A Programmer's Description of L^6

Kenneth C. Knowlton

Bell Telephone Laboratories, Murray Hill, New Jersey

Bell Telephone Laboratories' Low-Level Linked List Language L^6 (pronounced "L–six") is a new programming language for list structure manipulations. It contains many of the facilities which underlie such list processors as IPL, LISP, COMIT and SNOBOL, but permits the user to get much closer to machine code in order to write faster-running programs, to use storage more efficiently and to build a wider variety of linked data structures.

CONTENTS

Presented at an ACM Symposium on Symbolic and Algebraic Manipulation, Washington, D. C., March 29–31, 1966.

1. Introduction

Bell Telephone Laboratories' Low-Level Linked List Language (L^6, pronounced "L-six") contains many of the facilities which underlie such list processors as IPL [1], LISP [2], COMIT [3] and SNOBOL [4], but it permits the user to get much closer to machine code in order to write faster-running programs, to use storage more efficiently and to build a wider variety of linked data structures. Preliminary experiments indicate that L^6-coded programs may achieve running speeds an order of magnitude faster than the same problems coded in the higher languages, at the expense of slightly more tedious or detailed programming.

Important features of L^6 are: the availability of several sizes of storage blocks, a flexible means of specifying within them fields containing data or pointers to other blocks, a wide range of logical and arithmetic operations on field contents, and an instruction format in which remote data is referenced by concatenating the names of fields containing the succession of pointers leading to this data. In the following paragraphs these features are described briefly; remaining sections of this paper give the details and serve as a programmer's manual. Programming examples are presented in Section 6. L^6 is also described in two computer-produced movies [5].

L^6 data structures are made by fetching from a storage allocator blocks of many different sizes, and linking them by pointers which are planted in fields which the programmer himself defines. Blocks, when in use, are immobile in storage; they are never relocated automatically nor changed in size. Relative sizes of blocks go as powers of 2; thus the storage allocator can easily split large blocks of free storage into smaller ones and, conversely, can easily fit pieces back together to reconstitute large blocks if and when their parts are simultaneously free [6].

The formatting of data and pointers within blocks is left entirely to the programmer. He may define a number of fields each of which is the specific contiguous part of the particular word, of all blocks, which he specifies. To define a field is actually to cause a run-time compilation of short subroutines which the system uses for fetching, storing or pointer-chasing with respect to this field. To redefine the field during the run is thus to recompile the appropriate subroutines, in terms of numbers just computed if desired; automatic formatting of data within blocks is thus possible. (See Figure 1 for examples of these subroutines for 7094 L^6.)

The exact form of the subroutines depends on the particular machine and on the location of the field in the word: the subroutines are most efficient if the field corresponds to a part of the machine word which the particular computer can handle efficiently, as for example, the 15-bit "decrement" on the 7094. Much of the speed of L^6 programs is due to this optimization of basic subroutines.

In L^6 a remote data item is conveniently referenced by concatenating the single-character names of pointers leading to this data. Thus, if base register A points to a block whose B–field points to a block whose C–field points to a block whose D–field is to be referenced, the latter is called simply ABCD. This is in contrast with the more usual functional notation: D(C(B(A))). The L^6 order of mentioning pointer seems to be far more natural and its notation, slightly modified, has been adopted by PL/I [7].

The single-character restriction on field names precludes private mnemonics, but it has two definite advantages. First, it eliminates one level of punctuation otherwise needed to delimit field names in a sequence, thus making coding not only more legible but also sufficiently compact so that many operations can be strung out on a single line. The resulting program therefore appears much more like a two-dimensional flowchart than does a program written as a long vertical list of operations and tests. The second advantage of punctuation-free concatenation is that it forces on the programmer the higher-level mnemonics appropriate for his problem. Thus in the case cited, when he has used ABCD a few times he begins to think of it as a semantic unit, simplifying his conceptual framework from that in which he visualized the three intervening pointers.

L^6 has been implemented, by Macro FAP [8] programming, on the IBM 7094 at Bell Telephone Laboratories, and an extended version of it is being written for the GE-635 and GE-645 [9]. It is presently being used for a variety of purposes, including information retrieval, simulation of digital circuits, and automatic drawing of flowcharts and other graphical output. It has also been used in programming courses in teaching the principles of list processing, because its inner workings are more obvious than those of other languages.

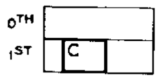
In general, L^6 is useful where storage allocation is microscopic and dynamic or where the programmer wants the pattern of pointers among data items to correspond closely to the physical or logical structure of the objects with which his program deals (electronic circuits, communication networks, strings of text, parsed sentences and formulas, search trees) as in simulation, game playing, symbol manipulation, information retrieval and graph manipulation. It can also serve as the means for implementing quickly, and in a relatively machine-independent way, higher-level list languages which contain more powerful operations for specific problem areas.

2. Data Structure

Data structures are built in L^6 by appropriating blocks of various sizes, defining fields (simultaneously in all blocks) and filling these fields with data and pointers to other blocks. These structures and operations are described in terms of a 36–bit word machine; in general, programs written for machines of a particular word size will run without reprogramming on all machines of larger word size.

2.1 Blocks. Available blocks are those whose lengths are one machine word (called "1–blocks"), 2 machine words ("2–blocks"), 4 machine words, and in general 2^n machine words up to 128 words. In describing blocks and fields in blocks, the word with the lowest machine address is called the 0th word, the next is the 1st word, etc. In each word, the leftmost bit is called the 0th bit, the next is the 1st bit, etc., and the rightmost bit is therefore the 35th bit.

