C A R N E G I E - M E L L O N U N I V E R S I T Y
D E P A R T M E N T O F C O M P U T E R S C I E N C E

L C C

R E F E R E N C E M A N U A L

Harold R. Van Zoeren

November 10, 1969

---- Introduction ----

LCC is a language for conversational computing which operates within the TSS monitor system on the IBM 360/67 computer at Carnegie-Mellon University. In its fundamental design, LCC began as an amalgamation of (1) the basic elements and statements of the algorithmic language ALGOL 60 and (2) the input-output, control, editing, and filing statements of the conversational language JOSS, but extensive modifications have been made to exploit as fully as possible the dynamic nature of conversational computing. The resulting language, with its underlying processing system, gives you, the LCC user, a very high degree of power and flexibility.

The working sentences of the LCC language are statements, a statement (abbreviated s) being a command which causes LCC to perform an action (e.g., a modification of data, an input/output operation, a modification of control). You may type an arbitrary number of statements, separated from one another by semicolons (;), on a single line. Such a statement list is called a step, and LCC will execute the statements within it from left to right.

Steps in LCC may be used in two different ways, either delayed or immediate. Delayed steps are translated and saved by LCC, and they may later be recalled and executed under programmer control. A delayed step is distinguished by the presence of a preceding decimal step number which indicates its relationship to other steps. A step number must lie between 0001.0001 and 9999.9999, and it is separated from the step text by either a colon (:) or a comma (,). Both its integer portion, from which leading zeros may be omitted, and its fractional portion, from which trailing zeros may be omitted, must lie between 0001 and 9999. A step number serves both as the editing designator for a step and as a control designator for the first statement in the step. In addition to the step number, any statement in a delayed step may have one or more labels associated with it as control designators, a label being an identifier which immediately precedes the statement and is separated from it by a colon (:). If a step has multiple step numbers, all must precede its first statement or label, and only the rightmost number will be used; if a statement has multiple labels, each will be significant. Some examples of delayed steps are:

```
3.7,  GO TO 3.5;

3125.0042: A ← B+1; LB06: C ← D*E; LBL: F ← G/3

27.85:   27.830, L: M: TYPE Y,Z; RETURN T
```

Delayed steps will be ordered according to their numbers, and they may be inserted, modified, or deleted freely while conversing. They may be typed in any order, and a newly typed step

will replace any previously saved step with the same number. For execution purposes, steps are grouped into parts, with a part being the ordered set of all steps whose numbers have the same integer portion. When executed, a part will be treated as an ALGOL block, i.e., variables which are declared and labels which are used in it will have local meanings which are valid only when it is active (i.e., it is being executed). All such local meanings will be erased when execution of the part is completed.

An immediate step, which is distinquished by the absence of a step number, is translated and executed when typed and is then discarded. Immediate steps are used to perform one-time or "desk calculator" calculations, to control the execution of the delayed steps of a program, and to perform various editing and debugging operations. An explicit transfer of control (GOTO) to an immediate statement is not allowed, and consequently immediate statements cannot be labelled.

Syntactically, any LCC statement may be used in either an immediate or a delayed step. When executed, however, each statement will be checked for validity in the currently existing context, and at that time, some statements will be treated as no-ops (e.g., an immediate 'PAUSE', a delayed 'GO'), and some will lead to errors (e.g., a global 'GOTO', a global 'RETURN').

An LCC statement may be empty, in which case it contains no non-blank characters and it performs no action. The various non-empty LCC statements are listed alphabetically by their initial keywords or metavariables and described below. Following the statement descriptions are descriptions of the subsidiary metavariables (expressions, literal constants, etc.) used in the language. The abbreviated syntax notation which has been used is described in Appendix A, and the complete syntax for LCC is summarized in Appendix B.

---- LCC Statements ----

statement ::= (one of the following -- pp. 3-29)

ALTER group | : | e_1 → e_2 , e_3 → e_4 , ... , e_(2*N-1) → e_(2*N)
| , |

The expressions e_I should evaluate to character strings. LCC will search the text of the group for substrings which match the given pattern strings e_1, e_3, ... , e_(2*N-1). Each substring which matches an e_(2*J-1) will be replaced by the corresponding e_(2*J), and the group will be retranslated with its altered text.

LCC will perform the search as in the following pseudo-LCC code:

```
FOR (each step in the group (ordered on step numbers)) DO
  {  FOR I FROM 1 BY 2 TO 2*N-1 DO
     {  START_OF_SEARCH_POINTER ← 1;
        AGAIN: IF (search finds substring e_I) THEN
           {  (replace substring by e_(I+1));
              START_OF_SEARCH_POINTER ← (position of
                1st char after replaced substring);
              GO TO AGAIN } }
     IF (any replacements were made) THEN (retranslate) };
```

For the search LCC will treat both text and pattern strings as sequences of either contiguous letters and/or digits or individual non-blank, non-alphanumeric characters, with blanks being ignored except insofar as they separate alphanumeric sequences from one another. As an example, the step

```
4.8: X←IF PQR THEN (TEMP+1) ELSE '15.64 FF';GO TO 4.41;
```

will be found to contain the substrings (among others)

'X←', '(TEMP', '+', '15', 'FF', 'GO TO',
'41; ', and '.' (twice)

but it will not contain the substrings

'8', 'Q', '.6', 'GOTO', or 'TO4'

Examples:

```
ALTER STEP 1.6 : 'X' → 'AX' , 'Y' → 'BY'
ALTER PARTS, 'P + Q' → R
ALTER 4.77 , 'A' → ''
```

```
ARRAY  ⊢ ⊢ ident ⊣ .,.  [ ⊢ e < : e > ⊣  .| ][ |.  ] ⊣ .,.
                                          | , |
```

LCC will assign to each ident in a list the multidimensional array structure specified by the bounds list which immediately follows it. Each item in a bounds list gives the limits on one subscript of an array structure. The number of items is thus the number of dimensions of the array. An item in a bounds list can be either a pair of expressions specifying the lower and upper limits on the subscript for that dimension or a single expression specifying the upper limit on that subscript (the lower limit will be implicitly 1).

Storage will not be allocated for an array until the array is used, and even then it will only be allocated for a given row when an element from that row is first accessed.

Examples:

```
ARRAY LA[1:N, -3:8*K]
ARRAY JIM, JOE[10,15,20], DAVE[0:8][4][-6:-1]
```

```
BEGIN  ⊢ s ⊣ .;.  END
```

The keywords 'BEGIN' and 'END' delimit a "block", whose list of arbitrary LCC statements will be treated as if it were in a part, i.e., there may be local variables valid only within it. LCC will perform a block entry, after which it will execute the statements from the list in sequence. This "block statement" will normally be terminated by "running off its end". A RETURN statement within it will first terminate the context of the block statement and then return from the context in which the block was embedded.

Examples:

```
BEGIN  STEP 4.8;  PART 251;  S ← T  END
BEGIN  NEW A,B;  PART 6;  PART 8  END
```

```
CASE  e  OF { s_1 ; s_2 ; ... ; s_N }
```

The expression e will be evaluated and rounded to an integer J. If $1 \le J \le N$, LCC will give control to statement s_J, from which control will normally pass to the successor of the CASE statement. It is an error if J is out of the range 1 to N.

Examples:

```
CASE J+1 OF { X ← F(A,B) + C ;
              X ← SQRT(Y) + D ;
              GO TO 6.2 ;
              ;
              X ← SIN(Y↑2) ;
              GO TO 6.2 ;
              X ← 0 }
```

CASE e OF { s_1 ; s_2 ; ... ; s_N ; OTHERWISE s_(N+1) }

The expression e will be evaluated and rounded to an integer J. If $1 \leq J \leq N$, action is as in the simple CASE statement without an OTHERWISE. If J is out of the range 1 to N, control will be given to statement s_(N+1).

Examples:

```
CASE I OF { X←5; OTHERWISE X ← ⍵+5 }
```

COMBINE < STEPS > num_1 TO num_2 AS e

A single string will be constructed by concatenating, in step number order, the text portions of all steps with numbers between num_1 and num_2 inclusive. During this concatenation process, a semicolon (;) character will be appended to any step which does not already terminate with one. LCC will then retranslate the new string as step e. Steps num_1 to num_2 will not be deleted and will be unaffected by the COMBINE statement (unless $num\_1 \leq e \leq num\_2$). As in a group, it is an error if num_1 > num_2 (unless num_2 < 1).

Examples:

```
COMBINE STEPS 6.7 TO 6.83 AS 6.7
```

COPY group AS e

If e evaluates to an integer, the set of steps from the specified group will be copied and retranslated as a new group, with the integer portion of each step number being replaced by the value of e (which must not be zero). If e does not evaluate to an integer, this statement is equivalent to the statement

```
COPY group AS e BY .01
```

All steps in the original group must be in the same part. The source text for the group will not be modified by the COPY, and the original group will not be deleted.

Examples:

```
COPY PART 3 AS 43
COPY STEP 5.61 AS 12.074
```

COPY group AS e_1 BY e_2

LCC will copy, renumber, and retranslate the ordered set of steps from the specified group. The renumbering will start with e_1 (or, if e_1 is an integer, with (e_1 + e_2)) and successive step numbers will be incremented by e_2 (whose value must lie between .0001 and .9999). The original group of steps will not be deleted by a COPY statement (though it may be changed if some of the new steps fall within the group). The source text for a copied step will not be modified during the COPY, and it is your responsibility to make sure that the renumbered steps do not contain spurious references to steps in the original group. To insure this, you should use labels rather than step numbers to refer from one statement in the group to another.

Examples:

```
COPY STEPS 14.371 TO 14.4305 AS 814.001 BY .002
```

DELETE FILE e

The expression e must evaluate to a string, which will be used as a file name. LCC will delete that file from your file catalog, and it will take back any storage which that file used.

Examples:

```
DELETE FILE 'AB'
```

DELETE ALL

This statement is effectively equivalent to (but slightly slower than) the step

```
EXIT ALL; DELETE STEPS; DELETE VALUES
```

Your working storage will be completely erased, and LCC will be re-initialized, just as if you had logged off and then logged back on.

DELETE | PARTS |
       | STEPS |

LCC will erase from working storage both the source and object codes for all steps (only values will remain).

DELETE VALUES

LCC will erase from working storage the current incarnation-value for each of your identifiers, giving every identifier in your program the meaning "undefined".

DELETE < STEPS > num_1 TO num_2

LCC will erase from working storage all steps whose numbers lie between num_1 and num_2 inclusive. As in a group, it is an error if num_1 > num_2 (unless num_2 < 1).

Examples:

```
DELETE 151.42 TO 151.536
```

DELETE < STEP > num

Equivalent to

```
DELETE STEPS num TO num
```

Examples:

```
DELETE STEP 4.231
```

DELETE PARTS num_1 TO num_2

Equivalent to

```
DELETE STEPS  (num_1 + .0001) TO (num_2 + .9999)
```

```
DELETE PART num
```

Equivalent to

```
DELETE PARTS num TO num
```

```
DELETE | < STEPS > |  ⊢ num_1 < TO num_2 > ⊣ .,.
       | PARTS     |
```

A DELETE statement may include a group list. LCC will then delete all steps in each of the specified groups.

Examples:

```
DELETE PARTS 4, 7 TO 10, 153, 48
DELETE 3.71, 3.814, A4 TO (A4 + P - .5)
```

```
DELETE ⊢ varid ⊣ .,.
```

LCC will replace the current incarnation-value for each varid in the list by "undefined". If a varid referred to a string or array (or any other item for which storage was allocated), the internal links to that storage will be cut, but the storage will not be taken back until the block within which it was allocated has been terminated.

Note that an array element can be deleted. This feature will be necessary before you can change the meaning of an array element which is a procedure, a reference pointer, or an array name.

Examples:

```
DELETE A,B
DELETE C[I,J,4]
```

```
DISPLAY FILE < CATALOG >
```

LCC will type out a list of the names of all of your LCC files. The names will be the full TSS names of your files, which are qualified by your user number and the internal name LCCFILE. Thus the file 'SAVAL' of user XYZ1ZZ13 will have the full name

```
XYZ1ZZ13.LCCFILE.SAVAL
```

DISPLAY RETURN < STEPS >

LCC will type out a list of the currently active steps, thus giving a map of the present control status. Step designators will be typed one per line, and the list will be ordered so that the innermost (most recent) step will be typed first. For steps inside parts, LCC will type the step number; for immediate steps LCC will type the characters '***'; for a procedure call LCC will type the procedure name. Thus LCC might type the lines

```
 ***
17.3
FUNCT
***
4.3
***
```

in response to your 'DISPLAY RETURN' statement.

DISPLAY ALL

Equivalent to

DISPLAY PARTS; DISPLAY VALUES

DISPLAY | PARTS |
| STEPS |

Equivalent to

DISPLAY STEPS 1.0001 TO 9999.9999

DISPLAY VALUES

LCC will type, in alphabetical order, the names and current meanings of all of your defined identifiers (i.e., the meanings atop each of your variable stacks). Appropriate formats will be used for values (numeric, logic, and string) and references (label, array, procedure, and pointer). Each displayed line will also include a prefixed level number which indicates the level of the block in which that identifier was declared, i.e., the outermost block level in which the current meaning will hold. For global variables, the level number of zero will be suppressed. An example of the displayed output is:

```
2     ARRA          ARRAY  [1:5,3:10,-2:6]
3     LAB           IN  3.6
1     LV            ¢000000FF
3     NAM           ⊃ABC
      NV            -1.234567₩15
      PROC          PROCEDURE
2     SV            'ST'
```

DISPLAY < STEPS > num_1 TO num_2

LCC will type in order the source images for all steps whose numbers are between num_1 and num_2 inclusive. As in a group, it is an error if num_1 > num_2 (unless num_2 < 1). Each step will begin on a new line and will include both its number and its text. Except for possible minor differences in the format of the step number, a displayed step will look exactly as it did when you typed it in.

