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. 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

- IF and NOT are synonymous with IFALL and IFNONE.
- 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.

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)
(0,INPUT)(DO,ORDER)(DO,OUTPUT)END
```

* University of London Institute of Computer Science, 44 Gordon Sq., London WC1.
The routines INPUT, ORDER, and OUTPUT in that order and the program is finished. The routines are entered at the line beginning with their name, and conclude dynamically, though not necessarily lexicographically, on encountering the word DONE. For example the routine INPUT first asks for a new block of size 4 words to be pointed to by bug W, and writes the number 10,000 into the B-field of this block. It then saves the contents of bug X and the definition of field 1 in system push-down stores, gets a block of size 1 word, defines field 1 as bits 0 through 5 of word 0 of any block and enters the loop at NEXT. Inside the loop it asks for a new block which it links to the previous block, effectively pushing down W and plants a backward link in the D field of the previous block. The B field is set equal to zero. Characters are read one at a time to the field X1. If the character read is not a space it is left-shifted into the B field of the head block of the list then control is transferred to RD, otherwise if it is the second consecutive space (in which case QB equals 0) then field definition 1 is restored, the block to which X points is freed, the field contents of X are restored and control returns to the main program to continue with (DO,OUTPUT). Otherwise, if the value of XB is less than the value of XDB then the contents of these two fields are interchanged, the pointer X set equal to the pointer inXD and control transferred to BACK, else X is set equal to the pointer in field XA and control transferred to ND. Note that XD and (X,A) are abbreviations for the operations (X,P,DX) and (X,P,XA) respectively.

The routine OUTPUT prints the ordered sequence of numbers and returns blocks of storage to the free space list.

Definition of Ls6X

This section defines the syntax of Ls6X in BCL. The reader who is not familiar with phrase structure languages or has difficulty in understanding BCL is advised to read the Appendix before proceeding. The following program compiles an intermediate code which is executed by a low-level interpreter.

Main program structure

LS6X is (INITIALISE, LS6XSTATS)

LS6XSTATS is (EITHER INSTR

OR DIRECTIVE

OR NONMATCH,GARBAGE),LS6XSTATS)

The routine INITIALISE initialises certain working variables before commencing the compilation. An LS6X program is defined as LS6X 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 reads over the remainder of the line. In LS6X newline is the instruction separator. Compilation of an LS6X program is terminated by the directive **ENTER**.

DIRECTIVE is (OSP., **ENTER**, LS6XEND)

LS6XEND causes entry to the program. 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.

INSTR is (INSTRSTRT, OSP.

,EITHER CONDNL

OR UNCONDNL

OR LABEL,(EITHER CONDNL

OR UNCONDNL

OR EOL ))

The routine INSTRSTRT allocates locations in the object area for descriptive information concerning the line to be matched and initialises certain variables.

Types of instruction

CONDNL is (CONDITION, TESTS

,(EITHER 'THEN', OPERATNS OR NIL.)

,TRANSFER, EOL)

UNCONDNL is ((EITHER 'THEN' OR NIL.), OPERATNS

,(EITHER TRANSFER OR NIL.),EOL)

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

The first lines of this program are concerned with initialisation—setting up storage from octal 20000000 to 20000400 in blocks of size 4 24-bit words and then defining fields D,A,B and Z as bits 0 through 23 of words 1,2,3 and 0 of each block respectively.

The third line calls the three routines INPUT, ORDER and OUTPUT in that order and the program is finished. The routines are entered at the line beginning with their name, and conclude dynamically, though not necessarily lexicographically, on encountering the word DONE. For example the routine INPUT first asks for a new block of size 4 words to be pointed to by bug W, and writes the number 10,000 into the B-field of this block. It then saves the contents of bug X and the definition of field 1 in system push-down stores, gets a block of size 1 word, defines field 1 as bits 0 through 5 of word 0 of any block and enters the loop at NEXT. Inside the loop it asks for a new block which it links to the previous block, effectively pushing down W and plants a backward link in the D field of the previous block. The B field is set equal to zero. Characters are read one at a time to the field X1. If the character read is not a space it is left-shifted into the B field of the head block of the list then control is transferred to RD, otherwise if it is the second consecutive space (in which case WB equals 0) then field definition 1 is restored, the block to which X points is freed, the field contents of X are restored and control returns to the main program to continue with (DO,ORDER). When the terminator single space is found, the contents of the field WB are converted from 6-bit character code to binary and control transferred to read the next number.

