
| LASTBL)          . . . .
```

The special treatment of the last block on a list could be avoided by enlarging the list head to include a backward pointer also, and using doubly linked circular lists.

There is one further detail to check before two mates can be recombined. Suppose that an 8-block is split to provide a 4-block and two 2-blocks, and that at some later stage the state of the 8-block is as shown in Fig. II.3.4.

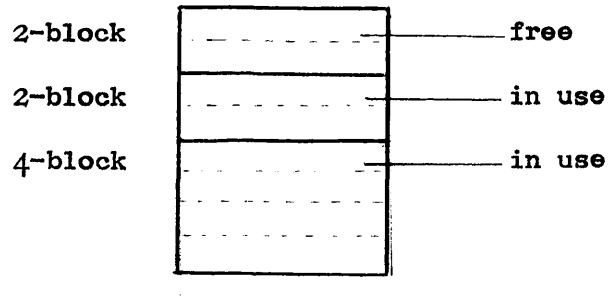


Fig. II.3.4

If the used 4-block then becomes free again before the 2-block, when the sign bit of word 0 of the mate is checked it indicates that the mate is free. Clearly we must also check the size of the mate. Even for free blocks then, the size must be immediately available. We know that the least significant octal digit of an Atlas whole word address is always zero so these three bits in word 0 can be used to store log2(size) just as they are for blocks which are in use.

Routine to free a block and recombine it with its mate if possible:

```
        DEFINE R FRBLOCK
|       DO
|       OCT := OP2
|       STOREIN1                    :: Assign new value of a2 to a1.
|       WS1 := COOF(OP1)            :: Get first word of block.
|       IF WS1 LT 0 GO TO NOTFREE   : If not already free.
|       O/P(NL.,'ATTEMPTING TO FREE A BLOCK WHICH IS ALREADY FREE')
|       STATEPRINT                  :: Output state of system.
|       RETURN
| NOTFREE) 165,LOGSIZE,WS1,0.7      :: Get log2(size).
|       FINDSIZE                    :: Locate free space list,
|                                   :: find size.
|       WR2 := 1
|       UPDTNDOT                    :: Update potential numbers.
| TRYMATE) IF SIZE= MAXSIZE GO TO LINKON
                                    :: If maximum size then no
                                    :: recombination.
```

```
|       124,SIZE,SIZE,0              :: Convert size to address units.
|       124,SIZE,SIZE,0
|       WS1 := OP1
|       126,WS1,SIZE,0               :: Address of mate.
|       WS2 := COOF(WS1)
|       127,WS2,0,*40000007          :: Get sign bit and log2(size).
|       IF WS2 = LOGSIZE GO TO MATE  :: If mate free.
| LINKON) WS1:= COOF(POINTER)
|       167,WS1,LOGSIZE,0            :: Record log(size).
|       COOF(POINTER):= OP1          :: Forward links.
|       COOF(OP1) := WS1
|       PLUS1(OP1) := POINTER        :: Backward links.
|       IF WS1 = LOGSIZE GO TO END   :: If end of list.
|       PLUS1(WS1) := OP1
| END)  RETURN
| MATE) COOF(PLUS1(WS1)) := COOF(WS1) :: Extract block.
|       IF COOF(WS1) = LOGSIZE GO TO LASTBLOCK
|                                    :: If last block on list.
|       PLUS1(COOF(WS1)) := PLUS1(WS1)
| LASTBLOCK) 167,OP1,SIZE,0
|       126,OP1,SIZE,0               :: OP1 points to
                                     :: recombined block.
|       POINTER := POINTER + 2       :: Point to next list.
|       SIZE := PLUS1(POINTER)       :: Get  new size.
|       LOGSIZE := PLUS2(POINTER)    :: Get new log2(size).
|       PLUS3(POINTER):=PLUS3(POINTER) + 1
|                                    :: Increase potential number.
|       GO TO TRYMATE                :: Go to try next mate.
|       END                          :: End of FRBLOCK.
```

A third version of the storage allocator.

A further disadvantage of the present storage allocation and bookkeeping method is that four bits of word zero are not available to the user. This seriously limits the usefulness of 2-blocks as word 1 is then the only possible link field and only 20 bits of word 0 are available for the storage of other information. It is not possible to simulate LISP-like systems using 2-blocks as two link fields are required for this. At present the Atlas LSIX system is used mainly for teaching purposes and it is important to be able to manipulate LISP-like lists. To facilitate this the four bits which are reserved for the system have been mapped into another area of the store. Each 48 bits of list storage is mapped into 6 bits. All 6 bits are zero unless the 48 bits constitute the first 48 bits of a data block. In this case the bits are allocated as follows:

bit	
1	used/free marker - 1 if block is in use;
2 }	available for use in automatic garbage
3 }	collection version to be described later;
4 }	
5 }	log2(size) of data block.
6 }	

This representation of 48 bits by 6 bits is very convenient as computation of the map address involves only a right shift of 3 binary places and addition of a base address. 6-bit patterns are easily manipulated on Atlas. The efficiency of this system with regard to space depends very largely upon the block size used. If a large number of 2-blocks are used there might be an overall saving in space.

The availability of blocks of general size

The advantages in restricting the available sizes of blocks to an integral power of two are obvious. The size, which must be readily available to the system, may be stored compactly and blocks are easily split to form smaller blocks or recombined with their mates when larger blocks are required. However, these advantages favour the implementor and not the general user. In the system described above, the user may specify any size of block between 2 and 128 words and is not aware of the fact that the sizes actually allocated are integral powers of two. A system which allocates any size of block between 2 and 128 words is not difficult to implement.

There are good reasons for organising the free space lists as before. The actual size would again be stored in word 0, but now occupying eight bits. With so much of the first word reserved for the system there is little use for a 1-block and it is proposed that block sizes should be a multiple of two. Thus if a block of 9 words is requested a 10-block is allocated. For this purpose the system requires a 16-block of which it immediately returns the first four words to the 4-block list, the next two to the 2-block list and allocates the last 10 words to the user. The immediate return of the first few unwanted words is effected by subtracting 9 from 16 to give (00000111) in binary from which it is clear that the unwanted parts are a 4-block and 2-block, the final bit being ignored.

The size of a block also determines the manner in which it is freed. Thus for the 10-block allocated above we have the size (00001010) in binary and starting with the right most 1 bit we return

first a 2-block and then an 8-block. If immediate recombination is possible then the fragmentation of store which is inherent in this generalisation is to a certain extent counteracted. Immediate recombination implies doubly linked free space. If the list heads also contain two pointers then there is some advantage in using circular free space lists, for then any odd blocks which are returned immediately in the allocation process may be linked to one side of the list head and allocations made from the other. This increases the likelihood that the odd blocks are free and available for recombination when their mates become free again.

An automatic garbage collector for LSIX

The major problem which arises when attempting to reclaim part of a list structure is that of knowing which data blocks are no longer needed. A number of solutions have been proposed. The first of these by Newell Simon and Shaw places the responsibility on the programmer. This language, IPL-V, includes instructions for erasing lists. In LSIX we have the equivalent free block instruction. A second solution is that used in SLIP in which a reference count is kept in the head of every list. For each additional reference to a list or part of a list the reference count in the list head is increased by one. It is always possible to gain access to the list head from any part of a list structure as SLIP is a symmetric list processor i.e. it uses doubly linked lists. However, the process of linking back to the list head is time consuming. There is the additional disadvantage that while any part of a list is shared by the other lists which are not free it is not possible to reclaim any of the list cells on that list. The third solution is that proposed

by McCarthy in which no cells are reclaimed until all of the free space is exhausted. Then a garbage collection routine is entered which scans all list structures which are in use and marks the data blocks attached to the lists. When the lists have all been scanned the whole of the list area is scanned again. Blocks which are not marked are free and may be reclaimed. At the same time the marks are erased from the blocks which are in use, so that the garbage collection routine may be reentered as often as necessary during the execution of a program.

The basic problem in McCarthy's method is that of scanning the lists. In general the lists will be branched and every branch must be traced. The natural way to process a list is by recursion but a recursive routine requires an indefinite amount of store. As the garbage collector is entered only when all, or nearly all, of the storage space has been exhausted it is most unlikely that sufficient storage will be available for a recursive garbage collector. An ingenious solution to this problem has been proposed by Schorr and Waite (1967). They describe a garbage collector for the WISP language (Wilkes, 1964) which uses only three registers for temporary storage. The process described is capable of scanning any list structure, however complex it may be. A slightly modified version has been used to collect garbage in a BCL program which processes binary trees (see section 2.4) and the Atlas LSIX garbage collector is a further extension of that described by Schorr and Waite.

The flexibility which LSIX provides in both the definition and the use of fields is one of its major advantages. It is this

flexibility which makes it difficult to collect garbage automatically. Any field which is of address length or longer may be used to store a link, and the contents of a field are interpreted according to the context in which they are used. At the time of storing a link the nature of the operand is known to the system, especially if the user is willing to preserve the semantic difference between the copy field and copy pointer operations although even this is not absolutely necessary. In the Atlas LSIX, the only fields which can possibly contain an (Atlas) address are full 24-bit fields. When a link is stored we require an extra bit, outside the 24 bit word, to record the fact that the word contains a pointer. Now in the mapping version of LSIX, the reader may remember that of the 6 bits representing each 48-bit block of list storage 2 bits were unused. These provide our two pointer flags for the possible address fields in the 48 bit block. Using these the system can keep a record of all links in the list structures.

For the basefields (bugs) there are no pointer flags. Any bug whose contents could possibly be an address in the list space is assumed to point to a list. Other pointers to lists may be stored temporarilly on the system's field contents stack. These pointers also must be taken into account by the garbage collector. The system first scans all list structures to which bugs point and then any lists pointed to from the field contents stack marking those 48 bit blocks which are in use . When all accessible blocks have been marked the whole of the list area is scanned again; any free blocks are collected up as 8-blocks, 4-blocks and 2-blocks; marks are erased from those blocks which are in use and finally, if the maximum size of block set up by the program is greater than eight,

the 8-blocks are recombined with their mates, if free, until blocks of the maximum size have been reconstituted.

In LSIX, as in other list-processing systems, it is usual to access the fields of a block via a pointer to the head or first word of the block. This is not the only way in which a field may be accessed. In particular, suppose that the same set of operations is to be applied successively to each word of a block. Each time round the loop it is necessary either to increase the word number in the field definition by redefining the field or to increment the block pointer so that it advances word by word through the block. The latter method is the more efficient as redefinition of a field is a lengthy process. However, the garbage collector becomes very complicated if blocks may be accessed by means of pointers to words other than the first. The LSIX user is advised that at all times there must be a pointer to the first word of every block which is in use as the garbage collector ignores pointers to words other than the first. In almost all cases, on completion of a loop in which a block pointer is incremented, the user will want to restore the pointer to its initial value so a copy will have been saved either on the field contents stack or in another field.

BCL routines for scanning list structures and for collecting the unmarked blocks are described in Appendix 4.

§2.4 List Processing in BCL

So far we have been concerned mainly with the use of BCL to implement LSIX. In this section we consider the use of BCL itself as a list processor. The basic operations in any compiler compiler include facilities for manipulating strings of input symbols and BCL is no exception. Through experience gained in the use of BCL as a compiler compiler for LSIX, it became clear that with a number of extensions BCL could be used as a high level symbol manipulation language having many of the facilities which are available in LISP. Also, through the use of symbolic machine orders for the machine concerned, BCL provides the same flexibility as low-level systems such as LSIX, with the possibility of manipulating bit patterns. The BCL List Processing system is particularly suitable for teaching as the student is able to define and build his own list processor using blocks of several different sizes which are defined by the program. Standard functions for manipulating list structures are easily defined by the user. A number of demonstration programs which have been used on a computer science course for M.SC. students are described below.

The version of BCL used in this section is that defined by the Atlas BCL compiler dated August 1968 which is a further development of the compiler used to implement LSIX. One of the most useful additions is the provision of labels and GO TO commands within groups of elements. Labels are defined only in the group and branch in which they are set and just as it is illegal to jump into a DO loop in FORTRAN, so also in BCL jumps into an alternative are not allowed, neither are jumps out of an alternative although the latter restriction is a temporary one.

To provide a list processing system based on this version of BCL the author has added functions and groups with parameters. The parameters implemented at present are of type A only (storing an address or an address length integer) and are called by value. Character variables of up to four characters in length may be used as actual parameters as there is no type check. Examples given in this section show that even this small subset of parameters provides a very powerful system. General parameters of any type with calls by name, reference or value will eventually be implemented in BCL and will improve the system still further. Functions have been added through the implementation of an EXIT statement which can have one actual parameter being the value to be returned. EXIT statements can be used in any level of alternative within a group and are effectively a RETURN or jump out of a group to the calling point. When an EXIT is used in a branch within a group then a stack of pointers must be unwound as in the case of a transfer out of a block in Algol. A more detailed description of these extensions to the BCL compiler is given in Appendix 5.

In the first example (see Table II.4.1) the LISP function CONS and the predicates EQ and NULL are defined in BCL; CAR and CDR are represented by HEAD and TAIL , and the complete program shows how functions such as APPEND, UNION and INTERSECTION are defined recursively. The program in Table II.4.1 shows how the principles developed in this section can be used in a system which requires simple linked lists in which the nodes (pointer words in LISP) consist of two address-size fields named HEAD and TAIL. More complex structures in which the head field may contain a link to a sublist will be discussed later. The tail of the last node in a

```
DECLVAR IS (A FREE, A COUNT, A WS, A RESULT,A P,A Q,A R)

PWORD(?) IS (A HEAD,A TAIL)

INITFREE IS (121,FREE,0,*20000000)

SETUPFREE(A COUNT) IS (TAIL(FREE) = 0,
                AGAIN: SETUP(PWORD,WS,FREE),
                       TAIL(FREE) = WS,
                       COUNT = COUNT - 1,
                       IF COUNT GT 1 GO TO AGAIN)

CONS(A X,A Y) IS (EITHER IF FREE = 0,
                         O/P(NL.,'FREE SPACE EXHAUSTED',NL.),STOP
                  OR     WS=FREE,
                         FREE=TAIL(FREE),
                         HEAD(WS)=X,
                         TAIL(WS)=Y,
                         EXIT(WS)  )

NULL(A X) IS (IF X = 0)

EQ(A X,A Y) IS (IF X = Y)

PRINTLIST(A X) IS (PRINT: IF X = 0 GO TO END,
                          WS = HEAD(X), O/P(WS,SP.(2)),
                          X = TAIL(X), GO TO PRINT,
                    END:  O/P(NL.)  )

APPEND(A X,A Y) IS (EITHER NULL(X),EXIT(Y)
                    OR     EXIT(CONS(HEAD(X),APPEND(TAIL(X),Y))))

MEMBER(A X,A Y) IS ((EITHER NULL(Y),RESULT = 0
                     OR     EQ(X,HEAD(Y)), RESULT = 1
                     OR     MEMBER(X,TAIL(Y))  ), IF RESULT = 1 )

UNION(A X,A Y) IS (EITHER NULL(X),EXIT(Y)
                   OR     MEMBER(HEAD(X),Y), EXIT(UNION(TAIL(X),Y))
                   OR     EXIT(CONS(HEAD(X),UNION(TAIL(X),Y)))  )

INTERSECTION(A X,A Y) IS (EITHER NULL(X), EXIT(0)
                          OR     MEMBER(HEAD(X),Y),
                                 EXIT(CONS(HEAD(X),
                                           INTERSECTION(TAIL(X),Y)))
                          OR     EXIT(INTERSECTION(TAIL(TAIL(X),Y))))
```

Table II.4.1 (a) Program defining some List Processing functions.

```
LISTPROGRAM IS (INITFREE, SETUPFREE(40),
                P=CONS(2,CONS(4,CONS(6,0))),
                O/P(NL.,'LIST P '), PRINTLIST(P),
                Q=CONS(4,CONS(6,CONS(8,0))),
                O/P(NL.,'LIST Q '), PRINTLIST(Q),
                R=APPEND(P,Q),
                O/P(NL.,'LIST P WITH LIST Q APPENDED '),
                PRINTLIST(R),
                R=UNION(P,Q),
                O/P(NL.,'UNION OF LISTS P AND Q '),
                PRINTLIST(R),
                R=INTERSECTION(P,Q),
                O/P(NL.,'INTERSECTION OF LISTS P AND Q '),
                PRINTLIST(R),
                STOP )

*ENTER(O/P(NL.,'LIST PROGRAM TEST ',NL.(2)),LISTPROGRAM)
```

Table II.4.1 (b) A simple program using the functions defined above.

```
LIST PROGRAM TEST


LIST P 2  4  6
LIST Q 4  6  8
LIST P WITH LIST Q APPENDED 2  4  6  4  6  8
UNION OF LISTS P AND Q 2  4  6  8
INTERSECTION OF LISTS P AND Q 4  6
```

Table II.4.2 Output from the program in Table II.4.1 (b).

list contains an easily recognizable symbol, zero in this case, which serves as a terminator.

Declaration of variables, nodes and fields

In the first line of the program in Table II.4.1 a number of variables to be used in the program are declared as type A and may therefore be either link variables storing an address or integer variables. They are also declared jointly to constitute the group DECLVAR.

Consider the next definition

PWORD(?) IS (A HEAD, A TAIL).

This says that PWORD (or pointer word in LISP) is a structure (or datagroup) consisting of two fields HEAD and TAIL each of type A (address). The query indicates that HEAD and TAIL are selector functions and not variable names. Thus HEAD refers to the first halfword and TAIL to the second halfword of a structure (an address occupies one half-word on Atlas). BCL is very free in mixing elements of different type in a group; in this work we prefer to distinguish fairly sharply between data-groups and command-groups (alias routines).

The nodes within a list may be referenced either directly by means of a link variable containing the address of the node or indirectly through the link field of another node. The fields within a node are referenced by writing the name of the field followed by the name of a pointer or link variable in parentheses. Thus if the link variable P points to a node, the head field of that node is referred to as HEAD(P) and the tail field as TAIL(P). If P points to the first node of a linked list, as in Fig. II.4.1, then nodes other than the first may be accessed via the pointers in the

TAIL fields. For example HEAD(TAIL(P)) refers to the head field of the node to which TAIL(P) points, i.e. to the head field of the second node of the list. Similarly TAIL(TAIL(P)) refers to the tail field of the second node of the list P (the list to which P points).

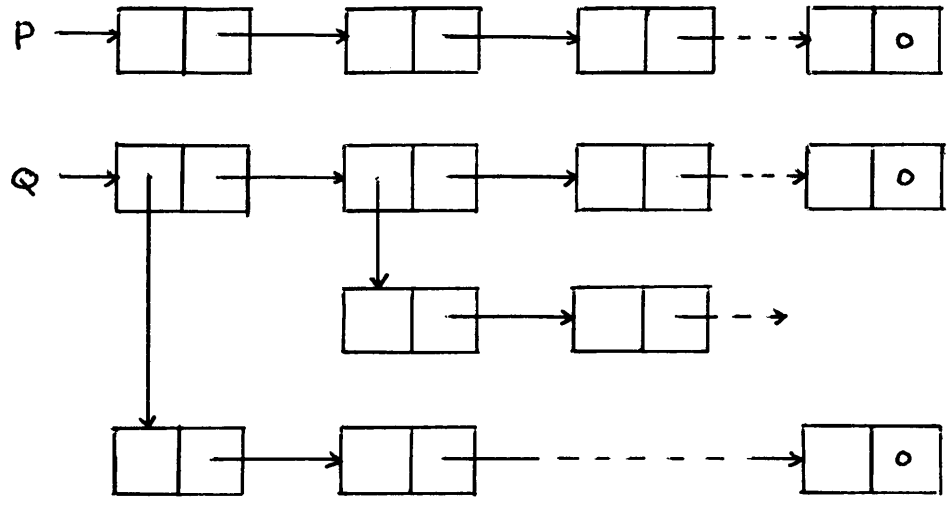


Fig. II.4.1 Examples of linked lists.

The field HEAD(HEAD(TAIL(Q))) refers to the head of the first cell of the second sublist of the list Q. Note that a field name is meaningless if used on its own; it must always be used with a pointer or link variable. The 'functions' HEAD and TAIL enable us to dissect any list structure however complex it may be.

Groups of commands, functions

Lists are constructed by getting new nodes and planting in them links to other nodes. Nodes which are available for constructing linked lists are usually stored as a linked list of free space. In Table II.4.1 the link variable FREE points to such a list of free space which is set up by calling the group of commands SETUPFREE. The function CONS gets a node from this list and plants values in

its head and tail fields. Note the method of branching used in this function. EITHER the free space list is empty, in which case the program is terminated (no garbage collector is defined in this simple program) OR the first node is unlinked from the free space list and the values of X and Y are written into the head and tail fields respectively. Finally the command EXIT causes a return, bringing with it the value of the link variable WS, so that the value of the function CONS is a pointer to a new node. A group of commands is called by writing its name followed by a list of zero or more actual parameters enclosed with parantheses and separated by commas.

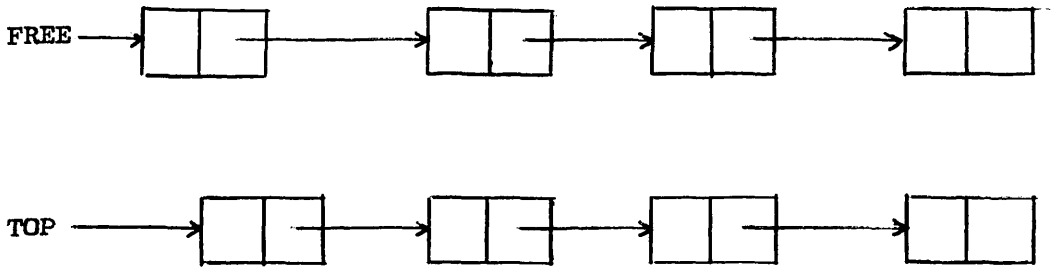
If TOP is a link variable which points to the top node of a stack then the value of the variable A may be stacked by means of the statement

```
TOP = CONS(A,TOP)
```

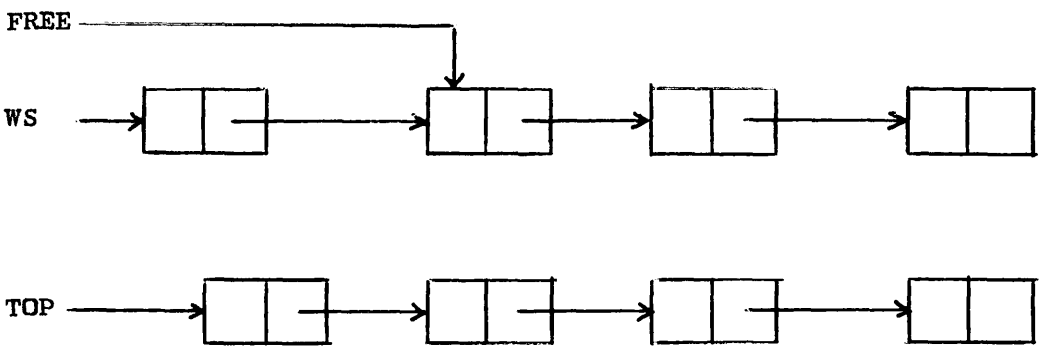
This statement is equivalent to the following sequence:

```
|                IF FREE NE 0 GO TO GETNODE,
|                O/P(NL., 'FREE SPACE EXHAUSTED'), STOP,
|      GETNODE:  WS = FREE,
|                FREE = TAIL(FREE),
|                TAIL(WS) = TOP,
|                HEAD(WS) = A,
|                TOP = WS,
|                 . . . .
```

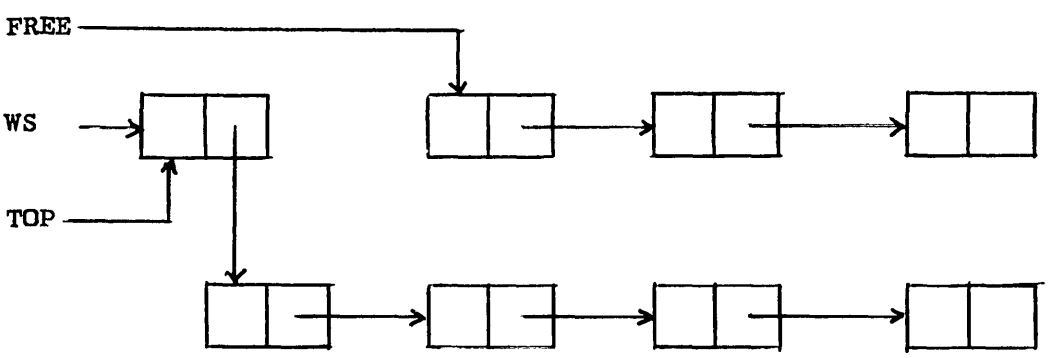
Fig. II.4.2 shows the state of the stack and the free space list at various stages during the execution of these statements.



(a) Before entering CONS.



(b) After executing FREE = TAIL(FREE).



(c) After leaving the CONS function.

FIG. II.4.2 The state of the Stack and Free Space list at various stages during the execution of CONS(A,TOP).

Returning to the remaining groups of commands in Table II.4.1, NULL(X) tests if the list X is empty by testing if pointer X is zero. EQ(X,Y) tests two symbols for equality and PRINTLIST(X) prints the elements of the simple linked list X .

The groups APPEND, MEMBER, UNION and INTERSECTION are more interesting. They are the BCL versions of the functions of the same names defined in the LISP 1.5 Programmer's Manual (McCarthy et al., 1965). Each of these groups is defined recursively and also uses previously defined groups. They are included to demonstrate the functional aspects of the system. The arguments of APPEND are both lists and the function constructs a new list which is the second list appended to a copy of the first without changing either of the original lists.

MEMBER is a predicate. It tests if the symbol X is a member of the list Y. EITHER Y is an empty list in which case the result is false, a fact which is recorded by setting RESULT to zero, OR X equals the head element of Y and RESULT is set to 1, OR MEMBER is called recursively to test if X is a member of the list TAIL(Y). Eventually either X is found on list Y and RESULT = 1 or the end of list Y is reached without finding X, and RESULT = 0. MEMBER is finally 'matched' or 'failed' by the condition IF RESULT = 1. The effect of this will become clearer when we consider the group UNION which calls MEMBER.

A simpler definition of MEMBER which uses IF Y NE 0, the equivalent of NOT(NULL(Y)), is

