USERS' GUIDE TO LITTLE No. 19

May 1973

Mike Brenner

I. INTRODUCTION

LITTLE is a computer language which is well-suited for systems programs, especially large programs which must run on more than one computer. Often a major problem with inventing complex new programs for modern computers is the extreme difficulty of running the program on a computer other than the one that they were originally debugged on. "Extremely difficult" means approaching the point where it would be less work to write the program all over again starting with the flow chart than it would be to modify it to work on the new computer.

Two of the major reasons for such difficulty in transporting programs across computer boundaries are the many nagging tiny differences in the languages and, of course, the different word sizes in computers.

To solve the problem of word size and other hardware incompatibilities, all code which is machine dependent should be isolated at the beginning of the program, in the first block of code, called the "machine block." LITTLE can do this easily with its excellent macro pre-processor, global quantities, and its user-defined variable lengths.

To solve the problem of slight differences in language creeping in between compilers written for allegedly identical

languages on different machines. LITTLE will bootstrap itself onto each different machine.

Bootstrapping traditionally is the art of leaving a quick-sand trap into which one has fallen, by grasping one's boot-straps and lifting upon them until one has pulled himself out. In more modern terms, the crane on top of the World Trade Center which has "pulled" the building up from the ground simulates what LITTLE is, namely, a program programming itself. This comes about because the LITTLE compiler is written in the LITTLE language itself. The object code generators can be inserted for the particular computer it is running on. Then the compiler can be compiled by an already existing LITTEL compiler, such as the one at NYU.

The current version of LITTLE (working at NYU as the lower level language which supports the SETL system) was invented by Professor Jack Schwartz in 1968 and originally specified in Cocke and Schwartz, "Programming Languages and Their Compilers", NYU 1970. It has developed over the years along with the experiences of the SETL group and this Users' Manual describes the current implementation. For acomplete discussion of the bootstrapping process and the implementation of LITTLE, see the System Programmers' Reference Manual for LITTLE, which will be available shortly.

II. LITTLE SOURCE DECK

Statements

LITTLE statements may appear on cards on on files which contain card images. Statements are written free-field in columns 1 through 72. Column 73 on may be used for identification information, such as that generated by an automatic UPDATE program. Blanks are ignored between tokens, but may not appear within tokens (except when used as characters within character strings). Statements are separated by semicolons (11-7-8 Punch). All statements may be continued on as many lines as you wish. There is no header statement, but the final statement must be FIN; or an end-of-file mark.

Comments

There are two forms of comments that may be used in a LITTLE program. PL/I type comments enclosed between /* and */ may appear anywhere that blanks may appear.

```
ABC = 3 /* comment */;
ABC = /* comment */ 3;
```

but not `AB /* comment */ C = 3;`

In addition, if a token begins with a $, the rest of that card is treated as a comment. Comments are lexically removed before any other manipulation of the source text is attempted, including macro expansion or macro definition recognition.

Labels

Statements may be labelled. Labels are enclosed in slashes and consist of not more than 200 aophanumeric characters beginning with an alphabetic character.

```
/Z1234/  A = B * C + 1;
```

III. LITTLE EXPRESSIONS

Constants

The only data type recognized by LITTLE is the bit string. All constants are bit strings, declarable in binary form, octal form, integer form, character form, or logical form. These forms are interchangeable and have exactly the same internal representation. The forms are different in the source text as a user convenience.

The binary form is a string of 1's and 0's followed by an L.

101001L 10000L 101L

The octal form is a string of digits from 0 to 7 followed by a B.

54B 1027B

The combined binary-octal form is a binary number followed by a B followed by an octal number. It would be unusual to have more than 2 binary bits preceding the B.

1B3 is 1011L iB77 is 1 111 111 L

The integer form is a positive decimal integer. LITTLE does not handle negative integers because that would introduce machine dependencies. At the present time, the result of integer operators may not exceed 2**60-1.

0 423 1989 4234567

The character form is enclosed in quotes (4-8 punch).