Examples:

```
DISPLAY 415.3 TO 415.7
```

DISPLAY < STEP > num

Equivalent to

```
DISPLAY STEPS num TO num
```

DISPLAY PARTS num_1 TO num_2

Equivalent to

```
DISPLAY STEPS (num_1 + .0001) TO (num_2 + .9999)
```

Examples:

```
DISPLAY PARTS 4 TO 6
```

DISPLAY PART num

Equivalent to

```
DISPLAY PARTS num TO num
```

```
DISPLAY  | < STEPS > |  ⊢ num_1 < TO num_2 > ⊣ .,.
         | PARTS     |
```

A DISPLAY statement may include a group list. LCC will then display all the specified steps or parts, ordering the groups for typing from left to right in the list.

Examples:

```
DISPLAY 3.4 TO 3.43, 3.8 TO 4.2, 4.513, 4.902
```

```
DISPLAY  ⊢ varid ⊣ .,.
```

LCC will type out the current meaning for each varid in the list. Each displayed "meaning" will take up a single line, and it will include exactly the same information that would be typed for that variable by a 'DISPLAY VALUES' statement. If no meaning has been assigned to a listed varid, the varid will be displayed as "undefined".

Examples:

```
DISPLAY A, C, P, X[1,1], X[4,7,3]
```

```
EXIT
```

An EXIT statement is used to delete the context of the part or step group which is currently active and give you control in the context of the part or step group which called it. A more precise description of an EXIT is as follows:

EXIT recognizes only contexts involving explicitly numbered steps and those involving the user (it regards you as the numbered step 0.0). An EXIT statement will delete all execution contexts down to and including that for the first non-zero numbered step on the context stack. It will then delete all contexts down to but not including the first numbered step. If that is a step 0.0, it gives you control; if not it adds a new step 0.0 context, which also gives you control. Thus an EXIT deletes all execution contexts down to, but not including, the first numbered step below the first non-zero numbered step, and it then gives you control.

EXIT ALL

LCC will perform successive EXITs until the global state is reached (i.e., there are no remaining group contexts) and it will then give control to you.

Examples:

```
IF ERROR6 THEN EXIT ALL
```

EXIT < TO > < PART > e

If part e is not currently active, LCC will type an error message and give control back to you. Otherwise LCC will perform an EXIT. If the resulting context is that of part e, control will be given to you. If not, LCC will perform another EXIT, etc.

Examples:

```
EXIT TO PART 3
EXIT 703
```

```
<| FOR ident <|FROM| e_1 >|> <|BY e_2 < TO e_3 >|> < WHILE e_4 > DO s
 |           |←   |      |   |TO e_3 < BY e_2 >|
 | FROM e_1                |
```

The statement s will be executed repeatedly as long as the expression e_4 is true and as long as the value of the controlled ident is within the specified range. With each repetition, the value of the explicit (ident) or implicit control variable will be modified as specified by the controlling for-clause. The phrase 'FROM e_1' may be omitted if e_1 = 1, 'BY e_2' may be omitted if e_2 = 1, 'TO e_3' may be omitted if the loop is to be terminated in some manner other than that of the controlled variable reaching a final value (i.e., if e_3 is infinite), and 'FOR ident' may be omitted if ident does not appear in e_4 or in s (in which case an implicit controlled variable will be used).

Operation of a complete iteration statement is equivalent to that of the LCC block

```
BEGIN NEW BYE ← e_2, TOE ← e_3;  ident ← e_1;
  L:  IF IF BYE ≥ 0 THEN ident ≤ TOE ELSE ident ≥ TOE
      THEN IF e_4 THEN  { s; ident ← ident + BYE;
                          GO TO L }  END
```

where the identifiers L, BYE, and TOE do not occur within any of the e_I or in s. Note that, unlike ALGOL 60, the increment and terminal expressions e_2 and e_3 are evaluated only once, when

execution of the iteration statement begins, and subsequent changes to any variables used in e_2 and e_3 will not affect the control of the iteration.

Examples:

```
FOR I FROM 1 BY G TO H+1 WHILE N ≠ 3 DO ST
WHILE B < C DO PART 2
TO T DO PART 345
DO PART 6543
FOR J ← K TO P BY -2 DO F(J,K)
```

GO

LCC will return control to the context from which you were called, resuming execution from the point of the call. A GO has meaning only after you have been called via a statement (PAUSE) or action (pressing the ATTN or BREAK key) which expects you to eventually return control to the caller.

```
| GO < TO > | e
| GOTO      |
```

If e is a label, it must be that of a statement in a currently active group. LCC will then EXIT to that group and transfer execution control to the labelled statement. If e is not a label, it must evaluate to a step number in a currently active group. If the step number is in the range of the group currently being executed, LCC will transfer control to the first statement in the designated step. If the number is not in range, LCC will EXIT from the current group context and repeat the above process.

Examples:

```
GO TO LABL3
GOTO 6.1
GO 1243.0001 + J
```

IF e THEN s

If the expression e evaluates as true, execution control is transferred to s (from which control will normally pass to the successor of the IF statement). If e evaluates as false, s is skipped. If e has a logic or arithmetic value, it will be considered as true if it is non-zero or as false if it is zero; strings will be converted to their equivalent arithmetic values.

Examples:

```
IF X < 4 THEN PAUSE
```

IF e THEN s_1 ELSE s_2

If e evaluates as true, execution control will be transferred to s_1, from which control will normally pass around s_2 to the successor of the IF statement. If e evaluates as false, execution control will pass around s_1 to s_2, from which control will pass to the successor of the IF statement.

Examples:

```
IF ~P ∨ Q THEN Z ← 5 ELSE ( T ← T + 1; TYPE T )
```

LINE < e >

LCC will upspace your paper (at your terminal) by one line or, if an expression e is supplied, by e lines.

Examples:

```
LINE
LINE 4-J
```

LOAD < FILE > e

LCC will open file e and, if the file was created by one or more SAVE statements, load into working storage whatever was SAVEd there. This is done by treating the information in the file as a set of lines of input text, each of which will be read and translated just as if it had been typed in by you.

LCC treats all files alike, regardless of whether they were created by SAVE or WRITE statements. Thus a file may contain immediate statements which were written (as strings) by a WRITE statement. These will be both translated and executed during a LOAD of that file. Any immediate statement may be written and LOADed, including another LOAD statement.

Examples:

```
LOAD FILE 'QQ13'
```

```
NEW ARRAY  ⊢⊢ ident ⊣ .,.  [ ⊢ e < : e > ⊣  .| ][ |.  ] ⊣ .,.
                                             | , |
```

This statement acts just like an ARRAY statement except LCC will construct a new incarnation-value for each ident before assigning it its specified array structure.

Examples:

```
NEW ARRAY A3, A4[10, 20, 5:30], A5[5]
```

```
NEW  ⊢ ident ⊣ .,.
```

This declaration statement causes a new incarnation-value (IV) with the meaning "undefined" to be constructed at the current nesting level for each ident in the list. In the usual case that the old IV is on a lower level, this new IV will be linked to the old one, which it will temporarily supersede. In case the old IV is on the current level (i.e., the ident is being redeclared in this block), it will be replaced by the new one.

A declaration holds only within the scope of the block in which it is executed. When that block is terminated, all IVs declared in it will be erased, and the meanings which the corresponding idents had before their declarations were executed will be restored.

Examples:

```
NEW A,B
```

```
NEW  ⊢ ident ← | e         | ⊣ .,.
              | pointer   |
              | procedure |
              | structure |
```

This statement acts much like a simple NEW statement, but instead of giving each newly constructed IV an undefined meaning, LCC will assign it a specified initial "value". Declarations and assignments will be made from left to right in the NEW list, but a "value" will be constructed before the ident to which it is to be assigned is declared. Thus, for example, in the statement

```
NEW A ← B + A
```

the old value of the variable A will be added to B to obtain the initial value of the new A.

Examples:

```
NEW S ← 'SS', T ← U - V, W ← ⊃X
NEW F ← ▽(A,B) PART 9  {NEW P ← ▽Q+R▽, S ← 6}▽
NEW A ← ARRAY[3,0:5], B, C ← 26, D ← ARRAY[X:Y]
```

NUMBER AS e_1 < BY e_2 >

LCC will automatically type out for you at the beginning of each input line a step number followed by a colon (:). Before translation, the supplied number will be appended as a prefix to whatever step text you type. The numbering sequence will normally start at e_1, with successive step numbers being incremented by e_2, but if any numbers in the sequence (including e_1) turn out to be integers, they will be skipped. Thus it is quite acceptable for the numbering to cross part boundaries. If e_2 is given, its value must lie between .0001 and .9999; if the 'BY' phrase is missing, e_2 will be assumed to be .01.

LCC's automatic numbering will continue until you turn it off; this is done by inputting an empty step, i.e., by pressing the RETURN key immediately after the step number.

Examples:

```
NUMBER AS 17.3 BY .002
```

NUMBER group AS e_1 BY e_2

LCC will renumber and retranslate the ordered set of steps from the specified group. The renumbering will start with e_1 (or, if e_1 is an integer, with (e_1 + e_2)) and successive step numbers will be incremented by e_2 (whose value must lie between .0001 and .9999). The original group of steps will be deleted by a NUMBER statement (otherwise this statement acts exactly the same as the corresponding COPY statement, which does not delete the group). The source text for a step will not be modified by a NUMBER statement, and it is your responsibility to make sure that the renumbered steps do not contain spurious references to steps in the original group. To insure this, you should use labels rather than step numbers to refer from one statement in the group to another.

Examples:

```
NUMBER STEPS 7.7 TO 8.2 AS 25 BY .02
```

NUMBER group AS e

If e evaluates to an integer, the set of steps from the specified group will be renumbered and retranslated as a new group, with the integer portion of each step number being replaced by the value of the expression e (which must not be zero). If e does not evaluate to an integer, this statement is equivalent to the statement

NUMBER group AS e BY .01

All steps in the original group must be in the same part. The source text for the group will not be modified by this NUMBER statement, but the original group will be deleted (otherwise this statement is identical to the statement 'COPY group AS e').

Examples:

```
NUMBER 8.07 AS 14.253
NUMBER STEPS 6.4 TO 6.5 AS 1016
```

NUMBER group BY e

Equivalent to

NUMBER group AS X BY e

where X is the truncated value of the first step number in the group. This statement is used mainly to tidy up the fractional step numbers for a part without changing its name (i.e., its part number).

Examples:

```
NUMBER PART 803 BY (INC * .0001)
```

NUMBER group

Equivalent to

NUMBER group BY .01

Examples:

```
NUMBER STEPS 43.001 TO 43.18
```

OFF

LCC will perform an 'EXIT ALL' and it will then log you off. A message will be written to indicate the elapsed time and the processor time used during your conversational session. Your automatic reload file will be erased by this OFF statement (see Appendix G).

Examples:

```
IF DONE THEN OFF
```

OFF SAVE

This statement acts just like a simple OFF statement except for its treatment of your automatic reload file, which will not be erased and thus may be reloaded when you begin your next conversational session (see Appendix G).

PART num

A new block context will be set up for the sequence of steps from num+.0001 to num+.9999. Execution will then begin within that context at the first step whose number is ≥ num+.0001 and it will normally continue through successively higher numbered steps. Control will be returned when the part "runs off its end" or when it executes a RETURN statement, an EXIT statement, or a GOTO which transfers out of its range.

A part may be called either as an operand in an expression (in which case it should return a result) or as a statement. In the latter case it should not return a result, but if it does, LCC will type the value of the result at your terminal.

Examples:

```
PART 5
TO PART 17 DO PART ABACAD
```

PART num { s_1 ; s_2 ; ... ; s_N }

A new block context will be set up for the sequence of steps in part num. Execution control will then be transferred to statement s_1, from which control will normally pass to s_2, s_3, ... in order up to s_N, from which control will pass to the lowest numbered step in part num. Thus the statement list within the

braces is treated as if it were a step numbered num+.00009 in a part context which is expanded to include that step.

Examples:

```
PART (J + 2) { NEW A←⊃B[C]; TYPE D + A; E←16 }
PART 3 { NEW A ← G-H ; NEW D ← ∇ R / PART 2 ∇ }
```

PAUSE

LCC will type a message giving the step number of the PAUSE statement, after which it will give control to you.

Examples:

```
IF X < 4 THEN PAUSE
```

PAUSE e

LCC will type out the string supplied by the expression e, after which it will give control to you.

Examples:

```
PAUSE 'HALF DONE'
```

PRINT < FILE > e

LCC will print (on the line printer in the computer room) the contents of file e, which must have been generated by an LCC SAVE or WRITE statement. File e will not be changed by being printed, but if you PRINT a file during a conversational session, you will not be allowed to delete it later on in that same session.

Examples:

```
PRINT FILE 'PRNTFIL'
```

RECOVER < e >

LCC will treat a RECOVER statement as a dummy statement unless you give it from a user state which was entered because of an error in a delayed step. In the latter case, your furnished expression e, which will only be acceptable if the operation which caused the error halt should have produced a result, will be used as that result, and LCC will resume execution from the point of

the error as if the operation had been completed. As an example, if your program halts with the error message

ERROR OR01 AT 4.1 DIVISION BY ZERO

you may resume execution by typing the statement

RECOVER $3_{10}20$

Execution will then continue just as if the divide operation had been completed normally and had yielded the result $3_{10}20$.