```
| MEMBER(A X,A Y) IS (IF Y NE 0, EITHER EQ(X,HEAD(Y))
|                                OR     MEMBER(X,TAIL(Y)) )
```

UNION constructs a list which is the union of its two arguments. Inside the group UNION, EITHER list X is empty in which case the result is list Y, OR if the first element of list X is a member of list Y the result is the union of the two lists TAIL(X) and Y. If the first element of list X is not a member of list Y then MEMBER(HEAD(X),Y) is failed by its final condition, IF RESULT = 1, and the third alternative of UNION is entered giving the result that UNION(X,Y) is CONS(HEAD(X), UNION(TAIL(X),Y)).

The value of the function INTERSECTION is the intersection of two lists. The use of these functions clearly demonstrates the power of the system. Actual parameters of a function may themselves involve further calls on functions to any depth.

In Table II.4.1(b), LISTPROGRAM is a group of commands to test the system which has been defined. It begins by calling INITFREE which initialises the start of free space, (the only command in this group is a symbolic machine order to assign to the variable FREE the octal address 20000000), and then SETUPFREE to set up a linked list of free space. It next constructs two simple lists P and Q and calls in turn APPEND, UNION and INTERSECTION with P and Q as actual parameters. The results are output after each statement and the test program then STOPs. Fig. II.4.3 shows the lists constructed. Note that the results of APPEND and UNION share the nodes of the original list Q.

Following the directive *ENTER is a command to output on a newline (NL.) a title followed by two newlines before entering the group LISTPROGRAM. The actual computer output for this simple test program is shown in Table II.4.2.

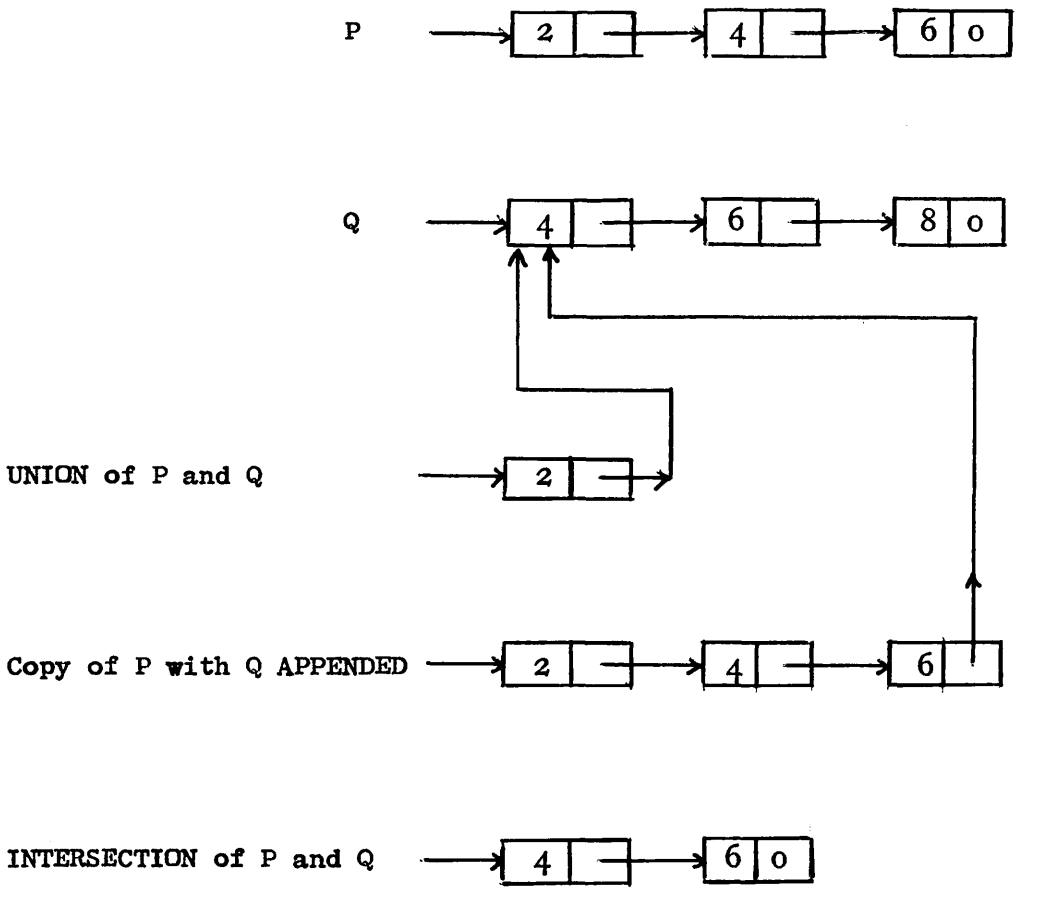


Fig. II.4.3 Results of APPEND(P,Q),UNION(P,Q) and INTERSECTION(P,Q).

Input, Output and Storage of Atoms

Consider the declarations (A INTEGER, 8C CHARVAR). The first of these declares INTEGER to be of type A; it can therefore store an address or an integer. The second declares CHARVAR to be a character variable which can store up to eight six-bit characters left justified in a 48-bit field. The appearance of variable names in the list of elements of a group which is entered in input or output mode causes values of the specified type to be transferred.

For example provided that the next characters in the input stream are of the appropriate type, the occurrence of INTEGER causes input of an integer to the variable INTEGER. If there are no digits in the input stream the transfer fails. Similarly CHARVAR causes the transfer of up to eight alphanumeric characters. The transfer is terminated either after eight such characters have been input or on finding a character which does not belong to the appropriate character set (digits and letters). If after reading at least one letter or digit an unacceptable character occurs before the eighth character is read, then the transfer is terminated and the remainder of the 48-bit field is filled with space characters. Variables to store more than eight characters may be declared and it is possible to input as atoms character strings of any length.

Once input an atom must be stored either in list space or in separate atom space. Storage in list space restricts character strings to four characters as the head field of a list-cell is only 24 bits.

Atomic symbols may be stored in separate atom space by setting up atom records defined as follows:

Atom Flag	Atom type	Atom length	Atom symbol
-1	0 or 1 or 2	1, 2 or more	

where the atom types 0, 1 and 2 represent integer, real and character strings respectively, and the length is 1 for integers, 2 for real numbers and is variable for character strings.

Thus for an integer atom we define the record

IATOM(?) IS (A FLAG, A TYPE, A LENGTH, A ISYMBOL)

and for character strings of up to 8 alphanumeric characters

CATOM(?) IS (A FLAG, A TYPE, A LENGTH, 8C CSYMBOL)

Real numbers are not yet implemented in the BCL prototype compiler used for this work but the record for a real atom might be

RATOM(?) IS (A FLAG, A TYPE, A LENGTH, R RSYMBOL)

To store the atom which is input, a new atom record is set up using the BCL command

SETUP (Recordname, Pointer, Pointer to Atomspace)

where the record name is either IATOM, CATOM or RATOM, pointer is a variable which points to the record after it has been set up and the pointer to atom space indicates the next available space (in consecutive store locations) for atoms. Thus to read and store an integer atom and return with a pointer to its record in atom space we define the group

```
|        IREAD IS (INTEGER,
|                  SETUP(IATOM, WS, ATOMSPACE),
|                  FLAG(WS) = -1,
|                  TYPE(WS) = 0,
|                  LENGTH(WS) = 1,
|                  ISYMBOL(WS) = INTEGER,
|                  EXIT(WS))
```

The first field of any atom record is a negative flag to indicate that the record is that of an atom. Thus the predicate ATOM(X) which is true if X is null or if X points to an atom is defined by

```
|       ATOM(A X) IS (EITHER NULL(X)
|                         OR      IF FLAG (X) LT 0)
```

If the head field of a node contains a pointer to a sublist it is necessary to distinguish it from atoms. The predicate ATOM defined above is suitable for this when atoms are stored in separate atomspace. If atoms are to be stored in the head field of a node then a bit must be reserved to distinguish between atoms and pointers to sublists. So that the full head field may be available for storing atoms it is convenient to extend our definition of PWORD to include a third field which stores a flag describing the contents of the head field. We now have

PWORD(?) IS (A FLAG, A HEAD, A TAIL) .

If the field named FLAG is zero for atoms and one for sublists then the predicate ATOM is redefined as

```
|       ATOM(A X) IS (EITHER IF FLAG(X)=0
|                         OR      IF HEAD(X)=0) .
```

§2.4.2 Manipulation of Algebraic Expressions.

The following program may be used to read polynomial expressions, store them as binary trees, output them in forward Polish notation, reverse Polish notation and in the normal infix form, and finally to differentiate such a polynomial with respect to a single variable and output its derivative after some simplification.

Groups of commands to input an expression are given in Table II.4.3. These use the syntax defined by

```
<constant>    ::= <integer>
<variable>    ::= <name>
<primary>     ::= <constant> | <variable> | ( <expression> )
<secondary    ::= <primary> ** <constant> | <primary>
<term>        ::= <term> * <secondary> |
                  <term> / <secondary> | <secondary>
<expression>  ::= <expression> + <term> |
                  <expression> - <term> | <term>
```

The nodes set up are three-field nodes of the form

If the symbol field contains an arithmetic operator the link fields point to the two operands involved. Constants and variable names are stored in the symbol fields of nodes of which both link fields are zero.

```
*BCL SOURCE

DECLVAR IS (A LISTSPACE,A POINTER,A INTEGER,A X,A OP,A WS,
            A PLUS,A MINUS,A MULT,A DIV,A EXPNT,A VARX,A WS1,
            A WS2,2C INAME)

NODE(?) IS (A LLINK, A SYMBOL,A RLINK)

CONS(A X,A Y,A Z) IS (SETUP(NODE,POINTER,LISTSPACE),
                      LLINK(POINTER)=X,
                      SYMBOL(POINTER)=Y,
                      RLINK(POINTER)=Z,
                      EXIT(POINTER)  )

VARIABLE IS (OSP.,INAME,OSP.,EXIT(CONS(0,INAME,0))  )

CONSTANT IS (OSP.,INTEGER,OSP.,EXIT(CONS(0,INTEGER,0))  )

PRIMARY IS ((EITHER X = CONSTANT
             OR     X = VARIABLE
             OR     '(',X=EXPRESSION,')'),EXIT(X)  )

SECONDARY IS (X=PRIMARY,EITHER '**',EXIT(CONS(X,'**',CONSTANT))
                        OR     EXIT(X)  )

TERM IS (X=SECONDARY, MORE: (EITHER '/',OP='/'
                             OR     '*',OP='*'
                             OR     EXIT(X)  ),
                             X=CONS(X,OP,SECONDARY),
                             GO TO MORE    )

EXPRESSION IS (X=TERM,MORE: (EITHER '+',OP='+'
                             OR     '-',OP='-'
                             OR     EXIT(X)  ),
                             X=CONS(X,OP,TERM),
                             GO TO MORE)
```

Table II.4.3. Groups of Commands to Input an Expression.

The expression

$$(2x + 1)^3 - 6x$$

is punched as

```
(2 * X + 1) **3 - 6 * X
```

and the effect of the statement X=EXPRESSION is to assign to X a pointer to a binary tree representing the input expression as shown in Fig. II.4.4.

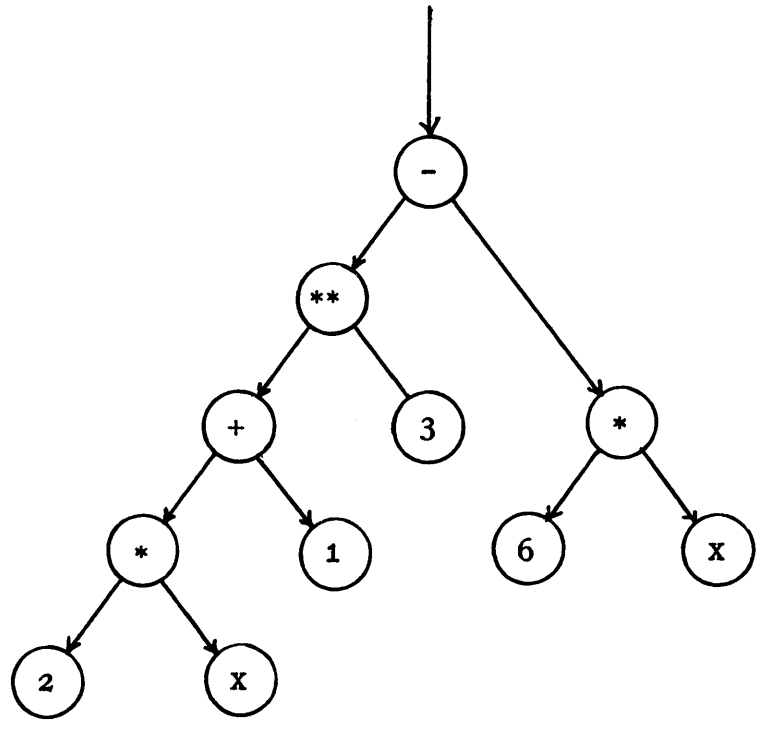


Fig. II.4.4 Tree representation of (2x + 1)**3 - 6x.

```
PRINT(A X,A LASTOP,A OP) IS (EITHER IF X=0
                         OR     IF LLINK(X)=0,IF RLINK(X)=0,
                               (EITHER IF SYMBOL(X) GT 0,
                                       WS=SYMBOL(X), O/P(WS)
                                OR     INAME=SYMBOL(X),
                                       O/P(INAME)   )
                                OR     OP=SYMBOL(X),
                                      (EITHER IF LASTOP=EXPNT
                                       OR    (EITHER IF LASTOP=MULT
                                              OR     IF LASTOP=DIV),
                                             (EITHER IF OP=PLUS
                                              OR     IF OP=MINUS)),
                                       O/P('('),PRINT(LLINK(X),OP),
                                       INAME=OP,O/P(INAME),
                                       PRINT(RLINK(X),OP),O/P(')')
                         OR     PRINT(LLINK(X),OP),INAME=OP,
                                O/P(INAME),PRINT(RLINK(X),OP)   )

NODEPRINT(A X) IS (EITHER IF LLINK(X)=0,IF RLINK(X)=0,
                          IF SYMBOL(X) GT 0,WS=SYMBOL(X),
                          O/P(WS,SP.(2))
                   OR     INAME=SYMBOL(X),O/P(INAME)   )

PREPRINT(A X) IS (EITHER IF X=0
                  OR     NODEPRINT(X),
                         PREPRINT(LLINK(X)),PREPRINT(RLINK(X))  )

ENDPRINT(A X) IS (EITHER IF X=0
                  OR     ENDPRINT(LLINK(X)),
                         ENDPRINT(RLINK(X)),NODEPRINT(X)  )

POSTPRINT(A X) IS (EITHER IF X=0
                   OR     POSTPRINT(LLINK(X)),
                          NODEPRINT(X),POSTPRINT(RLINK(X))  )
```

Table II.4.4. Groups of commands to output the information stored in a tree.

Commands to output the elements of a tree are given in Table II.4.4. PREPRINT outputs the expression in forward Polish notation, ENDPRINT in reverse Polish notation and POSTPRINT in infix form without parentheses. The terms PREorder, POSTorder and ENDorder are those defined and used by Knuth (1968). PRINT inserts parentheses in the infix form to remove any ambiguities. Typical results for the tree in Fig. II.4.4 are

Forward Polish:	-	**	+	*	2	X	1	3	*	6	X
Reverse Polish:	2	X	*	1	+	3	**	6	X	*	-
Infix without brackets:	2	*	X	+	1	**	3	-	6	*	X
Infix with brackets:	(2	*	X	+	1)	**	3	-	6	*	X

These different orders of output are obtained simply by traversing the tree in different orders. PREPRINT first visits the root then the left subtree and finally the right subtree. The group NODEPRINT is machine dependent and requires further comment. This group first tests if the node contains a constant or a variable name by testing the link fields. If both links are zero and if the contents of the symbol field are negative then the node contains a variable name which is output in character form by first transferring it to a character variable, INAME in this example. A positive value in the symbol field is the value of an integer which is first transferred to a variable of type A and then output. Nodes whose link fields are not zero store arithmetic operators which are output as characters after being assigned to a character variable.

(The reader may have noticed that PRINT, which incidentally does not use NODEPRINT, appears to have three formal parameters, X, LASTOP

and OP, whereas when it is called only two actual parameters are specified. The system as implemented at present does not check that the number of actual parameters is equal to the number of formal parameters nor does it provide for local variables and as the parameters are called by value this provides a convenient trick by which to introduce the latter.)

The function DIFF(X) (see Table II.4.5) uses the basic rules of differentiation to construct a tree representing the derivative of the expression to which the link variable X points. Note that the variable OP is a local variable and not a formal parameter. No simplification is performed within this group and the result of Y = DIFF(X) is shown in Fig. II.4.5.

```
DIFF(A X,A OP) IS (EITHER IF X=0,EXIT(0)
     OR     IF LLINK(X)=0,IF RLINK(X)=0,
           (EITHER IF SYMBOL(X)=VARX,EXIT(CONS(0,1,0))
            OR     EXIT(0)  )
     OR    (EITHER IF SYMBOL(X)=PLUS,OP='+'
            OR     IF SYMBOL(X)=MINUS,OP='-'),
            EXIT(CONS(DIFF(LLINK(X)),OP,DIFF(RLINK(X))))
     OR     IF SYMBOL(X)=MULT ,
            EXIT(CONS(CONS(LLINK(X),'*',DIFF(RLINK(X))),'+'
                              ,CONS(DIFF(LLINK(X)),'*',RLINK(X))))
     OR     IF SYMBOL(X)=DIV,
            EXIT(CONS(CONS(CONS(RLINK(X),'*',DIFF(LLINK(X))),'-',
                           CONS(LLINK(X),'*',DIFF(RLINK(X)))),
                         '/',CONS(RLINK(X),'**',CONS(0,2,0))))
     OR     IF SYMBOL(X)=EXPNT,
            EXIT(CONS(RLINK(X),'*',CONS(DIFF(LLINK(X)),'*',
            CONS(LLINK(X),'**',CONS(RLINK(X),'-',CONS(0,1,0))))))
```

Table II.4.5. Commands to differentiate a simple polynomial expression with respect to a single variable.

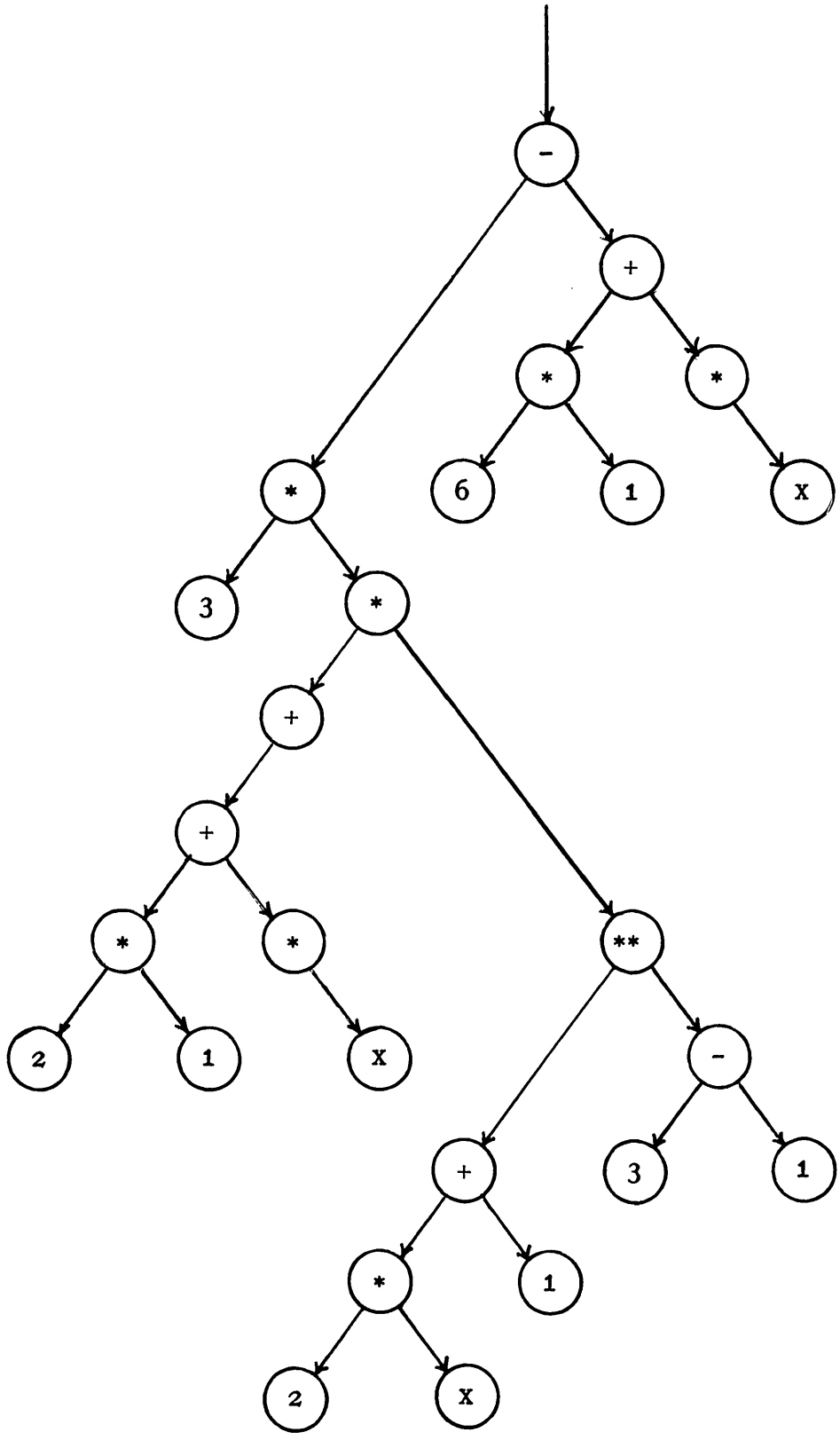


Fig. II.4.5. Derivative of (2x + 1)**3 - 6x before simplification.

The group 'EQUAL' used below tests two trees for equality.

```
IATOM(A X) IS (IF LLINK(X)=0,IF RLINK(X)=0,IF SYMBOL(X) GT 0  )

SPLUS(A X) IS (EITHER IF RLINK(X)=0, EXIT(LLINK(X))
                OR    IF LLINK(X)=0, EXIT(RLINK(X))
                OR    IATOM(LLINK(X)),IATOM(RLINK(X)),
                      WS=SYMBOL(RLINK(X))+SYMBOL(LLINK(X)),
                      EXIT(CONS(0,WS,0))
                OR    EXIT(X)  )

SMINUS(A X) IS (EITHER IF RLINK(X)=0,EXIT(LLINK(X))
                 OR    EQUAL(LLINK(X),RLINK(X)),EXIT(0)
                 OR    IATOM(LLINK(X)),IATOM(RLINK(X)),
                       WS=SYMBOL(LLINK(X))-SYMBOL(RLINK(X)),
                       (EITHER IF WS GT 0,EXIT(CONS(0,WS,0))
                        OR     LLINK(X)=0,SYMBOL(RLINK(X))=0-WS,
                               EXIT(X)  )
                 OR    EXIT(X)  )

SMULT(A X) IS (EITHER IF LLINK(X)=0, EXIT(0)
                OR    IF RLINK(X)=0, EXIT(0)
                OR    IF SYMBOL(LLINK(X))=1, EXIT(RLINK(X))
                OR    IF SYMBOL(RLINK(X))=1, EXIT(LLINK(X))
                OR    EQUAL(LLINK(X),RLINK(X)),
                      EXIT(CONS(LLINK(X),'**',CONS(0,2,0)))
                OR    IF SYMBOL(RLINK(X))=MULT,IATOM(LLINK(X)),
                      IATOM(LLINK(RLINK(X))),WS1=SYMBOL(LLINK(X)),
                      WS2=SYMBOL(LLINK(RLINK(X))),1302,WS1,WS2,0,
                      EXIT(CONS(CONS(0,WS1,0),'*',RLINK(RLINK(X))))
                OR    EXIT(X)  )

SDIV(A X) IS (EITHER IF LLINK(X)=0, EXIT(0)
               OR    EQUAL(LLINK(X),RLINK(X)),EXIT(CONS(0,1,0))
               OR    IF RLINK(X)=0,0/P(NL.,'DIVISION BY ZERO '),
                     EXIT(X)
               OR    EXIT(X)  )

SEXPNT(A X) IS (EITHER IF LLINK(X)=0, EXIT(0)
                 OR    IF RLINK(X)=0, EXIT(CONS(0,1,0))
                 OR    IF SYMBOL(RLINK(X))=1, EXIT(LLINK(X))
                 OR    EXIT(X)  )

SIMPLIFY(A X) IS (EITHER IF X=0,EXIT(0)
                   OR    IF RLINK(X)=0,IF LLINK(X)=0,EXIT(X)
                   OR    RLINK(X)=SIMPLIFY(RLINK(X)),
                         LLINK(X)=SIMPLIFY(LLINK(X))
                        ,EITHER IF SYMBOL(X)=PLUS,EXIT(SPLUS(X))
                         OR     IF SYMBOL(X)=MINUS,EXIT(SMINUS(X))
                         OR     IF SYMBOL(X)=MULT,EXIT(SMULT(X))
                         OR     IF SYMBOL(X)=DIV,EXIT(SDIV(X))
                         OR     IF SYMBOL(X)=EXPNT ,EXIT(SEXPNT(X))  )
```

Table II.4.6. Simplification routines.

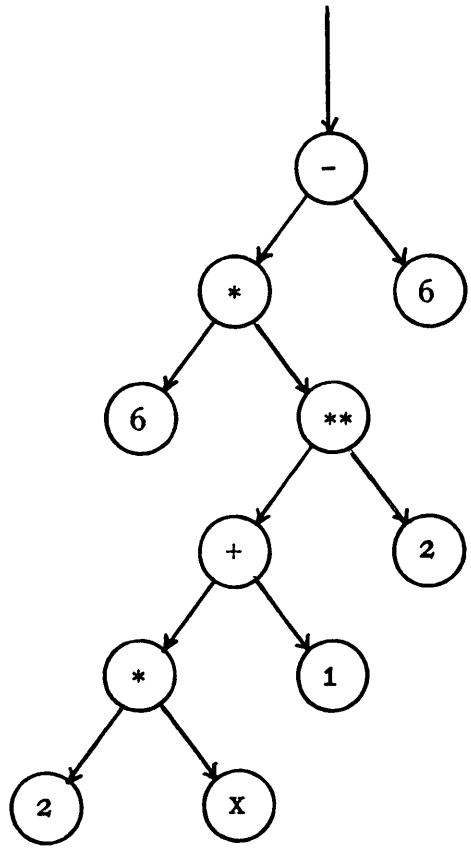


Fig. II.4.6. Derivative of (2x + 1)**3 - 6x after simplification.

Obviously some simplification could have been carried out during the actual construction of the derivative but simplification is itself quite instructive and is defined in separate groups of commands given in Table II.4.6. The result of Z = SIMPLIFY(Y) is the tree shown in Fig. II.4.6.

Table II.4.7 contains actual computer output from a program using the differentiation and simplification groups defined here. It will be seen that these results are not all in their simplest form. In particular the derivative of (X + Y) (X - Y) with respect to X is output as X + Y + X - Y . It is well known that the major part of any differentiation program is the simplification of the results. Further groups to collect terms could be included. The group DIFF is easily extended to deal with more general functions but the commands given here are sufficient to illustrate the techniques used.

```
DIFFERENTIATION TEST
THE DERIVATIVE OF 3* (X **2+ X) + 2* X **3
WITH RESPECT TO X IS 3* (2* X + 1) + 6* X **2
THE DERIVATIVE OF (X + Y)* (X - Y)
WITH RESPECT TO X IS X + Y + X - Y
THE DERIVATIVE OF 3* (2* X + 1)**3 - 2* X **2
WITH RESPECT TO X IS 18* (2* X + 1)**2 - 4* X
THE DERIVATIVE OF 3* (2* X + 1)**2 + 6* X **3
WITH RESPECT TO X IS 12* (2* X + 1) + 18* X **2
END OF PROGRAM
```

Table II.4.7. Output from the differentiation program in Table II.4.8.

```
LISTPROGRAM IS (DATAF=0,      ::  data trace off
                PLUS='+',
                MINUS='-',
                MULT='*',
                DIV='/',
                EXPNT='**',
                VARX='X',
           NEXT: 121,LISTSPACE,0,*2,    ::  Initialise list
                                        ::  space pointer.
               (EITHER NLS OR NIL.),
               (EITHER X=EXPRESSION,
                       O/P(NL.(3),'THE DERIVATIVE OF '),
                       PRINT(X)
                OR     O/P(NL.(2),'END OF PROGRAM'),STOP),
                O/P(NL.(2),'WITH RESPECT TO X IS '),
                PRINT(SIMPLIFY(DIFF(X))),
                GO TO NEXT  )

NLS IS (OSP.,NL.,EITHER NLS OR NIL.)

*ENTER(O/P(NL.,'DIFFERENTIATION TEST ',NL.(2)),LISTPROGRAM)
```

Table II.4.8. A differentiation program.

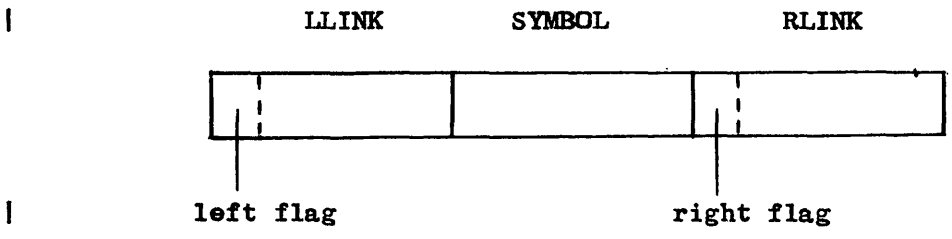
§2.4.3 An example of automatic garbage collection in BCL

In the expression manipulation program described above, space required for new nodes was allocated sequentially and not from a linked list of available space. Before the input of each new expression the pointer to the listspace is reset to its initial value and the same space is used several times over.

To demonstrate a method of automatic garbage collection another version of the program has been written in which space available for new nodes is organised in the form of a linear linked list pointed to by the link variable FREE. From time to time this list is exhausted and nodes which are no longer attached to active lists, and are therefore free, can be collected up and returned to the free space list by the garbage collector. In general the active lists will not be linear lists and any routine which scans an active list to determine which nodes are still accessible, and therefore in use, must scan all branches of the list. The simplest method is to use a recursive routine to scan the lists but recursive routines require an indefinite amount of work space for storing link information at each call. As the garbage collector is called only when all, or almost all, work space has been exhausted it is important to use a method which requires very little work space. The method used here requires only three working registers and uses the link fields of the nodes themselves to store any pointers which must be saved.

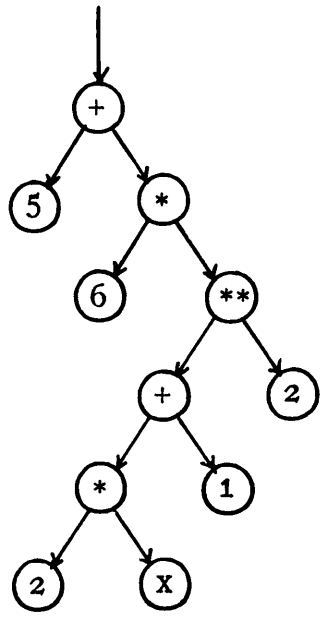
The nodes used are the same as before, consisting of the three fields LLINK, SYMBOL and RLINK. The garbage collector requires two additional one-bit fields for flags. It is convenient to use the

sign bit of each link field for these as no address in the listspace can be negative.

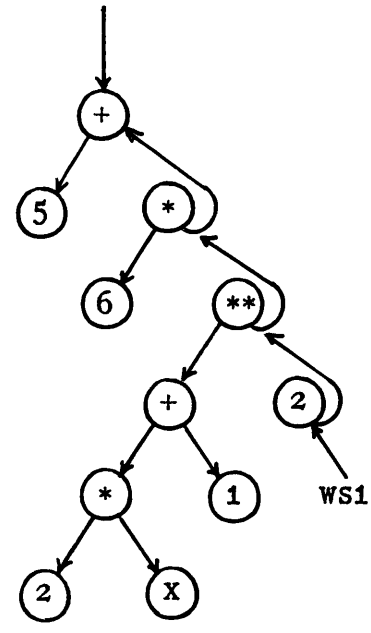


The SCAN routine first makes a forward scan along the right links ignoring other fields and reversing the right links until a branch end is found. The scan is then reversed, and as each node is passed its right flag is set to indicate that the node is in use. This backward scan is terminated on finding a node with a left subtree which is as yet unscanned. The left flag of this node is set and a forward scan made of the right links of the next (left) subtree, again terminating at a branchend. On the backward scan any left flag which has been set indicates a branch point. The effect of the scanning operation is illustrated by the tree diagrams on the following page, in which a 1 to the left (right) of a node indicates that the left (right) flag of that node is set.

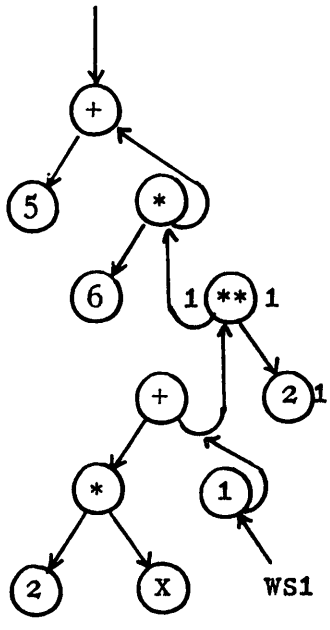
In addition to any trees pointed to by link variables there may be some pointers, to the subtrees of partially constructed trees, stored temporarily in the system work stack. These also must be scanned and this is the first operation of COLLECTGARBAGE. When all active trees and subtrees have been scanned the routine LINKFREE scans all list space from LISTART to LISTEND returning unflagged nodes to the free space list and resetting flags in nodes which are still in use.



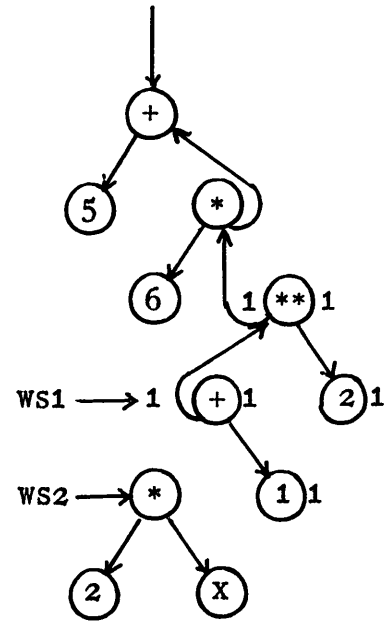
(a) Before scan commences.



(b) End of first forward scan.



(c) End of second forward scan.



(d) Start of third forward scan.

Fig. II.4.7. The state of the tree representing 5 + 6(2x + 1)**2 at various stages during the scanning procedure.

The program begins by calling SETUPFREE to set up a linked list of free space from location *2 onwards.