2.2 Definable Fields in Blocks. The programmer may define up to 36 *fields* in blocks, which have as names single digits or single letters: 0, 1, 2, $\cdots$, 9, A, B, $\cdots$, Z. Each of these fields is a specified contiguous part of a particular word of every block. Thus the C–field of every block may be defined as bits 6 through 20 of the 1st word of the block, as here indicated:



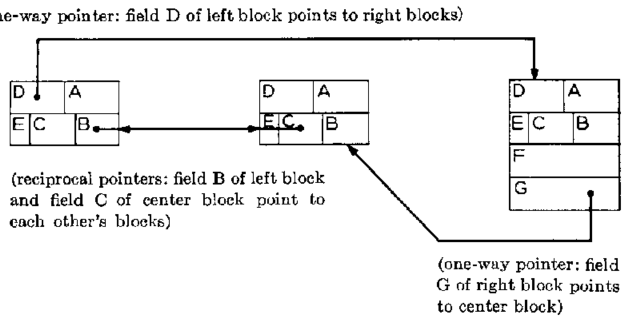
Some fields may fall outside of the smaller blocks, and the programmer must not attempt to use, for example, the C–field (as here defined) of a 1-block.

A field is defined or redefined by specifying its name, its word in the block (0–127), its leftmost bit (0–35) and its rightmost bit (0–35). A field may overlap or be part of another field; it may be a full machine word, or it may be null. If a field is defined or redefined at an intermediate point in a run, it may be defined in terms of quantities which have just been computed.

The contents of a field may be treated as alphanumeric information, a non-negative integer, a bit pattern to be manipulated by logical operations, or a pointer to another block, as described below. The type of information contained in a field is implicit in the way the program uses it; there is no need to identify it so that the L^6 system knows, for example, whether a certain field contains a pointer.

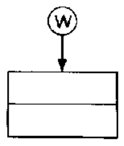
(One restriction on fields for current 7094 L^6 is that they may not include bits 0–2 or 18–20 of the 0th word. This restriction will be removed in subsequent versions and the examples in this paper do in fact violate it).

2.3 Pointers. Any field long enough to hold an address may contain a pointer to another block (technically the address of the 0th word of that block). For purposes of illustration, let us suppose that the programmer has defined fields A and D as 18–bit fields, B and C as 15–bit fields, E as a 6–bit field, and F and G as full-word fields as indicated in the diagram below. In addition the indicated symbolism is used for one-way pointers and for reciprocal pointers, i.e., pairs of pointers, each of which points to the other's block.


(one-way pointer: field D of left block points to right blocks)
D
A
E
C
B
(reciprocal pointers: field B of left block and field C of center block point to each other's blocks)
F
G
(one-way pointer: field G of right block points to center block)

2.4 Base Fields, called "Bugs". The L^6 system contains a set of base fields, each 36–bits wide and capable of holding a pointer, a number, a logical bit pattern or binary-coded alphanumeric information. There are 26 base fields, with names A, B, $\cdots$, Z; these names are distinguished from the names of definable fields in blocks by the way they are used, as described below.

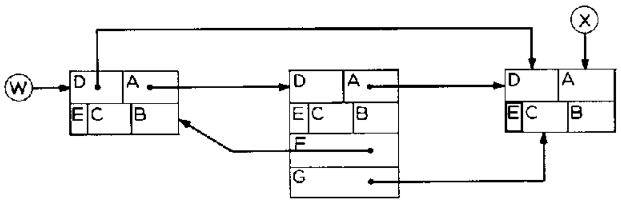
The base fields are called "bugs" because, by changing the pointers which they contain, they may be thought of as hopping or crawling through the intricate network of connected blocks, each "sitting" at any instant on the block which it is pointing to. In illustrations, a bug containing a pointer will be drawn as a small circle with bug name inside and an arrow to a block:



It is through pointers in bugs that access to all blocks and their data is achieved. By appropriate use of pointers in both bugs and blocks, useful data structures such as strings, loops, trees and graphs can be built and manipulated.

2.5 Referencing of Field Contents—Concatenation of Names. In L^6 coding, the contents of a bug are referred to simply by naming the bug, e.g., by writing W. If a bug points to a block, a particular field in that block is referred to by concatenating the names of the bug and the field. WD refers to the contents of the D–field of the block that bug W is pointing to.

A field more remotely positioned from a bug is referred to by concatenating names of the bug, the sequence of pointers, and the field. Thus the contents of the E–field of the right-hand block



may be referred to as

XE or WDE or WAAE or WAFAGE.

Normally, a pointer points to (i.e., is the address of) the 0th word of a block. However, if a number n is added to it so that it now points to the nth word of a block, then all fields accessed *via this pointer* will function as if they had been defined n words lower than their normal positions. Thus, by successively incrementing a pointer, the programmer can scan through a long block, applying the same field names successively to different words of the block.

2.6 Literals. Fixed quantities, i.e., literals, to be assembled into a program may be expressed in decimal, octal or alphanumeric form. A number is either a decimal or octal literal (it begins with a digit, not a bug name). Which type it is, depends upon the particular test or operation it is used in, as described in Sections 4 and 5.

An alphanumeric, or "Hollerith" literal, is a sequence of characters each of which is a digit, a letter or a period. Hollerith literals are recognized by the H which is appended to the particular test or operation code as described in Sections 4 and 5; they are commonly used in connection with input and output operations.

2.7 Read-Only Fields. Another set of special fields can only be read but not changed directly. These "read-only" fields contain the following information: "*T*." contains time in milliseconds since the beginning of the run; "*n*." contains the potential number of blocks of size n presently available from the storage allocator, where n may be 1, 2, 4, $\cdots$, 128. ("Potential number" here means the number of blocks which would result if all blocks larger than n were split into n-blocks.)

3. Instruction Format

Programming in L^6 involves the use of five different instruction types with similar formats. An example of one of these, the IFNONE type, is

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

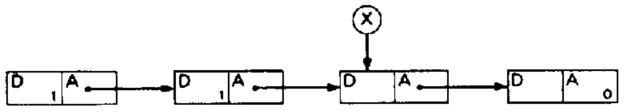
which says that

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 current contents of XA point,
  then go to the instruction labeled L2 (the same instruction in this case).

otherwise no operations are to be performed and control goes to the next line of coding.

This particular instruction might have been used to cause bug X to scan a string of blocks connected by A–fields, changing D–fields to 1's:



The scan terminates either upon reaching the end of the list (indicated by a 0 in the A–field) or upon finding a D-field whose contents equal the contents of bug Y.

The above sample instruction contains two tests and two operations; other numbers are possible provided that the entire instruction fits onto one punched card.