'ABC' 'al.-$/******+=//////**/+***' '/*'

The logical constants are .T. and .T., representing the values TRUE and FALSE. 'FALSE', or .F., is represented by all zero's; anything else is taken to be 'TRUE', .T..

Remember these are all forms of bit strings, they are not different data types. They all have the same internal representation and are treated alike by LITTLE operators.

Variables

Variable names are alphanumeric characters beginning with an alphabetic character. Variable names of arbitrary length are permitted. Each variable used in the LITTLE program must be declared by specifying its length in bits. This is done with a SIZE statement.

```
SIZE X(60),FLAG(1),LONG(106);
```

The length of a variable does not depend on the word size of the computer and can be from 1 to 511 bits long. However, some operations do not function across word-boundaries. For this reason, LITTLE allows one-dimensional arrays. These are declared using the DIMS statement, which must follow the statement SIZE-ing the array variable.

```
SIZE B(60);
DIMS B(34);
```

This declares an array called B, consisting of 34 variables each of length 60 bits.

Subscripting

Variables declared in a DIMS statement are subscripted in the normal way;

```
DIMS B(5);
A = B(3);
```

However, unlike FORTRAN,the subscript may be any valid LITTLE expression.

Name Scoping

Variables retain their size until re-SIZE-ed later in the program. In particular, they retain their size across subroutine boundaries. Thus, all variables are global from the point of their definition onwards. When a variable is re-SIZE-ed, the old declaration is lost and the new one is the only one known by the compiler. Note that thsi unwieldy name-scoping scheme is to be replaced shortly

Use of variables not appearing in a SIZE statement result in an informative diagnostic. In this event LITTLE proceeds SIZE-ing the variable to the word size of the computer.

The bits in LITTLE words are always counting that bit 1 is the right-most bit, bit 2 is the second from the right, etc.

Data Statements

Data statements, which store values at load time rather than execution time, have the form of a list of variables followed by equal signs and their values, saparated by slashes.

```
DATA X=10/Y=34B/Z='JIM'/A=1011L;
```

In the case of arrays, the whole array may be set in the data statement:

```
SIZE ZJ(2);DIMS AJ(3);
DATA AJ = 1L,10L,11L;
```

This sets AJ(1) to 1, AJ(2) to 2, and AJ(3) to 3.
A mechanism for specification of initial values of arrays, all of which are identical, will be introduced shortly.

Expressions

LITTLE expressions are composed of variable and function names, and constants, combined by operators. LITTLE operators, with the exception of the four arithmetic operators +,-,*,/, are all composed of a 1-,2-, or 3-letter alphabetic mnemonic, preceded and followed by a period.

If the right side of the assignment has a SIZE, R, greater than the SIZE of the left side, L, then the right side is truncated by only storing bits 1 to L into the address specified by the left side. If the left side has the larger SIZE then bits R+1 to L will be set to zero.

EXAMPLES:

```
SIZE A(3),B(6);
A = 101011L; /* now A is 011L */
B = 110L; /* now B is 000110L */
```

Monadic Operators (written as a prefix to an expression)

These three, and all operators in LITTLE, work on variables of any size from 1 bit up to the word size of the computer, without any change except in the variables' SIZE statements.

.NOT. Boolean minus. Results in bit by bit inversion of the operand.

.NB. Number-of-bits operator. Counts the number of non-zero bits in the operand.

.FB. First-Bit operator. Gives the position of the first hgih-order non-zero bit, that is, the first non-zero bit from the left.

EXAMPLE:

```
SIZE X(4),y(4);
DATA X=11;          /* 11 is 1011L */
Y = .NOT. X;        /* Y is now 0100L */
Y = .FB. X;         /* Y is now 4, because the fourth bit
                       from the left is the first non-zero bit */
Y = .NB. X;         /* Y is now 3, since there are 3 non-zero
                       bits */
```

Dyadic Operators

Dyadic operators are written in infix form, between the two operands.