```
| NODE(?) IS (A LLINK,A SYMBOL,A RLINK)

| SETUPFREE(A COUNT) IS (121,LISTART,0,*2,
|                        FREE:=LISTART,LISTEND:=LISTART,
|                  MORE: SETUP(NODE,WS,LISTEND),  :: set up node.
|                        RLINK(WS):=LISTEND,      :: plant link.
|                        COUNT:=COUNT-1,
|                        IF COUNT GT 0 GO TO MORE,
|                        RLINK(WS):=0  )          :: terminate list.
```

CONS gets new nodes and plants information in the three fields. It first asks if the free space list is empty and if so calls the garbage collection routine. If no free cells are found the program is abandoned, otherwise a new node is allocated.

```
| CONS(A X,A Y,A Z) IS (EITHER IF FREE:=0,
|                              COLLECTGARBAGE,
|                              IF FREE:=0,
|                              O/P(NL.,'FREE SPACE EXHAUSTED'),
|                              DUMP,STOP
|                       OR     WS:=FREE, FREE:=RLINK(FREE),
|                              LLINK(WS):=X,
|                              SYMBOL(WS):=Y,
|                              RLINK(WS):=Z,
|                              EXIT(WS)   )
```

The routine COLLECTGARBAGE sets a working pointer to the bottom location of the system stack and then proceeds to scan any lists whose addresses have been stacked. Note that list addresses lie in the range

| LISTART ≤ list address < LISTEND.

FREESPP marks the top of the system stack.

COLLECTGARBAGE next scans any other lists that are still required. These are specified by the program. In this case only the list X is used. After this has been scanned LINKFREE collects up all free nodes between LISTART and LISTEND.

```
| COLLECTGARBAGE IS (O/P(NL.(2),
|                    'GARBAGE COLLECTION ROUTINE ENTERED',NL.),
|                    WS4:=STACKBASE,
|             NEXT:  IF WS4 GE FREESPP GO TO STACKDONE,
|                    IF COOF(WS4) LT LISTART GO TO SKIP,
|                    IF COOF(WS4) GE LISTEND GO TO SKIP,
|                    SCAN(COOF(WS4)),
|             SKIP:  WS4:=WS4+ONE,
|                    GO TO NEXT,
|        STACKDONE:  SCAN(X),
|                    LINKFREE  )
```

```
| SCAN(A X) IS (WS1:=1, WS2:=X,          :: Initialise working
                                         :: pointers.

|        MORE:  IF WS2 LE 0 GO TO BRANCHEND,

|               WS3:=RLINK(WS2),

|               RLINK(WS2):=WS1,         :: Reverse link.

|               WS1:=WS2, WS2:=WS3,      :: Step down branch.

|               GO TO MORE,


|   BRANCHEND:  IF WS1=1 GO TO ENDSCAN,

|               WS3:=WS2, 167,WS3,0,*4,  :: Set rightflag to
                                         :: 'not free'.

|               WS2:=WS1,

|               IF LLINK(WS2) LT 0 GO TO ENDBRANCH,

|               WS1:=RLINK(WS1),         :: Step back up branch.

|               RLINK(WS2):=WS3,         :: Restore link.

|               IF LLINK(WS2)=0 GO TO BRANCHEND,

|               WS3:=LLINK(WS2),

|               LLINK(WS2):=-WS1,        :: Set left flag to mark
                                         :: branch.

|               WS1:=WS2, WS2:=WS3,

|               GO TO MORE,


|    ENDBRANCH: WS1:=-LLINK(WS1),

|               126,WS3,0,*4,           :: Reset left flag.

|               LLINK(WS2):=WS3,

|               GO TO BRANCHEND,

|      ENDSCAN: NIL.  )
```

In LINKFREE the effect of the system procedure DELETE(NODE,WS1) is to step the pointer WS1 back by an amount equal to the length of a node.

```
| LINKFREE IS (FREE:=0, WS1:=LISTEND,     :: Start at end of
|       NEXT: DELETE(NODE,WS1),           :: list area.
|              IF RLINK(WS1) LT GO TO NOTFREE,
|              RLINK(WS1):=FREE,          :: Linkon free node.
|              FREE:=WS1,
|              GO TO END,
|     NOTFREE: WS2:=RLINK(WS1),
|              126,WS2,0,*4,              :: Reset right flag to
|                                         :: zero.
|              RLINK(WS1):=WS2,
|        END:  IF WS1 GT LISTART GO TO NEXT  )
```

The garbage collector was tested in the expression differentiation program by setting up only forty nodes on the free space list initially and after the output of each derivative setting X to zero so freeing the tree to which X pointed. The octal output of the list area, given in Appendix 6, shows that the garbage collector worked satisfactorily.

§2.4.4 BCL program to build a tree-structured directory.

Linked memory techniques have important applications in building files. The following program builds and updates a tree structured file to store a number of alphabetic items of variable length. Instructions are included to search for an item in the tree, insert an item if it is not already in the tree, and delete an item from the tree.

After initialisation of the file the program stores the items

BAN, BACK, BANE, BARREL, BE, BAR, BANK, BANG,

BEEN, BARE, BARGAIN, BAND, BARREN, BARK, BEE, BARB,

BANDIT, BARN and BARGE

in a form representing the logical tree in Fig. II.4.8 in which each branch end is marked by an asterisk.

This tree is represented in the store by a binary tree using the usual convention that each node contains two links, a SON and a BROTHER. Each node may therefore have only one SON, any other sons being stored as brothers of the first SON. Fig. II.4.9 shows the representation in store using nodes defined by

NODE(?) IS (IC SYMBOL, A SON, A BROTHER)

The program data consists of lists of items preceded by one of the directives *INSERT, *FIND, *DELETE and *END. Test data and the corresponding output follow the program below.

The reader will note that each item is scanned twice, once as a full word and then one character at a time. It is convenient to output a whole item at once even though it is stored one character per node. The only process which is perhaps non-trivial is deletion. The whole or part of an item to be deleted may be a part

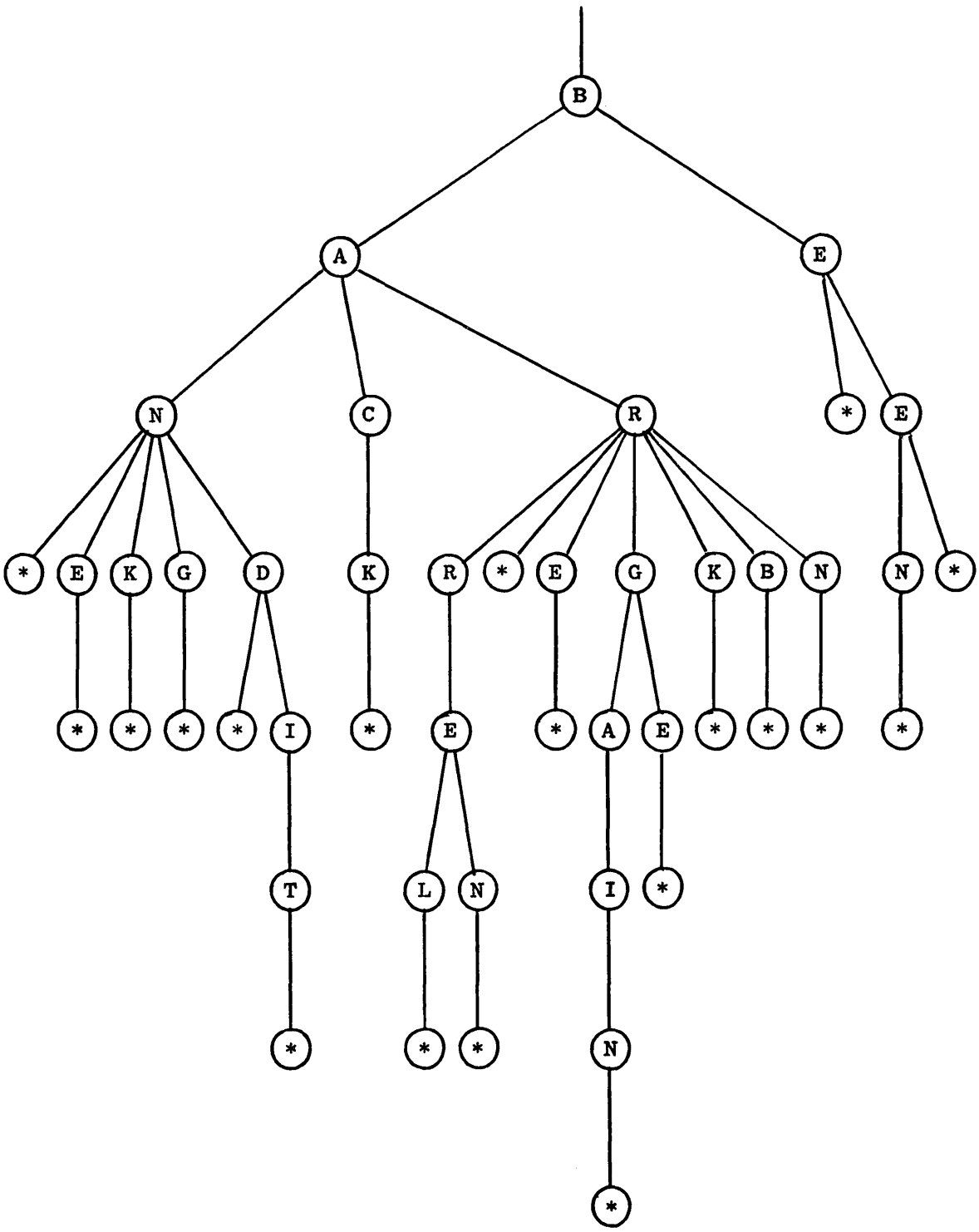


Fig. II.4.8. Logical tree representation of data.

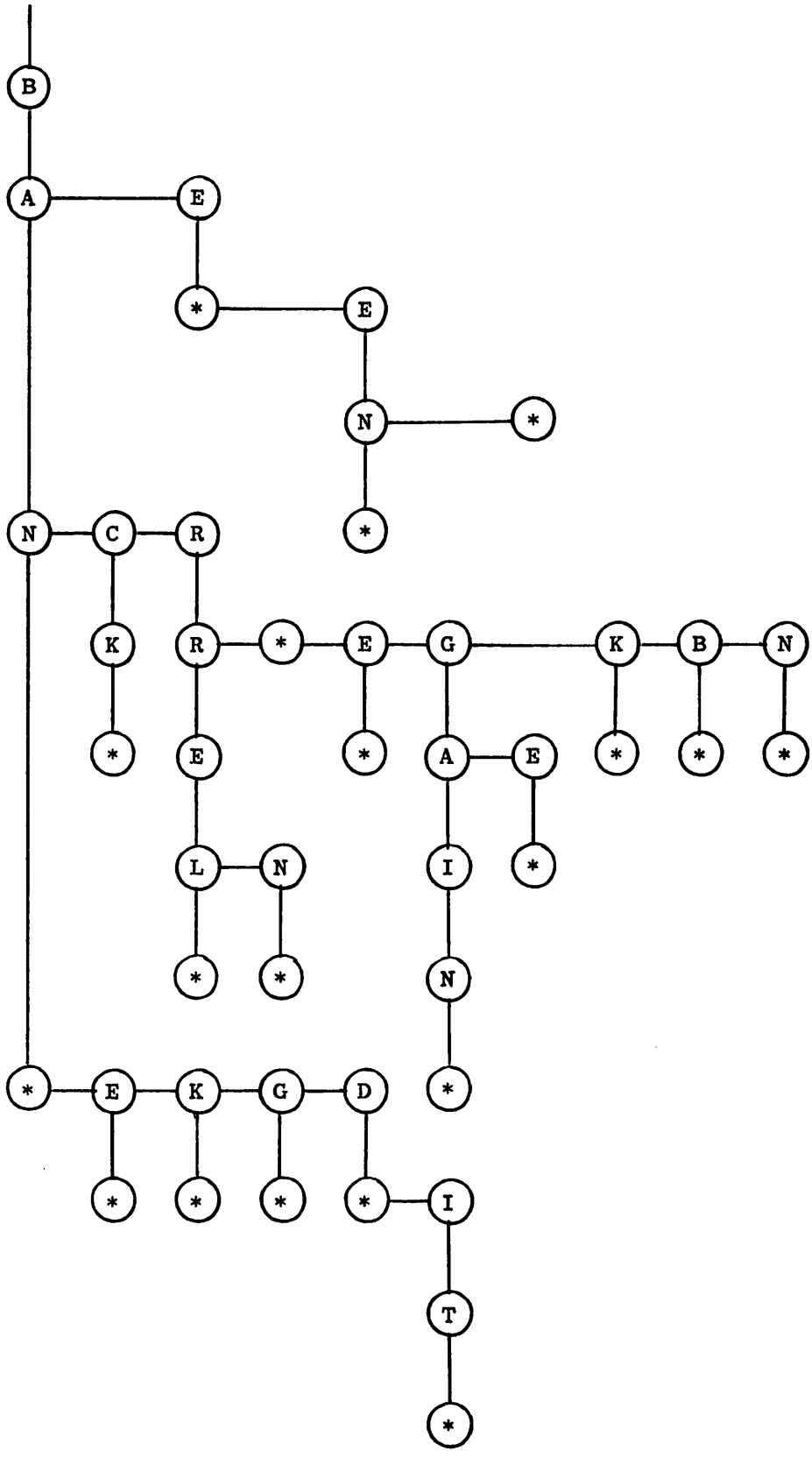


Fig. II.4.9. Binary tree representation of data.

or the whole of some other item which is not to be deleted. In searching for the item to be deleted the process notes, in OLDW, any part of the item which is a stem of one or more other items. This part must not be deleted. It is also necessary to note whether the link to be changed to effect the deletion is a SON link or a FATHER link. A SWITCH is set and reset to keep track of this. The FIND, INSERT and DELETE operations all involve the same search and are therefore written as one group of commands.

BCL program to build a tree-structured file.

```
DECLVAR IS (A LISTSPACE,A W,A ROOT,A WS,A OLDW,A FLAG,A SWITCH,
            1C K,8C NAME)

NODE(?) IS (1C SYMBOL,A SON,A BROTHER)

CONS(A X,A Y,A Z) IS (SETUP(NODE,WS,LISTSPACE),
                      SYMBOL(WS) = X,
                      SON(WS) = Y,
                      BROTHER(WS) = Z,
                      EXIT(WS) )

NLS IS (OSP.,NL., (EITHER NLS OR NIL.) )
```

```
PROCESSFILE IS  (DATAF = 0,     :: Switch off data trace
      NEXTITEM: (EITHER NLS OR OSP.),
                (EITHER '*INSERT',OSP.,FLAG = 1
                 OR     '*FIND'  ,OSP.,FLAG = 2
                 OR     '*DELETE',OSP.,FLAG = 3
                 OR     '*END',EXIT
                 OR     NIL.),

                (EITHER NAME,
                        O/P(NL.,(EITHER IF FLAG = 1, 'INSERT '
                                 OR     IF FLAG = 2, 'FIND   '
                                 OR     IF FLAG = 3, 'DELETE '),NAME),
                        IF 1= 0
                 OR     K),

                 W = ROOT,
      NEXTCHAR:  IF SYMBOL(W) = K GO TO FOUND,
                 IF BROTHER(W) = 0 GO TO NOTIN,
                (EITHER IF FLAG = 3, OLDW = W, SWITCH = 1
                 OR     NIL.),
                 W = BROTHER(W),
                 GO TO NEXTCHAR,

         FOUND:  IF K = ' ' GO TO IN,
                (EITHER IF FLAG = 3, IF BROTHER(SON(W)) NE 0,
                        OLDW = W, SWITCH = 0
                 OR     NIL.),
                 W = SON(W),
                (EITHER K OR SP., K = ' '),
                 GO TO NEXTCHAR,

            IN: (EITHER IF FLAG = 1,
                        O/P(NL.,'ITEM ',NAME,' ALREADY IN FILE')
                 OR     O/P(NL.,'ITEM ',NAME,' FOUND IN FILE') ),
                (EITHER IF FLAG = 3,
                       (EITHER IF SWITCH = 0,
                               SON(OLDW) = BROTHER(SON(OLDW)),
                        OR     BROTHER(OLDW) = BROTHER(BROTHER(OLDW)) ),
                        O/P(NL.,'ITEM ',NAME,' DELETED FROM FILE')
                 OR     NIL.),
                 GO TO  NEXTITEM,

         NOTIN: (EITHER IF FLAG = 1,BROTHER(W) = CONS(K,0,0),
                        W = BROTHER(W), IF K = ' ' GO TO INSERTED,
      READNEXT:        (EITHER K OR SP.,K = ' '),
                        SON(W) = CONS(K,0,0), W = SON(W),
                        IF K NE ' ' GO TO READNEXT,
      INSERTED:         O/P(NL.,'ITEM ',NAME,' ADDED TO FILE')
                 OR     O/P(NL.,'ITEM ',NAME,' NOT FOUND IN FILE'),
                        GO TO END,

          SKIP:         (EITHER K OR SP.,K = ' '),
           END:          IF K NE ' ' GO TO SKIP ),
                 GO TO NEXTITEM )
```

```
PROGRAM IS (121,LISTSPACE,0,*2,          :: Initialise list space.
            ROOT = CONS(' ',0,0),        :: Initialise file.
            PROCESSFILE, STOP)

*ENTER(PROGRAM)
```

DATA

```
*INSERT BAN BACK BANE BARREL BE BAR BANK BANG BEEN BARE
        BARGAIN BAND BARREN BARK BEE BARB BANDIT BARN BARGE BAN
*FIND   BANE BEEN BARGAIN BEAR
*INSERT BEER
*FIND   BEER
*DELETE BEER
*FIND   BEER
*DELETE BARREL BAN
*FIND   BARREN BARREL BAN BAND
*END
```

OUTPUT

```
INSERT BAN
ITEM BAN      ADDED TO FILE
INSERT BACK
ITEM BACK     ADDED TO FILE
INSERT BANE
ITEM BANE     ADDED TO FILE
INSERT BARREL
ITEM BARREL   ADDED TO FILE
INSERT BE
ITEM BE       ADDED TO FILE
INSERT BAR
ITEM BAR      ADDED TO FILE
INSERT BANK
ITEM BANK     ADDED TO FILE
INSERT BANG
ITEM BANG     ADDED TO FILE
INSERT BEEN
ITEM BEEN     ADDED TO FILE
INSERT BARE
ITEM BARE     ADDED TO FILE
INSERT BARGAIN
ITEM BARGAIN ADDED TO FILE
INSERT BAND
ITEM BAND     ADDED TO FILE
INSERT BARREN
ITEM BARREN   ADDED TO FILE
INSERT BARK
ITEM BARK     ADDED TO FILE
INSERT BEE
ITEM BEE      ADDED TO FILE
INSERT BARB
ITEM BARB     ADDED TO FILE
INSERT BANDIT
ITEM BANDIT   ADDED TO FILE
INSERT BARN
ITEM BARN     ADDED TO FILE
INSERT BARGE
ITEM BARGE    ADDED TO FILE
INSERT BAN
ITEM BAN      ALREADY IN FILE
```

```
FIND  BANE
ITEM BANE     FOUND IN FILE
FIND  BEEN
ITEM BEEN     FOUND IN FILE
FIND  BARGAIN
ITEM BARGAIN FOUND IN FILE
FIND BEAR
ITEM BEAR     NOT FOUND IN FILE
INSERT BEER
ITEM BEER     ADDED TO FILE
FIND BEER
ITEM BEER     FOUND IN FILE
DELETE BEER
ITEM BEER     FOUND IN FILE
ITEM BEER     DELETED FROM FILE
FIND BEER
ITEM BEER     NOT FOUND IN FILE
DELETE BARREL
ITEM BARREL   FOUND IN FILE
ITEM BARREL   DELETED FROM FILE
DELETE BAN
ITEM BAN      FOUND IN FILE
ITEM BAN      DELETED FROM FILE
FIND  BARREN
ITEM BARREN   FOUND IN FILE
FIND  BARREL
ITEM BARREL   NOT FOUND IN FILE
FIND  BAN
ITEM BAN      NOT FOUND IN FILE
FIND  BAND
ITEM BAND     FOUND IN FILE
```

§2.4.5 The Classical Transportation Problem.

Our final example is taken from an important class of linear programming problems known as transportation problems. A transportation problem is mathematically equivalent to the following:

> Given a certain amount of some commodity available at each of n sources and a certain amount required at each of m destinations, where the total amount available equals the total requirement, and given the cost of supplying each destination from each source (as so much per unit), find the cheapest way of meeting the requirements.

The problem can be formulated as a linear programming problem if we define $x[i,j] \geq 0$ as the number of units sent from source i to destination j . We must then minimize

$$z = \sum_{i,j} c[i,j] * x[i,j]$$

subject to $\sum_{j} x[i,j] = a[i]$ and $\sum_{i} x[i,j] = b[j]$

where c[i,j] is the cost of transporting one unit from source i to destination j, a[i] is the amount available at source i and b[j] the amount required at destination j . It is assumed that

$$\sum_{i} a[i] = \sum_{j} b[j] .$$

Such problems have been solved in many different ways the original formulation and solution being due to Hitchcock (1941). The method used here (sometimes called the u-v method) follows the logic of the simplex algorithm but keeps track of the situation in a more compact way.

For m origins and n destinations the constraints

$$\sum_{j} x[i,j] = a[i] \quad , \quad i = 1,2, \ldots , m$$

and

$$\sum_{i} x[i,j] = b[j] \quad , \quad j = 1,2, \ldots , n$$

constitute m+n equations in m*n unknowns. The coefficent matrix has rank m+n-1 and a basic solution to the constraints is one in which not more than m+n-1 of the x[i,j] are greater than zero. The m*n solution matrix is therefore sparse and since problems which arise in practice may involve hundreds of origins and thousands of destinations it is important to store all information as compactly as possible.

It is convenient to store the values of the m+n-1 basic variables x[i,j] on linked orthogonal lists, each element being a member of both a row list and a column list. The nodes used are six-field nodes containing a row and a column number, a row link and a column link, the value of the variable, and a sixth field which is available for perturbations in degenerate problems.

ROW	COL	ACROSS
VALUE	OTHER	DOWN

Each list is circular and has a list head also of six fields.

A typical configuration for three origins with availabilities 15, 12 and 18 units and four destinations with requirements 7, 12, 12 and 14 units is shown on Page 136. Note that a column list-head has a negative row number and a row list-head has a negative column number. There is a special base node (or HEADCELL) with both row and column number negative. This stores the sum of the availabilities and the sum of the requirements. Availabilities at sources and requirements at destinations are stored in the appropriate list heads. The particular basic solution shown on Page 136 is x[1,1] = 7,x[1,2] = 8, x[2,2] = 4,x[2,3] = 8, x[3,3] = 4 and x[3,4] = 14 ; all other x[i,j] are basic and therefore zero.

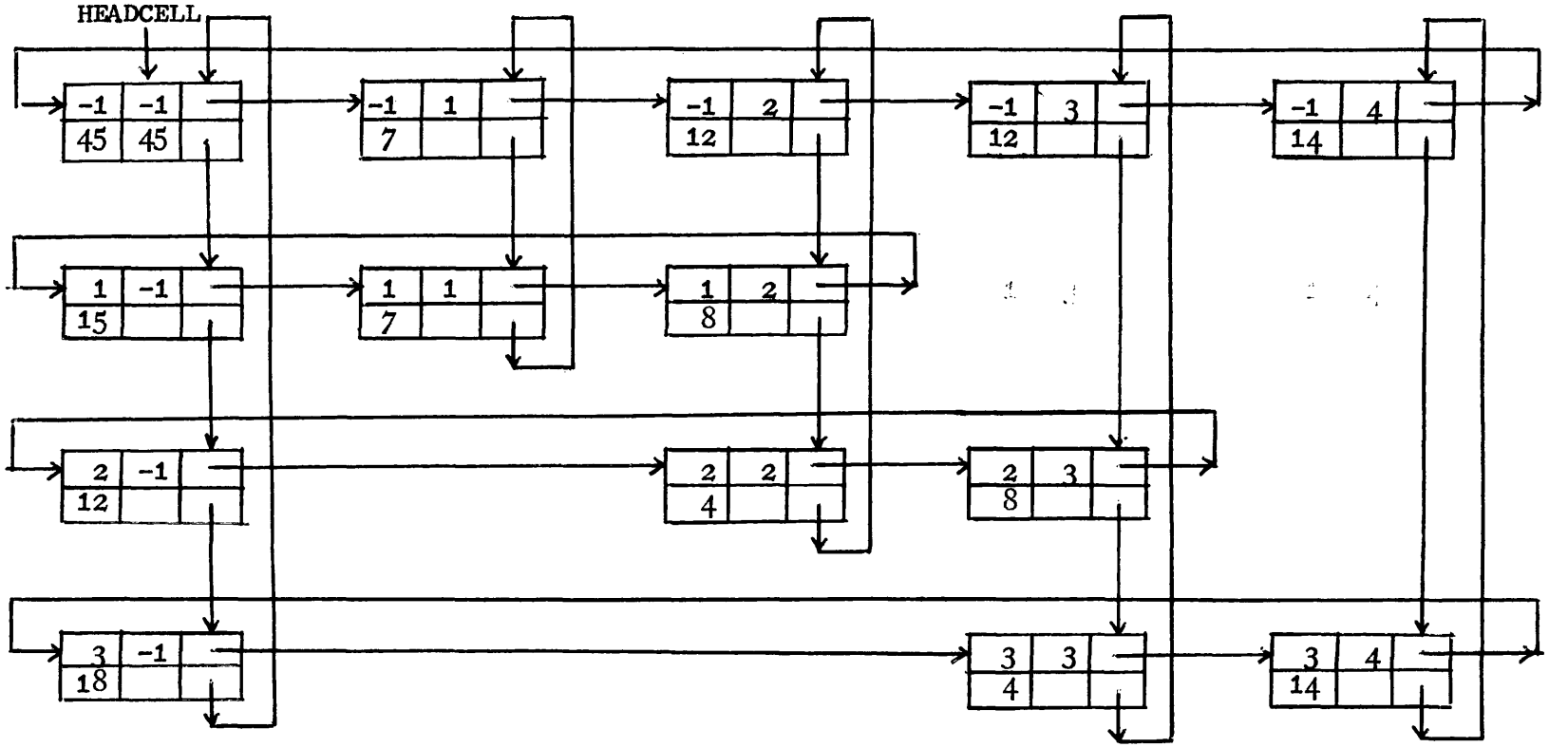


Fig. II.4.10 Orthogonal list representation of an initial basic solution for a 3 X 4 transportation problem.

The program begins by reading M and N the number of sources and destinations, the N requirements and the M availabilities, at the same time setting up list heads in which to store this information. If the sum of the availabilities is not equal to the sum of the requirements the program is abandoned.

Next the costs c[i,j] are read and stored in an M*N matrix in the conventional form. This matrix could be stored in linked form if sparse e.g. if many routes are forbidden, in which case no cost is given for them, but this program assumes that the cost matrix is stored sequentially row by row. [Note that arrays had not been implemented in BCL at the time of writing this program]. The input of the problem data and setting up the linked list-heads is the function of the group READVALUES given below.

The simplex algorithm starts with a basic solution, tests it for optimality and, if not optimal, transforms the basis to give a new solution which in general is closer to the optimum than is the original. The method used to set up an initial basic solution is the so called 'North West Corner' rule defined by the BCL group NWRULE. This rule pays no attention to cost and starting with the first source and first destination sends along this route the maximum possible number of units. This will either exhaust the available stock at source one or will satisfy the requirement at destination one but in general the supply and demand constraints will not be satisfied simultaneously. If the supply is exhausted then the next source is taken otherwise the process continues with the next destination until eventually the supply at origin M is exhausted and the demand at destination N met by sending the remaining units along route (M,N). The basic solution shown for the 3*4 problem above is an initial solution obtained by the northwest

corner rule. As each assignment is made a node is set up, the number of units x[i,j] stored, the row and column numbers set equal to those of the source and destination respectively and finally the node is linked into the orthogonal column and row lists.

As each basic solution is found it is output by the group PRINTSOLN.

The test for optimality and the method of constructing one basic solution from another are well written up in most of the standard texts on linear programming and the theory behind these operations will not be given here. A basic solution is transformed to a new basic solution by increasing the value of some non basic variable until one of the basic variables becomes zero. The roles of these two variables are then interchanged. The problem is to choose a non basic variable which when increased will reduce the total distribution cost. For each (non-basic) variable we define a relative cost factor being the increase in total cost per unit increase in the variable all other non-basic variables remaining at zero level.

Dantzig (Linear Programming and Extensions,1965) shows that the relative cost factor for a variable x[i,j] is

$$c[i,j] - u[i] - v[j]$$

where u[i] and v[j] are shadow prices associated with row i and column j respectively. The relative cost factors are zero for basic variables giving m+n-1 equations

$$c[i,j] = u[i] + v[j] \text{ , } x[i,j] \text{ basic ,}$$

in the m+n unknowns u[i] and v[j] . These are solved by setting u[i] arbitrarilly to zero and finding the remaining m+n-1 shadow prices from the m+n-1 equations.

Clearly, if no relative cost factor is negative then the current basic solution is optimal. The shadow prices are computed by CALCV and CALCU which call each other recursively.

| Let c[r,s] = Min[c[i,j] - u[i] - v[j]]

then if c[r,s] $\geq$ 0 the solution is optimal otherwise x[r,s] is the variable to enter the basis.

INBASIS links a node into position r,s , in the orthogonal lists, to store the value of the new variable and PATH constructs a linked list representing the (unique) closed path connecting the variables in the current basis whose values are to be changed by adding or subtracting the value of x[r,s] in such a way as to satisfy the row (supply) and column (demand) constraints. The maximum value of x[r,s] is the value for which some variable in the current basis becomes zero and leaves the basis. This value is found by MAXRS. In desk calculations on small problems, finding a closed path is trivial, but automatic methods cannot use visual aids and the group PATH is very time consuming. In practice, large transportation problems are not solved by this method but usually operate on the dual problem. PATH tries all possibilities starting from position (r,s), stepping alternately along rows and columns from one basic variable to another, and backtracking when no further progress can be made until it eventually finds the unique closed path for the variable x[r,s].

Finally the new basic solution is found by NEWBASIS which alternately adds and subtracts the value of x[r,s] to and from the basic variables around the closed path. The variable which becomes zero leaves the basis having been replaced by x[r,s] . This whole process is repeated until an optimal solution is found.

The following program gives details of the groups mentioned above together with the results for the simple problem with three origins and four destinations referred to on Page 135.