The routine INPUT therefore builds a data structure consisting of a doubly linked list of 4-word blocks in the B fields of which are stored the input numbers. At the start of the instruction labelled NEXT after the first two numbers have been input the data structure has the form shown in Fig. 1.

The routine ORDER has the form shown in Fig. 2. It uses the bug X, so it begins by saving (and ends by restoring) the field contents of X and then makes X point to the block to which field WA points (this is set by the routine INPUT). If the contents of field XA are zero then the list has been ordered, the contents of X are restored and control returned to the main program to proceed with (DO,OUTPUT). Otherwise, if the value of XB is less than the value of XDB then the contents of these two fields are interchanged, the pointer X set equal to the pointer in XD and control transferred to BACK, else X is set equal to the pointer in field XA and control transferred to ND. Note that (X,D) and (X,A) are abbreviations for the operations (X,P,DX) and (X,P,XA) respectively.

The routine OUTPUT prints the ordered sequence of numbers and returns blocks of storage to the free space list.
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 INSTRSTR. Test types and operand types are recorded in the variables k and J. At the end of each test, TESTSEND plants the values of k and J (note the difference between TESTSEND and TESTEND). TESTSTR 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, 'test', either tests or osp, testsend)

TESTS is (osp, test, either tests or osp, testsend)

TEST is ('test', teststr, 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 DILITERAL)

OR if j = 1, DILITERAL)

OR if j = 2, OLITERAL)

OR if j = 3, HILITERAL)

OR if j = 4, (either field

OR OLITERAL))

(')', teststr, field, sep)

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 (po, 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 type (k) and the number of operands (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 OPEO when the operation has been matched.

OPRATNS IS (osp, operatn, either operatns or osp.)

OPERATN IS ('(', opstart, osp,

(either arg, sep, arg, osp, 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, 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 opcode, oct is a working variable and plant plants information in the object area.

OR if k le 29, field, sep, opcode, osp, .

(either if k le 27

(either if j = 0, (either field

or DILITERAL)

OR if j = 1, DILITERAL)

OR if j = 2, OLITERAL)

OR if j = 3, HILITERAL)

OR if j = 4, (either field

OR OLITERAL))

OR nil, '.

)' OR if k le 31, if k gt 27

(either field or DILITERAL)

('osp', ',', (either if na = 3, '.

(either if j = 1, DILITERAL)

OR if j = 3, HILITERAL)

OR if j = 4, (either field

OR OLITERAL))

OR nil, '.

)' OR if k = 32, field, sep, opcode, sep, fldnames, oct := 0, plant

(either if na = 3, '.

(either field

or DILITERAL)

OR osp, '.

)' OR if k = 33, (either 'st' or 'r', k := 43), sep, 'fc', sep

(field, .'na := 1

OR if k = 34, (either 'st' or 'r', k := 44), sep, 'fd', sep

, fldnames, oct := 0, plant, osp, '.

na := 1

OR if k = 35, symbol, sep, 'do', sep, symbol, osp, '.

)' OR if k = 36, (either field or DILITERAL)

, sep, 'd', fldnames, oct := 0, plant

, sep, (either field or DILITERAL)

, sep, (either field or DILITERAL, '.

OR if k = 37, (*', OLITERAL, sep, opcode, sep, DILITERAL

, 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, opcode, sep

(either field or DILITERAL)

, (either if na = 3, sep, field or nil, '.

)' OR if k = 36, field, sep, opcode, sep

, (either field or DILITERAL)

, (either if na = 3, sep, field or nil, '.

)'
A list of operations with the corresponding values of $k$ is given in Table 2. When $bcl$ is fully implemented, then on the second pass (when $k$ is known), the appropriate branch of OPERATN will be selected using a facility similar to a computed goto, so improving the efficiency of this group.

**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 usually specifies the type of operation. During the shallow analysis the operation code is assigned to the variable opc and looked up in the operations table by OPSEARCH (a binary search) which returns with values of $k$ and $j$ specifying the operation type and operand type respectively. The values of $k$ and $j$ are used on the second pass. $k = 100$ indicates failure to find the operation code in the table. The code 0 for defining a field is dealt with separately.

**REJECT** is (IF 1 = 0)

A match may be deliberately failed by means of an impossible condition.

**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)

TIMEFIELD plants the address of the field in which time is stored.

READFIELD checks that the integer is an integral power of 2 and plants the field address in the object area.