.EQ.	equal (Result is .T.(=1L) or .F.(=0L) for the 6 comparison operators
.NE.	not equal
.GT.	greater than
.LT.	less than
.GE.	greater than or equal to
.LE.	less than or equal to
.OR.	Boolean 'OR' of two bit-strings of arbitrary length
.AND.	Boolean 'AND', e.g., a masking operation
.EX.	Boolean 'EXCLUSIVE OR' between bit-strings
.C.	concatenation of two bit-strings (not limited to word-size units)

EXAMPLES;

```
SIZE X(4),Y(4),Z(8);
DATA X = 1010L/y=1100L;
Z = X.OR.Y;          /* Z is 00001110L */
Z = X.AND.Y;         /* Z is 00001000L */
Z = X.EX.Y;          */ Z is 00000110L */
Z = X.C.Y;           /* Z is 10101100L */
Z = X.LE.Y;          */ X is interpreted as integer 10
                        Y is interpreted as integer 12
                        Z is .T.
                        Z is 11111111  */
```

Triadic Operator

.T. The field extractor is the triadic operator written in prefix form:

.T. EXP,CONSTANT,EXP2

EXP determines the first position of the extracted subfield of EXP2

CONSTANT is the length of the subfield

Remember that in LITTLE you count bits such that the right-most bit is number one

.F.2,3,A Extracts the 3 bits starting from the 2nd bit from the right of A.

.F.I,5,B(K) Extracts the 5 bits starting at bit I of the Kth element of array B.

In a field extractor on the right side of an assignment statement then EXP and EXP2 can be any LITTLE expression. If the field extractor appears on the left-hand side the EXP2 may be only a (possibly subscripted) variable name which may be preceded by field extractors.

```
SIZE A(5);
A = .F.3,1,1011B; /* A is now 00000L */
A = .F.1,2,1011B; /* A is now 00011L */
A = .F.4,2,1011B; /* A is now 00001L */
A = .F.160,5,1011B; /* A is now 00000L */
.F.2,3,A=5; /* stores 5 (i.e., 101L) in the 2,3,4 bits of A.
               A is now 01010L */
```

Precedence

As in other languages, precedence rules are provided to reduce the number of parentheses needed to specify a complicated expression. The following table shows the unusual precedence groups of LITTLE operators, arranged in order of increasing precedence (i.e., * is of greater precedence than +).

TABLE OF LITTLE OPERATORS IN ORDER OF INCREASING PRECEDENCE

1. The integer comparison operators

   .EQ. .NE. .GT. .LT. .GE. .LE.

   These operators have a non-standard precedence lower than .AND. and .OR., because unlike PL/I and FORTRAN, it is not expected that integers will be the most commonly used data forms. Therefore while IF(A.LE.B .AND. C.LE.D) is a convenient FORTRAN expression meaning IF((A.LE.B) .AND. (C.LE.D)), it is expected that in LITTLE the AND operator will be used much more frequently as a masking operation than as the logical AND of true-false expressions, which is its most common use in FORTRAN.

2. integer addition and subtraction + and -

3. Integer multiplication and division * and /

4. Boolean addition and concatenation of strings .OR. .C.

5. Boolean inversion, bit count, first-bit .NOT.,.NE.,.FB.

6. Boolean multiplication, exclusive or .AND.,.EX.
7. function and array references and field extractors

Executable statements

Having constructed the form of LITTLE expressions, here are some examples of LITTLE statements. First, there are four types of assignment statements. EXP1, EXP2, and EXP3 represent any valid LITTLE expressions. Address can be any variable name.

```
SIMPLE ASSIGNMENT          ADDRESS = EXP;
INDEXED ASSIGNMENT         ADDRESS(EXP) = EXP2;
PARTWORD ASSIGNMENT        .F. EXP,CONSTANT,ADDRESS = EXP2;
INDEXED PARTWORD           .F. EXP,CONSTANT,ADDRESS(EXP2) = EXP3;
```

Other Executable Instructions

```
CONTINUE;                  this is a no-op in LITTLE
GO TO label;               unconditional transfer to a label
GOBY exp(label-1,label-2,label-3,...,label-n);
                           computed GO TO  just like FORTRAN,
                              except that exp can be any expression
IF(expr) GO TO label;      conditional transfer on non-zero (true)
CALL name (expl,...,expn);    subroutine call (with aruments)
RETURN;                    FORTRAN-type non-recursive return
```

There are no recursive routines in LITTLE. The statements just described are the only executable statements in the LITTLE

language except for the INPUT/OUTPUT instructions described in another section.