A test in general is a question about how the contents of a field compares with a literal (as the 0 in the example), or with the contents of another field (as the Y in the example). An operation on the other hand generally specifies some change to be made in the contents of a field and it usually involves the contents of another field (as the XA above), or a literal (as the 1 above). The system's complete repertoire of tests is presented in Section 4 and that of its operations is given in Section 5; both are summarized in Table II, whose mnemonics are explained in Table I. The names of three other conditional instructions, similar in format, and the conditions under which they are satisfied are

IFANY,	satisfied IF ANY of the elementary tests are satisfied,
IFALL,	satisfied IF ALL of the elementary tests are satisfied, and
IFNALL,	satisfied IF NOT ALL of the elementary tests are satisfied

In each case, if the compound condition is satisfied, then the operations are performed; if the line ends with the symbolic name of an instruction (a "go-to"), control goes to that instruction, otherwise to the next line. If the compound condition is not satisfied, no operations are performed and control goes to the next line. The list of operations may be missing, in which case the delimiter THEN is also omitted but a go-to is obligatory.

TABLE I. Mnemonic Notation Used in Text and in Table II for Describing Elementary Tests and Operations—
with permissible ranges of arguments for current 7094 L^6

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, ···, ZZZZZZ.
a	"affected field", i.e., designation of a field whose contents are affected by an operation: a bug A, B ···, Z, or remote field, A0, A1, ···, ZZZZZZ.
Names	
f	name of a definable *f*ield: 0, 1, ···, 9, A, B, ···, Z
s	a program *s*ymbol (i.e., name of a program location)
Literals	
o	an octal number specified directly: 0, 1, ···, 777777777777
d	a decimal number specified directly: 0, 1, ···, 34359738267 ($2^{35}-1$).
h	a *H*ollerith literal: 0, 1, ···, ZZZZZZ. Permissible characters are the ten digits, 26 letters and period. Other characters must be specified in terms of their octal equivalences.
Alternatives	
cd	"contents or *d*ecimal"—i.e., either *c* or *d* as explained above.
co	"contents or *o*ctal"—i.e., either *c* or *o* as explained above.

A final instruction type is the unconditional THEN which is similar in format except that it contains no list of tests and no delimiter THEN. It also may have a null list of operations, in which case the go-to is obligatory. In addition to the five names given thus far, IF is a synonym for IFALL and NOT is synonymous with IFNONE: these are suggested where there is just one elementary test, to improve legibility of the program.

4. Elementary Tests

An elementary test in L^6 consists of three arguments, separated by commas, and enclosed in a pair of paren-

TABLE II. Resume of Tests and Operations of L^6
Lower-case mnemonics are explained in Table I.

	I. *TESTS*		
Equality	*Inequality*	*Greater Than*	*Less Than*
(*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*)
Pointers to Same Block		*One-Bits of*	*Zero-Bits of*
(c_1, P, c_2)		(*c*, O, *co*)	(*c*, Z, *co*)
		(*c*, OD, *d*)	(*c*, ZD, *d*)
		(*c*, OH, *h*)	(*c*, ZH, *h*)
	II. *OPERATIONS*		
	SETUP STORAGE, GET AND FREE BLOCKS		
Setup Storage	*Define Field*	*Get Block*	*Free Block*
(s_1, SS, *d*, s_2)	(cd_1, Df, cd_2, cd_3)	(*a*, GT, *cd*)	(*a*, FR, 0)
size	word	(*a*, GT, *cd*, a_2)	(*a*, FR, *c*)
first word	first bit		
last word	last bit		
	COPY BLOCKS AND FIELDS		
Copy Field	*Duplicate Block*	*Interchange Field Contents*	*Point to Same Block as*
(*a*, E, *cd*)	(*a*, DP, *c*)	(a_1, IC, a_2)	(*a*, P, *c*)
(*a*, EO, *o*)			(*a*, Δ) = abbrev. for (*a*, P, *a*Δ)
(*a*, EH, *h*)			
	ARITHMETIC OPERATIONS		
Add	*Subtract*	*Multiply*	*Divide*
(*a*, A, *cd*)	(*a*, S, *cd*)	(*a*, M, *cd*)	(*a*, V, *cd*)
(*a*, AO, *o*)	(*a*, SO, *o*)	(*a*, MO, *o*)	(*a*, VO, *o*)
(*a*, AH, *h*)	(*a*, SH, *h*)	(*a*, MH, *h*)	(*a*, VH, *h*)
	LOGICAL OPERATIONS		
Logical Or	*Logical And*	*Exclusive Or*	*Complement*
(*a*, O, *co*)	(*a*, N, *co*)	(*a*, X, *co*)	(*a*, C, *co*)
(*a*, OD, *d*)	(*a*, ND, *d*)	(*a*, XD, *d*)	(*a*, CD, *d*)
(*a*, OH, *h*)	(*a*, NH, *h*)	(*a*, XH, *h*)	(*a*, CH, *h*)
	SHIFTS AND BIT COUNTS		
Shift Left	*Shift Right*	*Locate Bits*	*Count Bits*
(*a*, L, *cd*)	(*a*, R, *cd*)	(*a*, LO, *c*)	(*a*, OS, *c*)
(*a*, L, *cd*, *co*)	(*a*, R, *cd*, *co*)	(*a*, LZ, *c*)	(*a*, ZS, *c*)
(*a*, LD, *cd*, *d*)	(*a*, RD, *cd*, *d*)	(*a*, RO, *c*)	
(*a*, LH, *cd*, *h*)	(*a*, RH, *cd*, *h*)	(*a*, RZ, *c*)	
	INPUT/OUTPUT AND CONVERSION		
Input	*Print*	*Convert*	*Microfilm*
(*a*, IN, *cd*)	(*cd*, PR, *co*)	(*a*, BZ, *c*)	($cd_{x\min}$, XR, $cd_{x\max}$)
	(*cd*, PRH, *h*)	(*a*, ZB, *c*)	
Print List		(*a*, BD, *c*)	($cd_{y\min}$, YR, $cd_{y\max}$)
(*c*, PL, *f*)	*Punch*	(*a*, BO, *c*)	(cd_{x_0}, TV^{H}, cd_{y_0}, *co*, *cd*)
(*c*, PL, *f*, *cd*)	(*cd*, PU, *co*)	(*a*, DB, *c*)	(cd_{x_0}, $\mathrm{TV}^{\mathrm{H}}\mathrm{H}$, cd_{y_0}, *h*, *cd*)
	(*cd*, PUH, *h*)	(*a*, OB, *c*)	(DO, ADVANC)
	PUSHDOWN AND POP-UP OPERATIONS		
Save and Restore Field Contents	*Save and Restore Field Definition*	*Do Subroutine*	*Go-To's for Exiting from Subroutine*
		(DO, *s*)	
		(s_2, DO, *s*)	
(S, FC, *c*)	(S, FD, *f*)	(DO, STATE)	DONE
(R, FC, *c*)	(R, FD, *f*)	(DO, DUMP)	FAIL