**Bug** is (LETTER, BUGADDR)

BUGADDR computes and plants the address of the specified field.

**FLDNAMES** is (FLDNAME, EITHER FLDNAMES OR NIL)

**FLDNAME** is (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)

**LTR** is (DGTEST, DGT)

**DGTEST** test if the next character is a letter or digit respectively, if so its Atlas internal value is assigned to LTR or DGT.

**Types of literal**

**DLITERAL** is (OSP., INTEGER, STCONST, PLANT

, OCT := 0, PLANT, OSP)

Decimal literals (positive integers in the range $0, 2^{24} - 1$) are assigned to the variable INTEGER. STCONST enters the constant in the constants table (if not already entered) and returns with its address which is planted in the object area.

**OLITERAL** is (OSP., WSL := 0, COUNT := 8, ODIGITS

,STCONST, PLANT, OCT := 0, PLANT, OSP)

Octal integers of not more than 8 digits may be assembled and stored in the constants table. WSL is a work space.

**ODIGITS** is (ODIGIT, COUNT := COUNT - 1, ASMBlODGT

,EITHER IF COUNT GT 0, ODIGITS OR NIL)

ASMBlODGT assembles octal numbers.

**ODIGIT** is (DIGIT, IF DIGIT LT 8)

**HLITERAL** is (WSL := 0, COUNT := 4, HCHARS, STCONST, PLANT

, OCT := 0, PLANT)

HCHARS accepts up to four characters (Atlas inner set) not including newline, comma and right bracket, packs them (right justified) in the work space WSL, and finally passes them to STCONST to be stored in the constants table. 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.

**Labels, label references**

**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. A label followed by newline is interpreted as a reference to a label. A label reference implies a transfer of control for which the 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.

**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)

LABELSET is (MATCH

,EITHER IF ADDR(CURRENT) = 0

,ADDRCURRENT := OBJECTP

,(EITHER IF REFADDR(CURRENT) = 0

,NCURRENT := REFADDR(CURRENT)

,REFADDR(CURRENT) := 0, PLUGLIST)

OR  OPLABEL('LABEL', LBL, 'SET TWOIC')

Match compares LBL with entries in the dictionary. If this is the first occurrence of the label name a new record is set up. When a label is set, any forward references are plugged by PLUGLIST. CURRENT and NCURRENT are pointers to dictionary records and forward reference records respectively.

**LABELREF** is (MATCH, OCT := ADDR(CURRENT)

,(EITHER IF ADDR(CURRENT) = 0

,SETUP(FWDREF, NCURRENT, DICEP)

,LINK(NCURRENT) := REFADDR(CURRENT)

,ADDRESS(NCURRENT) := OBJECTP

,REFADDR(CURRENT) := NCURRENT OR NIL)

,PLANT)

If the label has not been set, a record of the forward reference is 'SETUP' and inserted in the pluglist. DICEP is a pointer to a list of free space available for setting up new records.

**SYMBOLES** (LBL, JUNK, EITHER IF LBL = 'STATE', K := 41, NA := 0

OR IF LBL = 'DUMP', K := 42, NA := 0

OR LABELREF, OCT := 0, PLANT)

References to the two system subroutines 'STATE' and 'DUMP' are special cases. The operations (DO, STATE) and (DO, DUMP) do not use the normal subroutine entry and return.

**Some miscellaneous groups**

**PLANT** is (COOF(OBJECTP) := OCT, OBJECTP := OBJECTP + ONE)

The next location in the object area is set equal to the variable OCT and the object pointer advanced one word. This group is used by BUGADDR and FLADDR.