In the near future some user convenient statement types may be added to the LITTLE compiler. These would include IF..., THEN..., ELSE, DO, and WHILE statements.

IV. PROGRAM STRUCTURE

LITTLE prgorams are divided into subroutines and functions, each of which may use SIZE and DIMS and DATA statements to declare new variables. There is no "main program" in LITTLE. One subroutine must be named "START" and this is where the system initially transfers control. Each SUBR OR FNCT block is ended with an END statement. If the END statement is executed before a RETURN statement is encountered, the routine will not return, but the whole LITTLE program will be aborted.

EXAMPLE:

```
SUBR ADD(X,Y,Z);
SIZE X(60,Y(60),Z(60);
Z = X + Y;
RETURN;
END;
```

With this subroutine defined, another routine could call it:

```
CALL ADD(X,3,H);
```

This would add 3 to the value of X and store it in H.

```
FNCT ADD(X,Y);
SIZE X(70),Y(60),ADD(60);
ADD = X + Y;
RETURN: end;
```

With this function definition, another routine could call it:

```
SIZE ADD(60);  H = ADD(X,3);
```

This would have exactly the same effect as the subroutine call above. Notice that the function must be SIZED both in the function definition routine and in every routine that calls it. However, since the SIZE is global, it is actually only necessary to SIZE it in the defining FNCT statement and the first routine that calls it, assuming that no other routines SIZE a variable with the same name as the function. Only the fifst seven characters of LITTLE function and subroutine names are used by LOADER, though LITTLE itself can handle names up to 200 characters.

V. THE MACRO PRE-PROCESSOR

The lexcial scanner is capable of some rather sophisticated macro activity. Macro-definitions and calls may appear anywhere a blank may appear in a source program. The simplest type of macro is a simple substitution. As an example, let us say that you wish to SIZE a large number of variables to the word size of the computer at many random places, but you want the word size of the computer to change from one machine to another. In the macro section at the beginning of the LITTLE program, define a macro:

```
+ * WS = 60 **
```

Now in a SIZE statement, you may write:

```
SIZE X(WS);
```

When the lexical scanner comes to the WS in the SIZE statement, it looks it up in the hash-coded table of macros and expands WS to 60. It then turns only the constant 60 over to the parser, which never sees the token 'WS'.

More complicated macros can be devised which will substitute for parameters.

```
+ *NAME(A1,A2,A3,A4,A5,...,An) = BODY **
```

Macros may have from 0 to 15 parameters. The BODY is decomposed

into tokens. Each occurrence of an argument Aj is detected and flagged when the macro definition is stored in the internal hash table. The Aj's must be simple variables. To invoke a macro, it is called with the sequence:

```
NAME(ARGUM1,ARGUM2,ARGUM3,ARGUM4,...,ARGUMn)
```

The number of arguments in the calling sequence and the defining sequence must match, although null arguments in the calling sequence are permitted. That is, two commas immediately following each other indicate the argument between them is the null string.

The arguments in the calling sequence may be any valid LITTLE expressions which are balanced with respect to parentheses and which have no occurrences of commas or semicolons except enclosed within parentheses. If a semicolon appears, it is flagged with an error message, and the macro is not expanded, however this check can be turned off (see the section on LITTLE-SCOPE interface).

EXAMPLE:

```
+ *NAME(X,Y,Z) = X - Y * Z **        macro definition
NAME(X/Y,F(G),'BBB'.C.'CCC')         macro invocation
```

Macro invocations are expanded by substituting ARGUMj for Aj for each argument occurring in the macro definition, and issuing the resulting stream of tokens instead of the stream initially input. If this transformed stream is then found to

contain a macro invocation, this inner invocation is expanded, and so on recursively. Thus macro bodies may contain calls to other macros, though never macro definitions. Any error in a macro definition or call such as non-matching numbers of arguments will generate an informative diagnostic and the macro will not be expanded. If a macro is not closed with '**' before another is opened with '+*', a closure is assumed. This can save a lot of other macros from being passed over in the case of a single mistake.