In some cases it is possible to resume execution after errors where no explicit result is involved. In those cases you may use a simple RECOVER statement which furnishes no result expression. As an example, if you attempt to call part 3 when part 3 is empty, LCC will halt execution of your program with an error message such as

ERROR PC02 AT 5.2 PART 3 DOES NOT EXIST, YOU CANNOT CALL IT

You could then resume execution by typing the lines

```
3.1: LOAD STUFF
RECOVER
```

Examples:

```
RECOVER X + Y
```

RETURN

LCC will delete the current execution context and return control to its caller, resuming execution from the point of the call.

RETURN e

This statement acts just like a simple RETURN statement except the value of e is computed before the RETURN is performed, and that value is the result of the call.

Examples:

```
RETURN X - Y * 3
```

RETURN pointer

This statement acts just like a simple RETURN statement except the specified reference pointer is constructed before the RETURN is performed and that pointer is the result of the call.

Examples:

```
RETURN ⊃ VBL[I+1]
```

RETURN procedure

This statement acts like a simple RETURN statement except a reference to the given procedure is constructed before the RETURN is performed and that reference is the result of the call. Thus if a procedure PR, which is called via the statement

```
RED ← PR(X,Y,Z)
```

returns with the statement

```
RETURN ▽ (A,H) PART 66 ▽
```

the effect (except for possible side effects) is to perform the assignment

```
RED ← ▽ (A,H) PART 66 ▽
```

Examples:

```
RETURN ▽ STEPS 4.8 TO AZ ▽
RETURN ▽ (B,C) { PART 7; PART 25 } ▽
```

SAVE save-object

LCC will put the save-object (a list of steps and/or values) into the currently open file. A step will be SAVEd in the same form that would be used to DISPLAY it, which is, except for possible minor differences in the format of the step number, the same form that you used to type it in. The current meaning of a variable will be SAVEd as an assignment statement which assigns that meaning to the variable. Thus a SAVEd file can be reloaded merely by executing it; this is done by means of a LOAD statement. Note that no context information will be kept with a SAVEd variable, and it will be up to you to recreate the proper context into which to later load the file. Only variables whose meanings are values (numeric, logic, or string), pointers, or arrays will be SAVEd. An array will be saved as a structure assignment followed by assignment statements for each of its SAVE-able

elements.

A SAVE statement does not save numeric values to their full precision (about 17 digits) but only to the precision of the printing routines (10 digits). Thus a SAVEd and reLOADed program may not function exactly the same as if it had been run in a single session. This will not usually be noticeable, but it will show up if numbers such as PI and EE (which are initially accurate to the last bit) are saved or if, for example, X = 1/3 is SAVEd. In the latter case we would normally get 3 * (1/3) to print as 1 (due to rounding in the output routines; 3 * (1/3) = 1 is FALSE, however), but after saving and reloading X we would get 3 * X to yield .9999999999.

Any number of SAVE statements can be executed to generate a given file; each will append its lines at the end of those already in the file.

Examples:

```
SAVE STEPS 35.6 TO 35.8
SAVE X, Y[I,1], Y[I,2], Z
```

SAVE save-object AS < FILE > e

Equivalent to

```
USE FILE e;  SAVE save-object
```

Examples:

```
SAVE PARTS 45 TO 493 AS FILE 'CAT'
```

STEPS num_1 TO num_2

This "step call" is an "execute" statement, which may be used to perform steps from some other portion of your program as if they had been copied in-line in its place. As in a group, num_1 must be ≤ num_2 (unless num_2 < 1). LCC will set up a new group context (non-block) for the sequence of steps from num_1 to num_2. Execution will then begin at step num_1, and it will continue through successively higher numbered steps. This step call will normally be terminated either by a RETURN statement without a value or by "running off the end" of the step group. An EXIT statement will terminate the step call and return control to you in the context of its calling group.

Examples:

```
STEPS 3.8 TO 3.93
```

STEP num

Equivalent to the statement

STEPS num TO num

TYPE ⊢ type-object ⊣ ,,.

For a type-object consisting of an expression e, LCC will type the value of e, left-justified on a line. A numeric value will be rounded to 10 significant decimal digits and typed as an integer or a decimal number, with an exponent part being appended if necessary. A logic value will be typed as TRUE or FALSE or as an 8-digit hexadecimal number (i.e., it will have the form of a logic-literal). A string value will be typed as is without surrounding quote marks.

LCC will ignore an empty type-object in this unformatted TYPE statement.

A for-clause in a type-object merely specifies control over another type-object, but the controlled objects will be typed just as if the for-clause were outside the TYPE statement instead of inside. As an example, the type-object

(for-clause e_2 , e_3)

will, under control of the for-clause, type values for e_2, e_3, e_2, e_3, ... , one per line.

Examples:

```
TYPE A + B, , C
TYPE P, (FOR I TO 19 DO I, CAB[I[J]])
```

USE < FILE > e

The expression e must evaluate to a string whose body will be used as a file name. LCC will open that file and use it in any subsequent SAVE or WRITE statement which does not mention a file explicitly. Only one such file can be open at any time, so file e will be closed either by a logoff or by executing any filing statement (including another USE) which explicitly mentions a different file.

A file name must be an identifier (ident) of length ≤ 8 which

does not contain any lower case letters or underline (_) characters.

Examples:

```
USE FILE 'QWIC'
```

WRITE ⊢ type-object ⊣ .,.

This statement is just like a TYPE statement except the type-objects will be written on the currently open file instead of at your terminal. Any number of WRITE statements can be executed to write a given file; each will append its lines at the end of those already written.

Examples:

```
WRITE A, B*C
WRITE (FOR I TO 10 DO (FOR J TO 10 DO FISH[I,J]))
```

WRITE ⊢ type-object ⊣ .,. AS < FILE > e

Equivalent to

USE FILE e; WRITE ⊢ type-object ⊣ .,.

? ⊢ < string-literal > varid ⊣ .,.

For each varid in the list, the following process will be performed: LCC will type either a standard identifying message or, if you preceded the varid by a string-literal, the string which you supplied. It will then give control to you. You must type the text for an expression and return control to LCC (by pressing the RETURN key). Your expression will then be evaluated and assigned to varid.

Examples:

```
?A,B
? 'TYPE RANK' RNK[3], RNK[4]
```

? $ ⊢ < string-literal > varid ⊣ .,.

This statement acts like the regular ? statement except LCC will treat each of your typed expressions as the body of a string

(i.e., it will surround each expression by quote marks). Thus the value assigned to each varid will always be a string.

A slight variation is possible here in the use of single-quote marks, which need not be doubled to appear in your requested string body. Thus if you type

```
AB'C`D
```

in response to the statement

```
?$ ST
```

the effect will be exactly the same as if you had executed the statement

```
ST ← `AB''C``D'
```

Examples:

```
?$ S, `T STRING ' T
```

```
{ ⊢ s ⊣ .;. }
```

This "compound statement" will be treated as a single control unit whose sub-statements will be executed sequentially from left to right. A compound statement is not a block and it may not have its own local variables; therefore its main use is within a controlling statement such as an IF, CASE, or iteration.

Examples:

```
IF ~P ∨ Q THEN Z ← 5 ELSE  { T ← T + 1; TYPE T }
```

```
! e
```

The expression e must evaluate to a string, whose contents will be treated as statement data to the LCC translator. When a ! statement is executed, the string which it supplies will be processed just as if it were a step which was just typed in. If the string turns out to be an immediate step, it will be executed as the current statement. If not, it will be stored as usual for a delayed step and control will pass to the successor of the ! statement. This statement is useful mainly in programs which generate new program text during execution.

Examples:

```
!  `A ← B + C;    ⍝  Translate this later'
! S ⍎ T
! "STEP 8.44";  ⍝ Same as the statement  STEP 8.44
```

```
∆ < ⊢ character ⊣ ... >
```

No operation will be performed. The character sequence will be treated merely as a comment, with all characters following the first ∆ in a step being completely ignored.

Examples:

```
∆ THIS IS A COMMENT LINE.
```

```
var ← e
```

The variable designator var is first elaborated (cycling all the way down its pointer chain if it begins one) to obtain the "elaborated address" of a value (non-reference) entry. Then the expression e is evaluated to yield a numeric, logic, or string value. That value is assigned to the elaborated address of var, with no conversions of any sort being performed.

Examples:

```
K ← M ∧ ⍕F0
P[3] ← A + (B ← B + 1) + H(N)
I ← J ← K ← ''
```

```
var ←  ∇ < (  ⊢ ident ⊣ .,.  ) >  | e         | ∇
                                  | ⊢ s ⊣ .;. |
```

var is treated as in an expression-assignment. A reference to the given procedure will be constructed and assigned to (the elaborated address of) var. The procedure body is either the expression e or the statement list, and the listed idents are formal identifiers in that body. When the procedure is called, actual parameters must be supplied to replace the formal identifiers during execution of e or the list of statements s. For a procedure with no parameters, the formal identifier list is normally omitted. If so, parentheses cannot be used to surround a procedure-body expression, because they would be treated as parameter delimiters. To get around this syntactic ambiguity, LCC allows an empty formal parameter list to precede a procedure-body expression e (but not a statement list).

Once var has been made a procedure name, any mention of it in an expression or assignment will cause the procedure to be evaluated. Thus the meaning of the var cannot be changed unless it is first redeclared or DELETEd.

Examples:

```
PROC ← ∇(F,G) F + G * H∇
G ← ∇ PART 81 { NEW Z ← Z + 1; Q ← Q } ∇
C[I,J] ← ∇(X) PART 371∇
P3 ← ∇ { PART 4; PART 68; I ← I + 1 } ∇
F ← ∇ () IF X < 48.3 THEN T+1 ELSE T ∇
```

```
var ← ARRAY [  ⊢ e < : e > ⊣  .| ][ |.  ]
                                | , |
```

LCC will assign to var the multidimensional array structure specified by the given bounds list. The bounds list gives the number of dimensions of the array structure and the limits on each of its subscripts. An item in the bounds list can be either a pair of expressions specifying the lower and upper limits on the subscript for that dimension or a single expression specifying the upper limit on that subscript (the lower limit will be implicitly 1).

Storage will not be allocated for an array until that array is used, and even then it will only be allocated for a given row when an element from that row is first accessed. LCC keeps identifying information for each element in an array, and therefore arrays need not be homogeneous. Thus, for example, in a given row an array could contain elements which were procedures, pointers, numeric values, string values, and even arrays.

Note that if the var above is an identifier, this statement form is exactly equivalent to an ARRAY statement. Thus the two statements

```
A ← ARRAY[0:4,6]
ARRAY A[0:4,6]
```

are equivalent. However, if the var is subscripted, we can with this statement specify that an array element is to be itself an array, an effect which is not possible with an ARRAY statement.

Examples:

```
LA ← ARRAY[1:N, -3 : 8*K]
P[2,4] ← ARRAY[5,10,24]
```

```
var ← pointer
```

var is treated as in an expression-assignment. The specified reference pointer will be constructed and assigned to (the elaborated address of) var.

LCC cannot allow a variable to point to another which is declared in an inner (higher) nesting level; therefore such an assignment will lead to an error message and will be rejected. An assignment which would create a circular pointer chain, as in

```
A ← ⊃ B; B ← ⊃ A
```

will also be rejected.

Examples:

```
ND ← ⊃AH[I,J]
```

```
var  < ( < ⊢ | e         | ⊣ .,.  > ) >
             | pointer   |
             | procedure |
```

The procedure referenced by var is performed, using the items in the list as actual parameters. This is done by setting up a new block context, declaring as NEW all formal idents listed in the definition of var, assigning, in order, each actual parameter to the corresponding formal ident, and then transferring control to the body of var. Control will be returned when the procedure executes a RETURN statement, when it "runs off its end" (which causes an implicit RETURN to be executed), when it executes an EXIT statement, or when it executes a GOTO which transfers out of its body. A procedure may be called either as an operand in an expression (in which case it should return a result) or as a statement. A procedure statement should not return a result, but if it does, the value of the result will be typed out at your terminal.

As an example, suppose we have executed the procedure assignment

```
R ← ▽ (A, B, C) PART 3 ▽ ;
```

and we execute the step

```
R ( X - Z , ▽ (G) G * H / 3 ▽, ⊃ W ) ; S ;
```

A new block context will be opened, LCC will perform the statements

```
NEW A ← X - Z ;
NEW B ← ▽ (G) G * H / 3 ▽ ;
NEW C ← ⊃ W ;
```

and execution will begin in the new block context at the first step in part 3. After normal termination of the part, the block context will be closed and LCC will proceed with the successor to

the procedure call, i.e., statement S.

Procedures need not have parameters; thus the actual parameter list may be omitted. If more actual parameters than formals are supplied, the leftmost actuals will be used, with the extra ones being stacked for the duration of the procedure incarnation. If in a subsequent nested procedure call too few actual parameters are supplied, the extra actual parameters from outer procedure calls will be used, with those from the innermost calls being used first.