```
DECLVAR IS (A HEADCELL,A LISTSPACE,A SLISTSPACE,A COST,A COSTP,
            A ROWP,A COLP,A SROWP,A SCOLP,A SPARE,A PATHLIST,
            A P,                          ::  Pointers.
            A M,A N, A A,A B,A C,
            A  SUMA,A SUMB,               ::  Problem data.
            A UI,A UJ,                    ::  Simplex multipliers.
            A WS,A WS1,A WS2,A WS3,A COUNT,A MINCIJBAR,
            A MIN )                       ::  Working variables.

ROWHEAD(?) IS (A ROW,A COL,A ACROSS,A DOWN,A VALUE,A U)
COLHEAD(?) IS (A ROW,A COL,A ACROSS,A DOWN,A VALUE,A V)
NODE(?) IS (A ROW,A COL,A ACROSS,A DOWN,A VALUE,A FLAG)
PWORD(?) IS (A SYMBOL,A LINK)
CONS(A X,A Y) IS (SETUP(PWORD,WS,LISTSPACE),
                  SYMBOL(WS) = X, LINK(WS) = Y,
                  EXIT(WS)      )
NLS IS (OSP., NL., EITHER NLS OR NIL.)
SEPR IS (OSP., ( EITHER NLS OR NIL.), ',',
                (EITHER NLS OR NIL.), OSP.)
TRANSPORTATION IS (121,LISTSPACE,0,*2,          :: Initialise list
                                                :: space pointer.
                   (EITHER NLS OR NIL.),
                    READVALUES,                 :: Input problem data.
                    NWRULE ,                    :: Set up initial
                                                :: solution.
                    PRINTSOLN,                  :: Output initial
                                                :: solution.
                    SETUP(NODE,SPARE,LISTSPACE):: Get spare node.
                    SLISTSPACE = LISTSPACE,     :: Save pointer.
```

```
      AGAIN:    CALCV(DOWN(HEADCELL),0),    :: Compute multipliers.

                COSTP = COST,               :: Initialise pointer
                                            :: to cost matrix.
                MINCIJBAR = 0, CIJBAR,      :: Compute cost factors.

                IF MINCIJBAR GE 0 GO TO END:: If optimal solution.

                INBASIS,                    :: Insert spare node.

                R = ROW(SROWP),S=COL(SCOLP):: Note position of
                                            :: new node.
                P = SPARE,FLAG(P)=1,        :: Nodes on closed path
                                            :: are flagged.
                PATHLIST=CONS(SPARE,0),

                PATH,                       :: Find closed path.

                MAXRS,                      :: Maximum value of
                                            :: new variable.
                NEWBASIS, PRINTSOLN,        :: Print new basic
                                            :: solution.
                GO TO AGAIN,

          END:  O/P(NL. , 'OPTIMAL SOLUTION FOUND'), STOP)


READVALUES IS (SETUP(NODE,HEADCELL,LISTSPACE), :: Set up head cell.

                ROW(HEADCELL)=0-1,COL(HEADCELL)=0-1,

                M,SEPR, N, SEPR ,           :: Input dimensions.

:: The following instructions set up and initialise a linked list of
:: row listheads.

                COUNT=1, SUMA=0, WS=HEADCELL,

      NEXTR:    A,SEPR,SUMA=SUMA+A          :: Read next value of A

                ROW(LISTSPACE)=COUNT,

                COL(LISTSPACE)=0-1,

                VALUE(LISTSPACE)=A,

                FLAG(LISTSPACE)=0,

                ACROSS(LISTSPACE)=0, DOWN(WS)=LISTSPACE,

                SETUP(ROWHEAD,WS,LISTSPACE),

                COUNT=COUNT+1,

                IF COUNT LT M GO TO NEXTR,

                DOWN(WS)=HEADCELL,VALUE(HEADCELL)=SUMA,
```

```
:: The following instructions set up and initialise a circular linked
:: list of column list heads.

              COUNT=1, SUMB=0, WS=HEADCELL,

      NEXTC:  B,SEPR,SUMB=SUMB+B,              :: Read next value of B.

              ROW(LISTSPACE)=0-1,

              COL(LISTSPACE)=COUNT,

              VALUE(LISTSPACE)=B,

              FLAG(LISTSPACE)=0,

              ACROSS(WS)=LISTSPACE,DOWN(LISTSPACE)=0,

              SEPTUP(COLHEAD,WS,LISTSPACE),

              COUNT=COUNT+1,

              IF COUNT LE N GO TO NEXTC,

              ACROSS(WS)=HEADCELL,

              IF VALUE(HEADCELL)=SUMB GO TO SKIP,

              O/P(NL., 'SUMA NE SUMB, PROBLEM INFEASIBLE'),

              STOP,

:: The following instructions input the cost matrix.

              SKIP: COST= LISTSPACE,     :: Save pointer to cost
                                         :: matrix.
                    WS1 = M,

             NEXT2: WS2=N,

             NEXT1: C,COOF(LISTSPACE)=C,SEPR,

                    LISTSPACE=LISTSPACE+1,

                    WS2=WS2-1, IF WS2 GT 0 GO TO NEXT1,

                    WS1=WS1-1, IF WS1 GT 0 GO TO NEXT2)
```

```
NWRULE IS (ROWP=DOWN(HEADCELL),
          COLP=ACROSS(HEADCELL),        :: Start at NW corner.
          ACROSS(ROWP)=LISTSPACE,
          DOWN(COLP)=LISTSPACE,         :: Link on first node.
   MORE:  SETUP(NODE,WS,LISTSPACE),
          ROW(WS)=ROW(ROWP),COL(WS)=COL(COLP),
          IF VALUE(ROWP) GE VALUE(COLP) GO TO ACROSS,

          VALUE(WS)=VALUE(ROWP),        :: A supply exhausted.
          VALUE(COLP)=VALUE(COLP)-VALUE(ROWP),
          VALUE(ROWP)=0,
          ACROSS(WS)=ROWP,ROWP=DOWN(ROWP),
          ACROSS(ROWP)=LISTSPACE,       :: Link to next node.
          DOWN(WS)=LISTSPACE,
          GO TO MORE,

 ACROSS:  VALUE(WS)=VALUE(COLP),        :: A demand satisfied.
          VALUE(ROWP)=VALUE(ROWP)-VALUE(COLP),
          VALUE(COLP)=0,
          DOWN(WS)=COLP,COLP=ACROSS(COLP),
          IF COLP=HEADCELL GO TO END,
          DOWN(COLP)=LISTSPACE,         :: Link to next node.
          ACROSS(WS)=LISTSPACE,
          GO TO MORE,
    END:  ACROSS(WS)=ROWP  )
```

```
PRINTSOLN IS (O/P (NL., 'BASIC SOLUTION', NL.),
              P=ACROSS(DOWN(HEADCELL)),
       NEXT:  WS1=ROW(P),WS2=COL(P),WS3=VALUE(P),
              O/P(NL., 'X(',WS1,',',WS2,')=',WS3),
              P=ACROSS(P), IF COL(P) GT 0 GO TO NEXT,
              P=ACROSS(DOWN(P)),
              IF ROW(P) GT 0 GO TO NEXT,
              O/P(NL.)    )
```

Given P, a pointer to a basic node, and IU the shadow price associated with the row of which this node is a member CALCV computes VJ the shadow price associated with the column of which the node is a member. This value of VJ is then passed to CALCU which attempts to compute the UI for the next node in this column and if successful calls CALCV and so on until all shadow prices have been computed. The variable SAVEP used in these groups is in each case a local variable. Initially CALCU is called with P pointing to the first node in the first row and with UI = 0.

```
CALCV(A P,A UI,A SAVEP) IS (SAVEP=P,
                     START: P=ACROSS(P),
                            IF COL(P) GT 0 GO TO SKIP,
                            U(P)=UI,          :: Record shadow price.
                            IF P = DOWN(HEADCELL) GO TO END,
                            P = ACROSS(P),
                      SKIP: IF P=SAVE(P) GO TO END,
                            VJ=CIJ(ROW(P),COL(P)) - UI,
                            CALCU(P,VJ), GO TO START,
                       END: NIL. )
```

```
CALCU(A P,A VJ,A SAVEP) IS (SAVEP=P,
                    START: P=DOWN(P),
                           IF ROW(P) GT 0 GO TO SKIP,
                           V(P)= VJ, P=DOWN(P),
                     SKIP: IF P=SAVEP GO TO END,
                           UI=CIJ(ROW(P),COL(P))-VJ,
                           CALCV(P,UI), GO TO START,
                      END: NIL. )
```

```
:: CIJBAR computes the relative cost factors saving the value and
:: location of the minimum of these.
```

```
CIJBAR IS (SROWP=0, SCOLP=0, WS=0,
           ROWP=DOWN(HEADCELL), COLP=ACROSS(HEADCELL),
    NEXT:  IF COL(COLP) GT 0 GO TO A,
           COLP=ACROSS(COLP), ROWP=DOWN(ROWP),
           IF ROW (ROWP) LT 0 GO TO END,
      A:   WS=COOF(COSTP)-U(ROWP),
           WS=WS-V(COLP),
           IF MINCIJBAR LE WS GO TO B,
           SROWP=ROWP,SCOLP=COLP,MINCIJBAR=WS,
      B:   COL(P)=ACROSS(COLP), COSTP=COSTP+1,
           GO TO NEXT,
     END:  NIL.  )
```

```
::  INBASIS inserts a new node in position (SROWP,SCOLP) representing
::  the new variable entering the basis.

INBASIS IS (P=SROWP,

  NEXTCOL:  IF COL(ACROSS(P)) GT COL(SCOLP) GO TO RINSERT.

            IF COL(ACROSS(P)) LT 0 GO TO RINSERT,

            P=ACROSS(P), GO TO NEXTCOL,

  RINSERT:  ACROSS(SPARE)=ACROSS(P),ACROSS(P)=SPARE,

            P=SCOLP,

  NEXTROW:  IF ROW(DOWN(P)) GT ROW(SROWP) GO TO CINSERT,

            IF ROW(DOWN(P)) LT 0 GO TO CINSERT,

            P=DOWN(P), GO TO NEXTROW,

  CINSERT:  DOWN(SPARE)=DOWN(P),DOWN(P)=SPARE,

            ROW(SPARE)=ROW(SROWP),COL(SPARE)=COL(SCOLP),

            VALUE(SPARE)=0,FLAG(SPARE)=0  )
```

PATH starts at the new node and steps alternately along rows and columns searching for a closed path. R and S are row and column numbers of the new node.

```
PATH IS (NEXTROW: P=ACROSS(P),

                  IF COL(P) GT 0 GO TO SKIP1,

                  P=ACROSS(P),

           SKIP1: IF FLAG(P)=1 GO TO REJECTROW,

                  FLAG(P)=1, PATHLIST=CONS(P,PATHLIST),

                  IF COL(P)=S GO TO END,

                  GO TO NEXTCOL,

       REJECTROW: P=SYMBOL(PATHLIST),

                  PATHLIST=LINK(PATHLIST), FLAG(P)=0,

                  DELETE(PWORD,LISTSPACE), GO TO NEXTCOL,

        NEXTCOL:  P=DOWN(P),

                  IF ROW(P) GT 0 GO TO SKIP2,

                  P=DOWN(P),
```

```
          SKIP2: IF FLAG(P)=1 GO TO REJECTCOL,
                 FLAG(P)=1, PATHLIST=CONS(P,PATHLIST),
                 GO TO NEXTROW,
      REJECTCOL: P=SYMBOL(PATHLIST),
                 PATHLIST=LINK(PATHLIST), FLAG(P)=0,
                 DELETE(PWORD,LISTSPACE), GO TO NEXTROW,
            END: NIL. )

:: MAXRS finds the maximum value at which the new variable enters
:: the basis.

MAXRS IS (MIN=VALUE(SYMBOL(PATHLIST)),
          SPARE=SYMBOL(PATHLIST), WS=PATHLIST,
     NEXT: IF VALUE(SYMBOL(WS)) GE VALUE(SPARE) GO TO SKIP,
          MIN=VALUE(SYMBOL(WS)), SPARE=SYMBOL(WS),
     SKIP: WS=LINK(LINK(WS)),
          IF WS NE 0 GO TO NEXT )


NEWBASIS IS (NEXT: VALUE(SYMBOL(PATHLIST))=VALUE(SYMBOL(PATHLIST))-MIN,
                   FLAG(SYMBOL(PATHLIST))=0, PATHLIST=LINK(PATHLIST),
                   VALUE(SYMBOL(PATHLIST))=VALUE(SYMBOL(PATHLIST))+MIN,
                   FLAG(SYMBOL(PATHLIST))=0,PATHLIST=LINK(PATHLIST),
                   IF PATHLIST NE 0 GO TO NEXT,
                   WS=SPARE,
           GODOWN: WS=DOWN(WS), IF DOWN(WS) NE SPARE GO TO GODOWN,
                   DOWN(WS)=DOWN(SPARE),
                   WS=SPARE,
        GOACROSS: WS=ACROSS(WS),IF ACROSS(WS) NE SPARE GO TO GOACROSS,
                   LISTSPACE=SLISTSPACE)


*ENTER(TRANSPORTATION)
```

Results from a program using the foregoing routines to solve the 3*4 problem given on Page 135:

```
BASIC SOLUTION
X(1,1) = 7
X(1,2) = 8
X(2,2) = 4
X(2,3) = 8
X(3,3) = 4
X(3,4) = 14

BASIC SOLUTION
X(1,1) = 3
X(1,2) = 12
X(2,1) = 4
X(2,3) = 8
X(3,3) = 4
X(3,4) = 14

BASIC SOLUTION
X(1,2) = 12
X(1,3) = 3
X(2,1) = 7
X(2,3) = 5
X(3,3) = 4
X(3,4) = 14

BASIC SOLUTION
X(1,2) = 12
X(1,3) = 3
X(2,1) = 7
X(2,4) = 5
X(3,3) = 9
X(3,4) = 9

BASIC SOLUTION
X(1,2) = 3
X(1,3) = 12
X(2,1) = 7
X(2,4) = 5
X(3,2) = 9
X(3,4) = 9

OPTIMAL SOLUTION FOUND
```

PART III

The Implementation of Data Structures

The Implementation of Data Structures.

The preceding section was concerned with elementary list processing in BCL. The programs described illustrated the facilities in BCL which make it particularly suitable for the manipulation of linked structures and we emphasised that the shape and size of the nodes used and the manner in which they were linked together was defined by the program. In this section we shall consider extensions which would allow the user to define new types or classes of objects in his program. We are particularly interested in the definition of named classes of structured objects such as linked lists and trees and the operations which may be performed on them.

Before going into the details of specific extensions to BCL it is convenient to consider some recent developments in Algol which form a background to the developments proposed here. The chief impotencies of Algol 60 are well known; the most serious being input/output, string/character handling, lists and complex numbers. Since the publication of the revised version of Algol 60 in 1962, proposals for remedying some of these deficiencies have received considerable attention in the literature. D. Knuth (May, 1964), C.A.R. Hoare (November, 1965), and N. Wirth and C.A.R. Hoare (1966) have proposed modest extensions which have already been implemented on a few machines and are typical of the features proposed for Algol 6X. The most recent developments are embodied in Algol 68 which includes a number of important innovations but is not so closely related to Algol 60. It seems that there may still be a place for an Algol X, less ambitious than Algol 68, which can be implemented without departing significantly from the concepts of Algol 60.

A new type, character, should be provided so that implementors can organise storage more efficiently. System procedures for the input/output of whole lines could transfer characters to something like a character array. (They could probably be written trivially as for-statements at the cost of the inefficiency of calling a system character routine once per character.)

Actually something more than a character array is required. The array provides a means of allocating to a single identifier a block of storage much larger than a single variable; this is merely the essential first step. The next requirement is a means of structuring this, and the way it is met must make provision for other needs involving structures.

No implementor of Algol 60 would find it difficult to implement character arrays and standard system routines for input and output, and it is possible to take one further step without departing from the general scheme of organisation required for Algol 60. One could allow a declaration such as

structure (character[10], Y,Z; real W,X) h,j,k;

This declares three variables, h,j,k, each consisting of two 10-character strings and two reals, and allows references to the second string of the third variable by the notation Z(k). Such a system would, for example, allow a line for output to be built up by assignments to specific fields within it. By using this technique recursively,types are created which are the equivalent of a complete Cobol record, and moreover, one can declare as many instances of any type as one cares to find identifiers for,and can create and destroy them on the stack with the full freedom of the usual rules for local

storage in a block structure language. What, then, can not be done yet?

Complex numbers can be dealt with by declaring

structure (real Re, Im) z,y,x ...

and calling procedures such as sumcomplex(x,y,z), but z := x + y is not defined for variables of type complex. One can define

structure (character [10] f, character [20] g)

structure (character [20] h, character [10] k)

but one cannot define a 30-character string so that it can be analysed in either way at will (not without making very dangerous assumptions about the scopes of the selector function identifiers). One can define, perhaps,

structure (integer Head,Tail) array F[1:1000];

for i:=1 to 999 do Tail (F[i]) := i+1 ; etc.

and provide oneself with a list processing area, but one is then confined to working with simple lists of integers, in which, moreover, every list element has an unnecessarily explicit name. For several reasons the thing cannot be done properly by writing what a study of Algol 68 would suggest,

structure list = (union (atom, list) Head, list Tail) .

The first reason is that union has not been introduced. A second is that there is no provision for the naming of structures. Our objective is to develop a system in which named structures of any complexity can be defined in this way and handled as single units of data.

At this point it is convenient to return to BCL which is potentially more flexible than Algol 60. Input and output of structured data is already well defined in BCL, and, as we have

already seen, it is possible to represent data structures using facilities which already exist. The main deficiencies in BCL are deficiencies in the current implementations rather than in the language, although there is some confusion to be cleared up in the language itself, for example between indefinite groups and structure declarations. Types real and boolean are not yet implemented, arrays, functions, groups with parameters, and block structure are not generally available and the scope of identifiers is not well defined. Facilities already exist for handling any specified number of characters, for example, an identifier declared as type 7C may be used for the input/output and storage of from 1 to 7 characters. However, the space allocated for a string of 7 characters stored as a single unit declared as type 7C is not the same as that for 7 characters each of type 1C.

In section §2.4.1 various structures were declared using the BCL concept of an indefinite group. In particular we defined a simple two-field structure named PWORD as follows

```
PWORD(?) IS (A HEAD,A TAIL) .
```

The reader is reminded that the BCL compiler interprets this as the declaration of a group of variable declarations. In theory this group could be input or output according to the mode in which it is entered although in practice the result of inputting or outputting such a group is unpredictable. The query indicates that the group is indefinite and therefore any variables declared in the group are allocated stack space, not fixed space. Thus the variables HEAD and TAIL, each of type A (address) are allocated addresses relative to some stack pointer. Since, as it happens, HEAD is allocated a relative address (or offset) zero and TAIL an offset of one address

field it is convenient to use these names as selector function identifiers for referencing fields within a node or record. We note also that as the names HEAD and TAIL are in no way associated with PWORD in the identifier records set up by the existing BCL compilers, they may be used in conjunction with any base pointer. We see that in BCL there is some confusion between selector functions and variables, and between groups and structure declarations. We need to distinguish clearly between groups of objects for input and output and the definitions of named structures, or shapes, with which are associated selector functions, and for which no object code is generated. A structure declaration merely gives information to the compiler. We can allow a shape declaration of a named structure such as

SHAPE.(COMPLEX) IS (REAL RE,REAL IM)

This defines a structure named COMPLEX as a pair of real numbers. The field names, or selectors, RE amd IM are used to refer to the real and imaginary parts of any object which is declared to be of type COMPLEX. They are associated only with objects of type complex and in general have no meaning if used in conjunction with any object which is not of type complex. Having defined the type complex we can now write declarations such as

COMPLEX A

which says that A is an object which belongs to the class of objects named COMPLEX i.e. A is of type complex. The real part of A is referred to as RE(A) and the imaginary part as IM(A). The pair of real numbers RE(A) and IM(A) are together referred to as A and can be handled as a single object. Note that the declaration of SHAPE.(COMPLEX) does not result in any allocation of storage, nor is any object code generated; it merely specifies the amount of storage

to be allocated to any object which belongs to the class named COMPLEX. A dictionary record containing this information is set up for the name COMPLEX and the type of the name COMPLEX is 'SHAPE'. Dictionary records are also set up for RE and IM each of type REAL, and these records are linked to that of COMPLEX to indicate the association between the structure and its selectors. The declaration COMPLEX A causes space for two real numbers to be allocated to the object named A. Clearly, more complicated structures could be defined involving as many fields as are required. It may also be possible to specify alternative shapes belonging to the same class by means of a declaration such as

```
|        SHAPE.(NODE) IS ( (EITHER 8C HEAD1
|                           OR     A  HEAD2), A TAIL)
```

This says that an object of type node is a two field object of which the first is either an 8-character field or an address field and the second is an address field. In circumstances such as this, in which alternative structures require different amounts of storage, the maximum amount specified is always allocated.

We turn now to more complicated structured objects for which the total storage space required is indefinite. A simple example of such an object is a linear linked list. There are obvious advantages in being able to handle objects of type list as single units of data rather than as a number of separate elements but clearly the total amount of storage required depends upon the number of elements in the list and this may vary dynamically. We deal with

the problem in the following way

```
SHAPE.(LIST) IS (REF.(NODE) )
SHAPE.(NODE) IS (REF.(ATOM) HEAD, REF.(NODE) TAIL)
```

where an ATOM is user defined and may be an integer, a real number, a character string or any other object which the user may wish to define in his program. REF. is a new system word meaning 'the address of' its argument. Thus REF.(NODE) means storage for the address of an object of type NODE, but it also implies some check that the address is that of a NODE. The declaration LIST P results in the allocation of storage space for an address. This is initialised to zero, representing a null list, and in any future assignments the assigned value must be either zero or the address of a node. A NODE is in turn defined as a structure, or shape, consisting of two fields named HEAD and TAIL. Each of these is a reference to an object and therefore, when an object of type NODE is 'set up', the space allocated is for two address fields referred to as HEAD and TAIL. The fact that TAIL is also defined as a reference to a NODE does not lead to any complications. Any reference to an object requires an address size field whatever the object may be. As a zero address implies a null reference, a list is terminated by a node of which the TAIL is null.

A node could have been defined as