theses. The first argument always specifies the field on whose contents the test is being performed; the third argument is a literal or the specification of another field, whereas the second argument indicates what sort of comparison is to be made between these two quantities.

The exact form of the second argument also serves to indicate how the third argument is to be interpreted. If an O, D or H has been appended to it, then the third argument is taken as an octal, decimal or Hollerith literal; otherwise the "normal" interpretation is effective, in which numbers are taken as decimal in numeric tests but as octal for logical tests, and other third arguments are taken as field designators.

For the purpose of tests, all quantities, including field contents, are taken as non-negative integers (the largest integer that a n-bit field can hold being $2^n - 1$).

4.1 Numerical Tests. Four different numerical tests are used to determine whether the quantity specified by the first argument is Equal to, Not equal to, Greater than, or Less than the quantity specified by the third argument:

(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)

Here c stands for the specific field whose contents are being tested, cd for contents of another field or a *d*ecimal literal, and o and h for *o*ctal and *H*ollerith literals, respectively. Examples of these tests are:

(W, E, Y)	Is contents of W equal to contents of Y?
(WA, EH, Y)	Is contents of WA equal to Hollerith "Y"?
(ZAC, G, 53)	Is contents of ZAC greater than $(53)_{10}$?
(ZB, LO, 77)	Is contents of ZB less than $(77)_8$?

Whether the Pointer in c_1 points to the same block as the pointer in c_2 may be tested by

$$(c_1, \mathrm{P}, c_2).$$

At present (c_1, E, c_2) accomplishes the same test, but in future implementations of L^6, numerical equality of pointers may not always mean that they point to the same block, nor vice versa.

4.2 Logical Tests. There are two bitwise logical tests—one used to determine whether the first argument has One-bits in all positions where the third argument has ones, and the other to test whether the first argument has all of the Zero-bits of the third argument:

(c, O, co)	(c, Z, co)
(c, OD, d)	(c, ZD, d)
(c, OH, h)	(c, ZH, h)

If fields of different lengths are being compared, the test proceeds by aligning right ends of the fields and adding zero-bits to the left of the shorter field until the two are of equal length. Thus, for example, the "number" in a null field is said to contain all of the zero bits of any bit pattern whatever. For the purposes of logical tests and also for operations, literals and the contents of base fields (bugs) and read-only fields are considered to be 36-bit quantities.

5. Elementary Operations

Elementary operations of L^6 are coded by writing usually three but sometimes 2, 4 or 5 arguments which, as in tests, are separated by commas and enclosed in a pair of parentheses.

The first group of operations to be discussed are those for the manipulation of data structures and for copying data into bugs and fields. Subsequently discussed are arithmetic, logical, input/output operations, and finally operations that use pushdown lists which are part of the system.

In operations, the general rule for entering the results of a computation into a field is that the quantity is right-adjusted and any part of it which falls off the left end of the field is lost.

The most common form of elementary operation is

$$(a, op, cdo)$$

where a stands for the bug or field *a*ffected by the operation, op is the operation code, and cdo stands for a quantity involved in the operation: field *c*ontents or a *d*ecimal or *o*ctal literal. Integer literals, unless otherwise specified, are interpreted as decimal, except in logical operations, where they are taken as octal. As with tests, the exceptions are noted by appending a D for "decimal," an O for "octal" or an H for "Hollerith."

5.1 Operations for Structuring and Scanning Data

5.1.1 *Free Storage Set-up.* The Setting up of free Storage must be the first operation of an L^6 program. This is accomplished by the operation

$$(s_1, \mathrm{SS}, d, s_2)$$

where d is the maximum size of block which the program will require (1, 2, 4, 8, 16, 32, 64 or 128) and s_1 and s_2 are symbolic addresses of the beginning and end of the storage region to be used for list structures.

5.1.2 *Definition of Fields.* The defining of fields is usually the next thing to be done by a program. A field is Defined by

$$(cd_1, \mathrm{D}f, cd_2, cd_3)$$

where f is the name of the field being defined, cd_1 is the word of the block in which the field falls $(0, 1, \cdots, 127)$. cd_2 is the leftmost bit $(0, 1, \cdots, 35)$ and cd_3 the rightmost bit. For example, the fields A through G as illustrated in Section 2 could have been defined by

```
THEN   (0, DA, 18, 35) (1, DB, 21, 35) (1, DC, 6, 20) (0, DD, 0, 17)
THEN   (1, DE, 0, 5) (2, DF, W, 35) (3, DG, W, 35)
```

where the last two fields, just for the sake of example, have been defined to start in the bit position specified by the contents of bug W (initially zero). Examples of field-handling subroutines which these operations cause to be compiled are shown in Figure 1.