Examples:

```
FTN
F(A,X-Y)
FN(P+1, ⊃Q, ▽ R + PART 2 ▽ )
```

---- LCC Metavariables ----

```
binary-operator  ::=   |←|↑|*|/|↓|•|+|-|<|≤|=|≥|>|≠|↔|→|∧|∨|≡|¤|

unary-operator   ::=   | + | - | ↓ | ~ |

e  ::=  | primary                          |
        | unary-operator  e                |
        | e_1  binary-operator  e_2        |
        | IF  e  THEN  e_1  ELSE  e_2      |
```

An expression (e) in LCC is a combination of value entities (primaries) with operator symbols which acts as a rule for the computation of a value. Syntactically, an expression may be degenerate (i.e., a single primary), it may be a prefixed unary-operator acting on a value, it may be the combination of two values with an infix binary-operator, or it may be conditional, with a distributed operator (IF ... THEN ... ELSE ...) which selects one from a given pair of values.

The value of an LCC expression will normally be used as a constituent in a statement. However, if an expression appears in place of a statement (or if a syntactically correct LCC statement turns out to have a value), its computed value will be typed back to you. This gives LCC its "desk calculator" feature, whereby you need merely to type an expression to obtain its immediate evaluation -- there is no need to write a "program" to do so. Note, however, that if LCC, when scanning for a statement, finds as its first item an IF, ?, or !, it will treat what follows as a statement, not an expression. If that is not what you mean, you may use parentheses around your expression, and LCC will then treat it correctly.

A conditional (IF) expression acts much like an IF statement. If the expression e evaluates as true, the value of the conditional expression is e_1; if e evaluates as false, the value is e_2. Thus, if the variable AVAL = 1, the value of the expression

```
IF AVAL ≤ 5 THEN 825 ELSE 839
```

is 825.

The unary-operators are '+', '-', '↓', and '~'. A unary '+' is redundant, and +e = e no matter whether e is a number, a logic value, or a string. A unary '-' is a negation operator which changes the sign of any non-zero value to which it is applied (a zero is always positive). '↓' is a truncation operator whose result is the integer portion of the value to which it is applied. Thus ↓2.8 = 2, ↓-3.1 = -3, and ↓341 = 341. '-' and '↓' are arithmetic operators which can act only on numeric values; if they

are applied to logic values or to strings, those values will be converted to numbers before the operations are performed. '~' is a complement operator whose result is the bit-by-bit logical complement of the 32-bit value to which it is applied (i.e., each binary 1 becomes a zero and each binary 0 becomes a one). Thus

$$\sim TRUE = FALSE \quad (= {}_{\&}0), \qquad \sim_{16}FED0 = {}_{\&}FFFF012F$$

Note that multiple unary-operators may precede a primary; if so the operations which they represent will be performed from right to left. Thus

$$\downarrow -3.1 = \downarrow(-3.1) = -3 = -\downarrow 3.1 = -(\downarrow 3.1)$$
$$\downarrow + -3.1 = -\downarrow -\downarrow -3.1 = \downarrow\downarrow ---3.1 = \downarrow -+\sim_{\&}FFFFFFFC = -3$$

Like the unary-operators, the binary-operators can act only on values with the proper data attributes. If one is used with values having improper attributes, appropriate conversions (with a bias from string to logic value to number) will be automatically performed before the operator is executed.

The binary-operators '↑', '*', '/', '÷', '•', '+', and '-' are numeric operators; each acts on numeric values to produce a numeric result. '↑' denotes exponentiation, with e_1 as the base and e_2 as the exponent. The operators '+', '-', and '*' have the conventional meanings of addition, subtraction, and multiplication. '/' is the usual numeric (real) division, with a real result; '÷' (integer divide) and '•' (modulus, or remainder divide) cause a real division operation to be performed, but '÷' gives only the integer portion of the real result as its value (i.e., $A \div B = \downarrow(A/B)$) while '•' gives only the remainder (i.e., $A \bullet B = A - B * (A \div B)$). Thus

$$3.2 \div 2 = 1, \quad 3.2 \bullet 2 = 1.2$$
$$4.7 \div -3 = -1, \quad 4.7 \bullet -3 = 1.7$$

'∧', 'v', and '≡' are logic operators; each acts bit-by-corresponding-bit on logic values to produce a logic value. They have the conventional meanings of logical AND, OR, and equivalence.

'□' is a string concatenation operator which causes the body of string e_2 to be appended to that of e_1.

The operators '←←' and '→→' will shift a logic value or a string left or right. e_2, which will be truncated to an integer, is the length of the shift, while e_1 is the value to be shifted. Shifts will be by bits for logic values and by characters for strings. A shift of a (fixed length) logic value will cause any bits which are shifted out of the value to be lost; vacated positions at the other end of the value will be filled in with zeros. A string, however, does not have a fixed length. Characters shifted "off the end" will be lost, but there will be no "vacated positions" -- the string merely becomes shorter. Thus we will get

the following results:

```
'ABCDEFG' ↔ 4     =    'ABC'
'ABCDEFG' ↔ 2     =    'CDEFG'
'ABCDEFG' ↔ 2 ↔ 4     =    'C'
```

The relational operators '<', '≤', '=', '≥', '>', and '≠' can act on any operands with matching attributes. The meanings of the relations are obvious for numeric operands. Each produces as its result a Boolean value (TRUE or FALSE). For logic values, '=' and '≠' act bit-by-bit to produce logic values which will be, respectively, the logical equivalence and exclusive OR of their operands (i.e., L = M is the same as L ≡ M, and L ≠ M is the same as ~(L≡M)). If the other relations (<, ≤, ≥, >) are applied to logic values, those values will first be converted to numbers and then the usual rules for relations on numbers will be followed. Relations on strings will be performed character-by-character from left to right, with the shorter string being extended, if necessary, to the right with blanks. The normal 360 collating sequence will be used in comparing characters. The result of a string relation will be a Boolean value (TRUE or FALSE).

The assignment operator '←' in an expression takes as its left operand a var, i.e., a reference entity which specifies a variable name. Its right operand can be any expression. The value of an expression e_1 ← e_2 is the value of e_2, and as a side effect that value is also assigned to e_1. Note that a '←' in an expression takes as its left operand only that entity immediately to its left, while its right operand is the whole expression to its right. Thus the statement

```
A[0] ← A[1] ← B + C ← D * E ← F + G
```

will be performed as if it had been written

```
A[0] ← (A[1] ← B + (C ← D * (E ← F + G)))
```

Note also that a '←' in an assignment statement is treated differently from one in an expression in that it does not produce a result and its right-hand side need not be an expression.

If the sequencing of operations in an expression is not explicitly specified by the use of parentheses, the operations will be ordered within it from left to right, but with the following additional rules of precedence:

First:	↓ ~ +(unary) -(unary)
Second:	↑
Third:	* / ↑ •
Fourth:	+ -
Fifth:	< ≤ = ≥ > ≠
Sixth:	←→ →←
Seventh:	∧
Eighth:	∨
Ninth:	≡
Tenth:	◻
Eleventh:	← (as explained above)
Twelfth:	IF ... THEN ... ELSE ...

Thus the statement

```
X ← A - B ↑ 2 / C + ↓D
```

will be performed as if it had been written

```
X ← ((A - ((B ↑ 2) / C)) + (↓D))
```

If a conversion of a value to one of different attributes is necessary, it will automatically be performed by LCC as follows:

number → logic value: LCC will truncate the number and strip off its sign; the binary representation of the resulting integer is truncated to 32 bits to form the logic value. Thus

```
-25.7     becomes     ⍻19
```

number → string:
logic value → string: LCC will transform the internal representation of the number or logic value into its external form (that which would be typed by an output statement). That external form will be the body of the resulting string. Thus

```
-25.7     becomes     '-25.7'
⍻FF12     becomes     '⍻0000FF12'
```

logic value → number: LCC will use the logic value as the low-order 32 bits of the positive integer result. The other bits of the result will be zeros, and thus its value will be between 0 and 2↑32 - 1. As an example

```
⍻2F     becomes     47
```

string → number:
string → logic value: LCC will translate and evaluate the expression which is the body of the string. This must yield another value (possibly again a string) which may need another conversion, etc. Thus if A = 'B', B = 3,

B3 = 42.1, then

'A ¤ B' becomes 42.1

```
extractor ::=   | e_1  :  < e_2 > |
                |      :  e_2     |
```

If an entity has a logic or string value, it may be followed by an extractor, which will select a portion of that value for use as a primary. An extractor must have one of the forms listed above, where e_1 and e_2 are expressions which evaluate to integers, and $1 \le e\_1 \le e\_2 \le N$ (N is the number of bits or characters in the original entity value). If e_1 is missing, it is assumed to be 1; if e_2 is missing, its assumed value is N. Note that an extractor can follow any operand or parenthesized expression; it is not restricted to variables.

A logic value is a quantity whose 32 constituent bits are numbered, starting with 1, from left to right. When a subfield is extracted from a logic value LV, the result is a logic value consisting of those bits of LV with indices from e_1 to e_2 inclusive, right justified in a field of zeros. Thus if LV = ,FF00FF00, then

LV [5:12] = ,000000F0 (= ,F0)

The constituent characters in a string are also numbered, starting with 1, from left to right. When a substring is extracted from a string SV, the result is a string consisting of those characters of SV with indices from e_1 to e_2 inclusive. Thus if SV = 'PORCUPINE', then

SV [6:] = 'PINE'

If an extractor follows a subscript, the character pair '][' may be replaced by the single character ','. A value may not be extracted from an extracted value, and thus it is an error to follow one extractor with another.

Examples:

```
YELLOW[3:10] ¤ RED
Q[3, NN, I:J] ← R[:18]
(A + B + C)[5:8]
P(X, T+1)[35,14][23:]
```

```
for-clause ::= <|FOR ident <|FROM|e>|> <|BY e <TO e>|> <WHILE e> DO
                |           |←   |  |   |TO e <BY e>|
                |FROM e             |
```

See the iteration (FOR) statement description on page 12 for an explanation of the control exercised by a for-clause.

Examples:

```
FOR I FROM 1 BY G TO H WHILE N ≠ 3 DO S
TYPE P,(FOR I TO 10 DO(FOR J TO 5 DO C[I,J]),F[I])
```

```
group   ::=  | | PART  |  < num < TO num > > |
             | | PARTS |                     |
             | | STEP  |                     |
             | | STEPS |                     |
             | num < TO num >                |
```

A group is a specification of a step or a contiguous set of steps. A single step is normally specified by the keyword 'STEP' followed by a num, but if the group scanner finds a num without a preceding keyword, it will assume the presence of the word 'STEP'. A set of steps is specified as one of

```
STEP num TO num
STEPS num TO num
```

or merely as

```
num TO num
```

A part or set of parts is similarly specified as

```
PART num
```

or as one of

```
PART num TO num
PARTS num TO num
```

(the keywords 'PART' and 'PARTS' cannot be omitted).

In scanning for a group, as well as everywhere else in LCC, the translator always considers the keyword 'STEP' equivalent to 'STEPS' and the keyword 'PART' equivalent to 'PARTS'. Thus, for example, you can write

```
DISPLAY PARTS 6
DELETE STEP 4.7 TO 5.3
IF A < B THEN PARTS 6
```

Whenever the construction 'num_1 TO num_2' is used in LCC, you must have num_1 ≤ num_2, unless num_2 < 1, in which case LCC will increment it by the integer portion of num_1. Thus, for example,

```
DISPLAY STEPS 3.6 TO .9
```

is equivalent to

```
DISPLAY STEPS 3.6 TO 3.9
```

Examples:

```
ALTER STEP 1.6 : 'X' → 'AX' , 'Y' → 'BY'
COPY PART 3 AS 43
"STEPS 4.5 TO 4.73"
NUMBER 7.7 TO 8.2 AS 25 BY .02
```

```
digit   ::=   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

letter  ::=   |A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
              |a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|

ident   ::=   letter  < ⊢ | digit  | ⊣ ... >
                         | letter |
                         | ⊢ _ ⊣  |
```

Identifiers (idents) are used to name entities in LCC. An identifier consists of a sequence of one or more letters, digits, and/or underline(_) characters, the first of which must be a letter. Some identifiers are keywords in LCC and are reserved for that purpose; you cannot use them as names. Others, such as the names of the standard functions (see Appendix H) and the other built-in LCC functions and procedures (see Appendix I) are privileged identifiers in the sense that they are given meanings when LCC is initialized. You may use a privileged identifier as a variable name by declaring it, but if you do, its original meaning will be superseded and may be lost.

Even though an identifier can be arbitrarily long, LCC will retain only its first (leftmost) 8 characters, with all other characters being ignored. Thus identifiers must be uniquely distinguishable within their first eight characters.

Examples:

```
X
RED
ALGOL_60
RUMPELSTILTSKIN
```

```
hex-digit   ::=   | digit  | A | B | C | D | E | F |

logic-literal   ::=   | FALSE                              |
                      | TRUE                               |
                      | %  <  | L |  >  ⊢ hex-digit ⊣ ... |
                      |       | R |                        |
```

A logic-literal in LCC is written either as a hexadecimal value or as one of the Boolean values 'TRUE' or 'FALSE'. The 16 hexadecimal digits are specified by the decimal digits 0 through 9 and the letters A through F, with the "digits" 10 through 15 being represented by the letters A through F respectively. A logic value is represented in LCC as a 32 bit quantity; therefore a logic-literal can contain up to 8 hex-digits, which must be contiguous, i.e., imbedded blanks are not allowed. The optional letters 'L' and 'R' in a hexadecimal literal indicate left and right justification respectively. If neither letter is present, an 'R' will be assumed. Thus

```
%LFB1 = %FB100000
```

The Boolean values are equivalent to hexadecimal values as follows:

```
FALSE = %0                  (32 binary zeros)
TRUE  = %FFFFFFFF           (32 binary ones)
```

Note, however, that when tested in an IF clause, any non-zero value will be considered to have the quality "true". Thus the statement

```
TYPE  IF %123  THEN  'T'  ELSE  'F'
```

will print 'T', even though %123 ≠ TRUE.