```
SHAPE.(NODE) IS ((EITHER REF. (ATOM)
                  OR     REF. (NODE)) HEAD,
                         REF. (NODE)  TAIL )
```

in which case the head field may also point to a (sub)list so giving a binary tree structure. Alternatively, the programmer could include a NODE in his definition of an ATOM and use the former definition of a NODE.

With any class of structured objects, such as those defined as type list, in addition to the selector functions associated with the nodes, we also need a constructor function which is used to build structures dynamically. For example, consider

CONS.(NODE, X,Y)

Here CONS. is a system defined function which deals with the dynamic allocation of space for structures. The first parameter is the type of the object for which space is required and therefore specifies, indirectly, the amount of storage space required. The remaining parameters are the values to be stored in the fields of the specified object. Their number and type can be checked, at compile time, with the selector functions defined for the particular structure. The value of the function CONS. is the address of the structure set up. As an example of the use of the CONS. function, suppose that NODE is defined by

SHAPE.(NODE) IS (REF.(ATOM) HEAD, REF.(NODE) TAIL)

then an instruction to insert an additional element X at the front of a list, L, of such nodes is

L := CONS.(NODE,X,L).

The use of the function CONS. implies some mechanism for the dynamic allocation of storage. Under certain circumstances this space might be allocated on the run-time stack, but it may be necessary to allocate space from a separate pool (or heap in Algol

68 terms) of free space. We shall return to the organisation of free space later.

So far then, we can define new types of structured objects and operations to be performed upon them. For example, if X,Y and Z are of type LIST we can define a function

APPEND(LIST P,LIST Q),

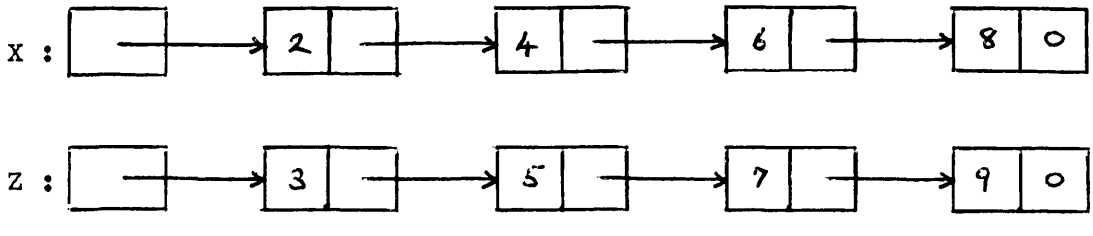
similar to that in §2.41 , and write Z := APPEND(X,Y). We can also write Z := X meaning 'copy the value of X into Z' (i.e. copy the address of a node from X to Z), the result being that Z points to the same node as X. If we wish to assign to Z a copy of the whole list referred to by X then we could use Z:= COPY.(X), where COPY. is a system defined function. Another useful system defined function would be WHOLE., used in the following way

WHOLE.(Z) := X

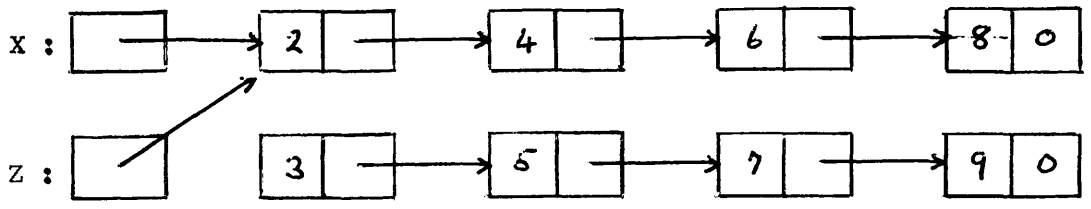
which is equivalent to

HEAD(Z) := HEAD(X), TAIL(Z) := TAIL(X) .

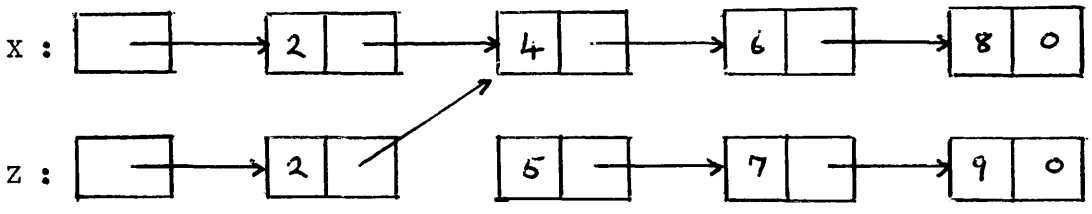
Note that when the function WHOLE. is used the system does not look into the structure or contents of the object referred to by X, it simply needs to know the size of the object, which must be the same as the size of object to which Z refers, and then copies it exactly. Suppose then that X and Z refer to linear linked lists of integers



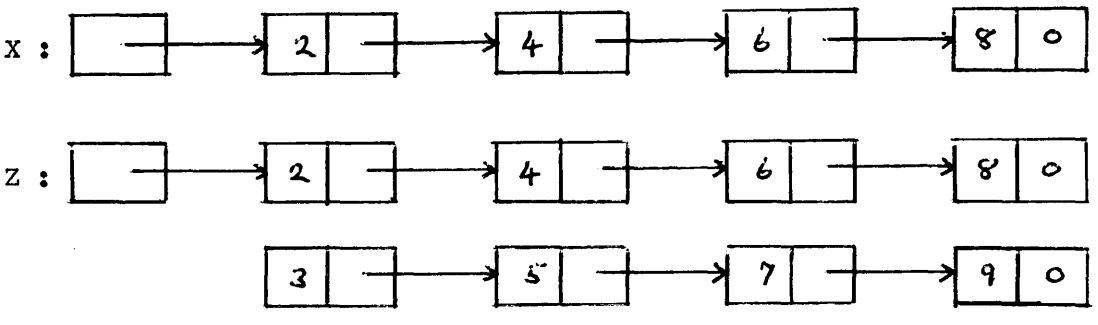
then Z := X gives



WHOLE.(Z) := X gives



and Z := COPY.(X) copies every node in the list giving



Two problems which arise in the scheme described so far are how to deal with

(a) denotations,

and (b) implicit type conversions.

It should be possible to assign to a complex number, Z, a pair of real numbers, being the real and imaginary parts, by writing

Z := (1.62,2.19)

but at present we must write

Z := CONS.(COMPLEX,1.62,2.19).

For a simple structure such as that defined as COMPLEX it would be easy to recognise (1.62,2.19) as an abbreviation for CONS.(COMPLEX,1.62,2.19), the type of the structure to be set up

being specified by the variable to which the value is to be assigned. In the case of structures which are defined recursively this is not so straightforward. A common written representation of list structures uses commas as atom separators, and brackets to denote substructures, for example

(2,3,(1,7,8),4,(5,2))

is a list with two sublists. It should be possible to assign such a list denotation to a variable of type LIST but how does the compiler interpret the right hand side of the statement

L := (2,3,(1,7,8),4,(5,2)) ?

A simple answer, which seems to be the only solution to the problem, is to define, for each type of denotation used in the program, a function specifying the procedure for converting the denotation to the appropriate internal representation, using the CONS. function to set up space as required. A function for the list denotation above would have as a single parameter the character string

'(2,3,(1,7,8),4,(5,2))' .

Such a user defined conversion function would be similar to that for reading the value of a list from an input stream of characters. If the character string is stored in the constants area in exactly the same format as that in which characters appear in the input stream, and if any area of store can be regarded as an input stream, then the evaluation of a denotation is exactly the same as 'inputting' a list from the constants area. For a list of nodes defined by

```
| SHAPE.(NODE) IS ((EITHER INTEGER
|                  OR      REF.(NODE)) HEAD,REF.(NODE) TAIL)
```

a suitable function for input of a list is

```
| READLIST IS (REF.(NODE) WS,      ::   declare a local ref. to NODE
|              OSP., '(', WS:=CONS.(NODE,HDELEMENT,TLELEMENT),')',
|                                                          EXIT(WS))

| HDELEMENT IS (INTEGER I,         ::   declare local integer I
|               OSP., EITHER INPUT(I), EXIT(I)
|                      OR     EXIT(READLIST) )

| TLELEMENT IS (OSP,.EITHER ',',EXIT(CONS.(NODE,HDELEMENT,TLELEMENT))
|                     OR     EXIT(0) )
```

These functions assume that characters are to be input from the normal input stream. It would be necessary to specify, by means of a parameter, the actual stream to be used. The only alternative to a set of user defined functions for dealing with denotations seems to be an explicit representation of the denotation, in terms of the CONS. function. In the example given above this is obviously much too cumbersome, the full representation being

```
CONS.(NODE,2,CONS.(NODE,3,CONS.(NODE,CONS.(NODE,1,CONS.(NODE,7,
      CONS.(NODE,8,0))),CONS.(NODE,4,CONS.(NODE,CONS.(NODE,5,
                                    CONS.(NODE,2,0)), 0)))))
```

There is no need to comment further on the unsuitability of such a representation.

The other problem mentioned above is that of type conversions. Conversions are easily accomplished explicitly, for example we can convert a real value X to complex form by means of the statement

```
Z := CONS.(COMPLEX,X,0)
```

where Z is of type complex, but is it meaningful to write

```
Z := X ?
```

Similarly, if both X and Y are real, may we write

Y := IM(X)

when what is meant is

Y := IM(CONS.(COMPLEX,X,0)) ?

In both of these examples it is not difficult to determine automatically the conversion implied. Again if X is of type integer and L of type LIST then obviously

L := X

is an abbreviation for

L := CONS.(NODE,X,0).

With the aid of a conversion table we could specify any implicit conversions which are valid, together with the results of such conversions, but this is complicated by the fact that we are allowing the user to define additional types, in terms of system defined types, and should therefore also allow him to extend the conversion table. This might be possible through the use of an INTERPRET statement used as follows

```
| INTERPRET.(TYPE.COMPLEX := TYPE.REAL)
        AS (TYPE.COMPLEX := CONS.(COMPLEX,TYPE.REAL,0))
```

It is immediately clear that such a statement would be very powerful, allowing the user to define, not only implicit conversions, but a wide range of infix operations on objects whose type may be defined in the program. It would be possible to extend the meaning of existing system defined operators to deal with new types, and to introduce new operators. However, this is where we draw the line in this work. The implementation of the INTERPRET statement presents interesting and challenging problems but is too

complicated to fit into the present scheme of things and we propose that at this stage, all conversions involving user defined types should be explicit.

We return now, very briefly, to the problem of storage allocation in a block structured system in which the user is allowed to define structured objects. Under certain circumstances a new instance of an object may result in the allocation of space on the run-time stack. This space is allocated and reclaimed according to the normal rules of block structure systems. What must be avoided at all times, is a situation in which the value of a reference is a stack address which is either undefined or has been redefined, i.e. at no time must we allow references to stack addresses which are above the stack pointer. To avoid this we have another pool of available storage which we shall call the heap, as in Algol 68. Simple rules for the allocation of storage are:

(1) At a declaration of an object, stack space is allocated and initialised to zero. The declaration is valid only in the block in which it is declared and anything assigned to the object is lost on exit from that block.

(2) Any allocation of anonymous storage through the use of the CONS. function is made from the heap.

(3) We could rule that no stack address may ever be assigned either to another stack address or to heap space but that when we need to assign a stack address the COPY. function is used to raise heap space into which all stack values involved in the transfer are copied. Consider first the following block of program which uses a mixture of Algol and BCL notations.

```
| BEGIN.      SHAPE.(LIST) IS (REF.(NODE))
|             SHAPE.(NODE) IS (INTEGER HEAD,
|                                 REF.(NODE) TAIL)
|             LIST A
|             FOR. I := 1 STEP. 1 UNTIL. 3 DO.
|             BEGIN. LIST B
|                    B := CONS.(NODE,I,A)
|                    A := B
|             END.
|             .......
|             .......
| END.
```

Any space allocated at a declaration is stack space. The first two declarations are SHAPE. declarations which merely give information to the compiler. The variable A is then declared as type LIST and is allocated an amount of stack space which is sufficient for an address. This field is initialised to NIL, which is represented by zero. Then stack space is allocated for the integer control variable which is initialised to 1 and the FOR. block is entered. Inside this block stack space is allocated to B. The statement

```
B := CONS.(NODE,I,A)
```

involves the allocation of heap space for a node, using the CONS. function. To the fields of this node we assign the value of the integer I and the value of the variable A. Now since A is of type LIST it is a reference to a node and all nodes are in heap space so the assignment involves a heap address and is in order. Next we assign to A the value of B which is again a reference to heap space.

On completion of this block the local stack space is lost and is raised again on reentry, however, the nodes have been set up in heap space and are still available so our rules are working efficiently in this case.

Now consider the following program block.

```
| BEGIN.     SHAPE.(LIST) IS (REF.(NODE))
|            SHAPE.(NODE) IS (REF.(INTEGER) HEAD,
|                             REF.(NODE) TAIL)
|            LIST A
|            FOR.   I:=1 STEP. 1 UNTIL. 3 DO.
|            BEGIN. INTEGER J
|                   LIST B
|                   J := I
|                   B := CONS.(NODE,J,A)
|                   A := B
|            END.
|            .......
|            .......
| END.
```

Here again, stack space is allocated to A and I and, inside the FOR. block, to J and B. The CONS. function again allocates heap space but now the value to be assigned to the HEAD field must be a reference to an integer so the value assigned is not the value of the integer J but the address of a location containing that integer value. We have ruled that under no circumstances should a stack address be assigned so a copy of the integer J is stored in heap

space and the address of this heap space is assigned to the HEAD field of the node. With these rules the result is exactly what is required, a list of references to three different integer values 1, 2 and 3.

We now ask if there are circumstances under which it would be safe to assign a stack address either to another stack location or to heap space. First consider the assignment of a stack address to a stack address. Clearly, it would be dangerous to make such an assignment to a stack address at a lower level since when the stack space is reclaimed on exit from a block we shall have a reference to undefined stack space. There seems to be no objection to the assignment of a stack address to another stack address at the same or a higher level on the stack, since in this case the reference becomes undefined at the same time as, or before, the value to which it refers. It would be safe to assign a stack address to heap space, only if all references to that heap space become undefined before, or at the same time as, the stack address becomes undefined. This the compiler cannot check and the responsibility must be placed firmly upon the programmer. Therefore we must not allow the programmer to allocate stack addresses either to the heap or to stack addresses at a lower level without realising what he is doing and it is suggested that such an assignment should be possible only through the use of a special system defined assignment function. With these rules it appears that the programmer is adequately protected and if there are special circumstances, and two way lists are a case in point, under which the programmer wishes to get round the general assignment rules the special assignment function would allow him to do so.

In conclusion, we may ask what we have achieved. The proposals put forward in this section for the future development of BCL provide a system in which data structures are easily defined, operations on such structures are available in the form of functions and the confusion between structure definitions and group declarations removed. Whilst the system still falls short of the proposals for Algol 68 it does provide most of the facilities that the ordinary programmer is likely to need. Furthermore, what has been proposed here is an extension of an existing system and fits quite naturally into the general scheme of BCL.

EPILOGUE

In retrospect it might seem that the most important part of this thesis has been concerned with LSIX and that BCL has been used merely as a tool for its implementation. However, although only one section is devoted to the use of BCL itself as a list processor, experience in teaching list processing in both LSIX and BCL has shown without any doubt that almost all students prefer BCL to LSIX. This might be because the particular group of students concerned are more interested in applications of computers and have not been particularly interested in working close to the machine itself, or it might be because the BCL list processor is a high level language, having all the facilities of most other high level languages, and at the same time it allows the user to include, anywhere in his program, assembly language instructions for the particular machine concerned, so providing the low-level facilities available in LSIX. It is the author's experience that whenever one attempts to write a program in LSIX one soon yields to the temptation to use BCL instead, leaving the system to deal with the chores associated with passing parameters, manipulating stacks for recursion and other low-level aspects of programming.

The original objective in this work was to design and implement a flexible, machine independent system for teaching list processing. Many list processing systems were in general use but none was sufficiently flexible to allow the user to define the shape and size of nodes from which list structures could be built. There was no general purpose language which included suitable list processing facilities and no list processing system with efficient arithmetic facilities. L6 was designed by Knowlton to provide a flexible, low-level, machine independent system which, it was

claimed, was ideal for teaching the fundamentals of list processing. L6 has a wide range of tests and operations, including the basic fixed point arithmetic operations, but no floating point arithmetic is included. The author's implementation of LSIX in BCL is largely machine independent. When BCL compilers become more generally available on other machines it will be a simple matter to transfer to those machines any systems written in BCL. Only eighteen per cent of the source statements in the LSIX compiler are machine dependent, in the object code these represent an even smaller percentage of the compiler, and only this part needs to be rewritten for the new machine. The Atlas BCL compiler is now written in BCL itself and likewise the task of providing BCL compilers for other machines is simpler than it might otherwise be.

The LSIX compiler described in PART II of this thesis is written in one of the first versions of BCL. Its efficiency would be greatly improved if it were updated to take advantage of recent developments in BCL. As the LSIX compiler is interpretive, a number of modifications can be carried out by rewriting some of the subroutines, and improvements such as the generalisation of storage organisation routines are the subjects of a number of projects currently being carried out by M.Sc. students. Such projects enable students to get to the heart of a list processing system and to understand the fundamental operations which underlie its implementation.

It has been claimed that LSIX is a convenient medium for the implementation of other programming languages but here again BCL has proved to be far superior. The implementation of LSIX in BCL was

the first use of BCL as a compiler-compiler. Soon afterwards BCL was written in itself and it is now being used to implement a wide range of languages including FORTRAN and Algol. The list processing facilities which have been added to BCL by the author have further increased its suitability for use in the teaching of elementary compilation techniques. After a short basic course in list processing in BCL, students have written programs ranging from the sorting and merging of simple linked lists to the simplification of algebraic formulae. In section 2.4.2 we gave an illustration of a program to input expressions and store them in the form of binary trees. Students have written similar programs to input and manipulate expressions and also to generate optimised machine code for the evaluation of expressions. BCL has been used in a file processing course to describe various internal structures associated with files which can be represented by directed graphs. Other applications include the definition, and some aspects of the implementation, of SOL, a simulation language, this also is the subject of an M.Sc. project.

To summarise, the main advantages of BCL as a language are:

(1) Students with no previous knowledge of computers and programming have found the language easy to learn.

(2) Programs written in BCL may consist of instructions to be obeyed in sequence in the conventional way, using labels and GO TO commands for the transfer of control, but they may also be written using a functional notation as in LISP. The functions used are easily defined by the user.

(3) The ease with which recursion may be used leads to ready acceptance of recursive programming techniques.

(4) BCL will eventually include arithmetical facilities at least as powerful as those of FORTRAN IV - an advantage which most list processors do not have.

(5) BCL is a high level language and programs written in it may be machine independent. Alternatively, if the user so wishes, he may get close to the machine by writing assembly language instructions for the machine concerned anywhere in his program.

(6) The emphasis on the input and output of structured data make BCL ideally suitable as a compiler-compiler.

(7) The language is equally suitable for systems analysis, commercial data processing and for writing mathematical programs.

The particular advantages of BCL as a list processor are:

(1) List processing facilities are available through the definition of simple groups of commands, and linked memory techniques can therefore be used as basic programming tools in any program.

(2) The list processing groups of commands are defined by the programmer himself, so allowing him to set up and manipulate list structures of any complexity using nodes of any desired size. The shape and size of a node are defined by the program. Several different sizes of node partitioned into fields in several different ways may be used in the same program.

The work described in this thesis constitutes the first round in an iterative process to investigate the effectiveness of systems for teaching list processing techniques. In the present year, students at the University of London Institute of Computer Science, and at Birkbeck College, both graduates and undergraduates, have been introduced to the BCL list processor and the results have exceeded all expectations. Whilst further modifications and extensions, including those proposed in Part III, may be carried out in the light of experience, and as the process converges, the author is already confident that in BCL we have a system whose power and flexibility are second to none.

BIBLIOGRAPHY

Bobrow, D.G. and Raphael, B. (1964)
'A comparison of list processing computer languages',
C.A.C.M., Vol. 7, pp. 231-240.

Dantzig, G.B. (1965)
'Linear Programming and Extensions', Chapter 14,
Princeton University Press, 1965.

Foster, J.M. (1967)
'List Processing', Macdonald, 1967.

Hendry, D.F. (1966)
'A Provisional BCL Manual',
University of London Institute of Computer Science.

Hendry, D.F. and Mohan, B. (1968)
'A BCL1 Manual',
University of London Institute of Computer Science,
ICSI 103.

Higman, B. (1969)
'ALGO - LXX',
University of London Institute of Computer Science,
ICSI 206.

Hitchcock, F.L. (1941)
'The Distribution of a Product from Several Sources
to Numerous Localities',
J. Maths. Phys., Vol. 20, pp. 224-230.

Hoare, C.A.R. (1965)
'Record Handling', Algol Bulletin 21.

Knowlton, K.C. (1965)
'A Fast Storage Allocator',
C.A.C.M., Vol. 9, pp. 616-625.

(1966)
'A Programmer's Description of L^6 ',
C.A.C.M., Vol. 9, pp. 616-625.

Knuth, D.E. (1964)
'A proposal for Input/Output Conventions in Algol 60',
C.A.C.M., Vol. 7, pp. 273-283.

– (1968)
'The Art of Computer Programming',
Vol. 1 Fundamental Algorithms, Chapter 2.
Addison-Wesley.

McCarthy, J. (1960)
'Recursive functions of symbolic expressions and
their computation by machine, PART I' ,
C.A.C.M., Vol. 3, pp. 184-195.

– (1962)
'LISP 1.5 Programmer's Manual'. M.I.T. Press

Newell, A., Shaw, C. and Simon, H. (1965)
'The Logic Theory Machine',
I.R.E. Transactions on Information Theory IT-2 pp. 61-70.

- (1960)
'Information Processing Language V, Manual',
Rand Corporation, P1918.

Schorr, H. and Waite, W.M. (1967)
'An Efficient Machine-Independent Procedure for
Garbage Collection in Various List Structures',
C.A.C.M., Vol. 10, pp. 501-506.

Weizenbaum, J. (1963)
'Symmetric List Processor', C.A.C.M., Vol. 6, pp. 524-544.

Wilkes, M.V. (1964)
'An experiment with a self-compiling compiler for
a simple list-processing language',
Annual Review in Automatic Programming,
Vol. 4, pp. 1-48, Pergamon Press.

Wirth, N. and Hoare, C.A.R. (1966)
'A contribution to the development of Algol',
C.A.C.M., Vol. 9, p. 413.

Woodward, P.M. (1966)
'List Processing' in 'Advances in Programming and
non-numerical computation'.
Pergamon, 1966.

APPENDIX 1

Output from a complete LSIX program

A complete list of tests and operations in LSIX has been published in Tables 1 and 2 of the author's paper entitled

'The definition and implementation of LSIX in BCL'

which forms Appendix 7 of this report.

The use of a few of these tests and operations is illustrated by a program to read a sequence of numbers, each of which is terminated by a single space, sort them into ascending order and output them. This program is written as three main subroutines, INPUT, ORDER and OUTPUT which are described in some detail in the paper referred to above.

In this Appendix we give actual output from the Atlas computer for the same program. Both source and object listings are requested to illustrate the object code generated for each source instruction and a number of diagnostic outputs are included, using the 'Print List' operation and the system subroutine 'STATE', to show the state of the system at various stages during the execution of the program.

LSIX program

```
*LSIX, SOURCE, LIST
( 25 ,PRH, PROGRAM TO SORT NUMBERS)
(*20000000,SS,4,*20000400)(3,DD,0,23)(1,DA,0,23)(2,DB,0,23)
            (DO,STATE)
            (DO,INPUT)(DO,STATE)(DO,ORDER)(DO,OUTPUT)(DO,DUMP) END
INPUT        (W,GT,4)(WB,E,32768)(S,FC,X)(X,GT,1)(S,FD,1)(0,D1,0,5)
            (DO,STATE)
NEXT        (W,GT,4,WA)(WAD,P,W)(WB,E,0)
RD          (X1,IN,1)(1,PR,X1)
        NOT (X1,EH, ) THEN (WB,L,6,X1) RD
        IF  (WB,E,0) THEN (R,FD,1)(X,FR,0)(R,FC,X)(W,PL,A,10) DONE
            (WB,DB,WB) NEXT
ORDER       (S,FC,X)(X,P,WA)
ND      IF  (XA,E,0) THEN (R,FC,X) DONE
BACK    IF  (XB,L,XDB) THEN (XB,IC,XDB)(X,D) BACK
            (X,A) ND
OUTPUT      (W,FR,WA)(S,FC,X)(1,PR,77)
            (W,PL,A,10)
ANYMORE IF  (WA,E,0) THEN (W,FR,0)(R,FC,X) DONE
            (X,BD,WB)(X,ZB,X)(6,PR,X)(W,FR,WA) ANYMORE
*ENTER
```

LSIX LISTING OF OBJECT CODE

```
*00110000  LINE 2
  00001007  00110060  00000000  00000020  00000360  00000030
  00102404  00000000  00102410  00000000  00102550  00000000
*00110060  LINE 3
  00004007  00110330  00000000  00000030  00000450  00000030
  00102554  00000000  00102560  00000000  00102564  00000000
  00000440  00000030  00102570  00000000  00100320  00000000
  00102574  00000000  00102600  00000000  00000440  00000030
  00102410  00000000  00100240  00000000  00102574  00000000
  00102600  00000000  00000440  00000030  00102604  00000000
  00100260  00000000  00102574  00000000  00102600  00000000
*00110330  LINE 4
  00001007  00110360  00000000  00000040  00000510  00000000
*00110360  LINE 5
  00005047  00110500  00000000  00000050  00000430  00000010
  00110500  00000000  00000510  00000000  00000430  00000010
  00111674  00000000  00000430  00000010  00112324  00000000
  00000520  00000000
*00110500  LINE 6
  00006007  00110744  00000000  00000060  00000460  00000020
  00101230  00000000  00102560  00000000  00000130  00000020
  00101230  00100260  00000000  00102610  00000000  00000410
  00000010  00101234  00000000  00000460  00000020  00101234
  00000000  00102410  00000000  00000420  00000010  00100020
  00000000  00000440  00000030  00102574  00000000  00100020
  00000000  00102574  00000000  00102614  00000000
*00110744  LINE 7
  00001007  00110774  00000000  00000070  00000510  00000000
```

```
*00110774  LINE 8
  00003007  00111154  00000000  00000100  00000460  00000030
  00101230  00000000  00102560  00000000  00101230  00100240
  00000000  00000140  00000020  00101230  00100240  00100320
  00000000  00101230  00000000  00000130  00000020  00101230
  00100260  00000000  00102574  00000000
*00111154  LINE 9
  00002007  00111264  00000000  00000110  00000330  00000020
  00101234  00100020  00000000  00102410  00000000  00000360
  00000020  00102410  00000000  00101234  00100020  00000000
*00111264  LINE 10
  40101017  00111410  00111334  00000120  00000010  00101234
  00100020  00000000  00102410  00000000  00000340  00000030
  00101230  00100260  00000000  00102620  00000000  00101234
  00100020  00000000  00111154
*00111410  LINE 11
  30104027  00111610  00111460  00000130  00000010  00101230
  00100260  00000000  00102574  00000000  00000540  00000010
  00100020  00000000  00000320  00000020  00101234  00000000
  00102574  00000000  00000530  00000010  00101234  00000000
  00000400  00000030  00101230  00000000  00100240  00000000
  00102624  00000000
*00111610  LINE 12
  00001017  00111674  00000000  00000140  00000270  00000020
  00101230  00100260  00000000  00101230  00100260  00000000
  00110774
*00111674  LINE 13
  00002007  00111770  00000000  00000150  00000410  00000010
  00101234  00000000  00000140  00000020  00101234  00000000
```

```
  00101230  00100240  00000000
*00111770  LINE 14
  30101027  00112060  00112040  00000160  00000010  00101234
  00100240  00000000  00102574  00000000  00000530  00000010
  00101234  00000000
*00112060  LINE 15
  30102017  00112244  00112140  00000170  00000040  00101234
  00100260  00000000  00101234  00100320  00100260  00000000
  00000010  00000020  00101234  00100260  00000000  00101234
  00100320  00100260  00000000  00000140  00000020  00101234
  00000000  00101234  00100320  00000000  00112060
*00112244  LINE 16
  00001017  00112324  00000000  00000200  00000140  00000020
  00101234  00000000  00101234  00100240  00000000  00111770
*00112324  LINE 17
  00003007  00112450  00000000  00000210  00000320  00000020
  00101230  00000000  00101230  00100240  00000000  00000410
  00000010  00101234  00000000  00000360  00000020  00102410
  00000000  00102630  00000000
*00112450  LINE 18
  00001007  00112530  00000000  00000220  00000400  00000030
  00101230  00000000  00100240  00000000  00102624  00000000
*00112530  LINE 19
  30102027  00112650  00112600  00000230  00000010  00101230
  00100240  00000000  00102574  00000000  00000320  00000020
  00101230  00000000  00102574  00000000  00000530  00000010
  00101234  00000000
*00112650  LINE 20
  00004017  00113044  00000000  00000240  00000250  00000020
```

```
  00101234  00000000  00101230  00100260  00000000  00000240
  00000020  00101234  00000000  00101234  00000000  00000360
  00000020  00102620  00000000  00101234  00000000  00000320
  00000020  00101230  00000000  00101230  00100240  00000000
  00112530  00000040  00000000  00000000  40000000

LSIX PROGRAM ENTERED


  PROGRAM TO SORT NUMBERS

SYSTEM SUBROUTINE STATE ENTERED FROM LINE  4

FIELD DEFINITIONS

NAME  WORD  LBIT  RBIT      MASK

A        1     0    23     77777777
B        2     0    23     77777777
D        3     0    23     77777777

STATE OF FREE SPACE LISTS

BLOCK     POTENTIAL
SIZE      NUMBER
  1         64
  2         32
  4         16
  8          0
 16          0
 32          0
 64          0
128          0

SUBROUTINE PUSH DOWN LIST EMPTY

LEVEL OF FIELD CONTENTS PUSH DOWN STORE IS 0


LEVEL OF FIELD DEFINITION PUSH DOWN STORE IS 0

OUTPUT OF STATE OF SYSTEM COMPLETED
```

```
SYSTEM SUBROUTINE STATE ENTERED FROM LINE  7

FIELD DEFINITIONS

NAME  WORD  LBIT  RBIT      MASK

1       0     0     5      77000000
A       1     0    23      77777777
B       2     0    23      77777777
D       3     0    23      77777777

BUG  W       CONTAINS  20000000  (OCTAL)

AND POINTS TO THE FOLLOWING BLOCK

00000002  00000000  00100000  00000000

BUG  X       CONTAINS  20000020  (OCTAL)

AND POINTS TO THE FOLLOWING BLOCK

00000000


STATE OF FREE SPACE LISTS

BLOCK     POTENTIAL
SIZE      NUMBER
  1         59
  2         29
  4         14
  8          0
 16          0
 32          0
 64          0
128          0

SUBROUTINE PUSH DOWN LIST

LEVEL  1 CALLED FROM LINE 5  OPERATION 5 FROM EOL

LEVEL OF FIELD CONTENTS PUSH DOWN STORE IS 1

LEVEL OF FIELD DEFINITION PUSH DOWN STORE IS 1

OUTPUT OF STATE OF SYSTEM COMPLETED

21 8123 9 94 415 416                          ( Data trace).
```

```
OCTAL PRINT OF LIST

FIRST BLOCK
00000002  20000160  00000000  00000000
NEXT BLOCK
00000002  20000140  00000640  20000200
NEXT BLOCK
00000002  20000120  00000637  20000160
NEXT BLOCK
00000002  20000100  00000136  20000140
NEXT BLOCK
00000002  20000060  00000011  20000120
NEXT BLOCK
00000002  20000040  00017673  20000100
NEXT BLOCK
00000002  20000000  00000025  20000060
NEXT BLOCK
00000002  00000000  00100000  20000040
END OF LIST PRINT

SYSTEM SUBROUTINE STATE ENTERED FROM LINE  5

FIELD DEFINITIONS

NAME  WORD  LBIT  RBIT      MASK

A       1     0     23     77777777
B       2     0     23     77777777
D       3     0     23     77777777

BUG  W       CONTAINS  20000200  (OCTAL)

AND POINTS TO THE FOLLOWING BLOCK

00000002  20000160  00000000  00000000

STATE OF FREE SPACE LISTS

BLOCK     POTENTIAL
SIZE      NUMBER
  1         32
  2         15
  4          7
  8          0
 16          0
 32          0
 64          0
128          0

SUBROUTINE PUSH DOWN LIST EMPTY

LEVEL OF FIELD CONTENTS PUSH DOWN STORE IS 0

LEVEL OF FIELD DEFINITION PUSH DOWN STORE IS 0

OUTPUT OF STATE OF SYSTEM COMPLETED
```

```
OCTAL PRINT OF LIST

FIRST BLOCK
00000002  20000140  00000011  20000200
NEXT BLOCK
00000002  20000120  00000025  20000160
NEXT BLOCK
00000002  20000100  00000136  20000140
NEXT BLOCK
00000002  20000060  00000637  20000120
NEXT BLOCK
00000002  20000040  00000640  20000100
NEXT BLOCK
00000002  20000000  00017673  20000060
NEXT BLOCK
00000002  00000000  00100000  20000040
END OF LIST PRINT

     9    21    94   415   416  8123                (Results)

SYSTEM SUBROUTINE DUMP ENTERED

SYSTEM SUBROUTINE STATE ENTERED FROM LINE  5

FIELD DEFINITIONS

NAME  WORD  LBIT  RBIT      MASK

A        1     0    23     77777777
B        2     0    23     77777777
D        3     0    23     77777777

STATE OF FREE SPACE LISTS

BLOCK     POTENTIAL
SIZE      NUMBER
  1         64
  2         31
  4         15
  8          0
 16          0
 32          0
 64          0
128          0

SUBROUTINE PUSH DOWN LIST EMPTY

LEVEL OF FIELD CONTENTS PUSH DOWN STORE IS 0

LEVEL OF FIELD DEFINITION PUSH DOWN STORE IS 0

OUTPUT OF STATE OF SYSTEM COMPLETED
```

OCTAL OUTPUT OF LINKED STORAGE AREA

```
20000040 00000000 00100000 20000040 20000024 00000000
00000000 00000000 20000060 20000000 00017673 20000060
20000100 20000040 00000640 20000100 20000120 20000060
00000637 20000120 20000140 20000100 00000136 20000140
20000160 20000120 00000025 20000160 20000200 20000140
00000011 20000200 20000220 20000160 00000000 00000000
20000240 00000000 40000000 00000000 20000260 00000000
40000000 00000000 20000300 00000000 40000000 00000000
20000320 00000000 40000000 00000000 20000340 00000000
40000000 00000000 20000360 00000000 40000000 00000000
00000000 00000000 40000000 00000000
```

DUMP COMPLETED

END OF JOB

APPENDIX 2

The Syntax of LSIX defined in BCL

A definition of the syntax of LSIX.