**EOL** is (OSP., NIL, EITHER EOL OR INSTREN.)
An LSIX instruction is terminated by one or more newlines. INSTRND 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

```
DEFINE R FINDFIELD
DO
WREG2 := 0
WREG1 := COOF(OBJECTP)
OBJECTP := OBJECTP+ONE
IF COOF(OBJECTP)= 0 GO TO END
WREG1 := COOF(WREG1)
MORE)WREG2 := COOF(OBJECTP)
OBJECTP := OBJECTP+ONE
IF COOF(OBJECTP)= 0 GO TO END
GETFIELD
GO TO MORE
END

Bug W  Definition
of field A  Definition of field D

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 GETFIELD
DO
IF WREG2 NE 0 GO TO REMOTE
WREG1 := COOF(WREG1)
RETURN
REMOTE)WREG1 := WREG1 + WORD(WREG2)
WREG1 := COOF(WREG1)
WREG3 := MASK(WREG2)
127,WREG1,WREG3,0
SHIFT := 23—RBIT(WREG2)
IF SHIFT=0 GO TO END
1342,WREG1,SHIFT,0
END
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
WREG1 := COOF(WREG1)
RETURN
REMOTE)WREG1 := WREG1 + WORD(WREG2)
WREG1 := COOF(WREG1)
WREG3 := MASK(WREG2)
127,WREG1,WREG3,0
SHIFT := 23—RBIT(WREG2)
IF SHIFT=0 GO TO END
1342,WREG1,SHIFT,0
END
RETURN
END
```

Downloaded from https://academic.oup.com/comjnl/article-abstract/12/1/15/311557 by guest on 13 March 2019
**Subroutine to store a field**

On entry WREG1 points to a block, WREG2 to a field definition (or zero). OCT contains the item to be stored in the specified field.

**DEFINE R STOREFIELD**

**DO**

**IF** WREG2 NE 0 GO TO REMOTE

**IF** WREG1 LT BUGBASE GO TO ERROR

COOF(WREG1) := OCT

**RETURN**

REMOTE)SHIFT := 23 — RBIT(WREG2)

1343,OCT,SHIFT,0

Protect read only fields.

WREG3 := MASK(WREG2)

Get mask from field definition.

127,OCT,WREG3,0

Mask operand to remove most significant bits if too long for specified field.

WREG1 := WREG1 +

WORD(WREG2)

Point to word in block.

WREG2 := COOF(WREG1)

Pick up contents of word.

126,WREG3,0,*77777777

Form complement of mask.

127,WREG2,WREG3,0

Clear specified field to receive new item.

167,WREG2,OCT,0

Add operand into cleared field.

COOF(WREG1) := WREG2

**RETURN**

ERROR)O/P('ATTEMPTING TO WRITE TO READ FIELD')

**RETURN**

**END**

The efficiency of Lsix depends largely upon the efficiency of these three field handling routines which are used for all operands. For this reason, in the implementation of Lsix on the Atlas computer, the inefficient shift extracodes 1342 and 1343 shown here are replaced by a jump into a table of shifts.

**Storage organisation**

An important feature of Lsix is the availability of several sizes of blocks. In general these are of length $2^n$ words where $n$ is in the range 0-7. One of the first operations in any Lsix program is to set up a linked list of blocks of free space of a specified size. These blocks are easily subdivided to form smaller blocks as required and when blocks in contiguous parts of the store are free simultaneously they may be recombined, if necessary, to form larger blocks. It is not possible to make more than a brief reference to storage organisation here, but a similar method has been described by Knowlton (1965) and it is hoped that a more detailed account of the allocation and freeing of storage, including an extension which allows any size of block, will be given in a future report.

**Conclusions**

In this paper we are concerned with two languages Lsix and BCL. The efficiency of an Lsix compiler written in BCL depends upon the BCL compiler for the machine concerned. Routines which are very efficient on some machines may be inefficient on others. As machine orders may be written anywhere in a BCL program, inefficiencies arising from the BCL compiler are easily removed by replacing the BCL commands by machine orders. This is particularly important in the inner loops of the interpreter routine.

Some Lsix operations are necessarily machine dependent especially the bit manipulation routines and input and output. In most implementations the routines for performing such operations are written in the basic language of the machine concerned. The analysis and recognition of Lsix instructions and many of the tests and operations are machine independent and can therefore be programmed in a high level language such as BCL with the obvious advantage that reasonably efficient compilers can be written quickly and with little effort. Only a few machine dependent routines need be written in basic machine language. Roughly 18 per cent of the Lsix compiler is in machine code. To make it available on another machine, only this fraction has to be written in the new machine language. As to the availability of the rest on a new machine, this depends on the provision of a BCL compiler, but as much of the BCL compiler is similarly written in BCL itself, this likewise is a smaller job than it would otherwise be. This application of BCL leaves no doubt as to its power and suitability for compiling compilers.

**Acknowledgements**

I should like to express my thanks to Mr. Bryan Higman for many helpful discussions, as a result of which I first became interested in L6, and for the interest he has shown in this work. Thanks are also due to Mr. David Hendry, who is responsible for the design and development of the BCL language, and to Derek Brough and Anne Meredith, members of the BCL group, whose help and enthusiastic support has been greatly appreciated.
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 . . . ZZ (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.
- 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.