Examples:

```
FALSE
%9AB7
%LFF
```

```
num  ::=   | int  ⊢ . ⊣  int |
           | ident            |
           | ( e )            |
```

A num is used to specify the number of a step or a part. It will usually be a decimal number, i.e., a number without an exponent. However, it may also be an ident whose value is a step number, or a parenthesized expression which evaluates to a step number.

A part or step cannot have a negative number; therefore LCC will take the absolute value of each evaluated num before using it.

Examples:

```
STEP 1420.35
PART (J + 2)
DELETE STEPS A TO B
```

```
int  ::=  ⊢ digit ⊣ ...

number-literal  ::=  | | int  < ⊢ . ⊣ < int > > | < ₁₀ < | + | > int > |
                     | | ⊢ . ⊣  int              |       | - |         |
                     | ₁₀ < | + | > int                              |
                     |      | - |                                    |
```

A decimal arithmetic constant in LCC is written as a number-literal. A number-literal is a sequence of digits, possibly including a decimal point, optionally followed by an exponent part. An exponent part consists of the delimiter character ₁₀ followed by an optionally signed decimal exponent. As a special case, if the base value of a number is to be 1, the number-literal can be written using only an exponent part. Thus

$$_{10}-15 = 1_{10}-15$$

Blank spaces are not allowed within a number-literal; thus 3.7 ₁₀-5 and 5_{10} 14 are illegal.

Numeric values will be stored by LCC as long (double word) floating-point System/360 quantities. This allows a precision of about 17 decimal digits, though for output LCC will usually round a number to 10 digits. The maximum absolute value of a number is approximately $7.237_{10}75$; the minimum non-zero absolute value is approximately ₁₀-75.

Examples:

```
15
7.36
₁₀25
6.2₁₀-5
138.
.5
```

```
operand  ::=  | BEGIN  ⊢ s ⊣ .;.  END                                  |
              | CASE e OF (  ⊢ e ⊣ .,.  < , OTHERWISE e > )            |
              | PART num  < {  ⊢ s ⊣ .;.  } >                          |
              | | STEP  | num < TO num >                               |
              | | STEPS |                                              |
              | ident                                                  |
              | logic-literal                                          |
              | number-literal                                         |
              | string-literal                                         |
              | var  < ( <  ⊢ | e          | ⊣ .,.  > ) >             |
              |               | pointer    |                          |
              |               | procedure  |                          |
              | ? < $ > < string-literal > < ident >                   |
              | ! e                                                    |
              | {  ⊢ s ⊣ .;.  }                                        |
              | " group "                                              |
```

Most of the operands are described individually below (starting on page 46). For ident and logic-, number-, and string-literal, see the descriptions of the individual metavariables. For the part call and the var call, see the descriptions of the corresponding statements.

An LCC operand may be characterized most simply as an entity which returns a result; a statement is an entity which does not return a result. In many cases, operands and statements look alike (e.g., a part or step call, a procedure call, a block) and the distinction between them must be made by context or it may have to be made dynamically during execution.

```
pointer  ::=  ⊃ varid
```

A pointer is used to indirectly reference an incarnation of a variable. It is thus an object which acts as an alias for the object to which it points. Whenever a variable containing a pointer is used in an expression or an assignment, the object to which it eventually points will be accessed or modified, not the original variable or the pointer. A pointer may point to another pointer, and thus we may have pointer chains. A pointer chain must end at a non-pointer (cycles will not be allowed) and it is that final element to which any pointer in the chain refers. As an example, after we execute the statements

```
A ← ⊃B;  B ← ⊃C;  C ← 17
```

the value of A + B + 1 will be 35. If we then execute the assignment statement

```
A ← 'FISH'
```

the value of C will be changed to the string 'FISH'.

Pointers may be assigned, RETURNed, or passed as actual parameters. Their main uses are to construct list structures or to refer to particular incarnation-values which might otherwise be unavailable in inner blocks of a program. Moreover, if a procedure is to store a result into a variable which is to be passed to it as a parameter, that parameter must not be the variable name but rather a pointer to it.

Examples:

```
T[0,6] ← ⊃Z
NEW PTR ← ⊃Q, Q ← 5
RETURN  ⊃AR3[2,I,-4]
PR(5, ⊃X, N)
```

```
primary ::= | operand | < [ | extractor                          | ] >
            | ( e )   |     | subscript-list < | ][ | extractor > |
                            |                  | ,  |             |
```

A primary begins with either an operand or an expression enclosed in parentheses, and it may be optionally followed by a subscript-list and/or an extractor. A primary is a value entity (numeric, logic, string) as distinguished from a reference entity (label, procedure, array name, pointer), though this distinction cannot be checked by LCC until the primary is executed.

Examples:

```
X[COLOR, SIZE, WT-2]
GREEN
YELLOW[B3:10]
Q[3][N][I:J]
(A + B - C)[5, :10]
FN(A,B)[C]
```

```
procedure  ::=  v < (  ⊢ ident ⊣ .,.  ) > | e          | v
                                          | ⊢ s ⊣ .;. |
```

The procedure body is the expression e or the statement list, and the listed idents are formal identifiers in that body. When the procedure is called, actual parameters must be supplied to replace the formal identifiers during execution of e or the statement list. For a procedure with no parameters, the formal identifier list may be omitted (see the description of the procedure assignment statement on page 26).

Examples:

```
PROC ← ∇(F,G) F + G * H∇
G ← ∇ PART 81 { NEW Z ← Z + 1; NEW Q } ∇
```

```
s   ::=   | statement |
          | ident : s |
```

Any statement in a delayed step may be preceded by one or more label identifiers which name that statement and allow other statements to branch or 'GO' to it. Labels are not usually necessary, because step numbers can also be used as transfer points for GOTO's, but they are useful for naming statements within a step or for naming statements in a part or group which is to be renumbered.

Labels do not always work correctly in LCC, and at present there are some situations which must be avoided. The known incorrect cases (as of October 24, 1969) are listed below.

1. Labels in steps called via step calls (as in STEPS 3.7 TO 3.8) do not work correctly and if used will usually lead to errors later on in your conversation.

2. Labels in a (BEGIN-END) block statement or expression do not work correctly and the errors they lead to will not normally be caught by LCC.

3. If a step containing labelled statements is added to an active part, the labels will not be declared during the current activation of that part. In future activations, however, they will operate correctly.

4. Labels in the statement list of a procedure or inside the braces of a part call do not work correctly and will normally be ignored by LCC.

Examples:

```
3.7: A: B ← 3
13.452, I ← I + 1; L: M: J ← J + 1
F: G: H: K ← J + 1
```

```
save-object  ::=  | ALL                                          |
                  | PARTS                                        |
                  | STEPS                                        |
                  | VALUES                                       |
                  | < | PART  | >  ⊢ num < TO num > ⊣ .,.  |
                  |   | PARTS |                               |
                  |   | STEP  |                               |
                  |   | STEPS |                               |
                  | ⊢ varid ⊣ .,.                            |
```

A save-object, which may be either SAVEd, DELETEd, or DISPLAYed, can be a set of contiguous steps (as in a group), a list of sets of steps or parts, a list of the meanings associated either with selected variables or with all the variables in your program (VALUES), or a combination of all of your steps and all of your values (ALL). As in a group, if the word 'PART' or 'STEP' is missing before a num in a save-object, the word 'STEP' will be assumed. Note that if a save-object begins with an identifier, that identifier will be treated as the first such in a varid list rather than the first num in a step list.

Examples:

```
DISPLAY VALUES
DISPLAY X, Y, Z[4,J]
DELETE STEPS 4.6, 7.1 TO 10.6, 15.3, 4.8902
SAVE PARTS 45 TO 493 AS FILE 'CAT'
```

statement

See the descriptions of the individual statements, starting on page 3.

```
string-character  ::=  | any-CMU-character-but-a-quote |
                       | ``                            |
                       | ''                            |

string-literal  ::=  ' < ⊢ string-character ⊣ ... > '
```

A string-literal in LCC is written as a sequence of zero or more string-characters enclosed within left and right single-quote characters. The legal string-characters are the 88 characters on the "CMU Type-Ball"

```
! ∀ ∃ $ | ⍎ ⍞ ⍕ ⍝ _ ?
1 2 3 4 5 6 7 8 9 0 − +

≡ ⊂ ⊃ = ≠ ∨ ∧ ⍋ ⍒ ∇ Δ
Q W E R T Y U I O P ⍟

~ ⍀ ⌿ < > [ ] ( ) : ⍫
A S D F G H J K L ; ←

∈ ⍵ ⍷ ≤ ≥ ⊢ ⊣ { } \
Z X C V B N M , . /
```

plus the 26 lower case letters and the space (blank) character. In order to avoid ambiguity, you must type in two successive left or right single-quote characters to get one inside your string. Thus if you execute the step

```
S ← `AB''''CD```;  TYPE S
```

LCC will type back the value

```
AB''CD`
```

which is the body of S. An exception to this rule is the treatment of a string body which is typed in response to a string read (?$) request. Single-quotes need not be doubled to appear in such a string.

The lower case letters cannot be typed out at your terminal by a CMU type-ball, although they can be printed by the line printer in the computer room (via a PRINT statement). A lower case letter can be typed in from your terminal by preceding the corresponding upper case letter by a vertical bar (|) which acts as an "escape character". Thus the string

```
`|AB|C|D|EFGH'
```

will be printed on the printer as

```
aBcdeFGH
```

Lower case letters will be typed out on your terminal as their upper case equivalents. Thus the above string would be typed as

```
ABCDEFGH
```

Because of the use of the vertical bar as an escape character, you must always type two successive vertical bars to get one into your string. Thus if you type in

```
`||+||-||⍋||~||'
```

LCC will type back the string body

|+|-|↓|~|

Other than for lower case letters, you will not need to use escape characters with the regular CMU type-ball. Escape characters will, however, be necessary if you use some other type ball or if you use a teletype for your conversation with LCC, but these uses will not be described here.

Examples:

```
'BLUE'
'ABC' ¤ 'DEF' ¤ S
! 'A ← B + C;   ▽ TRANSLATE THIS LATER '
```

```
structure  ::=  ARRAY  [  ⊢ e < : e > ⊣  .| ][ |.  ]
                                          | , |
```

A structure specifies the dimension and the subscript bounds which are to be assigned to a given var, thus making that var into an array. See the var ← ARRAY . . . statement on page 27 for a more complete description.

Examples:

```
LA ← ARRAY[1:N, -3 : 8*K]
NEW A ← ARRAY[3,0:5], B, C ← 26, D ← ARRAY[X:Y]
A[B,C] ← ARRAY [1:5][0:6]
```

```
subscript-list  ::=  ⊢ e ⊣  .| ][ |.
                             | , |
```

Any array designator (an array name or a reference to an array) may be followed by a subscript-list, which will select an element from the array. Each expression in the subscript-list will be evaluated to a number, rounded to an integer, and used as an index to obtain a constituent from the array, with the validity of the indexing being determined dynamically. The selected constituent element may again be an array, and the subscription process may then be repeated. When multiple subscripts are used, any character pair '][' may be replaced by the equivalent single character ','.

Examples:

```
X[COLOR, SIZE, WT-2] ← Y[3]
P(A,B+1) [I][J] ← D[1,2][3] / K
```

```
empty  ::=                          (i.e., the null string of characters)

type-object  ::=  | e                                          |
                  | empty                                      |
                  | ( for-clause ⊢ type-object ⊣ .,. )         |
```

See the description of the TYPE statement (page 23).

Examples:

```
TYPE C[3], , DEF + 1, 'STR'
WRITE (FOR I TO 100 DO A[I], B[I]) AS 'FILE6'
TYPE (FOR I TO 10 DO (FOR J TO 10 DO A[I,J]))
```

```
var  ::=  | operand |  < [  subscript-list  ] >
          | ( e )   |
```

A var begins with either an operand or an expression enclosed in parentheses, and it may be optionally followed by a subscript-list. A var must be a reference entity which specifies a variable name, though LCC cannot check whether or not the var is a reference until it is executed.

Examples:

```
P[3] ← A + (B ← B + 1) + H(N)
I ← J ← K ← ''
(A + B) [C][D] ← 3
(?P)[Q] ← 5
H[J](1,2)[3,4](5) ← 6
```

```
varid  ::=  ident  < [  subscript-list  ] >
```

A varid is an identifier optionally followed by a subscript-list, i.e., a varid is the designator of a variable. Expressions in the subscript-list may be separated from one another either by a comma or by the character pair '][' (i.e., a subscripted varid is also a varid, which may again be subscripted).

Examples:

```
ND ← ⊃AH[I,J]
A[B][C]
? A1, B2[3, J+1], C, D[K][9]
```

---- LCC Operands ----

BEGIN ⊢ s ⊣ .;. END

The keywords 'BEGIN' and 'END' delimit a "block", whose list of arbitrary LCC statements will be treated as if it were in a part, i.e., there may be local variables valid only within it. LCC will perform a block entry, after which it will execute the statements from the list in sequence. This "block expression" will normally be terminated by a RETURN statement which supplies a value. Such a RETURN will terminate the block context, and the returned value will be used as that of the operand. A RETURN statement without a value will first terminate the context of the block expression and then return from the context in which the block is embedded.