In addition to the normal variable names (labels) and integers, you may use, macros are capable of generating their own variable names (labels) and counters. If one of the special names ZZZA, ZZZB,...,ZZZ_Z appears in a macro body, the following action takes place. One of 26 counters in a common block in the compiler is incremented, but only the first time the particular name occurs in the given macro expansion. If the current value of the ZZA counter, for instance, is 65, then each occurrence within the macro body of ZZZA is replaced by an occurrence of ZZA ØØØ65.

Similarly the special names ZZYA,ZZYB,...,ZZYZ will be replaced by 5 digit integers.

EXAMPLES:

```
      +*LIMIT(A)= IF(A.LT.ALIM) GO TO ZZZB;
      CALL EXIT;
/ZZZB/                                **
```

This could lead to the actual expansion:

```
        IF(XYZ.LT.ALIM) GO TO ZZZB12345;
        CALL EXIT;
    /ZZZB12345/.....
```

This macro may be used several times in a subroutine without generating duplicate labels. Another example:

```
        +*A = ZZZYZ**
```

This will make what appears to be a variable in the source deck to be a constant which will be 1 the first time the ZZZYZ counter is called, 2 the next time, etc. Thus

```
        I = A; J = A
```

could expand into:

```
        I = 00005; J = 00006;
```

These macro-generated tokens are particularly useful when used as subscripts to long lists of simple array assignments.

Macros may be redefined, and a non-fatal message will be issued. The old definition is lost. To drop a macro, redefine it without an equal sign.

```
        +*NAME**
```

This removes it from macro status. If you want a macro to expand to the null string, i.e., to be ignored by the lexical scanner, define it like this:

```
+*NAME = **
```

If it is defined to be the null string, then

TEXT1 NAME TEXT2 will expand to TEXT1 TEXT2.

To get around the restriction that a macro definition may not appear within a macro, the macro-expanded output is permitted to pass through the macro-definition detector. This permits an indirect definition to appear, although a direct definition issues a fatal diagnostic. We prefer a roundabout method to the direct method which might be made available, since allowing such combinations as

```
+*A = "*B = ...
```

acn lead to errors.

The roundabout method is as follows. First define a lexical concatenator.

```
+*Q3(W1,W2,W3) = W1 W2 W3**
```

Then, to include a macro definition with a macro body, we may then write some such construction as

```
+*DEFINE(WD,TEXT) = Q3(+,*WD TEXT*,*)**
```

The power of this kind of structure is indicated by the language extensions to elementary LITTLE that can be lexically added through this type of construction. We can build WHILE and DO loops, and implement a macro-generated IF...THEN...ELSE... ENDIF statement which all translate into the IF(EXP) GO TO label; primitive.

To implement a macro-generated non-nestable DO loop of the form

```
DO(J,1,N);
TEXT
ENDO;
```

all we have to do is define two macros:

```
+*DO(J,A,B) = J = a;
/ZZZA/   If (J.GT.(B)) GO TO ZZZB
Q3(+,*AXXX = ZZZA*,*)
Q3(+,*BZZZ = ZZZB*,*)
Q3(+,*CZZZ = J*,*)**
```

and also

```
+*ENDO = CZZZ = CZZZ + 1; GO TO AZZZ;
/BZZZ/ CONTINUE**
```

VI. INPUT/OUTPUT FACILITIES

There are 6 I/O verbs in the LITTLE language (not counting the additional verbs available through PUP, the Print Utility Package which may be loaded with your LITTLE program to increase the I/O power of LITTLE)

READ N,LIST;	read BCD from tape unit N
WRITE N,LIST;	write BCD
READB N,LIST;	read binary
WRITEB N,LIST;	write binary
REWIND N;	rewind tape N
ENDFILE N;	put and end-of-file mark on tape unit N

In the above, N must be an integer signifying the logical tape number of a tape unit that the operating system recognizes as existing at the time your program is being executed. The lists are lists of arbitrary length composed of arbitrary LITTLE expressions separated by commas. Input and output are both assumed to be in 8A10 format in the READ and WRITE cases.