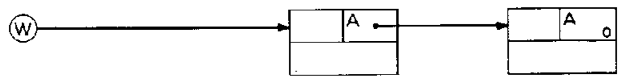
5.1.3 *Getting Blocks from Free Storage.* The program may GeT a block from the storage allocator by an operation of the form

$$(a, \mathrm{GT}, cd) \quad \text{or} \quad (a, \mathrm{GT}, cd, a_2)$$

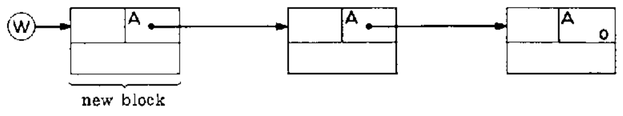
where a is the field which is to point to the new block and cd specifies the size of the desired block (directly if a decimal integer, indirectly if contents of a field.) If the indicated size is not an integral power of 2, the next larger power of 2 is understood. A fourth argument a_2, if present, indicates the field in which the original contents of a is then planted. Thus the get–block operation

$$(\mathrm{W}, \mathrm{GT}, 2, \mathrm{WA})$$

can act as a pushdown operation by changing the structure:



to:



A new block, as delivered by the allocator, is completely cleared to zero bits.

5.1.4 *Freeing of Blocks.* Blocks must be explicitly FReed (returned to the allocator); it is the programmer's responsibility to free a block when it is no longer needed. The coding is

$$(a, \mathrm{FR}, c) \quad \text{or} \quad (a, \mathrm{FR}, 0)$$

where a is the field pointing to the block and c, if not zero, specifies the contents, before the operation, which replace the contents of a after the operation. Thus the "pop-up" operation which would be precisely the inverse of the get–block operation illustrated in 5.1.3 would be (W, FR, WA).

5.1.5 *Duplication of Blocks.* A block may be DuPlicated by

$$(a, \mathrm{DP}, c)$$

where a after the operation contains a pointer to a new block which is a duplicate in size and contents of the block that c points to.

5.1.6 *Field-copying and Pointer-copying Operations.* The contents of two fields, a_1 and a_2, may be InterChanged (exchanged) by

$$(a_1, \mathrm{IC}, a_2).$$

```
A.TRAC   CAL  0,7            {Enter with IR7 pointing to a block.
         PAC  0,7            {Exit with IR7 pointing where field-A
         TRA  1,4            {of that block points.
           .
A.GET    CAL  0,7            {Enter with IR7 pointing to a block.
         ANA  A.ONES         {Exit with contents of field-A of that
         TRA  1,4            {block flush right in AC.
           .
A.STOR   STA  0,7            {Enter with IR7 pointing to a block.
         STT  0,7            {Exit with as much of AC as will fit
         TRA  1,4            {stored in field-A of that block.
           .
           .                 {Room for largest possible A.STOR
           .                 {routine.
A.ONES   OCT  000000777777
A.ZROS   OCT  777777000000
A.LGR    LGR  18             {Executed by XEC for right-shift and
A.LGL    LGL  18             {left-shift operations

C.TRAC   CAL  1,7            Note: Field C is neither flush
         ALS  3              left or flush right in a machine
         PDC  0,7            word, nor does it correspond to
         TRA  1,4            part(s) of the word which per-
C.GET    CAL  1,7            tain to special hardware-imple-
         ANA  C.ONES         mented instructions, as does
         ARS  15             field A. Consequently, it illus-
         TRA  1,4            trates the general forms of field-
C.STOR   ALS  15             handling subroutines.
         ANA  C.ONES
         ORS  1,7
         ORA  C.ZROS
         ANS  1,7
         TRA  1,4
C.ONES   OCT  007777700000
C.ZROS   OCT  770000077777
C.LGR    LGR  15
C.LGL    LGL  15
```

Fig. 1. Examples of field-handling subroutines compiled by current 7094 L[6]. These are the subroutines for fields A and C as defined by the operations (O, DA, 18, 35) and (1, DC, 6, 20) cited in the text. Calls to subroutines like these are illustrated in Figure 2.

```
LAC   A.BUG,7      (IR7 POINTS WHERE A POINTS)
TSX   B.TRAC,4     (IR7 POINTS WHERE AB POINTS)
TSX   C.TRAC,4     (IR7 POINTS WHERE ABC POINTS)
TSX   D.GET,4      (ACCUMULATOR CONTAINS ABCD)
XCL                (ABCD STORED IN MQ)
LAC   F.BUG,7      (IR7 POINTS WHERE F POINTS)
TSX   G.TRAC,4     (IR7 POINTS WHERE FG POINTS)
XCL                (ABCD TO ACCUMULATOR)
TSX   H.STOR,4     (ABCD STORED IN FGH)
```

Fig. 2. 7094 code compiled for the operation (FGH, E, ABCD). TSX calls are to subroutines such as those illustrated in Fig. 1.

The operation

$$(a, \text{E}, cd)$$
$$(a, \text{EO}, o)$$
$$(a, \text{EH}, h)$$

may be used to set the contents of a single field, a, Equal to the *c*ontents of another field or equal to a *d*ecimal, *o*ctal, or *H*ollerith literal.

To illustrate further how the L^6 system works, Figure 2 shows the 7094 code compiled for the field-copying operation (FGH, E, ABCD).

To make field a Point to the same block that field c points to, the coding is

$$(a, \text{P}, c).$$

At present (a, E, c) would accomplish the same thing, but in future implementations of L^6, as previously noted, pointers to the same block may not always be numerically equal. Therefore programmers are encouraged to observe the semantic distinction between pointers pointing to the same block vs. numerical equality of field contents.

An abbreviation may be used for a special case of the P–operation, in which a pointer is caused to move down a list structure by some such operation as (WA, P, WABC) where the first argument is identical with the initial part of the third. Here the special 2–argument form (WA,BC) could be used, or in general coding of the form

$$(a, \Delta), \quad \text{meaning } (a, \text{P}, a\Delta)$$

subject to the limitation that a must not be DO a special code for 2-argument operations involving subroutines, described later.