Examples:

```
X ← Y + BEGIN NEW A; PART 6; RETURN A  END  - Z
```

CASE e OF (e_1 , e_2 , ... , e_N)

The expression e is evaluated and rounded to an integer K. If $1 \leq K \leq N$, the value of this CASE expression is the value of e_K. It is an error if K is out of the range 1 to N.

CASE e OF (e_1 , e_2 , ... , e_N , OTHERWISE e_(N+1))

This statement operates like the ordinary CASE expression above except if K is out of the range $1 \leq K \leq N$, the value is e_(N+1).

Examples:

```
G ← CASE I-J OF (A, B+1, C-D, OTHERWISE E/6) * H
```

STEPS num_1 TO num_2

As in a group, num_1 must be ≤ num_2 (unless num_2 < 1). LCC will set up a new group context (non-block) for the sequence of steps from num_1 to num_2. Execution will then begin at step num_1 and it will continue through successively higher numbered steps. The context for this step group operand will normally be terminated by a RETURN statement, whose result will be the value

of the operand. It is an error for the group to return without a value. An EXIT statement will terminate the step group context and return control to you in the context of its calling group.

Note that there is a possible syntactic ambiguity when a step group operand is used inside an iteration clause. An example is the statement

```
FROM STEPS 3.5 TO 3.8 BY 2 DO PART 8
```

In any such ambiguous cases, the keyword 'TO' will always be associated with the step call rather than with the iteration clause.

Examples:

```
M ← X - STEPS 5.3 TO 5.46
```

STEP num

Equivalent to the operand

```
STEPS num TO num
```

Examples:

```
TEMP ← STEP 1420.35 * Z
```

?

If LCC encounters a question mark as an operand, it will type a message and give control to you. You must then type an expression and return control (by pressing the RETURN key). The typed expression will be translated and evaluated, and its result will be the value of the operand. Note that the typed expression may involve your program variables, whose current meanings will be used in its evaluation.

Examples:

```
Y ← ?A + ? + ?'LENGTH' LNG + ?$'READ STRING'
```

? string-literal

This operand performs like a simple ? operand except LCC will type out the user-supplied message string instead of the system message.

Examples:

```
T ← ? 'TIME'
```

? < string-literal > varid

This operand is equivalent to one of the expressions

```
(varid ← ?)
(varid ← ? string-literal)
```

Varid must be an optionally subscripted variable identifier. You will be asked for a value as for the simple ? operands described above. That value will be assigned to varid before being used as the value of the operand.

Examples:

```
X ← ?Y - 3 * ?''Z
```

? $ < string-literal > < varid >

This operand is the same as an ordinary ? operand except LCC will treat your typed response as the body of a string (i.e., it will surround the characters which you typed with quote marks). Thus the value of a ?$ operand will always be a string. As an example, if you respond with the character sequence

```
ALPHA + BETA
```

to LCC's request for the operand ?$PQ in the statement

```
T ← S ◻ ?$PQ
```

the effect will be to perform, in order, the assignments

```
PQ ← 'ALPHA + BETA';
T ← S ◻ 'ALPHA + BETA';
```

A slight variation is possible here in the use of single-quote marks, which need not be doubled to appear in your requested string body. Thus if you were to type

```
B'*'7+
```

in response to the above request for ?$ PQ, the effect would be to perform the assignment

```
PQ ← 'B''*''7+'
```

Examples:

```
G ← ?$ 'INPUT N' EN ◻ EM
```

! e

The expression e must evaluate to a string, whose contents will be treated as expression data to the LCC translator. When a ! operand is executed, the string which it supplies will be translated and converted to a value. That value will then be used as the value of the operand. Thus an operand !ST, where ST has a string value, has the same effect as the expression

```
(ST + 0)
```

which forces the value of ST to be converted from a string to a number before the addition can be performed.

Examples:

```
XY ← FF(1-SIN(Z), !P) * 3
```

; { ⊢ s ⊣ .;. }

LCC will treat the statement sequence from this "compound expression" as a single control unit whose sub-statements will be executed sequentially from left to right. A compound expression is not a block and does not have its own local variables. It will normally be terminated by a RETURN statement, whose value will be the value of this operand. A RETURN statement without a value will first terminate the context of the compound expression and then return from the context in which that expression is embedded.

Examples:

```
YZ5 ← T + { FOR K TO N DO F(K,L,N); RETURN K } / 2
```

" group "

The value of this operand is a string consisting of the text of the specified group. That string will contain only the source text for a step -- not its number. If the group includes more than one step, the strings for the individual steps will be concatenated in step-number order to form the operand, with no semicolons, blanks, or any other characters being inserted between successive text strings.

Examples:

```
S ← "STEPS 4.5 TO 4.73" ¤ "STEP 6.1"
! "14.301"
```

---- Explanation of Syntax Notation ----

< >

Optional presence -- These delimiters surround a construct which may either be present or absent.

| |

Alternatives -- These delimiters surround a set of alternatives, one and only one of which must be present. The alternatives are usually listed vertically, but for a few metavariables, such as "digit" and "letter", where there are many alternatives, they will be listed horizontally and separated from one another by | | delimiters.

⊢ ⊣

Grouping -- These bracketing delimiters are used for grouping only.

.,.

.;.

.|][|.
 | , |

Repetition -- The immediately preceding syntax construct, which will be surrounded by ⊢ ⊣ brackets, may be optionally repeated a number of times, with the construct between the dots (a comma, a semicolon, or either a comma or the character pair '][') being used to separate the individual constructs. Thus the notation

⊢ e ⊣ .,.

could mean any of the following

e , e , e , e
e
e , e

...

Repetition -- The immediately preceding construct may be optionally repeated a number of times, with no separators (or spaces) between the individual constructs.

::=

This separator may be read 'is defined to be'. It is used in the same sense as in Algol 60 syntax notation (BNF) for defining LCC metavariables.

In the syntax descriptions, lower-case words or phrases are used to name metavariables. As used here, a metavariable is a description-language variable which is used to simplify the description of LCC. A metavariable is not itself an LCC construct, but it is defined (often recursively) in terms of LCC elements. Whenever a metavariable is used in the syntax description of LCC, it must be replaced by a set of LCC characters satisfying its definition in order to obtain a valid LCC construct. As an example, the metavariable "digit" can be any of the atomic characters 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9.

The upper case words used in the syntax are primitive LCC elements which must be used (and spelled) exactly as written (except for the equivalent LCC words 'PART' and 'PARTS', which may be used interchangeably, and 'STEP' and 'STEPS', which may also be interchanged). These primitive "keywords" are reserved identifiers in LCC, and they may not be used to name variables. The current LCC keywords are the following:

ALL
ALTER
ARRAY
AS
BEGIN
BY
CASE
COMBINE
COPY
DELETE
DISPLAY
DO
ELSE
END
EXIT
FALSE
FILE
FOR
FORM
FROM
GO
GOTO
IF
IN
LINE
LOAD
NEW
NUMBER
OF
OFF
OTHERWISE
PART
PARTS
PAUSE
PRINT
PUNCH
READ
RECOVER
RETURN
SAVE
SHARE
STEP
STEPS
THEN
TO
TRUE
TYPE
USE
VALUES
WHILE
WITH
WRITE

---- LCC Syntax ----

```
binary-operator  ::=   |←|↑|*|/|÷|•|+|-|<|≤|=|≥|>|≠|↔|↠|↣|∧|∨|≡|¤|

digit  ::=   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

e   ::=   | primary                  |
          | unary-operator  e        |
          | e  binary-operator  e    |
          | IF  e  THEN  e  ELSE  e  |

empty  ::=                          (i.e., the null string of characters)

extractor  ::=   | e : < e > |
                 |   : e     |

for-clause ::= <|FOR ident <|FROM| e >|> <|BY e < TO e >|> <WHILE e> DO
                 |            |←   |    |   |TO e < BY e >|
                 |FROM e                 |

group   ::=   | | PART  | < num < TO num > > |
              | | PARTS |                    |
              | | STEP  |                    |
              | | STEPS |                    |
              | num < TO num >               |

hex-digit   ::=   | digit | A | B | C | D | E | F |

ident   ::=   letter  < ⊢ | digit  | ⊣ ... >
                          | letter |
                          | ⊢ _ ⊣  |

int   ::=   ⊢ digit ⊣ ...

letter  ::=   |A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
              |a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
```

```
logic-literal   ::=   | FALSE                                  |
                      | TRUE                                   |
                      | % < | L | > ⊢ hex-digit ⊣ ... |
                      |     | R |                            |

num   ::=   | int ⊢ . ⊣ int |
            | ident         |
            | ( e )         |

number-literal ::= | | int < ⊢ . ⊣ < int > > | < ₁₀ < | + | > int > |
                   | | ⊢ . ⊣ int             |      | - |        |
                   | ₁₀ < | + | > int                          |
                   |      | - |                                |

operand   ::=   | BEGIN ⊢ s ⊣ .;. END                                  |
                | CASE e OF ( ⊢ e ⊣ .,. < , OTHERWISE e > )            |
                | PART num < { ⊢ s ⊣ .;. } >                           |
                | | STEP  | num < TO num >                             |
                | | STEPS |                                            |
                | ident                                                |
                | logic-literal                                        |
                | number-literal                                       |
                | string-literal                                       |
                | var < ( < ⊢ | e         | ⊣ .,. > ) >              |
                |           | pointer   |                            |
                |           | procedure |                            |
                | ? < $ > < string-literal > < ident >                 |
                | ! e                                                  |
                | { ⊢ s ⊣ .;. }                                        |
                | " group "                                            |

pointer   ::=   ⇒ varid

primary ::= | operand | < [ | extractor                                  | ] >
            | ( e )   |     | subscript-list < | ][ | extractor > |
                        |                    | ,  |               |

procedure   ::=   ▽ < ( ⊢ ident ⊣ .,. ) > | e         | ▽
                                          | ⊢ s ⊣ .;. |

s   ::=   | statement |
          | ident : s |
```

```
save-object  ::=   | ALL                                              |
                   | PARTS                                            |
                   | STEPS                                            |
                   | VALUES                                           |
                   | < | PART  | >  ⊢ num < TO num > ⊣ .,. |
                   |   | PARTS |                                      |
                   |   | STEP  |                                      |
                   |   | STEPS |                                      |
                   | ⊢ varid ⊣ .,.                                    |

statement   ::=        (see list of statements starting on next page)

string-character   ::=   | any-CMU-character-but-a-quote |
                         | ``                            |
                         | ''                            |

string-literal   ::=   '  < ⊢ string-character ⊣ ... >  '

structure   ::=   ARRAY [  ⊢ e < : e > ⊣  .| ][ |.  ]
                                           | , |

subscript-list   ::=   ⊢ e ⊣  .| ][ |.
                               | , |

type-object   ::=   | e                                              |
                    | empty                                          |
                    | ( for-clause ⊢ type-object ⊣ .,. )             |

unary-operator   ::=   | + | - | ↓ | ~ |

var   ::=   | operand |  < [  subscript-list  ] >
            | ( e )   |

varid   ::=   ident  < [  subscript-list  ] >
```

statement ::= one of the following syntactic forms

```
ALTER group  | : |  ⊢ e → e ⊣ .,.
             | , |

< NEW > ARRAY ⊢ ⊢ ident ⊣ .,.  [ ⊢ e < : e > ⊣  .| ][ |. ] ⊣ .,.
                                                 | , |

BEGIN  ⊢ s ⊣ .;.  END

CASE e OF { ⊢ s ⊣ .;. < ; OTHERWISE s > }

COMBINE < STEPS > num TO num AS e

COPY group AS e < BY e >

DELETE  | FILE e      |
        | save-object |

DISPLAY  | FILE < CATALOG > |
         | RETURN < STEPS > |
         | save-object      |

EXIT  < | ALL                | >
        | < TO > < PART > e  |

for-clause  s

| GO < TO > |  < e >
| GOTO      |

IF e THEN s < ELSE s >

LINE < e >

LOAD < FILE > e
```

```
NEW  ⊢ ident  < • | e         | > ⊣ .,.
                 | pointer   |
                 | procedure |
                 | structure |

NUMBER  | AS e           |  < BY e >
        | group < AS e > |

OFF  < SAVE >

PART num  < {  ⊢ s ⊣ .;.  } >

PAUSE < e >

PRINT  < FILE > e

RECOVER < e >

RETURN  < | e         | >
          | pointer   |
          | procedure |

SAVE  save-object  < AS < FILE > e >

| STEP  |  num < TO num >
| STEPS |

TYPE  ⊢ type-object ⊣ .,.

USE < FILE > •

WRITE  ⊢ type-object ⊣ .,.  < AS < FILE > e >

? < $ > ⊢ < string-literal > varid ⊣ .,.

{  ⊢ s ⊣ .;.  }

! e
```

```
A < ⊢ character ⊣ ... >

var ← | e         |
      | pointer   |
      | procedure |
      | structure |

var < ( < ⊢ | e         | ⊣ .,. > ) >
            | pointer   |
            | procedure |
```

---- Procedure for Logging On to the LCC System ---- ---- at a 2741 Terminal ----

1. Set the power switch (at right of keyboard) to ON.

2. Make sure the terminal mode switch (on left side of 2741) is set to COM. It will be set to COM if and only if the keyboard is locked, which you can easily test by trying to press the RETURN key.

3. Push the TALK button on your Data-Phone.

4. Lift the phone receiver and dial the computer, which will answer and then emit a continuous tone. When you hear the tone (a beep), press the DATA button and replace the receiver. You are now connected with the TSS monitor system, which will, after a short delay, type back to you a message similar to

   ```
   B001 TSS AT CMU TASKID=0031 09/23/69 17:31 8345 SDA=0053
   ```

5. Type your 8-character user number and press the RETURN key. TSS will respond with a one or two line greeting message and, on a new line, an initial underline character (_) followed by a backspace, leaving the typing element positioned at the first position on the line.