Names

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 = 1 decimal literal
- J = 2 octal literal
- J = 3 general literal
- J = 4 either field or octal literal
- J = 5 field
- J = 6 other special cases such as a field name (a single letter or digit).

Fig. 2. Flow diagram for the subroutine ORDER

Table 2

L6 Tests and Operations with the corresponding K-values used in the BCL implementation. Lower-case mnemonics are explained in Table 1.

<table>
<thead>
<tr>
<th>OPERATIONS</th>
<th>TESTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy field,</td>
<td>Equality,</td>
</tr>
<tr>
<td>K = 1 (a, E, cd)</td>
<td>K = 1 (c, E, cd)</td>
</tr>
<tr>
<td>(a, EO, o)</td>
<td>(c, NO, o)</td>
</tr>
<tr>
<td>(a, EH, h)</td>
<td>(c, NH, h)</td>
</tr>
<tr>
<td>Divide,</td>
<td>Inequality,</td>
</tr>
<tr>
<td>K = 5 (a, V, cd)</td>
<td>K = 2 (c, N, cd)</td>
</tr>
<tr>
<td>(a, VO, o)</td>
<td>(c, NO, o)</td>
</tr>
<tr>
<td>(a, VH, h)</td>
<td>(c, NH, h)</td>
</tr>
<tr>
<td>Complement,</td>
<td>Greater than,</td>
</tr>
<tr>
<td>K = 9 (a, C, co)</td>
<td>K = 3 (c, G, cd)</td>
</tr>
<tr>
<td>(a, CD, d)</td>
<td>(c, GO, o)</td>
</tr>
<tr>
<td>(a, CH, h)</td>
<td>(c, GH, h)</td>
</tr>
<tr>
<td>Note one bits from left, K = 13</td>
<td>Less than,</td>
</tr>
<tr>
<td>(a, LO, c)</td>
<td>K = 4 (c, L, cd)</td>
</tr>
<tr>
<td>(a, RO, c)</td>
<td>(c, GO, o)</td>
</tr>
<tr>
<td>(a, LH, h)</td>
<td>(c, GH, h)</td>
</tr>
<tr>
<td>Locate zero bits from right, K = 14</td>
<td>Point to same block,</td>
</tr>
<tr>
<td>(a, LK, c)</td>
<td>K = 7 (c, X, d)</td>
</tr>
<tr>
<td>(a, ND, d)</td>
<td>(c, XH, h)</td>
</tr>
<tr>
<td>Count one bits, K = 17</td>
<td>Locate zero bits from right,</td>
</tr>
<tr>
<td>(a, OS, c)</td>
<td>K = 16 (a, RZ, c)</td>
</tr>
<tr>
<td>(a, ZS, c)</td>
<td>(a, ND, d)</td>
</tr>
<tr>
<td>(a, BZ, c)</td>
<td>(a, XH, h)</td>
</tr>
<tr>
<td>Binary to Octal</td>
<td>Inequality,</td>
</tr>
<tr>
<td>K = 21 (a, BD, c)</td>
<td>K = 30 (c, N, cd)</td>
</tr>
<tr>
<td>(a, BO, c)</td>
<td>(c, ND, d)</td>
</tr>
<tr>
<td>(a, DB, c)</td>
<td>(c, XH, h)</td>
</tr>
<tr>
<td>Shift Left,</td>
<td>Greater than,</td>
</tr>
<tr>
<td>K = 26 (a, FR, 0)</td>
<td>K = 42 (c, XH, h)</td>
</tr>
<tr>
<td>(a, FR, c)</td>
<td>(c, ND, d)</td>
</tr>
<tr>
<td>(a, LH, h)</td>
<td>(c, XH, h)</td>
</tr>
<tr>
<td>Print,</td>
<td>Less than,</td>
</tr>
<tr>
<td>K = 30 (cd, PR, co)</td>
<td>K = 4 (c, X, d)</td>
</tr>
<tr>
<td>(cd, PRH, h)</td>
<td>(c, ND, d)</td>
</tr>
<tr>
<td>(cd, PUH, h)</td>
<td>(c, XH, h)</td>
</tr>
<tr>
<td>Save field contents,</td>
<td>Point to same block,</td>
</tr>
<tr>
<td>K = 33 (S, FC, c)</td>
<td>K = 35 (a, RD, cd)</td>
</tr>
<tr>
<td>(S, FD, f)</td>
<td>(c, RH, cd)</td>
</tr>
<tr>
<td>(DO, s)</td>
<td>(s2, DO, s)</td>
</tr>
<tr>
<td>Set up storage,</td>
<td>Define field,</td>
</tr>
<tr>
<td>K = 37 (a, GT, cd)</td>
<td>K = 36 (cd, DF, cd)</td>
</tr>
<tr>
<td>(a, GT, cd, a2)</td>
<td>(a, RH, cd)</td>
</tr>
<tr>
<td>Restore field contents,</td>
<td>Not used</td>
</tr>
<tr>
<td>K = 43</td>
<td>(a, GT, cd, a2)</td>
</tr>
<tr>
<td>K = 44</td>
<td>(a, RH, cd)</td>
</tr>
<tr>
<td>(R, FC, c)</td>
<td>(R, FD, f)</td>
</tr>
<tr>
<td>(R, FD, f)</td>
<td>(R, RH, f)</td>
</tr>
</tbody>
</table>
Object pointer:= Address of next description word. 