5.2 Arithmetic Operations. The contents of a field may be altered by the arithmetic operations of Adding, Subtracting, Multiplying by, or diViding by a quantity specified as the contents of a field, or as a *d*ecimal, *o*ctal or *H*ollerith literal:

Add	Subtract	Multiply	diVide
(a, A, cd)	(a, S, cd)	(a, M, cd)	(a, V, cd)
(a, AO, o)	(a, SO, o)	(a, MO, o)	(a, VO, o)
(a, AH, h)	(a, SH, h)	(a, MH, h)	(a, VH, h)

5.3 Logical Operations

5.3.1 *Logical Or, And, Exclusive Or, and Complement.* The bit-by-bit logical functions provided include those for Oring, aNding, and eXclusive oring of a specified quantity onto the contents of the affected field, and for replacing by the Complement of a quantity:

logical Or	logical aNd	eXclusive or	Complement
(a, O, co)	(a, N, co)	(a, X, co)	(a, C, co)
(a, OD, d)	(a, ND, d)	(a, XD, d)	(a, CD, d)
(a, OH, h)	(a, NH, h)	(a, XH, h)	(a, CH, h)

5.3.2 *Bit-counting Operations.* The operations

$$(a, \text{OS}, c) \quad \text{and} \quad (a, \text{ZS}, c)$$

may be used to set the contents of the affected field a equal to the number of One–bitS or Zero–bitS in the contents of field c.

5.3.3 *Bit-Locating Operations.* A field a may be set to the position of the Rightmost One–bit or Zero–bit or the position of the Leftmost One–bit or Zero–bit of another field c by

$$(a, \text{RO}, c) \qquad (a, \text{LO}, c)$$
$$(a, \text{RZ}, c) \qquad (a, \text{LZ}, c)$$

where the position count is from the right or left end of the field, respectively. (Thus a field containing the binary number 1111011100 has its rightmost 1 in position 3, its rightmost 0 in position 1, its leftmost 1 in position 1, and its leftmost 0 in position 5.) A zero answer results if the field contains no bits of the sort specified.

5.3.4 *Left- and Right-Shift Operations.* A quantity may be Left-shifted into the right end of field a by a shift of cd bit positions by

$$(a, \text{L}, cd, co)$$
$$(a, \text{LD}, cd, d)$$
$$(a, \text{LH}, cd, h)$$
$$(a, \text{L}, cd)$$

where the shift is performed as follows. A copy of the quantity co, d or h followed by an indefinite number of zeros is positioned just right of the affected field, and then the entire string of bits, including original field a contents, are moved left cd bit positions; the bits then falling within the field are the new field contents and material shifted off the left end of the field is lost. If no fourth argument is given, the quantity shifted in is simply a string of zero bits.

Similarly a quantity may be Right-shifted into a bug or field by the corresponding coding,

$$(a, \text{R}, cd, co)$$
$$(a, \text{RD}, cd, d)$$
$$(a, \text{RH}, cd, h)$$
$$(a, \text{R}, cd)$$

where the operation proceeds as before except that the quantity shifted in is first positioned to the *left* of the affected field (both fields taken at actual size), *preceded* by an indefinite string of zero–bits.

Note for both left- and right-shift operations, that if the quantity shifted in comes from another field, that field is unaffected by the operation.

5.4 Input/Output and Conversion. Input and output operations are necessarily the most machine-dependent features of L^6. The following description of these operations pertains specifically to the present 7094 implementation, in which alphanumeric characters are binary-encoded into 6–bit quantities. Accordingly, these operations apply to fields whose widths are multiples of 6–bits, and the counts specified for them are numbers of *characters* – not bits –to be input or output.

5.4.1 *Input.* The input mechanism delivers 72 characters per card, followed by the special character $(77)_8$ meaning "end-of-card," followed by 72 characters from the next card, etc. The one and only INput instruction is

$$(a, \mathrm{IN}, cd)$$

where a is the field into which cd characters are left-shifted, as in the logical left-shift operation (5.3.4). The shift is unconditionally stopped, however, at the point where a $(77)_8$ has just entered the right end of a. Thus (a, IN, 73) will always position the input mechanism so that a subsequent input operation will begin with the first column of the next card.

5.4.2 *Printed and Punched Output.* Output operations do not affect any field contents. The formats of output operations therefore follow a different convention, with the first argument cd specifying not an affected field but the number of characters to be PRinted or PUnched:

$$\begin{array}{lcl} (cd, \mathrm{PR}, co) & & (cd, \mathrm{PU}, co) \\ (cd, \mathrm{PRH}, h) & \text{or} & (cd, \mathrm{PUH}, h) \end{array}$$

If the directly or indirectly specified number of characters cd is less than the field co holds, the characters for output are taken from the *right* end of the field. If cd is more than the field holds, then the required number of leading blanks are output first. The special octal-coded character $(77)_8$ is recognized as a signal to terminate the present print line or punched card and begin a new one.

5.4.3 *Microfilm Output.* Microfilm facilities permit the drawing of points, line segments and typed text anywhere within a rectangular "viewing window" which is mapped onto the total plotting area of the microfilm recorder. The X-Range and Y-Range of the window are set by

$$(cd_{x\min}, \mathrm{XR}, cd_{x\max}) \quad \text{and} \quad (cd_{y\min}, \mathrm{YR}, cd_{y\max}).$$

To Draw a Line from (x_0, y_0) to (x_1, y_1), the operation

$$(cd_{x_0}, \mathrm{DL}, cd_{y_0}, cd_{x_1}, cd_{y_1})$$
$$(cd_{x_0}, \mathrm{DL}, cd_{y_0})$$

is used, where a rudimentary line, i.e. a point, is drawn if x_1 and y_1 are not given.

A string of alphanumeric characters specified by co or h may be Typed Horizontally or Typed Vertically, starting at (x_0, y_0), by

$$(cd_{x_0}, \mathrm{TH}, cd_{y_0}, co, cd_{\max})$$
$$(cd_{x_0}, \mathrm{THH}, cd_{y_0}, h, cd_{\max})$$
$$(cd_{x_0}, \mathrm{TV}, cd_{y_0}, co, cd_{\max})$$
$$(cd_{x_0}, \mathrm{TVH}, cd_{y_0}, h, cd_{\max})$$

where typing continues until $cd_{\max}$ characters have been typed, or until encountering the character $(77)_8$, whichever comes first. As with printed and punched output, if $cd_{\max}$ is less than the literal or field provides, the characters are taken from the right end of the field. If, however, $cd_{\max}$ is larger, then the typing starts with the specified field and continues through the remainder of the machine word and then through successive machine words.