```
:: Main program structure
LSIX IS (LSIXSTATS)
LSIXSTATS IS ((EITHER INSTR
               OR     DIRECTIVE
               OR     O/P(NL.,'LSIX STATEMENT NOT RECOGNISED',NL.),
                      GARBAGE), LSIXSTATS)
GARBAGE IS (EITHER EOL
            OR     CHSET=CHSET1,NEXTCH,GARBAGE)
DIRECTIVE IS (OSP.,EITHER '*ENTER', STOP
                  OR     '*LSIX', OSP., OPTIONS)
OPTIONS IS ((EITHER 'SOURCE'
             OR     'LIST'
             OR     NIL.), (EITHER SEP,OPTIONS
                            OR     EOL) )
SEP IS (OSP.,',',OSP.)
INSTR IS (OSP.,EITHER CONDNL
               OR     UNCONDNL
               OR     LABEL, (EITHER CONDNL
                                 OR     UNCONDNL
                                 OR     EOL) )

:: Types of instruction
CONDNL IS (CONDTION,TESTS, (EITHER 'THEN', OPERATNS OR NIL.),
           TRANSFER,EOL)
UNCONDNL IS ((EITHER 'THEN' OR NIL.), OPERATNS,
             (EITHER TRANSFER OR NIL.), EOL)
```

```
:: Types of condition
CONDITION IS (EITHER 'IFANY',
              OR     'IFNALL',
              OR     'IFALL',
              OR     'IFNONE',
              OR     'IF',
              OR     'NONE',)

:: Analysis of Tests.
TESTS IS (OSP., TEST, EITHER TESTS OR OSP.)
TEST IS ('(', FIELD,SEP,
         (EITHER(EITHER 'K' OR 'N' OR 'G' OR 'L'),
                (EITHER 'O',J:=2 OR 'H',J:=3 OR J:=0)
          OR    (EITHER 'O' OR 'Z'),
                (EITHER 'D',J:=1 OR 'H',J:=3 OR J:=4)
          OR     'P', J:=0),
          OSP.,',',
         (EITHER IF J=0, (EITHER FIELD
                                 OR     DLITERAL)
          OR     IF J=1, DLITERAL
          OR     IF J=2, OLITERAL
          OR     IF J=3, HLITERAL
          OR     IF J=4, (EITHER FIELD
                                 OR     OLITERAL)),')')

:: Analysis of operations.
OPERATNS IS (OSP.,OPERATN, EITHER OPERATNS OR OSP.)
```

```
OPERATN IS ('(', OSP., EITHER 'DO', SEP, (EITHER 'STATE'
                                          OR     'DUMP'
                                          OR     SYMBOL),OSP.,')'
                    OR     FIELD, SEP, FLDNAMES, OSP., ')'
                    OR     ARG, SEP, OPCODE, REJECT
                    OR     IF K LE 27, FIELD, SEP, ARG, SEP,
                          (EITHER IF J=0, FIELD
                           OR     IF J=1, DLITERAL
                           OR     IF J=2, OLITERAL
                           OR     IF J=3, HLITERAL), ')'
                    OR     IF K GE 28, IF K LE 29,
                           FIELD, SEP, ARG, SEP,
                          (EITHER FIELD OR DLITERAL),
                          (EITHER OSP., ',',
                                 (EITHER IF J=1, DLITERAL
                                  OR     IF J=3, HLITERAL
                                  OR     IF J=4,(EITHER FIELD)
                                                OR     OLITERAL))
                           OR     NIL.), ')'
                    OR     IF K GE 30, IF K LE 31,
                          (EITHER FIELD OR DLITERAL), SEP, ARG,
                           OSP., ',' (EITHER IF J=3, HLITERAL
                                      OR     FIELD
                                      OR     OLITERAL), ')'
                    OR     IF K=32, FIELD, SEP, 'PL', SEP,
                           FLDNAME,(EITHER SEP, FIELD
                                    OR     SEP, DLITERAL
                                    OR     OSP.), ')'
```

```
OR      IF K=33, (EITHER 'S' OR 'R'), SEP, 'FC', SEP, FIELD, ')'
OR      IF K=34, (EITHER 'S' OR 'R'), SEP, 'FD', SEP,
                 FLDNAME, OSP., ')'
OR      IF K=35,  SYMBOL,SEP, 'DO', SEP, SYMBOL, OSP., ')'
OR     (EITHER FIELD OR DLITERAL), SEP,
       'D', FLDNAME, SEP, (EITHER FIELD OR DLITERAL),
                    SEP, (EITHER FIELD OR DLITERAL), ')'
OR      IF K=37, '*', OLITERAL, SEP, 'SS', SEP, DLITERAL,
                SEP, '*', OLITERAL, ')'
OR      IF K=38, FIELD, SEP, 'GT', SEP,
                (EITHER FIELD OR DLITERAL),
                (EITHER SEP, FIELD OR NIL.), ')' )

OPCODE IS (EITHER(EITHER 'E', K=1
                  OR     'A', K=2
                  OR     'S', K=3
                  OR     'M', K=4
                  OR     'V', K=5),
                 (EITHER 'O', J=2 OR 'H', J=3 OR J=0)
           OR    (EITHER 'O', K=6
                  OR     'N', K=7
                  OR     'X', K=8
                  OR     'C', K=9),
                 (EITHER 'D', J=1 OR 'H', J=3 OR J=4)
           OR    (EITHER 'DP', K=10
                  OR     'IC', K=11
                  OR     'P',  K=12
                  OR     'LO', K=13
                  OR     'RO', K=14
```

```
                OR      'LZ', K=15
                OR      'RZ', K=16
                OR      'OS', K=17
                OR      'ZS', K=18
                OR      'BZ', K=19
                OR      'ZB', K=20
                OR      'BD', K=21
                OR      'BO', K=22
                OR      'DB', K=23
                OR      'OB', K=24), J=5
        OR      'FR', K=26, J=0
        OR      'IN', K=27, J=0
        OR     (EITHER 'L', K=28 OR 'R', K=29),
               (EITHER 'D', J=1 OR 'H', J=3 OR J=4)
        OR     (EITHER 'PR',K=30 OR 'PU', K=31),
               (EITHER 'H', J=3 OR J =4)
        OR      'PL', K=32
        OR      'FC', K=33
        OR      'FD', K=34
        OR      'DO', K=35
        OR      'D', FLDNAME, K=36
        OR      'SS', K=37
        OR      'GT', K=38 )

REJECT IS (IF 1=0)
```

```
:: Types of field
FIELD IS (OSP.,(EITHER 'T.'
                OR     BUG,(EITHER FLDNAMES OR NIL.)
                OR     POTNUMBER), OSP.)
POTNUMBER IS (EITHER '1.' OR '2.' OR '4.' OR '8.' OR '16.'
              OR     '32.' OR '64.' OR '128.')
BUG IS (LETTER)
FLDNAMES IS (FLDNAME,(EITHER FLDNAMES OR NIL.))
FLDNAME IS  (EITHER LETTER OR DIGIT)
LETTER IS (EITHER 'A' OR 'B' OR 'C' OR .....
                            OR  'X' OR 'Y' OR 'Z')
ODIGIT IS (EITHER '0' OR '1' OR '2' OR '3' OR '4'
           OR     '5' OR '6' OR '7')
DIGIT IS (EITHER ODIGIT OR '8' OR '9')
DLITERAL IS (OSP., DIGIT, EITHER DLITERAL  OR OSP. )
OLITERAL IS (OSP., COUNT = 8, ODIGITS, OSP.)
ODIGITS IS (ODIGIT, COUNT=COUNT-1, (EITHER IF COUNT GT 0, ODIGITS
                                    OR     NIL.)  )
HLITERAL IS (COUNT=4, HCHARS)
HCHARS IS (EITHER ',',COUNT = -1, REJECT
           OR     ')',COUNT = -1, REJECT
           OR     IF COUNT GT 0,
                  CHSET = CHSET1, NEXTCH,COUNT = COUNT - 1,
                 (EITHER IF COUNT GT 0, HCHARS OR NIL.)  )
LABEL IS (LBL, JUNK, SP., OSP.)
JUNK IS (EITHER JNK, JUNK OR NIL.)
TRANSFER IS (OSP., LBL, JUNK)
SYMBOL IS (LBL, JUNK)
EOL IS (OSP., NL., EITHER EOL OR NIL.)
```

APPENDIX 3

BCL routines corresponding to LSIX operations.

Basic routines called from the main interpreter routine

```
DEFINE R OCTPRINT

:: OCTPRINT outputs in octal the contents of the variable OCT.
:: WR1 is a working variable.  The contents of OCT are
:: destroyed.

DO

COUNT := 3

165,WR1,OCT,*70707070

127,OCT,0,*07070707              :: Alternate octal digits in
                                 :: WR1 and OCT.
163,WR1,0,0

163,WR1,0,0

163,WR1,0,0                      :: Right justify contents of WR1.

REPEAT) 125,WR1,0,0

125,OCT,0,0                      :: Circular shift WR1,OCT 6 bits left.

1064,0,WR1,2                     :: Output least significant 6 bits.

1064,0,OCT,2                     :: Output least significant 6 bits.

COUNT := COUNT - 1

IF COUNT GE 0 GO TO REPEAT       :: If not finished, go back.

RETURN

END

DEFINE R STACKPRINT

:: Outputs in octal the contents of all locations from START
:: to FINISH.  The value of START is destroyed.

DO

AGAIN) OCT := COOF(START)        :: Fetch contents of next location.

O/P(SP.(2),OCTPRINT)             :: Output 2 spaces followed
                                 :: by octal number.
START := START + ONE             :: Advance START by one (half)word.

IF START LT FINISH GO TO AGAIN   :: Repeat if not finished.

RETURN

END
```

```
DEFINE R STATEPRINT

:: This routine is used for diagnostic purposes.  It outputs all
:: field definitions, the contents of non zero bugs, the blocks to
:: which bugs point, and the state of the system pushdown stores.
:: A typical output from STATEPRINT is given in Appendix 1.

DO

O/P(NL.(2),'SYSTEM SUBROUTINE STATE ENTERED',

        'FROM LINE ', LINE,NL.) :: Output message with line number.

POINTER := FNAMEBASE            :: Point to name of first field.

CURRENT := FLDBASE              :: Point to first field definition.

O/P(NL.(2),'FIELD DEFINITIONS',NL.,

           'NAME   WORD   LBIT   RBIT    MASK',NL.)

                                :: Column headings.

STATE2)FNAME := COOF(POINTER)   :: Get next field name.

      IF MASK(CURRENT)=O GO TO STATE1

                                :: If field not defined, skip.

      WR1 := WORD(CURRENT)      :: Get word number.

      124,WR1,WR1,O             :: Convert to 21-bit integer.

      WR2 := LBIT(CURRENT)      :: Get left-most bit number.

      WR3 := RBIT(CURRENT)      :: Get right-most bit number.

      OCT := MASK(CURRENT)      :: Get mask.

:: The following command outputs the field name in character form,
:: WR1, WR2 and WR3 as integers and the mask in octal.

O/P(FNAME,SP.(3),WR1,SP.(5),WR2,SP.(5),WR3,SP.(5),OCTPRINT,NL.)

STATE1) POINTER := POINTER+ONE  :: Advance pointer to
                                :: next field name.
       CURRENT := CURRENT + 2   :: Point to next field
                                :: definition.
      IF FNAME NE 'Z' GO TO STATE2
                                :: If not finished, go back.

:: End of output of field definitions.  Continue with bugs.
```

```
        POINTER := BNAMEBASE     :: Point to first bug name.

        CURRENT := BUGBASE       :: Point to first bug i.e.'A'.

STATE4) FNAME := COOF(POINTER)   :: Get bug name.

        IF COOF(CURRENT) = 0 GO TO STATE3
                                 :: If bug is zero, skip print.
        OCT := COOF(CURRENT)     :: Get bug contents in OCT.

        WR2 := OCT

        O/P(NL.,'BUG ',FNAME,SP.(2),'CONTAINS ')

        IF OCT LT STARTLIST GO TO STATE3A        :: If not pointer.

        IF OCT GE ENDLIST GO TO STATE3A          :: If not pointer.

:: The following instructions output the bug contents in octal
:: followed by the contents of the block to which the bug points.

        O/P(OCTPRINT,' (OCTAL) AND POINTS TO THE FOLLOWING BLOCK',NL.)

        OP1:= WR2                :: Transfer bug contents to OP1.

        PRINTBLOCK               :: Routine to print block to
                                 :: which OP1 points.
        GO TO STATE3

:: The following instructions output the bug contents as an integer.

STATE3A) 124,WR2,WR2,0           :: Convert to 21-bit integer.

        124,WR2,WR2,0

        124,WR2,WR2,0

        O/P(WR2,'(DECIMAL)',NL.):: Output integer.

STATE3) CURRENT := CURRENT+ONE   :: Advance pointers.

        POINTER := POINTER+ONE

        IF FNAME NE 'Z' GO TO STATE4

                                 :: If bugs not finished.

:: The following instructions output the state of the free space
:: list giving block size and potential number of blocks.

        SIZE := 1

        CURRENT := FREEHDR               :: Point to 1 - blocks.
```

```
        O/P(NL.,'STATE OF FREE SPACE LISTS',
                NL.,'BLOCK  POTENTIAL',
                NL.,'SIZE   NUMBER',NL.)
STATE5) WR1 := PLUS3(CURRENT)            :: Get potential number.
        O/P(NL.,SP.(2),SIZE,SP.(6),WR1) :: Output size and number.
        CURRENT := CURRENT + 2           :: Point to next sublist.
        SIZE := SIZE + SIZE              :: Double SIZE.
        IF SIZE LE MAXSIZE GO TO STATE5 :: If not finished.
:: The following instructions output the state of the system
:: pushdown stores.
        O/P(NL.(2), 'SUBROUTINE PUSH DOWN LIST')
        WR1 := SUBL                      :: Get subroutine level.
        IF SUBL GT 0 GO TO STATE6        :: If stack not empty.
        O/P('EMPTY')
        GO TO STATE7
STATE6) POINTER := SUBP - 3              :: Point to top record
                                         :: on stack.
STATE8) WR2 := PLUS5(POINTER)            :: Get line number of call.
        WR3 := PLUS3(POINTER)            :: Get position in line of call.
        O/P(NL.,'LEVEL ',WR1,'CALLED FROM LINE',
                WR2,' OPERATION ',WR3,' FROM EOL')
        POINTER := POINTER - 3           :: Point to next record.
        WR1 := WR1 - 1                   :: Decrease working level
                                         :: number.
        IF WR1 GT 0 GO TO STATE8         :: If not finished, go back.
STATE7) O/P(NL.(2),'LEVEL OF FIELD CONTENTS STACK IS',FCL)
                                         :: Output level of field
                                         :: contents stack.
        O/P(NL.(2),'LEVEL OF FIELD DEFINITION STACK IS',FDL)
                                         :: Output level of field
                                         :: definition stack.
        O/P(NL.(2),'OUTPUT OF STATE OF SYSTEM COMPLETED',NL.(2))
        RETURN
        END
```

```
        DEFINE R PRINTBLOCK

:: This routine gives an octal output of the block to which OP1 points.

        DO

        OCT := COOF(OP1)                    :: Get first word of block,
                                            :: (contains log(size)).
        127,OCT,0,0.7                       :: Get log(size).

        SIZE := 1

PL2)    IF OCT=0 GO TO PL3                  :: Go to print when size found.

        122,OCT,0,0.1                       :: Subtract 0.1 from log(size).

        SIZE := SIZE + SIZE                 :: Double size.

        GO TO PL2

PL3)    OCT := COOF(OP1)                    :: Get next word.

        O/P(SP.(2),OCTPRINT)                :: Octal output.

        SIZE := SIZE - 1

        IF SIZE = 0 GO TO END               :: If end of block.

        OP1 := OP1 + ONE                    :: Point to next word.

        GO TO PL3

END)    RETURN                              :: End of PRINTBLOCK.

        END



DEFINE R STOREIN1

:: This routine stores the contents of the variable OCT in the
:: field specified by BP1 and FP1.  Routines STOREIN2 and STOREIN3
:: are defined similarly for fields specified by BP2,FP2 and
:: BP3,FP3 respectively.

DO

WREG1 := BP1

WREG2 := FP1

STOREFIELD              :: Call routine to store field.

RETURN

END
```

BCL instructions corresponding to the LSIX operations

The following sets of instructions are entered via the multi-way switch labelled OPSPLIT1 in the interpreter routine. Each set of instructions is terminated with a GO TO OPRTN. The operations for setting up, allocating, freeing and copying data blocks are not described in this section but various methods of storage organisation are discussed in detail in section 2.3.

Definition of fields

The field definition operation is

(cd1,Df,cd2,cd3).

cd1 specifies the word number of the field, cd2 and cd3 the left most and rightmost bits within this word. D is the operation code representedby K = 36 and f is the name of the field to be defined. This is the only operation involving 4 operands - three integers and a field name. At the time of entry to the equivalent BCL instructions, only three operands have been fetched, cd1 is in OP1, the address of the field definition f is in BP2 and cd2 is in OP3. The first instructions FIND and GET cd3.

```
D)      FINDFIELD               :: Locate fourth operand.
        GETFIELD                :: Get fourth operand ( in WREG1).
        124,OP1,OP1,0           :: Convert cd1 to 22-bit integer,
        124,OP1,OP1,0           :: 1 LSIX word = half Atlas word.
        124,OP3,OP3,0           :: Convert cd2 to 21-bit integer.
        124,OP3,OP3,0
        124,OP3,OP3,0
        124,WREG1,WREG1,0       :: Convert cd3 to 21-bit integer.
        124,WREG1,WREG1,0
        124,WREG1,WREG1,0
```

```
        IF OP1 GT 64 GO TO NULL :: Max 128(half)words in any block.

        IF OP1 LT 0  GO TO NULL :: Validate definition.

        IF OP3 LT 0 GO TO NULL

        IF WREG1 GT 23 GO TO NULL

        IF OP3 GT WREG1 GO TO NULL

        WORD(BP2) := OP1          :: Store word number

        LBIT(BP2) := OP3          :: Store left bit number.

        RBIT(BP2) := WREG1        :: Store right bit number.

:: The following instructions create a mask for the field.

        121,WR1,0,*4              :: Load working register with a
                                  :: left-justified 1-bit.
        COUNT := WREG1-OP3        :: Initialise count.

DMORE)  IF COUNT = 0 GO TO DEND   :: If sufficient 1-bits.

        163,WR1,0,0               :: Right shift 1 bit.

        167,WR1,0,*4              :: Add 1 bit at left hand end.

        COUNT := COUNT-1          :: Count.

        GO TO DMORE

DEND)   1342,WR1,OP3,0            :: Right shift mask into position.

        IF OP1 NE 0 GO TO D1      :: If not word number 0.

        127,WR1,0,*77777770       :: Protect 3 least significant bits
                                  :: reserved for compiler.
D1)     MASK(BP2) := WR1          :: Store mask.

        GO TO OPRTN

NULL)   WORD(BP2) := 0            :: Invalid defn. nullified.

        LBIT(BP2) := 0

        RBIT(BP2) := 0

        MASK(BP2) := 0

        O/P(NL.,'*** INCORRECT FIELD DEFINITION IN LINE',LINE,NL.)

        GO TO OPRTN
```

```
:: The following instructions copy and interchange fields.

EQ)     OCT := OP2              :: Get second operand.

        STOREIN1                :: Store in field specified by first.

        GO TO OPRTN

:: The copy pointer operation which follows is implemented as copy
:: field contents.

OPP)    OCT:= OP2               :: Get pointer from second operand.

        STOREIN1                :: Store in first.

        GO TO OPRTN

IC)     OCT := OP2              :: Interchange field contents.

        STOREIN1

        OCT := OP1

        STOREIN2

        GO TO OPRTN

:: The Arithmetic operations are performed on the specified
:: operands and the result overwrites the first operand.

ADD)    OCT := OP1 + OP2

        STOREIN1

        GO TO OPRTN

SUB)    OCT := OP1 - OP2

        STOREIN1

        GO TO OPRTN

MPY)    1312,OP1,OP2,0          :: Multiply 24-bit integers.
                                :: result in OP1
        OCT := OP1

        STOREIN1

        GO TO OPRTN

DIV)    1314,OP1,OP2,0          :: Divide OP1 by OP2, lose remainder.

        OCT := OP1

        STOREIN1

        GO TO OPRTN
```

```
:: Logical operations OR, AND and nonequivalence operate on two
:: operands the first of which is overwritten by the result.

OR)     OCT := OP1              :: Get first operand.
        167,OCT,OP2,0           :: Logical OR with second.
        STOREIN1                :: Store result.
        GO TO OPRTN

AND)    OCT := OP1              :: Get first operand.
        127,OCT,OP2,0           :: Logical AND with second.
        STOREIN1                :: Store result.
        GO TO OPRTN

XOR)    OCT := OP1              :: Get first operand.
        126,OCT,OP2,0           :: Non equivalence with second.
        STOREIN1                :: Store result.
        GO TO OPRTN

:: The operation Complement stores the complement of the second
:: operand in the field specified by the first.

C)      OCT := OP2              :: Get second operand.
        126,OCT,0,*77777777     :: Complement.
        STOREIN1                :: Store result.
        GO TO OPRTN

:: In addition to the logical operations described above, LSIX
:: provides operations for counting 1-bits and 0-bits and
:: operations for locating the leftmost and rightmost 1-bit
:: and 0-bit of any operand.  These six operations have many
:: things in common.  They are implemented using one sequence
:: of BCL instructions with 6 different entry points and
:: switches are used where the operations diverge. Operations
:: apply to OP2 and results overwrite the first operand.
```

```
OS)       FLAG := 1                  :: Entry point for count 1-bits.

BITS)     LENGTH := 24               :: Max field width.

          IF FP = 0 GO TO BITS2      :: If operand is not remote field.

          LENGTH := RBIT(FP2) - LBIT(FP2)

          LENGTH := LENGTH+1         :: Compute field width for remote field.

BITS2)    OCT :=0                    :: Initialise bit count.

MOREBITS) LENGTH := LENGTH - 1       :: Count down bits in field.

          IF LENGTH LT 0 GO TO ENDBITS
                                     :: If whole field scanned.
          GO TO OS1,RO1,LO1,LO1,VIA FLAG
                                     :: Switch via flag.
OS1)      163,OP2,0,0                :: Circular right shift OP2  1 bit.

          IF OP2 GE 0 GO TO MOREBITS    :: If leading bit is zero.

          OCT := OCT + 1             :: Count 1 bit.

          GO TO MOREBITS

NOBITS)   OCT := 0

ENDBITS)  STOREIN1                   :: Store result.

          GO TO OPRTN

ZS)       126,OP2,0,*77777777        :: Entry for 0-bits.

          GO TO OS                   :: Complement and count 1-bits.

RO)       FLAG := 2                  :: Entry for locate 1-bit right,
                                     :: set flag.
          GO TO BITS

RO1)      OCT := OCT + 1             :: Count bit.

          163,OP2,0,0                :: Circular right shift OP2 1 bit.

          IF OP2 LT 0 GO TO ENDBITS  :: If leading bit is now 1.

          IF LENGTH = 0 GO TO NOBITS :: If whole field scanned
                                     :: without success.
          GO TO MOREBITS

RZ)       FLAG := 2                  :: Locate 0-bit from right.

          GO TO BITS1
```

```
LZ)     FLAG := 3                   :: Locate 0-bit from left.

LZ1)    IF FP2 = 0 GO TO LZ2        :: If not remote field.

        SHIFT := 23 - RBIT(FP2)

        SHIFT := SHIFT + LBIT(FP2)

        1343,OP2,SHIFT,0            :: Left justify operand in
                                    :: 24 bit field.
LZ2)    GO TO BITS,BITS,BITS1,BITS VIA FLAG     :: Switch.

LO)     FLAG := 4                   :: Locate 1-bit from left.

        GO TO LZ1

LO1)    OCT := OCT + 1              :: Count bits.

        IF OP2 LT 0 GO TO ENDBITS   :: If leading bit is a 1.

        1343,OP2,0,1                :: Left shift 1 bit (circular).

        IF LENGTH = 0 GO TO NOBITS  :: If whole field scanned.

        GO TO MOREBITS
```

```
:: In their most general form the LSIX shift operations have three
:: operands.  The first is the field to be shifted, the second
:: specifies the number of places to shift and the third operand
:: (zero if not specified) is the field to be shifted into the
:: first field specified.  Two subroutines are defined for shift
:: operations.

        DEFINE R LSHIFT

        DO

        WR1 := OP1              :: Fetch fields to be used in shift.

        WR2 := OP3

        IF OP2 := 0 GO TO END   :: If no shift required.

        IF FP3=0 GO TO READY    :: If OP3 already left justified.

        SHIFT := 23 - RBIT(FP3)

        SHIFT := SHIFT + LBIT(FP3)

        1343,WR2,SHIFT,0        :: Left justify OP3 in WR2.

READY)  SHIFT := OP2            :: Number of shift places.

        124,SHIFT,SHIFT,0       :: Convert to 21 bit integer.

        124,SHIFT,SHIFT,0

        124,SHIFT,SHIFT,0

MORE)   1343,WR1,0,1            :: Circular left shift WR1 1 place.

        1343,WR2,0,1            :: Circular left shift WR2 1 place.

        127,WR1,0,*77777776     :: Clear bit 23 of WR1.

        165,WR3,WR2,0.1         :: Get bit 23 of WR2 in WR3.

        167,WR1,WR3,0           :: Add it to WR1.

        127,WR2,0,*77777776     :: Clear bit 23 of WR2.

        SHIFT := SHIFT - 1      :: Count down.

        IF SHIFT GT 0 GO TO MORE:: If not finished.

        OCT := WR1              :: Result in OCT.

        STOREIN1                :: Store result.

END)    RETURN

        END
```

```
        DEFINE R RSHIFT
        DO
        WR1 := OP1              :: Get operands.
        WR2 := OP3
        IF OP2 = 0 GO TO END    :: If number of shifts is zero.
        IF FP1=0 GO TO READY    :: If OP1 already left justified.
        SHIFT := 23 - RBIT(FP1)
        SHIFT := SHIFT + LBIT(FP1)
        1343,WR1,SHIFT,0        :: Left justify OP1 in WR1.
READY)  SHIFT := OP2            :: Number of shift places.
        124,SHIFT,SHIFT,0       :: Convert to 21-bit integer.
        124,SHIFT,SHIFT,0
        124,SHIFT,SHIFT,0
MORE)   163,WR1,0,0             :: Circular shift right 1 place.
        163,WR2,0,0
        127,WR1,0,*37777777     :: Clear bit 0 of WR1.
        165,WR3,WR2,*4          :: Get bit 0 of WR2 in WR3.
        167,WR1,WR3,0           :: Add this bit to WR1.
        127,WR2,0,*37777777     :: Clear bit 0 of WR2.
        SHIFT := SHIFT -1       :: Count down.
        IF SHIFT GT GO TO MORE  :: If not finished.
        IF FP1 = 0 GO TO FLUSHRT:: If first operand is 24 bit field.
        SHIFT := 23 - RBIT(FP1)
        SHIFT := SHIFT + LBIT(FP1)
        1342,WR1,SHIFT,0        :: Right justify result.
FLUSHRT) OCT := WR1             :: Get result in OCT.
        STOREIN1
END)    RETURN
        END
```

```
:: The shift routines are called for the LSIX shift operations.