Pick up: description word, address of next description word in sequence and the address of the first operation in the current instruction.

Unpack description word:
NT:= number of tests;
NO:= number of operations;
COND:= type of condition (or zero);
GOTOFLAG:= type of transfer (or zero);

Find first operand (FINIFIELD)
Get first operand (GETFIELD)

Find second operand
Get second operand

Perform specified test (Switch via K). Result:= true or false.

Switch via COND

Object pointer:= address of first operation (or transfer)

NT:=NT-1

NT=0?
Yes
Switch via COND 1,2

No

NT=0?
Yes
Switch via COND 1,2

No

K:= operation code;
NA:= number of operands.

Locate operands (FINIFIELD).

Switch via GOTOFLAG.

NO:=NO-1

Switch via K to subroutine to pick up operands and perform operation.

Object pointer := Transfer address.

Subroutine exit.

Subroutine fail exit.

STOP, end of run.

COND=1 represents IF ANY
COND=2 := IF ALL
COND=3 represents IF FALL, IF
COND=4 := IF ELSE, ICT

Fig. 3. Operation of the interpreter routine
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 compared with objects in the group FIELD. The first object, OSP., is a built in group which recognises a group of field names and computes and plants the address 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.

```
| Bug T | Definition of field A | Definition of field 4 | zero terminator |
```

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 'r.' 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


Book Review

Indices and Primitive Roots, by A. E. Western and J. C. P. Miller 1968; 385 pages. (London: C.U.P., 26 0s. Od.)

This work incorporates and supersedes Haupt-Exponenten, Residue-Indices, Primitive Roots, and Standard Congruences, published in 1922 by the late Lt-Col. Cunningham in collaboration with H. J. Woodall and T. G. Creak. It may also be regarded as a continuation of Jacobi's Canon Arithmeticus.

The editors denote by \( g, g', h \) respectively the least positive, the least negative, the least primitive root modulo \( P, P \) being an odd prime. It would be convenient to define also \( G = g \) if \( g < g' \), \( G = -g' \) if \( g' > g \). With this, the index of \( a \) (prime to \( P \) given in the main tables is the least \( n \geq 0 \) such that \( a = G^n (mod \ P) \). The tables give (i) the complete factorisation of \( P - 1 \); (ii) \( g, g' \) and \( h \); (iii) the indices of certain \( a; \) and (iv) the residue-indices \( v = g.c.d. \)

\( (\text{ind } a, P - 1) \).

Table 1 covers all \( P \) up to 50021, Table 2 the \( P \) between 50000 and \( 10^3 \) and \( P \equiv 1 \) (mod 24). Table 3 goes up to 250000, with the stronger restriction \( P = 1 \) or 49 (mod 120); and in Table 4 \( P = 1 \) (mod 120) and \( P < 10^8 \). This large range of \( P \) is made possible by restricting the range of \( a; \) in Table 1, \( a \) ranges over primes up to 37 and 6, 10, 12. With this information it is not too difficult to calculate the indices of other \( a; \) as explained in the introduction.

The original calculations were all done by hand or with a desk machine, and the method is explained in detail, with some subsidiary tables, so as to enable the reader to investigate primes \( P \) not given in the main tables. All the entries have, however, been checked at least twice, on the ACE computer at the National Physical Laboratory. As a result, the surviving editor (Dr. Miller) hopes that very few errors remain; the reviewer is unable to say whether he is right.

The tables should be very useful to workers in the field. They provide evidence for many plausible conjectures, e.g. that \( g = g(P) \) defined above is of very low order of magnitude for large \( P \).

G. L. WATSON (London)