The film is *advanced* to the next frame simply by calling the subroutine ADVANC in the manner described in 5.5.3:

$$(\mathrm{DO}, \mathrm{ADVANC})$$

5.4.4 *Conversion for Input and Output.* One operation used for input conversion is

$$(a, \mathrm{BZ}, c)$$

which sets the contents of a equal to the contents of c, with leading Hollerith Blanks $(60)_8$ converted to Zeros $(00)_8$.

A subsequent conversion which will accept a sequence of Hollerith-coded Decimal digits and convert them to Binary form is:

$$(a, \mathrm{DB}, c)$$

Similarly a series of Hollerith-coded Octal digits is converted to Binary form by the operation:

$$(a, \mathrm{OB}, c)$$

The converse operations, in preparation for output, are the "Binary-to-(Hollerith-coded) Decimal" operation

$$(a, \mathrm{BD}, c)$$

and the "Binary-to-(Hollerith-coded)Octal" operation:

$$(a, \mathrm{BO}, c)$$

A final output conversion which may be used to change leading Zeros $(00)_8$ to Blanks $(60)_8$ is:

$$(a, \mathrm{ZB}, c)$$

If the field from which the number for conversion comes is less than 36 bits wide, then for the leading-blanks-to-zeros operation BZ, the contents before conversion are assumed to be a 36-bit quantity with leading blanks added, whereas for all other conversions, the contents before conversion are assumed to be 36-bit quantities with leading *zeros* added.

5.4.5 *Printing of Lists.* For diagnostic purposes, the entire contents of a list of blocks may be printed in octal if these blocks are linked by pointers in the same field of successive blocks. The operation

$$(c, \mathrm{PL}, f) \quad \text{or} \quad (c, \mathrm{PL}, f, cd)$$

will Print the List, where c is a field pointing to the first block and f is the single-character name of the linking field for finding successive blocks. The fourth argument, cd, if present, is the maximum number of blocks to be printed, otherwise the output continues down the list until encountering a zero in field f.

5.5 Pushdown and Pop-up Operation. The L^6 system contains three pushdown lists which operate on the customary last-in-first-out principle, one for saving the *contents* of fields; another for saving the *definitions* of fields; and the third for recording *subroutine returns*, i.e., program locations to which control returns after completion of subroutines.

5.5.1 *Field Contents Pushdown.* The operation

$$(\mathrm{S}, \mathrm{FC}, c)$$

will Save a copy of the Field Contents c on the field-contents pushdown; the field itself is undisturbed by the operation. The inverse operation,

$$(\mathrm{R}, \mathrm{FC}, c)$$

will Restore Field Contents c from the field contents pushdown, i.e., the latest entry on this pushdown which has not as yet been removed is now removed and used to replace the contents of field c.

5.5.2 *Field Definition Pushdown.* The operation

$$(\mathrm{S}, \mathrm{FD}, f)$$

will Save on the Field-Definition pushdown the triplet of numbers which currently define field f: word in the block, leftmost and rightmost bits. The current definition of the field remains effective.

A field may be redefined either by the field-defining operation as described in Section 5.1.2, or by "Restoring" a triplet previously saved on the Field-Definition pushdown. The operation

$$(\mathrm{R}, \mathrm{FD}, f)$$

causes field f to be *r*estored by removing from the *f*ield-*d*efinition pushdown the latest triplet not yet removed and using these three numbers to define the field f.

As examples of the manipulation of field definitions, field B may be changed to coincide with field C by the sequence (S, FD, C) (R, FD, B), whereas the definitions of fields F and G can be exchanged by the sequence (S, FD, F) (S, FD, G) (R, FD, F) (R, FD, G).

5.5.3 *Subroutines.* A subroutine may be executed by the "elementary" operation

(DO, *subentry*) or (*failexit*, DO, *subentry*)

where *subentry* is the name of an entry point of a subroutine and *failexit,* if given, is the name of the program location to which control should go if the subroutine is exited by the special go-to FAIL.

The more common means of exit from a subroutine is the special go-to DONE, which returns control to that point in the program just beyond the DO operation from which the subroutine was entered. And since these locations are recorded on a pushdown list, a subroutine may execute itself, as is illustrated in the programming examples in Section 6.

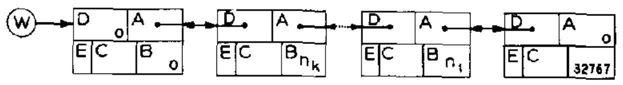
5.5.4 *System Subroutines STATE and DUMP.* Two subroutines for diagnostic purposes are provided as part of the system. The operation (DO, STATE) causes a printout of the state of the system, including the location of the (DO, STATE) call, time from beginning of run, depths of system pushdowns, locations of subroutine calls to this depth, amount of free storage, contents of bugs and contents of blocks which bugs point to.

A more extensive printout, caused by (DO, DUMP) includes all of the above information and in addition a listing in octal of the contents of all blocks currently in use.

6. Programming Examples

To illustrate L^6 coding, four programming examples are given below, each in the form of a closed subroutine. For these subroutines, fields are assumed to have been defined as illustrated in Section 2.3.

6.1 Inputting a List of Numbers. Let us assume that a series of numbers are to be read from successive 6-column fields of a single card, where the list is terminated by the first field containing only blanks. All numbers are assumed to be larger than 0 and smaller than 32767. The job of the subroutine is to compose a doubly-linked list of 2–blocks with these numbers in binary form in field B, and with the list beginning with dummy entry 0 and ending with dummy entry 32767:



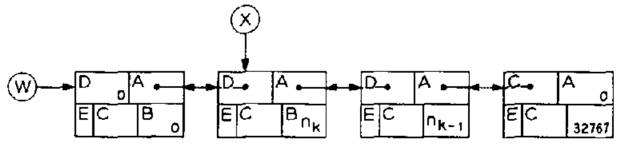
The entire subroutine can be coded in five lines as follows:

```
INP  THEN  (W, GT, 2) (WB, E, 32767) (S, FC, X)
RD   THEN  (X, IN, 6) (X, BZ, X)
     THEN  (W, GT, 2, WA) (WAD, P, W) (WB, DB, X)
     NOT   (X, E, 0) RD
     THEN  (X, IN, 73) (R, FC, X) DONE
```