6. If this is to be your first session with LCC, type the characters

   ```
   SHARE USER,LCC,USER
   ```

   and press RETURN. TSS will respond with another underline-backspace. This SHARE command needs to be typed only once, and on subsequent runs you should omit it.

7. Type the characters

   ```
   DDEF LCC,VP,USER.LCC,OPTION=JOBLIB
   ```

   and press RETURN. TSS will again respond with an underline-backspace.

8. Type the characters

```
LCC
```

and press RETURN. After a short delay, LCC will respond with a polite greeting such as

```
LCC: GOOD AFTERNOON
```

It will then indent four spaces and give you control. You are now communicating directly with the LCC processor, which will analyze all succeeding lines which you type.

The complete logon record for your first LCC run will thus be similar to the following:

```
B001 TSS AT CMU TASKID=0031 09/23/69 17:31 8345 SDA=0053
XYZ1ZZ13
15:22 23SEP 69-TSS UP TILL 24:00
SHARE USER,LCC,USER
DDEF LCC,VP,USER.LCC,OPTION=JOBLIB
LCC
LCC: GOOD AFTERNOON
```

For subsequent runs, everything will look the same except for the omission of the 'SHARE' line.

---- Typing LCC Text at a 2741 ----

The characters, including blanks, which you type will be sent to LCC line-by-line in the order you type them. However, if you discover before you finish typing a line that you have made an error on that line, you may backspace past the incorrect characters, thus deleting them from the line being sent to LCC (though not, obviously, from your typed page). You may then complete the line by typing the correct characters or, if no correction is needed, merely press the RETURN key. Each time you press BACKSPACE, you will delete one character from the line; thus five BACKSPACEs would erase the last five characters (including blanks) which you typed. After backspacing, you should manually upspace the paper in your 2741 to avoid any confusion which would be caused by strikeovers.

If your whole line is wrong, you may cancel it all by pressing RETURN immediately after typing either the character '¤' or the character '/'. LCC will completely ignore the line, and it will merely unlock the keyboard for the next line -- it will not indent the typing element after such a line cancellation. Note that a '¤' and a '/' will act to cancel a line only when they are followed immediately by a RETURN. In all other cases they are sent along as legitimate LCC characters.

When you complete a line, you must terminate it by pressing the RETURN key. This will cause the sequence of characters which you typed to be sent to the LCC processor for syntactic analysis and possible action. LCC will scan your line from left to right in order to translate it into an internal interpretable code. If your line is syntactically incorrect, an error message will be typed back to you, indicating (by a '|') the position in the line of the item which had just been scanned when the error was encountered and (by a number) the kind of error which was found (see Appendix E). If your line is correct, LCC will determine whether it is a complete step or whether you plan to supply an additional line to continue it. You must indicate such continuation by typing a hyphen or minus character ('-') just before pressing RETURN. The next line will then be concatenated with the current line such that its first character will follow directly after the last character before the hyphen, and the hyphen will be deleted.

Each line will be analyzed as above until a step is found to be complete. LCC will then determine whether the step is immediate or delayed by checking its step number. If it has a number, the step is delayed, and it will be saved internally so that it may be called into execution at some later time. If it has no number, the actions specified by the step will be performed immediately. When all such actions have been completed, LCC will indent one or more spaces, unlock the keyboard, and return control to you.

---- Error Messages ----

Translator (syntax) errors -- A vertical bar character (|) will be typed under the position in your step text which had just been scanned by the translator when it discovered the error, and a message of the form

```
ERROR SXnn text
```

will be written. "nn" is a two digit number which specifies the translator error which has been encountered, and "text" is an abbreviated description of the error (see Appendix F for some expanded descriptions of the errors). The error message will be left-justified on the line containing the '|' marker unless the marker occurs within the first 10 characters on the line, in which case the message will be typed to the right of the marker.

Execution errors -- Execution error messages are of the form

```
ERROR mmmm text
```

where "mmmm" is a four character internal error designator and "text" is a string which describes the error which has been encountered. Examples are

```
ERROR UN01 V[45,1] IS UNDEFINED
ERROR GO03 STEP 2.15 NOT IN AN ACTIVE CONTEXT
ERROR VE03 SUBSCRIPT OUT OF RANGE
ERROR OR01 AT 61.4  DIVISION BY ZERO
```

A complete listing of all the errors caught by LCC, with explanations of their causes and descriptions of any possible recovery options, may be found in the reference document "LCC Error Messages".

---- LCC Syntax (SX) Error Descriptions ----

1: This should have been a statement, but it isn't one.
2: This literal constant is malformed.
3: This must be an operand. It isn't one.
4: This must be an operator or a delimiter. It isn't one.
5: No `[' to match this `]'.
6: An extracted value may not be subscripted.
7: In the current language context, this is meaningless.
8: This should be a statement terminator (END, }, ;, ELSE, v).
9: No `(' to match this `)'.
10: No `BEGIN' to match this `END'.
11: No `IF' to match this `THEN'.
12: No `THEN' to match this `ELSE'.
13: Your " must meet its match here.
14: You need a step or part number here.
15: A controlled variable must be an identifier.
16: Your CASE statement needs a `{' here.
17: Your CASE expression needs a `(' here.
18: The `OTHERWISE' must be last in a CASE list.
19: You can't store into an extracted value.
20: You can't have more than an expression here.
21: You need `AS' here.
22: A parameter may only be delimited by `,' or `)'.
24: This step is missing an `END'.
25: This step is missing a `}'.
26: No `{' to match this `}'.
27: You need to specify some subscript bounds here.
28: You can only request input to a variable, not an expression.
30: You need `FROM' or `IN' or a statement terminator here.
32: No `v' to match this one.
34: You need a `)' to end this formal parameter list.
35: You need a save-object or a group designator here.
36: You need a group designator here.
38: This must be an identifier.
39: This must be a `←'.
40: You need a `:' or a `,' to delimit this ALTER list.
43: This can't follow an iterated output element.
44: This should be a step number, but it isn't one.

96: Whoops -- the first phase of the translator has just had a stack indexing error, which should be impossible. Please show your listing to an LCC implementor.
97: The translator has just run into some sort of a semantic error. It could be due to something simple, like an unmatched `END', but if you can't find a mistake, please ask an LCC implementor for some help.
99: Congratulations: you have just found an error in the LCC syntax tables. Please tell an LCC implementor about it.

---- Automatic Reload File ----

There is a possibility that during a conversational session a hardware or software failure will kill LCC and/or TSS and break off your conversation. In that case LCC will lose all of its temporary records of your interactions, which would normally include all of your delayed steps and all "values" which had been assigned to your variables as well as all the stacked information on the status of your program's execution at the instant of the system failure. The values and the execution information will be irretrievably lost, but LCC includes a special feature to save your delayed steps, thus lessening the catastrophic effects of the system crash.

This feature is the 'automatic reload file', a file on which your delayed steps are automatically saved while your conversation progresses. If there are no system failures during your session, this file will be deleted when you log off (unless you explicitly retain it with an 'OFF SAVE' statement), but if the system fails, the file will not be deleted and thus will be available for reloading when you next call LCC. Each time you call LCC, a check will be made to determine whether your automatic reload file exists. If it does not, nothing is done, but if it does, you will be given control after the message

AUTOMATIC RELOAD? Y OR N

You then have the option either to restore your delayed steps by loading the file (by typing 'Y' and pressing the RETURN key or by merely pressing RETURN) or to ignore the file and delete it (by typing 'N' and pressing RETURN). Steps will be added to the reload file in sets of 5 in the order you type them; thus you may lose your last five typed steps after a crash, but no more. Remember that no values or context information will be automatically kept, so you may have to perform a lot of initialization to resume execution from the point of the crash.

---- Standard Functions ----

The standard functions which are included in LCC as predefined procedures are listed below. Each requires as an argument (ARG) one actual parameter which must evaluate to a number. The arguments of the trigonometric functions (and the results of the inverse trigonometric functions) must be in radians.

Name	Function	Definition
ABS	Absolute value	| ARG |
ARCCOS	Arccosine	arccos(ARG)
ARCSIN	Arcsine	arcsin(ARG)
ARCTAN	Arctangent	arctan(ARG)
COS	Cosine	cos(ARG)
COTAN	Cotangent	cotan(ARG)
ENTIER		largest integer ≤ ARG
EXP	Exponential	e ↑ ARG
LN	Natural logarithm	ln(ARG)
LOG	Common logarithm	$\log_{10}$(ARG)
SGN	Sign	IF ARG > 0 THEN 1 ELSE IF ARG < 0
SIGN	Sign (same as SGN)	THEN -1 ELSE 0
SIN	Sine	sin(ARG)
SQRT	Square root	ARG ↑ (1/2)
TAN	Tangent	tan(ARG)

---- Built-in LCC Functions and Procedures ----

The special functions and procedures which are included in the LCC system are described below. To use the name of a standard or built-in function as a variable, you must declare it as 'NEW'. The function's original meaning will then be lost for as long as your declaration is in effect. If you declare one of these identifiers on level zero, its original meaning will be lost for the duration of your conversational session unless you reinitialize your LCC environment by executing a 'DELETE ALL' statement.

COLLATE (arg)

Arg must be an expression which evaluates to a string. The value of the function COLLATE is an integer associated with the leftmost character of the value of arg. A unique integer is returned for each valid LCC character, and the integers will be ordered according to the System/360 EBCDIC collating sequence for the associated characters. The space or blank character comes first in the collating sequence and thus has the smallest associated integer. The other valid LCC characters are listed below in order of ascending collating sequence (left to right and top to bottom).

≠ ∧ ∨ [↑ ↓ → ← ∃] . < (+ | • { } ≤ ≥ ⊢ ⊣ ⊂ ⊃ ∀ !

$ *) ; ~ − / ∈ ⍵ , ∆ _ > ? ≡ ` ´ ¨ ' ⍎ : ⍱ ⍲ ∨ = \

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

Examples: The following steps define a function ALPHA which returns the value TRUE if the first character in its argument string is alphabetic (lower or upper case); otherwise it returns FALSE.

```
1.5: ALTER 1.6, 'LL'←COLLATE('|A'), 'UL'←COLLATE('Z');
1.6: ALPHA ← ∨ (X) ((X ← COLLATE(X)) ≥ LL) ∧ (X ≤ UL) ∨;
```

EE

This parameterless function has as a constant value the base of the natural logarithms, i.e., 2.718281828 ... Its value is as accurate as is possible in a System/360 double-word.

EXTERNAL (arg)

This procedure allows you to temporarily add to your LCC environment a non-LCC procedure or function which is to be called from your LCC program. Its argument must be a pointer to the name of the procedure or function to be added (e.g., ⇒ NAM). The effect of EXTERNAL is temporary and lasts only until you log off or reinitialize with a 'DELETE ALL' statement.

The external procedure or function to be added must satisfy the standard TSS (FORTRAN) linkage conventions and its name must appear as an entry point in one of your effective TSS job-library stack members. The value which it returns (if any) must be a double-word number placed in floating-point register zero. All FORTRAN double-precision library functions which do not involve arrays satisfy these conditions and are acceptable EXTERNAL functions. Any other experimentation is at your own risk.

Examples: The following statements indicate to LCC that you wish to use the FORTRAN procedures 'DSIN' and 'DCOS'.

```
EXTERNAL( ⇒DSIN );
EXTERNAL( ⇒DCOS );
```

INTERNAL (arg_1 , arg_2)

This procedure should not be called by a normal user. Its name is included here merely to forestall possible naming conflicts.

LENGTH (arg)

Arg must be an expression which evaluates to a string. The function LENGTH will have as its value the length (in number of characters) of that string.

Examples:
The value of LENGTH('XYZ') is 3.
The value of LENGTH(S ¤ 1234), where S = 'CMU', is 7.

PI

This parameterless function has as a constant value the mathematical constant pi, i.e., 3.141592653 ... Its value is as accurate as is possible in a System/360 double-word.

SCANN (arg_1 , arg_2 , arg_3)

SCANN is a procedure which scans a string to obtain its first atomic element. Its first argument (arg_1) must be an expression which evaluates to a string, and arg_2 and arg_3 must be pointers (i.e., ⊃V and ⊃W, where V and W are arbitrary variables). SCANN will search the string supplied by arg_1 for its first (leftmost) atom. It will store that atom into the variable pointed to by arg_2 (i.e., V), and it will store into W a string consisting of everything from arg_1 which is to the right of its first atom.

For scanning purposes, an atom is one of the following:

1. A contiguous string of alphabetic and/or numeric characters (e.g., 'ABCD', '345', 'P42G', '64AB2').

2. A single non-alphanumeric character (e.g., '*', '.', '-', '(', ';').

Blanks which precede an atom will be ignored, and an atom will be terminated by a blank, another atom, or the end of the string which contains it.

Examples: The step

SCANN(' AB +ABC*DE', ⊃L, ⊃R); SCANN(R, ⊃LL, ⊃RR)

will set L to 'AB', R to ' +ABC*DE', LL to '+', and RR to 'ABC*DE'.