EXAMPLES:

```
READ1,X;
WRITE2,'Xis',X;
```

VII. LITTLE-SCOPE INTERFACE AT NYU

Here is the complete deck set-up to run a LITTLE program which adds one and one.

```
ID,CM150000,T10. NAME          JOB CARD
LOADER                         USE THE MACE LOADER
ATTACH(LEX,LTLLEXF)            GET LITTLE FRONT-END
LEX.                           EXECUTE THE FRONT-END
RFL,150000                     NEED SPACE FOR COMPILER
ATTACH(LITTLE,LITTLE)          GET THE COMPILER
LITTLE                         COMPILE
COLLECT,LGT,LTLL1BF            SATISFY EXTERNAL REFERENCES
RFL(50000)                     REDUCE STORAGE
LGO                            EXECUTE
END-OF-RECORD                  GREEN CARD
SIZE C(12);
C = 1 + 1;
WRITE 2,'1+1' = ',C;           THE PROGRAM
FIN;
END-OF-FILE                    RED CARD
```

Note that the RFL(50000) is used because the loader will otherwise RFL up to the job card CM before reducing to the minimum necessary for execution of LGO. Very large LITTLE program will require more field length.

A closer examination of the 'LEX' card is in order. The default parameters for LEX are:

```
LEX(INPUT,OUTPUT,LISTING,BINFILE)(XA=Ø,XE=Ø,CC=$,SL=Ø1,MASC=Ø,
            SLO=Ø,TF=1,NLO=1,ABT=0)
```

INPUT	SOURCE FILE
OUTPUT	LIST AND STATISTICS WRITTEN ON THIS FILE
LISTING	MACRO-EXPANDED SOURCE TEXT WRITTEN HERE
BINFILE	INPUT TO THE PARSER

Thus if a very long program is in UPDATE format, and only a lexical scan is desired, the following control cards would be used:

```
ID,CM120000,T50,NAME
LOADER
ATTACH(OLDPL,YOURLITTLEPROGRAMFILE)
UPDATE
ATTACH(LEX,LTLLEXF)
LEX(COMPILER)(TF=0)
END-OF-RECORD
UPDATE CONTROL CARDS
END-OF-FILE
```

The secondary parameters on the LEX card may be altered as follows. Setting something to 1 is the same as setting ti to 'ON' or 'TRUE'. Zero is equivalent to either 'OFF' or 'FALSE'.

XA 1	LITTLE will produce a full global cross-reference map of all names used in the program.
XE 1	LITTLE will produce a local cross-reference map for each routine that is for each 'SUBR' and 'FNCT'

MASC 1 This enables the macro-argument test which gives a-error if a semi-colon appears as a macro agrument not enclosed in parentheses. If the test is off and right-parenthesis terminating a macro-argument list is omitted, the remaining program text may be seen as an argument to the macro. This will cause the message 'OVERFLOW -ASTL-'. The test will catch the error at the end of the next statement ending in a semi-colon.

TF 0 This will turn off the token-file generation (used as input to parser via the file called BINFILE, the fourth argument of the program LEX). This should be turned off when only scanning, with no following parse, to increase the scanning rate.

ABT 1 Abort on lexical errors to avoid parsing erroneous text. This is only used in debugging macros. During normal debugging it is advantageous never to use ABT on or TF off, but rather always to get a complete parse immediately, to catch all the errors at once.

SLO 1 SETLISTC OVERRIDE: ignore SETLISTC cards throughout the program, using only the initial value of the SL parameter, described next.