LL)     LSHIFT

        GO TO OPRTN

R)      RSHIFT

        GO TO OPRTN

:: LSIX provides only one operation for input.  The number of
:: characters to be input is specified by OP2.  Input characters
:: are left shifted into the field specified by the first operand.
:: Input is terminated on reading an end of line character.

IN)     OCT := OP1                  :: Current OP1 in OCT.

INMORE) IF OP2=0 GO TO INEND        :: If finished.

        125,OCT,0,0                 :: Left shift OCT 6 bits to
                                    :: receive next character.
        127,OCT,0,*777777           :: Clear least significant 6 bits.

        1054,WR1,0,(INEOL)          :: Input character to least
                                    :: significant end of WR1,
                                    :: if newline character go to INEOL.
        167,OCT,WR1,0               :: Add character to OCT.

        122,OP2,0,0.1               :: Count down (24 bit integer).

        GO TO INMORE

INEOL)  167,OCT,0,7.7               :: Newline char. is represented by (77).

INEND)  STOREIN1                    :: Store result of input.

        GO TO OPRTN

:: The contents of fields and octal literals may be output in
:: character form by means of the PRint and PUnch operations.
:: Hollerith character strings are dealt with separately in
:: this implementation.

PU)     IF OPFLAG=1 GO TO OUTPUT:: If stream 1 already selected.

        OPFLAG := 1

        1060,0,0,1                  :: Select output stream 1.

        GO TO OUTPUT

PR)     IF OPFLAG=0 GO TO OUTPUT:: If stream 0 already selected.

        OPFLAG := 0

        1060,0,0,0                  :: Select output stream 0.
```

```
OUTPUT) IF OP1=0 GO TO OPRTN     :: Return if no chars. for output.
        124,OP1,OP1,0            :: Convert number of characters
        124,OP1,OP1,0            :: to 21-bit integer.
        124,OP1,OP1,0
        WR1 := OP2               :: Get field contents for output.
        COUNT := OP1             :: Get character count.
        IF OP3 GT 0 GO TO HOUTPUT   :: If output Hliteral.
PR1)    IF COUNT LE 4 GO TO PR2  :: If not more than 4 characters.
        COUNT := COUNT-1         :: Count down.
        O/P(SP.)                 :: Output leading space.
        GO TO PR1
PR2)    IF FP2=0 GO TO PR9       :: If 24-bit operand.
        165,WR2,WR1,*76          :: Convert any leading
        IF WR2 NE 0 GO TO PR9    :: binary zeros to blanks.
        167,WR1,0,*01
        165,WR2,WR1,*0076
        IF WR2 NE 0 GO TO PR9
        167,WR1,0,*0001
        165,WR2,WR1,*000076
        IF WR2 NE 0 GO TO PR9
        167,WR1,0,*000001
PR9)    GO TO PR6,PR7,PR8,PR5 VIA COUNT   :: Switch.
PR7)    125,WR1,0,0              :: Shift ready for output.
PR8)    125,WR1,0,0
PR5)    125,WR1,0,0
PR6)    165,WR2,WR1,7.7          :: Get first character.
        126,WR2,0,7.7            :: Complement this character.
        IF WR2=0 GO TO PR10      :: If character is newline.
        1064,0,WR1,0             :: Output 1 character.
```

```
PR11)   COUNT := COUNT-1          :: Count down.
        IF COUNT GT 0 GO TO PR5   :: If not finished.
        GO TO OPRTN
PR10)   1065,0,0,2.1              :: Output newline.
        GO TO PR11
:: Strings of characters of any length may be output. During
:: input of the program such character strings are stored
:: in the constants area.  They are output by the following
:: instructions.  The length of the string is assigned to OP3.
:: BP2 points to the character string.

HOUTPUT) IF OP1 LE OP3 GO TO H1   :: If no leading spaces.
        1064,0,0,0.1              :: Output space.
        OP1 := OP1-1              :: Count down.
        GO TO HOUTPUT
H1)     IF OP1 = OP3 GO TO H2
        OP3 := OP3-1              :: Decrease length of string.
        BP2 := BP2+ONE            :: Skip next character.
        GO TO H1
H2)     WR1 := COOF(BP2)          :: Get next character.
        1064,0,WR1,0              :: Output it.
        OP3 := OP3-1              :: Decrease length of string.
        BP2 := BP2+ONE            :: Point to next character.
        IF OP3 GT 0 GO TO H2      :: If not finished.
        GO TO OPRTN
```

```
:: Another output operation which is intended for diagnostic use
:: is the Print List operation.  OP1 points to first block, BP2
:: specifies the link field and OP3 is the number of blocks ( if
:: specified ).

PL)     O/P(NL.,'OCTAL PRINT OF LIST',NL.,'FIRSTBLOCK',NL.)
        IF OP3 GT 0 GO TO PL1   :: If number of blocks is specified.
        OP3 := OP3 - 1          :: Set negative OP3.
PL1)    WREG1 := OP1            :: Get pointer.
        WREG2 := BP2
        PRINTBLOCK              :: Print next block.
PL4)    122,OP3,0,0.1           :: Count down.
        IF OP3=0 GO TO PLEND    :: If finished.
        GETFIELD                :: Get link to next block.
        IF WREG1=0 GO TO PLEND  :: If end of list.
        OP1 := WREG1            :: Pointer to block.
        O/P(NL.,'NEXT BLOCK',NL.)
        GO TO PL1
PLEND)  O/P(NL.,'END OF LIST PRINT',NL.)
        GO TO OPRTN
```

```
:: Associated with the LSIX input and output operations are a
:: number of conversion operations:  Convert leading spaces to
:: zeros, leading zeros to spaces, Hollerith coded decimal and
:: octal to binary and vice versa. These six conversion operations
:: are described below.

BD)     OCT := 0                    :: Entry for Binary to Decimal and
                                    :: Binary to Octal.
        K := K-20                   :: K is now 1 for BD and 2 for BO.

        124,OP2,OP2,0               :: Left shift operand three places.

        124,OP2,OP2,0

        124,OP2,OP2,0

        COUNT := 4                  :: Initialise count.

BD1)    GO TO BD2, BO VIA K         :: Switch.

BD2)    1304,OP2,0,10               :: Divide by 10, remainder in B97.

BD3)    167,OCT,97,16               :: Add character form of
                                    :: remainder to OCT.
        125,OCT,0,0                 :: Left shift OCT 6 binary places.

        COUNT := COUNT-1            :: Count down.

        IF COUNT GT 0 GO TO BD1     :: If not finished go back.

        163,OCT,0,0                 :: Right shift 3 binary places.

        163,OCT,0,0

        163,OCT,0,0

BDEND)  STOREIN1                    :: Store result.

        IF OP2 =0 GO TO OPRTN

BDERR)  O/P(NL.,'OVERFLOW IN NUMERIC CONVERSION',NL.)

        GO TO OPRTN

BO)     165,97,OP2,7                :: Fetch next octal digit to B97.

        127,OP2,0,*777777           :: Clear least significant
                                    :: digit from OP2.
        163,OP2,0,0                 :: Divide OP2 by 8.

        163,OP2,0,0

        163,OP2,0,0

        GO TO BD3
```

```
DB)     OCT := 0                    :: Entry for conversions to binary.

        K := K-20                   :: K now 3 or 4.

        COUNT := 3                  :: Initialise count.

DB1)    121,WS1,0,1.1               :: WS1 assigned '9' as 24 bit integer.

        125,OP2,0,0                 :: Left shift operand 6 places.

        165,WR1,OP2,7.7             :: Get next character in WR1.

        IF WR1=0 GO TO DB5

        WR1:=WR1-2                  :: Convert WR1 to integer form.

        IF WR1 LT 0 GO TO DBERR     :: If negative, then error.

DB5)    IF K=4 GO TO OB             :: If octal to binary.

        IF WR1 GT WS1 GO TO DBERR
                                    :: If integer greater than '9'.
        1312,OCT,0,1.2              :: Multiply by 10 (24 bit).

DB2)    OCT := OCT+WR1              :: Add to OCT.

        203,127,COUNT,(DB1)         :: If not finished, count
                                    :: down and go back.
        STOREIN1                    :: If finished, store result.

        GO TO OPRTN

DBERR)  O/P(NL.,'NONNUMERIC CHARACTER IN NUMERIC CONVERSION')

        GO TO OPRTN

OB)     IF WR1 GE 1 GO TO DBERR     :: If digit not octal.

        124,OCT,OCT,0               :: Multiply OCT by 8.

        124,OCT,OCT,0

        124,OCT,OCT,0

        GO TO DB2
```

```
:: The following instructions convert leading spaces to zeros.
BZ)     COUNT := 5              :: Initialise count.
        OCT := OP2              :: Fetch operand.
BZ1)    165,WREG1,OCT,*76       :: Check that character is
                                :: not space or zero.
        IF WREG1 NE 0 GO TO BZEND
        127,OCT,0,*00777777     :: Space to zero.
        167,OCT,0,*20
        125,OCT,0,0             :: Left shift 6 places.
        COUNT := COUNT - 1      :: Count down.
        IF COUNT NE 1 GO TO BZ1 :: If not finished.
BZEND)  GO TO BZ5,BZ2,BZ3,BZ4,BZ5, VIA COUNT
BZ4)    125,OCT,0,0             :: Circular shift 6 places left.
BZ3)    125,OCT,0,0             :: Circular shift 6 places left.
BZ2)    125,OCT,0,0             :: Circular shift 6 places left.
BZ5)    STOREIN1
        GO TO OPRTN             :: End of spaces to zeros.
:: Convert leading zeros to spaces
ZB)     COUNT := 5              :: Initialise count.
        OCT := OP2              :: Fetch operand.
ZB1)    165,WREG1,OCT,*57       :: Check next character.
        IF WREG1 NE 0 GO TO BZEND   :: If not zero or space.
        127,OCT,0,*00777777     :: Clear first character.
        167,OCT,0,*01           :: Insert space character.
        125,OCT,0,0             :: Shift (circular) 6 places left.
        COUNT := COUNT-1        :: Count down.
        IF COUNT NE 2 GO TO ZB1 :: If not finished.
ZBEND)  GO TO BZEND             :: Go to shift and store result.
```

```
:: LSIX provides two system subroutines STATE and DUMP for diagnostic
:: purposes.  These and the user defined subroutines are entered via
:: a DO operation.  The DO operation for user defined routines is
:: described first.  SUBP is the subroutine stack pointer.

DO)     COOF(SUBP) := OBJECTP    :: Stack current object pointer.

        PLUS1(SUBP) := NDESCRWD  :: Stack address of next
                                 :: description word.
        PLUS2(SUBP) := DESCRWD   :: Stack current description
                                 :: word.
        PLUS3(SUBP) := NO        :: Stack number of operations
                                 :: remaining.
        PLUS4(SUBP) := 0         :: Stack a null fail return
                                 :: address (temporary).
        PLUS5(SUBP) := LINE      :: Stack line number.

        NDESCRWD := BP1          :: Pick up transfer address.

        IF BP2=0 GO TO DO1       :: If no fail return address.

        PLUS4(SUBP) := BP1       :: Stack fail return address.

        NDESCRWD := BP2          :: Correct transfer address.

DO1)    SUBP := SUBP+3           :: Push-down stack.

        SUBL := SUBL+1           :: Increment subroutine level.

        GO TO GAMMA              :: Transfer to subroutine.

:: The system subroutine STATEPRINT is called by (DO,STATE).

STATE)  STATEPRINT

        GO TO OPRTN

:: (DO,DUMP) causes an octal dump of the list area preceded by
:: the state of the system.

DUMP)   O/P(NL.(2),'SYSTEM SUBROUTINE DUMP ENTERED')

        STATEPRINT               :: Output state of system.

        START := STARTLIST

        FINISH := ENDLIST

        STACKPRINT               :: Octal output of list area.

        O/P(NL.(2),'END OF OCTAL OUTPUT OF LIST AREA',NL.(2))

        GO TO OPRTN
```

```
:: The system has two pushdown stores for saving Field Contents
:: and Field Definitions.  These are manipulated by the Save and
:: Restore Field Contents and Field Definition operations.

SFC)    COOF(FCP) ::= OP1          :: Save field contents, OP1.

        FCL := FCL+1               :: Increase field contents level
                                   :: number.
        FCP := FCP+ONE             :: Pushdown

        GO TO OPRTN                :: Return.

RFC)    FCP := FCP-ONE             :: Pop-up field contents stack.

        FCL := FCL-1               :: Decrease level number.

        OCT := COOF (FCP)          :: Retrieve field contents.

        STOREIN1                   :: Store result.

        GO TO OPRTN

SFD)    COOF(FDP) := WORD(BP1)     :: Save field defn.

        PLUS1(FDP) := LBIT(BP1)

        PLUS2(FDP) := RBIT(BP1)

        PLUS3(FDP) := MASK(BP1)

        FDL := FDL+1               :: Increase level number.

        FDP := FDP+2               :: Pushdown.

        GO TO OPRTN

RFD)    FDP := FDP+2               :: Pop-up field defn. stack.

        FDL := FDL-1               :: Decrease level number.

        WORD(BP1) := COOF(FDP)     :: Restore field defn.

        LBIT(BP1) := PLUS1(FDP)

        RBIT(BP1) := PLUS2(FDP)

        MASK(BP1) := PLUS3(FDP)

        GO TO OPRTN                :: Return.

:: The only other operations implemented are the storage allocation
:: and freeing operations which are described in detail in §2.3.
```

APPENDIX 4

Routines for Automatic Garbage Collection in LSIX

Routines for Automatic Garbage Collection in LSIX

The following routines, which are an extension of the garbage collector due to Schorr and Waite, have been tested on the Atlas computer as a part of an M.Sc. student's project.

```
        DEFINE R COLLGARB
        DO
        P4 := BUGBASE                   :: Pointer to first bug.
        BUGMAX := P4+13                 :: Pointer to last bug.
SCANBUG) SCANLIST                       :: Scan list to which bug
                                        :: points.
        124,P4,0,0.4                    :: Address of next bug.
        IF P4 LE BUGMAX GO TO SCANBUG
        IF FCL=0 GO TO LINK
        P4 := FCP
        WR1 := FCL
        163,WR1,0,0
        P4 := P4-WR1                    :: Point to bottom of stack.
SCANSTACK) SCANLIST                     :: Scan list.
        124,P4,0,0.4
        IF P4 LT FCP GO TO SCANSTACK    :: If not top of stack.
LINK)   LINKFREE                        :: Link unmarked blocks onto
                                        :: free space lists.
        RECOMBINE                       :: Recombine free mates if
                                        :: necessary.
        RETURN
        END
```

```
        DEFINE R SCANLIST
        DO
        P1 := 0
        P2 := COOF(P4)
        IF P2 LT STARTLIST GO TO ENDSCAN :: If not list pointer.
        IF P2 GE ENDLIST GO TO ENDSCAN   :: If not list pointer.
        165,WS1,P2,*77777770
        IF WS1 NE P2 GO TO ENDSCAN       :: If not pointer to word 0.
        GETMAP
        IF WS2 = 0 GO TO ENDSCAN         :: If not pointer to word 0.
        165,FLAG,WR1,4                   :: Get 'used' flag
        IF FLAG NE 0 GO TO ENDSCAN       :: If block already scanned.
        165,PFLAG,WR1,2                  :: Get pointer flag for
                                         :: word 0 .
SETFLAG) WS2 := WS2+WR1                  :: Reconstruct map
        WS2 := WS2+4                     :: Set used flag.
        STMAP                            :: Restore map.
NEXTWORD) IF PFLAG = 0 GO TO NOBRANCH    :: If not branch.
        WS1 := COOF(P2)                  :: Address of next block
        IF WS1 = 0 GO TO NOBRANCH        :: If null link.
        127,WS1,0,*77777770
        IF WS1 NE COOF(P2) GO TO NOBRANCH:: If not word 0.
        GETMAP                           :: Map for next block
        IF WS2 = 0 GO TO NOBRANCH        :: If not word 0.
        165,FLAG,WR1,4
        IF FLAG NE 0 GO TO NOBRANCH      :: If already scanned.
        165,PFLAG,WR1,2                  :: Get pointer flag for word 0
        COOF(P2) := P1                   :: Plant reverse pointer.
```

```
        P1 := P2                          :: Advance pointer.
        P2 := WS1                         :: Address of next block.
        GO TO SETFLAG
NOBRANCH) 124,P2,0,0.4                    :: Advance pointer P2 by
                                          :: one word.
        165,WS1,P2,*77777770              :: P2 odd or even?
        IF WS1 NE P2 GO TO ODD            :: If odd.
        GETMAP
        IF WS2 NE 0 GO TO ENDBLOCK        :: If end of current block.
        165,PFLAG,WR1,2                   :: Get pointer flag.
        GO TO SETFLAG
ODD)    GETMAP
        165,PFLAG,WR1,1                   :: Get pointer flag.
        GO TO NEXTWORD
ENDBLOCK) IF P1 = 0 GO TO ENDSCAN         :: If scan completed.
        121,SIZE,0,0.4                    :: Determine size.
DOUBLE) 124,SIZE,SIZE,0
        WS1 := P2- SIZE
        GETMAP
        IF WS2 = 0 GO TO DOUBLE
        P3 := P2-SIZE                     :: Point to start of block.
        P2 := P1                          :: Step back P2.
        P1 := COOF(P1)
        COOF(P2) := P3                    :: Restore forward link.
        GO TO NOBRANCH
ENDSCAN) RETURN
        END
```

```
        DEFINE R LINKFREE

:: This routine scans the whole of the list storage area from
:: STARTLIST to ENDLIST and collects up any unmarked, i.e. free,
:: space as 2-blocks, 4-blocks and 8-blocks.
        DO
        WR1 := STARTMAP                 :: Initialise map pointer.
        WR2 := STARTLIST                :: Initialise list pointer
        WS1 := FREEHDR+2                :: Pointer to 2-block list.
        WS2 := WS1+2                    :: Pointer to 4-block list.
        WS3 := WS2+2                    :: Pointer to 8-block list.
        COUNT2 := 0                     :: 2-block count.
        COUNT4 := 0                     :: 4-block count
        COUNT8 := 0                     :: 8-block count.
NEXT)   CONST := COOF(WR1)              :: Get next map word.
        165,WREG1,CONST,*40404040       :: Get flags.
        IF WREG1 NE 0 GO TO NOT8        :: If 8-block not free.
        COOF(WS3) := WR2                :: Link on free 8-block.
        WS3 := WR2
        COUNT8 := COUNT8+1              :: Count 8-block.
        121,CONST,0,*03                 :: Map for 8-block.
        GO TO PLANT
NOT8)   126,WREG1,0,*40404040           :: Complement flags.
        IF WREG1 = 0 GO TO PLANT        :: If no blocks free
        COUNT := 1
        165,WREG2,WREG1,*4              :: Check first flag.
        IF WREG2 = 0 GO TO FLAG2        :: If not free.
        COUNT := COUNT+1
FLAG2)  165,WREG2,WREG1,*004            :: Check second flag.
        IF WREG2 = 0 GO TO FIRST4       :: If not free.
```

```
        COUNT := COUNT +2
FIRST4) GO TO FLAG3,FIRST1,FIRST2,FIRST3, VIA COUNT
FIRST1) COOF(WS1) := WR2                    :: Link-on first 2-block.
        WS1 := WR2
        COUNT2 := COUNT2+1
        127,CONST,0,*07777777               :: Clear pointer flags for
                                            :: 2-block.
        GO TO FLAG3
FIRST2) COOF(WS1) := WR2+1                  :: Link-on second 2-block.
        WS1 := WR2+1
        COUNT2 := COUNT2+1                  :: Count 2-block.
        127,CONST,0,*77077777               :: Clear pointer flags.
        GO TO FLAG3
FIRST3) COOF(WS2) := WR2                    :: Link-on first 4-block.
        WS2 := WR2
        COUNT4 := COUNT4+1                  :: Count 4-block.
        127,CONST,0,*00007777               :: Clear pointer flags.
        167,CONST,0,*02007777               :: Set size for 4-block.
FLAG3)  COUNT := 1
        165,WREG2,WREG1,*00004              :: Check third flag
        IF WREG2 = 0 GO TO FLAG4            :: If not free.
        COUNT := COUNT+1
FLAG4) 165,WREG2,WREG1,4.0                  :: Check fourth flag.
        IF WREG2 = 0 GO TO LAST4            :: If not free.
        COUNT := COUNT+2
LAST4)  GO TO PLANT,LAST1,LAST2,LAST3, VIA COUNT  :: Switch.
LAST1)  COOF(WS1) := WR2+2                  :: Link-on 2-block.
        WS2 := WR2+2
        COUNT2 := COUNT2+1                  :: Count 2-block.
        127,CONST,0,*77770777               :: Clear pointer flags.
        GO TO PLANT
```

```
LAST2)  COOF(WS1) := WR2+3              :: Link on 2-block.
        WS1 := WR2+3
        COUNT2 := COUNT2+1              :: Count 2-block.
        127,CONST,0,*77777707           :: Clear pointer flags.
        GO TO PLANT
LAST3)  COOF(WS2) := WR2+2              :: Link on 4-block.
        WS2 := WR2+2
        COUNT4 := COUNT4+1              :: Count 4-block.
        127,CONST,0,*77770000
        167,CONST,0,*00000200           :: Form new map.
PLANT)  127,CONST,0,*37373737           :: Clear flags.
        COOF(WR1) := CONST              :: Plant map
        WR2 := WR2+4                    :: Increment list pointer.
        124,WR1,0,0.4                   :: Increment map pointer.
        IF WR1 LT ENDMAP GO TO NEXT     :: Go back if not finished.
        COOF (WS1) := 0                 :: Terminate lists.
        COOF(WS2) := 0
        COOF(WS3) := 0
        WS1 := FREEHDR + 6
        PLUS3(WS1) := COUNT8            :: Plant 8-block count.
        COUNT8 := COUNT8+COUNT8
        COUNT4 := COUNT4+COUNT8
        WS1 := WS1-2
        PLUS3(WS1) := COUNT4            :: Plant 4-block count.
        COUNT4 := COUNT4+COUNT4
        COUNT2 := COUNT2+COUNT4
        WS1 := WS1-2
        PLUS3(WS1) := COUNT2            :: Plant 2-block count.
        RETURN                          :: Return, end of LINKFREE
        END
```

```
        DEFINE R RECOMBINE
:: If the maximum size of block set by the program is not greater
:: than 8 then control is returned immediately. Otherwise, starting
:: with 8-blocks, taking each list in turn any mates which are free
:: simultaneously are recombined to form a block of the next larger
:: size. Advantage is taken of the fact that mates, if free, are
:: consecutive blocks on the free space lists. The process terminates
:: when the maximum size is reached or when no further recombination
:: is possible.
        DO
        POINTER := FREEHDR+6            :: Point to 8-block list.
        P2 := POINTER+2                 :: Initialise P2.
NEXTLIST) IF COOF (POINTER) = 0 GO TO END:: If empty list.
        IF PLUS1(POINTER) GE MAXSIZE GO TO END
                                        :: If maximum size reached.
        COOF(P2) := 0                   :: Terminate list.
        P1 := POINTER                   :: Initialise working
                                        :: pointer P1.
        POINTER := POINTER+2            :: Point to next list.
        COOF(POINTER) := 0              :: Initially empty.
        P2 := POINTER                   :: Initialise working
                                        :: pointer P2.
        SIZE := PLUS1(P1)               :: Get current size.
        124,CSIZE,CSIZE,0               :: Convert to address units.
        124,CSIZE,CSIZE,0
        PLUS3(POINTER) := 0             :: Initialise block count.
NEXTBLOCK ) IF COOF(P1) = 0 GO TO NEXTLIST
                                        :: If end of current list.
        WR1 := COOF(P1)                 :: Address of next block.
        126,WR1,CSIZE,0                 :: Address of mate.
```

```
        IF WR1 = COOF(COOF(P1)) GO TO RECOMBINE
                                          :: If mates free.
        P1 := COOF(P1)                    :: Step down current list
        GO TO NEXTBLOCK
RECOMBINE) PLUS3(POINTER) := PLUS3(POINTER)+1
                                          :: Count new block.
        WS2 := PLUS2(POINTER)
        WS1 := COOF(P1)
        STMAP                             :: Store new map for 1st half.
        WS1 := COOF(COOF(P1))
        WS2 := 0
        STMAP                             :: Store new map for 2nd half.
        COOF(P2) := COOF(P1)              :: LInk block into next list.
        P2 := COOF(P1)
        COOF(P1) := COOF(WS1)             :: Detach from current list.
        GO TO NEXTBLOCK
END)    RETURN                            :: End of RECOMBINE.
        END
```

In addition to the routines given above, the implementation of an automatic garbage collector for LSIX also involves modifications to some of the sets of BCL instructions described in Appendix 3. Any LSIX instruction involving the storing of pointers must be ammended to plant pointer flags as required in the maps.

APPENDIX 5

Extensions to the BCL Compiler

Notes on the implementation of functions and groups with parameters in BCL.

(1) The subset of parameters implemented consists only of parameters of type A which are called by value.

(2) Functions and groups with parameters are assigned type 2 i.e. they are treated as indefinite groups. Functions without parameters are definite groups (type 1) unless declared as indefinite.

(3) Name records are redefined as

```
NAMEREC(?) IS (A DICLINK,A ADDR,A TYPE,8C NAME,A LENGTH,
                                    A ELTREC,A PARLIST)
```

(4) Formal parameters are declared with the name of the group to which they belong. For example in

```
CONS(A X,A Y) IS (.................)
```

X and Y are formal parameters of CONS and are local to this group. The name records of the formal parameters of a group are stored as a sublist of the name record of the group and are accessed via the field PARLIST.

(5) Formal parameters are allocated stack space as if they were variables of type AX (i.e. declarations which cause no input) declared as the first elements of the group.

(6) When used in the body of the group, the formal parameters are automatically offset by the group stack pointer (GROUPP) unlike other variables which are declared in an indefinite group.

(7) The groups MATCH and LOOKUP have been redefined so that the current list of formal parameters is searched before the main dictionary when a name is encountered.

(8) MATCH is also modified to allow forward references to functions. If when analysing an expression, an undefined name is found it is assumed to be that of a function and a forward reference record is set up for it by the group MATCH.

(9) NAMESUBS and SUBSC (groups which deal with variables in expressions) are also redefined to allow references to functions (type 1 or 2).

(10) There is no check that the number of actual parameters used in a function call is equal to the number of formal parameters in the function definition. Advantage can be taken of this in that local variables can be defined as formal parameters for which no actual values are passed.

(11) In ASSCOM (the group dealing with assignment commands) the expression on the right hand side has the general form

<operand> +<operand> | <operand> - <operand> | <operand>

As the contents of the accumulator and modifier are not saved when a function is evaluated, any second operand in an expression should not involve a call on a function. This restriction can easily be removed when more general expressions are implemented in BCL.

(12) The actual parameters may themselves involve further calls on functions and to any depth.

(13) At a function call or before entry to a group with parameters the values of the actual parameters are transferred to the run-time stack where they are found when the group is entered.

(14) The value of a function is returned by the EXIT function the parameter of which is the operand whose value is to be returned. The effect is simply to assign to a variable named RESULT the value to be returned, whence it may be picked up immediately after leaving the function body. The function EXIT may appear anywhere in the

Body of the function. If it appears in the middle of a set of alternatives, the system pointer stack is reset as if the group had been left in the normal way. The definitions of group and branch linkage records have been modified to simplify the resetting of the pointer stack.

The full details of the modifications to the BCL compiler are of interest only to the reader who is familiar with the implementation of BCL in itself and are not given here.

APPENDIX 6

Garbage Collection - Output from a BCL Program.

THE DERIVATIVE OF (2* X + 1)**3- 6* X

WITH RESPECT TO X IS 6* (2* X + 1)**2- 6

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

ADDRESS	LLINK	SYMBOL	RLINK
*20000000	00000000	00000020	20000014
*20000014	00000000	70010101	20000030
*20000030	20000000	16010101	20000044
*20000044	00000000	00000010	20000060
*20000060	20000030	35010101	20000074
*20000074	00000000	00000030	20000110
*20000110	20000060	16160101	20000124
*20000124	00000000	00000060	20000140
*20000140	00000000	70010101	20000154
*20000154	20000124	16010101	20000170
*20000170	20000110	36010101	20000204
*20000204	00000000	00000010	20000220
*20000220	20000000	16010101	20000234
*20000234	00000000	16010101	20000250
*20000250	20000000	35010101	20000264
*20000264	20000000	35010101	20000300
*20000300	00000000	00000010	20000314
*20000314	20000074	36010101	20000330
*20000330	20000060	16160101	20000344
*20000344	20000000	16010101	20000360
*20000360	20000074	16010101	20000374
*20000374	00000000	00000010	20000410
*20000410	20000124	16010101	20000424
*20000424	00000000	16010101	20000440
*20000440	20000124	35010101	20000454
*20000454	20000520	36010101	20000470
*20000470	00000000	00000020	20000504
*20000504	00000000	00000060	20000520
*20000520	20000504	16010101	00000000

THE DERIVATIVE OF 3* (X **2+ X)+ 2* X **3

WITH RESPECT TO X IS 3* (2* X + 1)+ 6* X **2

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

ADDRESS	LLINK	SYMBOL	RLINK
*20000000	20000660	16010101	20000014
*20000014	20000644	35010101	20000030
*20000030	00000000	00000010	20000044
*20000044	00000000	00000010	20000060
*20000060	20000564	36010101	20000074
*20000074	20000550	16160101	20000110
*20000110	20000030	16010101	20000124
*20000124	20000564	16010101	20000140
*20000140	00000000	00000010	20000154