The first line gets a 2–block and puts 32767 in its B-field, and then saves the contents of bug X so that the bug may be used temporarily by the subroutine but restored to its original condition just before exit; thus the subroutine is transparent to bug X. The second line reads a 6-column field into bug X, and converts leading blanks to zeros. In line 3 another 2–block is pushed down on the list, the previous first entry is backward-linked to the new 2–block and a decimal-to-binary conversion of the input number is inserted into field B. In line 4, if the last number read is not zero (the result of 6 blanks), control is directed back to the second line in order to read the next 6-column field. If and when a zero results from the read operation, control falls to line 5, where an attempt is made to read 73 columns. (This readin will stop short at the end-of-card symbol, thus positioning the input mechanism at the beginning of the next card.) Finally, bug X is restored to its state on entry, and exit from the subroutine is via the special DONE go-to, which returns control to the point from which the subroutine was called, an "elementary" operation of the form:

(DO, INP)

6.2 Ordering and Removing Duplicates. A subroutine for ordering the numbers and removing duplicates from the list produced in 6.1 can be devised by temporarily using a bug, say bug X, to scan the list, comparing numbers in adjacent blocks:



The subroutine, with entry point ORDER, may be coded as follows:

```
ORDER THEN (S, FC, X) (X, P, WA)
ND     IF     (XA, E, 0) THEN (R, FC, X) DONE
BACK   IF  (XB,E,XDB)THEN(XDA,P,XA)(XAD,P,XD)(X,FR,XA)ND
       IF     (XB, L, XDB) THEN (XB, IC, XDB) (X, D) BACK
       THEN (X, A) ND
```

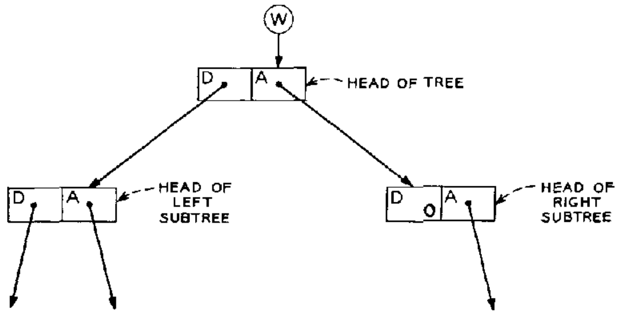
The first line saves contents of bug X and places bug X on the second block of the list that W points to. The second line tests whether X is at the end of the list and, if so, it restores original contents of bug X and exits. The third line tests whether this number, XB, is equal to the previous number, XDB. If so, the block is removed from the list by making its predecessor and successor in the list point to each other; then the block is freed and bug X moved to what was the successor before the freeing operation. The fourth lines tests whether the number XB is less than the preceding number; if so these numbers are interchanged and bug X moves left, following this number for as many interchanges as are required either until it is found to be a duplicate and eliminated, or until it fits into proper order. If and when X is sitting on a number which is greater than its predecessor (line 5), it moves right to test the next number, if any.

6.3 Outputting of Results of 6.2. The ordered list of numbers which results from 6.2 can be output, and the list returned to free storage, by the subroutine:

```
OUTPUT  THEN (W, FR, WA) (S, FC, X)
ANYMOR IF     (WA, E, 0) THEN (W, FR, 0) (1, PR, 77) (R, FC, X) DONE
        THEN (X,BD,WB)(X,ZB,X)(6,PR,X)(W,FR,WA)ANYMOR
```

The first line discards the initial dummy entry and saves contents of bug X. In the second line, if only the final dummy entry is present, it is freed, the special end-of-line character $(77)_8$ is output, bug X restored as on entry, and the subroutine is exited. Otherwise (line 3) bug X is filled with a binary-to-decimal conversion of the number, leading zeros are changed to blanks, all 6 characters are output, the block is freed, and control goes back to the test for the end of the list.

6.4 Returning a Tree to Free Storage. As an example of a subroutine which uses itself recursively, consider one which returns to free storage the entire binary tree pointed to by bug W and connected as indicated:



where a zero "pointer" at any level means no subtree below this point on this side.

The entire subroutine is the following:

```
RTRE  IF      (W, E, 0) DONE
      THEN  (S, FC, WA) (W, FR, WD) (DO, RTRE) (R, FC, W) RTRE
```

On initial entry into the subroutine, W will not contain zero but will point to a tree; hence control will fall to the second line. The second line simply saves a pointer to the right subtree, frees the head block and jumps W to the head of the left subtree, performs the RTRE subroutine on this subtree, refills W with the saved pointer to the right subtree, and now, with the left subtree gone, continues as if this were the head of the tree to be returned. If and when bug W is filled with a zero "pointer," the subroutine is exited from that level by virtue of the test in the first line.

REFERENCES

1. Newell, Allen, Ed. *Information Processing Language-V Manual.* Prentice Hall, Englewood Cliffs, N.J., 1961.
2. McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.
3. COMIT programmers reference manual. Research Lab. of Electronics, MIT, Cambridge, Mass., Nov. 1961.
4. Farber, D. J., Griswold, R. E., and Polonsky, I. P. SNOBOL: a string manipulation language. *J. ACM 11* (Jan. 1964), 21–30.
5. Knowlton, K. C. "L^6: Bell Telephone Laboratories Low-Level Linked List Language," 16min. film, and "L^6: Part II. An Example of L^6 Programming," 30min film. Both are 16 mm, black and white, with sound; available on loan from Technical Information Libraries, Bell Telephone Laboratories, Inc., Murray Hill, N.J.
6. ——. A fast storage allocator. *Comm. ACM 8,* (Oct. 1965), 623–625.
7. PL/I: Language Specifications. Form C28-6571-2, IBM Corp., 1271 Avenue of the Americas, New York, N.Y., Jan. 1966.
8. 7094 Bell Telephone Laboratories programmer's manual. Bell Telephone Laboratories, Inc., Murray Hill, N.J.
9. GE-635 programming reference manual. General Electric Computer Dept., Phoenix, Ariz., 1964.