SL xy Initial setting of the SETLISTC flag 'xy' represents a two-digit octal constant, that is, 6 binary bits. The value of the SETLISTC flag may be changed during the scanning of the input text by the appearance of the symbol SETLISTCxy anywhere in the input stream. The symbol SETLISTCxy itself will never be listed, but will only have the immediate effect of changing the value of the flag. In describing the meaning of the flag, remember that LITTLE always counts the rightmost bit as bit number 1.

if bit 1 is 1 then all cards read are listed
if bit 2 is 1 then macro-generated text is also listed
if bit 3 is 1 then macros are listed with macro generated text
if bit 4 is 1 then macro text (as listed) is also to be punched (72 columns)
if bit 5 is 1 then save the old value of SETLISTC before giving it the new value
if bit 6 if 1 then restore the old value of SETLISTC, saved by a previous setting of bit 5 to 1. Only one value can be saved at a time, the most recent save.

EXAMPLES:

SL 03 causes source text and macro-generated text to be printed in an interspersed manner. (SETLISTC03)

SL 15 causes macro-generated text to be punched without the macros showing and can be used for deck editing. (SETLISTC15)

If bit 1 is off, then the only listing that will occur is that when an error occurs, the previous ten tokens are printed. However, the line-identifying information in columns 73 to 80 is not printed out with it, so in large programs the card number, say 4589, does not help too much in locating the error. This problem is unwittingly solved by the fact that almost all errors detected by the lexical scanner also result in parse errors and will again be detected in the back end of the compiler

Note that macro expanded listing caused by bits 2 and 3 of SL are written to file LISTING, not output. To get them you must add the following cards:

```
REWIND(LISTING)
COPYBF(LISTING)
```

Finally,

CC = % This would change the end-of-line character(which indicates that the rest of the line is a comment)

IN SUMMARY:

ABT	abort on lexical errors
CC	comment character
MASC	macro-argument semi-colon test
SL	set initial value of SETLISTC
SLO	override occurrences of SETLISTCxy in the text, using only SL value

TF token-file write flag

XA global cross-reference map

XE local cross-reference map

A closer examination of the 'LITTLE' card follows. The default parameters on the 'LITTLE' card are:

```
LITTLE(BINFILE,OUTPUT,LITINIT)(A=1,SL=1,ABT=0,MAP=0)
```

BINFILE scanned output table from 'LEX' - the input to the parser

OUTPUT list file

LININIT This is the LITTLE "main program" provided by the System. It calls the user supplied subroutine START to begin program execution. If the user wishes to modify this main program, he should add these four cards between the JOB-card and the "LOADER." card:

```
ATTACH(OLDPL,LTLSYSLIBPL)
UPDATE(Q,8)
RUN(S,,,COMPILE,OUTPUT,LITINIT)
RETURN(COMPILE,OLDPL)
```

In addition the second record of his deck, that is, the record following the first end-of-record card, will contain any update directives he wishes to change the file. The following card would be included.

```
*COMPILE LITINIT
```

A=0	Do not produce binary object code on file LGO
A=1	Produce LGO file
SL=0	Do not list original source program
SL=1	List source cards
ABT=N	Abort to *exit card if parser detects at least N errors
ABT=0	Do not abort no matter how many parse errors
MAP=1	list storage map

EXAMPLE: To abort if any parse errors:

```
LITTLE. (ABT=1)
```

Finally, let us examine the "LGO" card. By default it is

```
LGO(INPUT,OUTPUT,TAPE3,TAPE4,TAPE5,TAPE6,TAPE7)
```

where TAPE1 is equivalenced to INTPUT and
TAPE2 is equivalenced to OUTPUT.

Thus, the system environment set up for LITTLE at NYU has five tape units, a printer, and a card reader (all simulated by disk files) available to the LITTLE user community. The computer also supports the SETL group, NYU students, and other federally sponsored projects. In order to prevent conflicts we always try to speed up the over-all turnaround time for all users. This is why the RFL(50000) card appears. Another little embellishment to speed up the operating system by decreasing the size of the file-name table is to put in this card after the "LEX" card:

```
RETURN(OLDPL,COMPILE,LEX)
```

and this one after the 'COLLECT' card:

```
RETURN(LITTLE,BINFILE,LITINIT,LTLLIBF)
```

IF the whole job runs in 100 seconds or less, then it is not worth putting in these return cards.