```
*20000154  20000124 35010101 20000170
*20000170  20000534 16010101 20000204
*20000204  00000000 16010101 20000220
*20000220  20000170 35010101 20000234
*20000234  00000000 00000010 20000250
*20000250  00000000 00000010 20000264
*20000264  20000710 36010101 20000300
*20000300  20000674 16160101 20000314
*20000314  20000234 16010101 20000330
*20000330  20000710 16010101 20000344
*20000344  20000660 16010101 20000360
*20000360  00000000 16010101 20000374
*20000374  20000454 35010101 20000410
*20000410  20000170 35010101 20000424
*20000424  00000000 00000020 20000440
*20000440  00000000 00000060 20000454
*20000454  20000440 16010101 20000470
*20000470  00000000 00000010 20000534
*20000534  00000000 00000030 20000550
*20000550  00000000 70010101 20000564
*20000564  00000000 00000020 20000600
*20000600  20000550 16160101 20000614
*20000614  00000000 70010101 20000630
*20000630  20000600 35010101 20000644
*20000644  20000534 16010101 20000660
*20000660  00000000 00000020 20000674
*20000674  00000000 70010101 20000710
*20000710  00000000 00000030 20000724
*20000724  20000674 16160101 00000000

THE DERIVATIVE OF (X + Y )* (X - Y )

WITH RESPECT TO X IS X + Y + X - Y

THE DERIVATIVE OF 3* (2* X + 1)**3- 2* X **2

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

 ADDRESS    LLINK     SYMBOL   RLINK
*20000000  20000504 35010101 20000014
*20000014  00000000 70010101 20000030
*20000030  00000000 71010101 20000044
*20000044  20000014 36010101 20000060
*20000060  20000000 16010101 20000074
*20000074  00000000 00000010 20000110
*20000110  20000074 36010101 20000124
*20000124  20000000 16010101 20000140
*20000140  00000000 00000010 20000154
*20000154  20000140 35010101 20000170
*20000170  20000140 16010101 20000204
*20000204  20000000 35010101 20000504
*20000504  00000000 70010101 20000520
*20000520  00000000 71010101 00000000
```

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

ADDRESS	LLINK	SYMBOL	RLINK
*20000030	00000000	00000010	20000044
*20000044	00000000	00000010	20000060
*20000060	20000424	36010101	20000074
*20000074	20000410	16160101	20000110
*20000110	20000030	16010101	20000154
*20000154	00000000	16010101	20000504
*20000504	00000000	00000010	00000000

WITH RESPECT TO X IS 18* (2* X + 1)**2- 4* X

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

ADDRESS	LLINK	SYMBOL	RLINK
*20000000	00000000	16010101	20000014
*20000014	20000154	35010101	20000030
*20000030	20000520	16010101	20000044
*20000044	00000000	00000020	20000060
*20000060	00000000	00000060	20000074
*20000074	20000060	16010101	20000110
*20000110	00000000	00000220	20000124
*20000124	20000424	16010101	20000140
*20000140	20000374	16010101	20000154
*20000154	20000110	16010101	20000170
*20000170	20000030	35010101	20000204
*20000204	20000154	36010101	20000220
*20000220	00000000	00000030	20000234
*20000234	00000000	00000020	20000250
*20000250	00000000	70010101	20000264
*20000264	20000234	16010101	20000300
*20000300	00000000	00000010	20000314
*20000314	20000264	35010101	20000330
*20000330	00000000	00000030	20000344
*20000344	20000314	16160101	20000360
*20000360	20000220	16010101	20000374
*20000374	00000000	00000020	20000410
*20000410	00000000	70010101	20000424
*20000424	00000000	00000020	20000440
*20000440	20000410	16160101	20000454
*20000454	20000374	16010101	20000470
*20000470	20000360	36010101	20000520
*20000520	00000000	00000040	20000534
*20000534	00000000	00000010	20000550
*20000550	20000234	16010101	20000564
*20000564	00000000	16010101	20000600
*20000600	20000234	35010101	20000614
*20000614	20000234	35010101	20000630
*20000630	00000000	00000010	20000644
*20000644	20000330	36010101	20000660
*20000660	20000314	16160101	20000674
*20000674	20000234	16010101	20000710
*20000710	20000330	16010101	20000724
*20000724	20000220	16010101	00000000

```
THE DERIVATIVE OF 3* (2* X + 1)**2+ 6* X **3

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

 ADDRESS     LLINK     SYMBOL    RLINK
*20000520 00000000 00000010 20000534
*20000534 00000000 00000010 20000550
*20000550 20000170 36010101 20000600
*20000600 20000520 16010101 20000644
*20000644 00000000 16010101 00000000

GARBAGE COLLECTION ROUTINE ENTERED

NEW FREE SPACE LIST

 ADDRESS     LLINK     SYMBOL    RLINK
*20000250 00000000 00000010 20000264
*20000264 20000000 16010101 20000300
*20000300 00000000 16010101 20000314
*20000314 20000000 35010101 20000330
*20000330 20000000 35010101 20000344
*20000344 00000000 00000010 20000360
*20000360 20000074 36010101 20000374
*20000374 20000060 16160101 20000410
*20000410 20000000 16010101 20000424
*20000424 20000074 16010101 20000454
*20000454 00000000 16010101 20000534
*20000534 00000000 00000010 20000614
*20000614 20000170 16010101 20000630
*20000630 20000140 16010101 20000660
*20000660 20000520 35010101 00000000

WITH RESPECT TO X IS 12* (2* X + 1)+ 18* X **2

END OF PROGRAM
```

APPENDIX 7

The Definition and Implementation of LSIX in BCL.

(Published in the Computer Journal,

Vol. 12, Number 1, February 1969)

Reprinted from The Computer Journal, Vol. 12, No. 1, February 1969

The definition and implementation of LSIX in BCL

By R. J. W. Housden*

This paper describes the implementation on the London University Atlas computer of the Bell Telephone Laboratories low level linked list language L6. A syntactical definition of L6 is given in terms of BCL, a general purpose programming language with special emphasis on data structures. The description of the implementation in BCL includes details of the general field handling routines.

(First received February 1968 and in revised form September 1968)

LSIX is a London version of the Bell Telephone Laboratories low-level list processing language L6 (Knowlton, 1966). This paper describes an implementation of LSIX using BCL, a general purpose programming language with special emphasis on data structures (Hendry, 1966). The BCL used is that defined by the prototype compiler which was available in January, 1967. Both the definition and the implementation are in BCL; the former is freely annotated but for those not familiar with BCL a few words of explanation are given in an Appendix. It is considered that the ability to define a language so precisely in this way is one of the interesting features of this paper.

LSIX instructions are compiled into an intermediate code which is executed by a low-level interpreter. The definition is followed by an outline flow diagram of the interpreter and details of three general field handling routines to find fields, get fields and store fields.

A complete LSIX program which has been run on the Atlas computer is given to illustrate the form of the data to be analysed. For a more detailed description of the language the reader is referred to Knowlton's description of L6.

Special features of LSIX

The most important features of LSIX which distinguish it from other list processors such as IPL, LISP, COMIT and SNOBOL are the availability of several sizes of storage blocks and a flexible means of specifying within them fields containing data or pointers to other blocks. Data structures are built by appropriating blocks of various sizes, defining fields (simultaneously in all blocks) and filling these fields with data and pointers to other blocks. Available blocks are of lengths 2^n machine words where n is an integer in the range 0–7. The user may define up to 36 fields in blocks, which have as names single letters or digits. Thus the D field may be defined as bits 5 through 17 of the first word of any block. Any field which is long enough to store an address may contain a pointer to another block. The contents of a field are interpreted according to the context in which they are used.

The LSIX system contains 26 base fields called *bugs*. The contents of a bug are referred to by naming the bug (a single letter). If the bug contains a pointer to a block, a particular field in that block is referred to by concatenating the names of the bug and the field. For example WD refers to the D field of the block to which W points. A field more remotely positioned from the bug is referred to by concatenating the names of the bug, the sequence of pointers and the field. Thus if bug X points to a block whose B field points to a block whose A field points to a block whose D field is to be referenced, the latter is called XBAD.

Instruction format

In general an LSIX instruction consists of an optional label followed in order by optional tests, optional operations and an optional transfer of control. An example given by Knowlton is

L2 IFNONE (XD, E, Y)(XA, E, 0) THEN (XD, E, 1)(X, P, XA) L2

which says that

IF NONE	of the following is true: that the contents of XD equals the contents of Y or that the contents of XA equals 0
THEN	perform the following operations: set the contents of XD equal to 1, make X point where the current contents of XA point then go to the instruction labelled L2 (the same instruction in this case).
OTHERWISE	no operations are to be performed and control goes to the next line of coding.

Other conditions are

IFALL	satisfied IF ALL of the elementary tests are satisfied
IFNALL	satisfied IF NOT ALL of the elementary tests are satisfied
IFANY	satisfied IF ANY of the elementary tests are satisfied.
IF and NOT	are synonymous with IFALL and IFNONE.

The other instruction type is the unconditional instruction consisting of a sequence of operations to be performed.

A complete list of tests and operations is given in **Tables 1** and **2**. Some of these are illustrated by the following complete program which reads, sorts into ascending sequence and outputs numbers each terminated by a single space. The sequence of numbers is terminated by a double space. For simplicity the numbers are restricted to the range 0–9999.

```
(*20000000,SS,4,*20000400)
(1,DD,0,23)(2,DA,0,23)(3,DB,0,23)
(0,DZ,0,23)
(DO,INPUT)(DO,ORDER)(DO,OUTPUT)END
```

* *University of London Institute of Computer Science, 44 Gordon Sq., London WC1.*

Analysis of tests

TESTS is a series of tests defined in the usual manner. As each takes space in the object area, and the number is unknown, when no more tests are found, the address of the first operation is planted, by TESTSEND, in one of the locations reserved by INSTRSTRT. Test types and operand types are recorded in the variables K and J. At the end of each test, TESTEND plants the values of K and J (note the difference between TESTSEND and TESTEND). TESTSTRT initialises certain variables. An argument is defined by ARG as any combination of characters not including comma and right bracket. The arguments of tests and operations are separated by commas.

```
SEP IS (OSP.,',',OSP.)
TESTS IS (OSP.,TEST,EITHER TESTS OR OSP.,TESTSEND)
TEST IS ('(',TESTSTRT,FIELD,SEP
      ,(EITHER(EITHER 'E',K := 1
                OR    'N',K := 2
                OR    'G',K := 3
                OR    'L',K := 4)
              ,(EITHER 'O', J := 2
                OR    'H',J := 3
                OR       J := 0)
   OR       (EITHER 'O',K := 5
                OR    'Z',K := 6)
              ,(EITHER 'D', J := 1
                OR    'H', J := 3
                OR       J := 4)
   OR       'P', K := 7,J := 0)
```

Completes first argument and predicate. Continue with separator and second argument.

```
      ,OSP.,',',(EITHER IF J=0, (EITHER FIELD
                                    OR    DLITERAL)
                    OR    IF J=1, DLITERAL
                    OR    IF J=2, OLITERAL
                    OR    IF J=3, HLITERAL
                    OR    IF J=4,(EITHER FIELD
                                    OR    OLITERAL))
      ,')',TESTEND)
```

Literal operands and FIELDS are defined below.

Analysis of operations

In general operations have either three or four arguments the second of which is the mnemonic function code but there are two special cases (DO, symbol) and (a, Δ) an abbreviation for (a, P, aΔ) with only two arguments. Matching an operation involves two passes. On the first pass no information is planted in the object area. A shallow analysis determines the operation code (K) and the number of arguments (NA). This first attempt to match sets certain values (particularly NA) and is then deliberately failed by using the group REJECT. The results from the first pass are used during the detailed analysis on a second pass. This technique for making several passes is commonly used in BCL programming. OPSTART sets operation type (K) and number of operands (NA) to zero and allocates a location into which this information is planted by OPEND when the operation has been matched.

```
OPERATNS IS (OSP.,OPERATN,EITHER OPERATNS OR OSP.)
OPERATN IS ('(',OPSTART,OSP.
,(EITHER ARG,SEP,ARG,OSP.,')',NA := 2, REJECT
```

Shallow analysis for two argument operations completed. Go on to deep analysis of two argument operations.

```
OR IF NA=2,(EITHER 'DO', SEP
                  ,(EITHER 'STATE',K := 41,NA : = 0
                   OR    'DUMP', K := 42,NA : = 0
                   OR    SYMBOL, K := 35,NA : = 1)
             OR    (EITHER FIELD,SEP,REJECT
                   OR    FIELD,SEP
                         ,OBJECTP := OBJECTP-ONE
                         ,FLDNAMES,K : = 12)),OSP.,')'
```

Analysis of two argument operations completed. 'STATE' and 'DUMP' are system subroutines. SYMBOL is defined below. Note the special technique for dealing with the operation (FIELD, FLDNAMES). The object area pointer (OBJECTP) is set back one word and FIELD matched a second time. In this way the abbreviated operation (a,Δ) is expanded to its full form (a, P, aΔ) in the object area. Go on to shallow analysis of three and four argument operations

```
OR    ARG,SEP,OPCODE,SEP,ARG,NA := 2,SEP,ARG,NA := 3
      ,(EITHER IF K= 10 ,K := 36
        OR     IF K= 23 ,K := 36
        OR     IF K= 35 ,K := 36),REJECT
```

Shallow analysis completed. Certain ambiguities arising in the group OPCODE (defined below) are removed once the number of operands is known and the K values are then corrected before going on to the deep analysis of three and four argument operations. In the deep analysis which follows the operation code is assigned to the variable OPCD, OCT is a working variable and PLANT plants information in the object area.

```
OR    IF K LE 29,FIELD,SEP,OPCD,OSP.,','
      ,(EITHER IF K LE 27
            ,(EITHER IF J=0, (EITHER FIELD
                                OR    DLITERAL)
             OR    IF J=1, DLITERAL
             OR    IF J=2, OLITERAL
             OR    IF J=3, HLITERAL
             OR    IF J=4, (EITHER FIELD
                             OR    OLITERAL)
             OR    IF J=5,FIELD),')'
   OR IF K GT 27
      ,(EITHER FIELD OR DLITERAL)
      ,(EITHER IF NA=3,','
            ,(EITHER IF J=1,DLITERAL
              OR    IF J=3,HLITERAL
              OR    IF J=4,(EITHER FIELD
                            OR OLITERAL))
      OR    NIL.),')'
OR    IF K LE 31,IF K GT 29
            ,(EITHER FIELD OR DLITERAL)
            ,',',OSP.,OPCD,OSP.,','
            ,(EITHER IF J=3,HLITERAL
              OR    FIELD
              OR    OLITERAL),')'

OR    IF K=32,FIELD,SEP,OPCD,SEP,FLDNAME,OCT:=0,PLANT
            ,(EITHER IF NA=3,',',(EITHER FIELD
                                    OR    DLITERAL)
             OR    OSP.),')'
OR    IF K=33,(EITHER 'S' OR 'R', K := 43),SEP,'FC',SEP
            ,FIELD,')',NA := 1
OR    IF K=34,(EITHER 'S' OR 'R', K := 44),SEP,'FD',SEP
            ,FLDNAME,OCT := 0,PLANT,OSP.,')',NA := 1
OR    IF K=35,SYMBOL,SEP,'DO',SEP,SYMBOL,OSP.,')'
OR    IF K=36,(EITHER FIELD OR DLITERAL)
            ,SEP, 'D',FLDNAME,OCT := 0, PLANT
            ,SEP,(EITHER FIELD OR DLITERAL)
            ,SEP,(EITHER FIELD OR DLITERAL),')'
OR    IF K=37,('*',OLITERAL,SEP,OPCD,SEP,DLITERAL
            ,SEP,'*',OLITERAL),')'
```

*followed by octal digits is an octal integer in BCL. Its use here is as an octal address in the Atlas computer.

```
OR    IF K=38,FIELD,SEP,OPCD,SEP
            ,(EITHER FIELD OR DLITERAL)
            ,(EITHER IF NA=3,SEP,FIELD OR NIL.),')')
,OPEND)
```

An LSIX instruction is terminated by one or more newlines. INSTREND plants descriptive information (number of tests, operations, etc.) in locations allocated at the start of the instruction by INSTRSTRT.

NLS IS (OSP.,NL.,EITHER NLS OR NIL.)

NLS is similar to EOL but no information is planted.

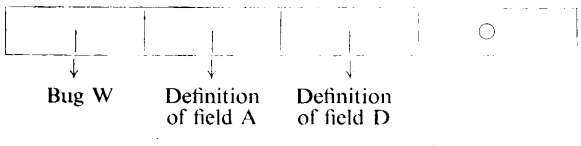
LSIXEND IS (FINISH := OBJECTP,OBJECTP := START
,OBJECTPRINT,EXECUTE)

Compilation is completed, the contents of the object area printed and execution commenced.

The Execution of an LSIX Program

During the analysis and recognition of LSIX source instructions descriptive information is planted in the object area. For each source instruction this information includes the type of instruction (conditional or unconditional), the number of tests, the number of operations and the type of transfer of control (normal transfer, subroutine return, or no transfer). For each test and operation is stored the test or operation code and the addresses of operands. The outline flow diagram in **Fig. 3** describes the operation of the interpreter routine.

Each operand, whether a base field (bug), remote field or a constant (decimal, octal or hollerith literal), is specified by a sequence of one or more addresses terminated by a zero. For example the remote field WAD is represented by the sequence of pointers



Three general field handling subroutines FINDFIELD, GETFIELD and STOREFIELD are used during execution to pick up and store operands. A field is defined at run time by its word number, left most bit and right most bit. For example, the operation (2,D6,3,17) defines field 6 of any block as bits 3 through 17 of word number 2. The execution of such an operation results in the setting up of a field definition, including a 24-bit mask, which is used by the field handling subroutines. Because of the complete generality of field definitions no attempt is made to use the few special hardware facilities for handling special cases. The only special case which might have been worth detecting is the field which spans the full 24 bits of the word.

Any field in the data structures may be specified by two pointers—one to the first word of the block containing the field and the other to the definition of the field concerned. Other operands, basefields and bugs, are specified directly by the first of these pointers and the second pointer is set to zero. In the three subroutines which follow the two pointers are stored in WREG1 and WREG2 respectively.

Subroutine to find a field

On entry OBJECTP points to the first of a sequence of addresses. On exit WREG1 points to the block containing the field and WREG2 to the definition of the field (conventionally zero for base fields and constants).

```
DEFINE R FINDFIELD
DO
WREG2 := 0
WREG1 := COOF(OBJECTP)              Pick up the first address.
OBJECTP := OBJECTP+ONE              Advance object pointer.
IF COOF(OBJECTP)= 0 GO TO END       If next address is
                                    zero go to end.
WREG1 := COOF(WREG1)                Get address of block
                                    to which WREG1 points.
MORE)WREG2 := COOF(OBJECTP)         Get address of next
                                    field definition.
OBJECTP := OBJECTP+ONE              Advance object pointer.
IF COOF(OBJECTP)= 0 GO TO END       If next address is
                                    zero go to end.
GETFIELD                            Get contents of the
                                    field specified
                                    by WREG1 and WREG2.
GO TO MORE
END)OBJECTP := OBJECTP+TWO          Advance object pointer
                                    to next item of
                                    information.
RETURN
END
```

Subroutine to get the contents of a field

On entry WREG1 points to a block and WREG2 to the definition of a field in that block (zero for base fields and constants). On exit WREG1 contains the contents of the field right justified and WREG2 is unchanged.

```
DEFINE R GETFIELD
DO
IF WREG2 NE 0 GO TO REMOTE          If more than one
                                    address then field
                                    is remote.
WREG1 := COOF(WREG1)                Otherwise pick up
RETURN                              contents (basefield or
                                    constant) and return.
REMOTE)WREG1 := WREG1 +             Point to word
                WORD(WREG2)         containing the field.
WREG1 := COOF(WREG1)                Pick up word
                                    including the field.
WREG3 := MASK(WREG2)                Copy the mask from
                                    the field definition.
127,WREG1,WREG3,0                   Machine order to
                                    mask the field.
SHIFT := 23-RBIT(WREG2)
IF SHIFT=0 GO TO END                Right justify the
1342,WREG1,SHIFT,0                  field in WREG1
END)RETURN
END
```

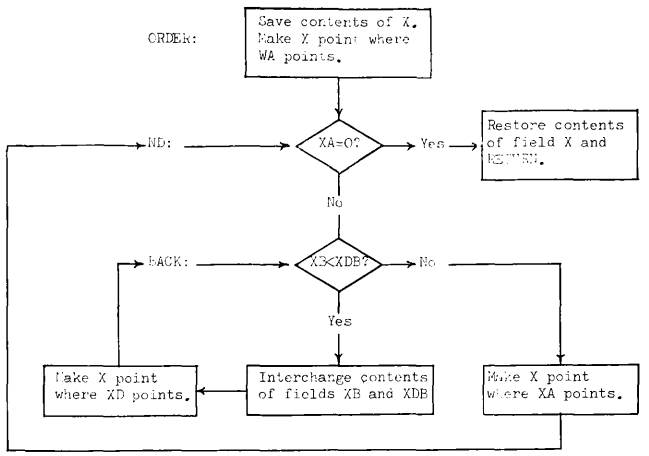


Fig. 2. Flow diagram for the subroutine ORDER

Table 1

Mnemonic notation used in Table 2 for describing L6 tests and operations. The notation is that used by Knowlton in the original description of L6. The ranges of arguments are those for the current Atlas LSIX.

Field Designators

c 'contents', i.e. designation of a field whose contents are used in a test or operation: either a bug, A, B, . . ., Z or a remote field, A0, A1, ..., ZZ . . . ZZZ (the number of characters is limited only by the length of a line of program.

a 'affected field', i.e. designation of a field whose contents are affected by an operation.

Names

f name of a definable field: 0, 1, ..., 9, A, B, ..., Z.

s a program symbol (label, name of a program location).

Literals

o an octal number specified directly: 0, 1, ..., 77777777.

d a decimal number specified directly: 0, 1, ..., $2^{24} - 1$.

h a general literal: 0, 1, ..., ZZZZ. All characters (Atlas inner set) except newline are permissible; comma and right bracket must be written as (,) and ( )) respectively. Newline must be specified by its octal equivalent. In the case of output operations Print and Punch the number of characters in a general literal is restricted only by the length of a line of program.

Alternatives

cd either c or d as defined above.

co either c or o.

The Atlas LSIX compiler records operand types in the variable J as follows:

J = 0	either field or decimal literal	J = 4	either field or octal literal
J = 1	decimal literal	J = 5	field.
J = 2	octal literal	J = 6	other special cases such as a field name (a single letter or digit).
J = 3	general literal		

Table 2

L6 Tests and Operations with the corresponding K-values used in the BCL implementation. Lower-case mnemonics are explained in Table 1.

TESTS

Equality, K = 1	Inequality, K = 2	Greater than, K = 3	Less than, K = 4
(c, E, cd)	(c, N, cd)	(c, G, cd)	(c, L, cd)
(c, EO, o)	(c, NO, o)	(c, GO, o)	(c, LO, o)
(c, EH, h)	(c, NH, h)	(c, GH, h)	(c, LH, h)
One-bits of, K = 5	Zero-bits of, K = 6	Pointers to same block, K = 7	
(c, O, co)	(c, Z, co)		
(c, OD, d)	(c, ZD, d)		
(c, OH, h)	(c, ZH, h)	(c1, P, c2)	

OPERATIONS

Copy field, K = 1	Add, K = 2	Subtract, K = 3	Multiply, K = 4
(a, E, cd)	(a, A, cd)	(a, S, cd)	(a, M, cd)
(a, EO, o)	(a, AO, o)	(a, SO, o)	(a, MO, o)
(a, EH, h)	(a, AH, h)	(a, SH, h)	(a, MH, h)
Divide, K = 5	Logical Or, K = 6	Logical And, K = 7	Exclusive Or, K = 8
(a, V, cd)	(a, O, co)	(a, N, co)	(a, X, co)
(a, VO, o)	(a, OD, d)	(a, ND, d)	(a, XD, d)
(a, VH, h)	(a, OH, h)	(a, NH, h)	(a, XH, h)
Complement, K = 9	Duplicate block, K = 10	Interchange field contents, K = 11	Point to same block as, K = 12
(a, C, co)	(a, DP, c)	(a, IC, a2)	(a, P, c)
(a, CD, d)			
(a, CH, h)			
Locate one bits from left, K = 13	Locate one bits from right, K = 14	Locate zero bits from left, K = 15	Locate zero bits from right, K = 16
(a, LO, c)	(a, RO, c)	(a, LZ, c)	(a, RZ, c)
Count one bits, K = 17	Count zero bits, K = 18	Blanks to zero, K = 19	Zeros to blanks, K = 20
(a, OS, c)	(a, ZS, c)	(a, BZ, c)	(a, ZB, c)
Binary to Decimal K = 21	Binary to Octal K = 22	Decimal to Binary K = 23	Octal to Binary K = 24
(a, BD, c)	(a, BO, c)	(a, DB, c)	(a, OB, c)
Free block, K = 26	Input, K = 27	Shift Left, K = 28	Shift Right, K = 29
(a, FR, 0)		(a, L, cd)	(a, R, cd)
(a, FR, c)	(a, IN, cd)	(a, L, cd, co)	(a, R, cd, co)
		(a, LD, cd, d)	(a, RD, cd, d)
		(a, LH, cd, h)	(a, RH, cd, h)
Print, K = 30	Punch, K = 31	Print List K = 32	
(cd, PR, co)	(cd, PU, co)	(c, PL, f)	
(cd, PRH, h)	(cd, PUH, h)	(c, PL, f, cd)	
Save field contents, K = 33	Save field definition, K = 34	Do subroutine K = 35	Define field, K = 36
(S, FC, c)	(S, FD, f)	(DO, s)	(cd, Df, cd, cd)
		(s2, DO, s)	
Set up storage, K = 37	Get block, K = 38	(DO, State), K = 41	(DO, Dump), K = 42
(s1, SS, d, s2)	(a, GT, cd) (a, GT, cd, a2)		
Restore field contents, K = 43	Restore field definition, K = 44	Not used K = 25, K = 39, K = 40.	
(R, FC, c)	(R, FD, f)		

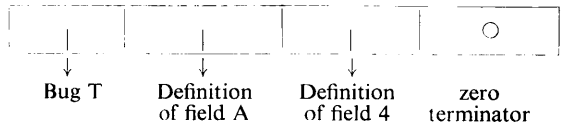
Appendix A note on BCL and the analysis of LSIX instructions

BCL is a general purpose programming language with special emphasis on data structures. Consider the sequence

```
FIELD IS (OSP., (EITHER 'T.', TIMEFIELD
                 OR     BUG, (EITHER FLDNAMES OR NIL.)
              OR INTEGER ,'.', IF INTEGER LE 128, READFIELD)
              ,OSP., OCT := 0, PLANT )
```

which occurs in the main text of this report. The first two words indicate that this is a definition of the 'name' FIELD. That the rest of it is a parenthesised structure with commas indicates that FIELD denotes a structure of the type known as a 'group'. The commas between the 'objects' denote juxtaposition, and for alternatives the notation EITHER. . . . OR. . . . is used. The objects within a group may be literals or names. Character literals are enclosed with primes, numeric literals are obvious, also literal commands such as $x := z$, and literal groups (in parentheses). Names, which must of course be defined somewhere, but can be defined *passim*, may be names of variables, routines or groups. Group definitions may be recursive, i.e. the name of a group may appear in its own list of objects.

Suppose we encounter the object 'FIELD' when in the course of reading in, and the next characters in the input stream are TA4, a remote field. These characters are matched with objects in the group FIELD. The first object, OSP., is a built in group which recognises and skips over any number (including zero) of spaces. Next we have the first of three alternatives. The next two characters in the input stream are compared with the literal 'T.'. T is matched but period is not so this match fails and the second alternative is tried. The group BUG recognises T as the name of a bug or base-field and plants its address in the object area. The second object in this branch is itself a pair of alternatives, (EITHER FLDNAMES OR NIL.), which matches any number of field names and computes and plants the addresses of the corresponding field definitions. In this example, field names A and 4 are recognised and the corresponding addresses planted. Finally, after the successful matching of the second alternative, OSP. reads over any spaces, the variable OCT is assigned the value zero and the group PLANT plants the value of OCT in the object area. Thus as a side effect of the recognition of the remote field TA4 the following sequence of pointers is planted in the object area.



A second example is the special read-only field 64. (an integral power of two terminated by a period). As the first character is a digit, attempts to match 'T.' and BUG fail and the third alternative is tried. The object INTEGER is an integer variable to which the integer 64 is assigned. Then the period is matched and if the condition INTEGER LE 128 is satisfied the routine READFIELD tests that the input integer is an integral power of two and computes and plants the address of the field '64.'.

When BCL is used as a compiler compiler, commands written as objects in a group may generate and plant object coding as soon as source language instructions are matched. Alternatively the user may, if he so wishes, construct analysis records.

References

HENDRY, D. F. (1966). *A Provisional Manual for the BCL Language*, University of London Institute of Computer Science (Internal report).

KNOWLTON, K. C. (1965). A Fast Storage Allocator, *Communications Assoc. Comp. Mach.*, Vol. 8, pp. 623–625.

KNOWLTON, K. C. (1966). A Programmer's Description of L^6, *Communications Assoc. Comp. Mach.*, Vol. 9, pp. 616–625.

UNWIN BROTHERS LIMITED, WOKING AND LONDON