SPLITT (arg_1 , arg_2 , arg_3 , arg_4)

SPLITT is a function which searches a string (of atoms) for a

specified substring. Its value will be TRUE if the substring can be found or FALSE if it cannot. Its first two arguments must be expressions which evaluate to strings, and its last two arguments must be pointers (i.e., ⊃V and ⊃W, where V and W are arbitrary variables). SPLITT will treat both strings as sequences of atoms (see the SCANN procedure above) and, searching from left to right, it will attempt to find a sequence of atoms in arg_2 which matches the atomic sequence arg_1. If such a sequence is found, SPLITT will return the value TRUE and, as side effects, it will store all of arg_2 to the left of the match into the variable pointed to by arg_3 (i.e., V), and it will store everything to the right of the match into W. If no matching subsequence is found, V and W will be left unchanged.

Note that the matching done by SPLITT is atom-by-atom rather than character-by-character. This means that the character string arg_1 need not be contained exactly in arg_2 to obtain a match, though it must be except for blanks which may surround atoms (i.e., the strings 'A+B', ' A +B', 'A + B' are all equivalent in this atomic sense). Effectively then, all extraneous blanks in arg_1 are deleted before the match is performed, and arg_2 cannot be searched for sequences of blanks.

Examples: The operand

```
SPLITT('AB', 'ABC:AB←AB+1', ⊃L, ⊃R)
```

has the value TRUE and it sets L to 'ABC:' and R to '←AB+1'. The operand

```
SPLITT('3 . 4  ', '3.4 :A + B', ⊃LL, ⊃RR)
```

has the value TRUE and it sets LL to '' (the null string) and RR to ' :A + B'.

---- Example LCC Conversation ----

```
    Δ THIS IS THE RECORD OF AN ACTUAL CONVERSATION BETWEEN A USER
    Δ    (AT A REMOTE 2741 TYPEWRITER) AND THE LCC SYSTEM.

    Δ THE FOLLOWING ARE NUMBERS (LITERAL NUMERIC CONSTANTS) IN LCC:

    15
15
    7.36
7.36
    .00065
.00065
    1234567890.
1234567890

    Δ WE CAN APPEND AN EXPONENT TO GET LARGER (OR SMALLER) NUMBERS:

    6.2₁₀12
.62₁₀+13
    3.721₁₀-5
.00003721
    6.35₁₀-42
.635₁₀-41
    12345.2₁₀+65
.123452₁₀+70

    Δ AN EXPONENT ALONE IS ALSO A NUMBER.

    ₁₀-4
.0001
    ₁₀15
.1₁₀+16

    Δ NUMBERS ARE OPERANDS WHICH CAN BE COMBINED INTO EXPRESSIONS,
    Δ    USING THE UNARY PREFIX OPERATORS (WHICH ARE WRITTEN TO THE
    Δ    LEFT OF AN OPERAND):
    Δ          -      NEGATE
    Δ          +      (HAS NO EFFECT)
    Δ          ⌊      TRUNCATE (STRIP OFF THE FRACTIONAL PART)

    Δ    AND THE BINARY INFIX OPERATORS (WRITTEN BETWEEN TWO OPERANDS):
    Δ          +      ADD
    Δ          -      SUBTRACT
    Δ          *      MULTIPLY
    Δ          /      DIVIDE
    Δ          ↑      RAISE TO A POWER

    Δ IF WE TYPE IN AN EXPRESSION, LCC WILL EVALUATE IT AND TYPE BACK
    Δ    THE RESULT.  THUS WE CAN USE LCC TO PERFORM 'DESK CALCULATOR'
    Δ    OPERATIONS.
    Δ LET'S TRY A FEW EXPRESSIONS TO SEE WHAT WILL HAPPEN.
```

```
      2+2
4
      3*8
24
      -5
-5
      2345-876
1469
      1/3
.3333333333
      2/7
.2857142857
      2↑5
32
      2↑32
4294967296
      2345.6789↑     ∆ I GOOFED. TO CANCEL THIS LINE I'LL TYPE ¤ AND RETURN
ERROR SX03    |
      ∆ I GOOFED AGAIN -- I HIT THE RETURN KEY FIRST INSTEAD OF THE `¤'
      ∆    KEY, SO LCC TRIED TO TRANSLATE THE LINE.  ITS TRANSLATOR FOUND
      ∆    THAT I HAD A MISSING OPERAND, WHICH I ALREADY KNEW.
      ∆    I'LL TRY IT AGAIN ON THIS LINE -- ¤
∆ LCC IGNORED THAT LINE AND MERELY UNLOCKED THE KEYBOARD TO LET ME
      ∆    TYPE ANOTHER ONE.  LCC WILL NEVER INDENT AFTER A CANCELLED
      ∆    LINE.  EITHER A `¤' OR A `/' WILL CANCEL A LINE, BUT TO DO
      ∆    SO IT MUST BE TYPED IMMEDIATELY BEFORE A CARRIER RETURN.
      ∆    AN EMBEDDED `¤' OR `/' HAS NO SUCH CANCELLATION PROPERTIES.
      ∆ LCC WILL ALSO IGNORE BLANK LINES AND ANY LINES (SUCH AS THESE)
      ∆    WHICH BEGIN WITH A DELTA (∆).  THUS COMMENT LINES MAY BE
      ∆    TYPED WITHOUT ANY ANALYSIS FROM THE LCC SYSTEM.
      NOTE THAT IF I FORGET THE `∆' ON A COMMENT LINE, LCC WILL OBJECT.
ERROR SX04 |
      ∆ IT SAYS `THAT' ISN'T AN OPERATOR, WHICH IS CERTAINLY TRUE.  AN
      ∆    ENGLISH SENTENCE DOESN'T USUALLY TURN OUT TO BE A VALID
      ∆    LCC STATEMENT.

      ∆ IF YOU MAKE AN ERROR AND NOTICE IT BEFORE YOU SEND THE LINE TO
      ∆    LCC (I.E., BEFORE YOU HIT THE RETURN KEY), YOU CAN CORRECT
      ∆    THE ERROR BY BACKSPACING TO THE LEFTMOST BAD CHARACTER AND
      ∆    RETYPING IT AND ALL THE CHARACTERS WHICH FOLLOWED IT.  ANY
      ∆    CHARACTERS BACKSPACED OVER (NOT JUST THE LEFTMOST ONE) WILL
      ∆    BE DELETED FROM THE LINE.  I'LL SHOW YOU AN EXAMPLE:
      12.34,56   THE `,' SHOULD BE A `+'.  I'LL BACKSPACE AND RETYPE IT.
          +56    ∆ I UPSPACED MANUALLY TO AVOID STRIKEOVERS.
68.34
      ∆ STRIKEOVERS WON'T BOTHER LCC, BUT I WOULDN'T BE ABLE TO READ
      ∆    WHAT I TYPED.

      ∆ NOW LET'S TRY SOME MORE EXPRESSIONS.

      ↓2345.876
2345
      +345
345
```

```
    234 + 12.5 * 54.2 / 6_3 - 2
232.1129167

    Δ UNARY OPERATIONS ARE NORMALLY DONE BEFORE ↑'S, WHICH ARE DONE
    Δ    BEFORE * AND /, WHICH IN TURN ARE DONE BEFORE + AND -.
    Δ    HOWEVER, WE CAN CHANGE THIS IMPLICIT HIERARCHY OF OPERATIONS
    Δ    BY USING PARENTHESES.

    12.78 * (92.5 / .341 - .00058) ↑ (3 * .788)
7228636.11
    Δ THIS WAS DONE AS IF IT HAD BEEN WRITTEN
    12.78 * ( ( ( 92.5 / .341) - .00058) ↑ (3 * .788) )
7228636.11

    Δ BESIDES THE UNARY AND BINARY OPERATORS WE CAN USE SOME OF THE
    Δ    STANDARD MATHEMATICAL FUNCTIONS SUCH AS
    Δ              SQRT     SQUARE ROOT
    Δ              SIN      SINE          (ARGUMENT IN RADIANS)
    Δ              COS      COSINE        (ARGUMENT IN RADIANS)
    Δ              LN       LOGARITHM     (BASE E)
    Δ              EXP      EXPONENTIAL   (E↑ARGUMENT)
    Δ              ARCTAN   ARCTANGENT    (ANGLE IN RADIANS)

    Δ LET'S TRY A FEW OF THEM.

    SQRT(3)
1.732050808
    SQRT(234)
15.29705854
    SIN(5)
-.9589242747
    LN(2)
.6931471806
    EXP(1)
2.718281828

    Δ THUS FAR IN THIS CONVERSATION, NO VALUES HAVE BEEN RETAINED BY
    Δ    LCC, BUT IF WE WISH TO KEEP A COMPUTED NUMERIC VALUE, WE CAN
    Δ    STORE IT INTO A VARIABLE.  VARIABLES ARE DESIGNATED BY
    Δ    IDENTIFIERS, WHICH YOU CAN CHOOSE FREELY (EXCEPT FOR LCC
    Δ    KEYWORDS LIKE 'TYPE' AND 'IF', WHICH HAVE SPECIAL MEANINGS).
    Δ    AN IDENTIFIER MUST BEGIN WITH A LETTER AND IT CAN CONTINUE
    Δ    WITH LETTERS, DIGITS, OR UNDERLINE (_) CHARACTERS.  IDENTIFIERS
    Δ    CAN BE AS LONG AS YOU LIKE, BUT LCC WILL IGNORE ANY CHARACTERS
    Δ    AFTER THE FIRST 8.
    Δ I'LL PICK SOME IDENTIFIERS AND STORE VALUES INTO THEM.  NOTE THAT,
    Δ    UNLIKE ALGOL, LCC DOES NOT REQUIRE ME TO DECLARE AN IDENTIFIER
    Δ    BEFORE I USE IT.

    A ← 5 ; B ← 4 ; LCC ← 111868 ; FISH ← 0 ; NOVEMBER ← 18 ; A_B_C ← 35
    Δ WE CAN CHECK THE VALUES WHICH WERE STORED BY TYPING THEM OUT.

    TYPE A,B,LCC,FISH,NOVEMBER,A_B_C
5
```

```
4
111868
0
18
35
     ∆ NOW WE CAN USE THESE VARIABLES AS OPERANDS IN FURTHER CALCULATIONS
     A+B
9
     SQRT(B+FISH)
2
     LCC / NOVEMBER - (LCC * A_B_C)
-3909165.111

     ∆ WE CAN CHANGE THE VALUE OF A VARIABLE WHENEVER WE WISH:
     A ← -742.8 ; B ← B-1; FISH←34-B;      TYPE A, B,FISH
-742.8
3
31

     ∆ THE CONSTRUCTION      A ← 5      IS A STATEMENT, IN PARTICULAR, AN
     ∆    ASSIGNMENT STATEMENT.  THE 'TYPE' STATEMENT IS ANOTHER KIND OF
     ∆    STATEMENT WHICH CAUSES EACH OF A LIST OF EXPRESSION VALUES TO
     ∆    BE TYPED BACK TO US (ONE VALUE PER LINE).  WE CAN PUT MORE THAN
     ∆    ONE STATEMENT ON A LINE BY SEPARATING THE SUCCESSIVE STATEMENTS
     ∆    BY SEMICOLONS (AS ABOVE).  A SEMICOLON AFTER THE LAST STATEMENT
     ∆    ON A LINE IS OPTIONAL.
     ∆ WE CAN MAKE AN ASSIGNMENT INSIDE AN EXPRESSION, OR WE CAN BOTH
     ∆    TYPE AND ASSIGN IF WE WISH.

     T ← A / (C ← B - 1) + 100;  TYPE T,C
-271.4
2
     TYPE P ← LCC + 1
111869
     TYPE CAT ← DOG - 3;
ERROR UN01 DOG IS UNDEFINED
     ∆ THAT DIDN'T WORK BECAUSE I FORGOT TO GIVE A VLAUE TO THE VARIABLE
     ∆    DOG.  I'LL DO SO AND TRY AGAIN.  NOTE THE ERROR MESSAGE FROM
     ∆    LCC'S EXECUTOR, WHICH WAS UNABLE TO CONTINUE AFTER FINDING AN
     ∆    UNDEFINED VARIABLE.
     DOG ← ¯5
     TYPE CAT ← DOG - 3
99997
     I←J←K←L←M←N←0;  ∆ WE CAN ASSIGN A VALUE TO A WHOLE SET OF VARIABLES.
     TYPE I+J+K+L+M+N;  ∆ THEY WILL ALL BE ZERO.
0
     IJKLMNOPQRSTUVWXYZ ← 5; TYPE IJKLMNOP; ∆ LCC IGNORES THE REST.
5
     ∆ WE CAN TEST THE VALUES OF VARIABLES BY MEANS OF AN 'IF' STATEMENT.
     ∆    EXAMPLES ARE:

     IF A < B THEN TYPE 3 ELSE TYPE 0
3
     IF B*P ≠ LCC THEN TYPE 9999
```

```
9999

     ∆ IF WE WANT TO PERFORM MORE THAN ONE ACTION DEPENDING ON A
     ∆    CONDITION, WE CAN COMBINE A SET OF STATEMENTS INTO A SINGLE
     ∆    COMPOUND STATEMENT VIA THE STATEMENT BRACKETS  {  AND  }.
     ∆ THUS WE CAN TYPE:

     IF A/B ≤ P THEN { T ← 3 ; W ← 4 ; TYPE T+W };
7
     IF T = P THEN IF A ≠ B THEN TYPE 3 ELSE TYPE 4 ELSE TYPE 5
5
     ∆ NOTE THAT ANY STATEMENT (EVEN AN IF STATEMENT) CAN FOLLOW A
     ∆    'THEN' (OR AN 'ELSE').

     ∆ SO MUCH FOR THE BASIC 'DESK CALCULATOR' FEATURES OF LCC.  SUPPOSE
     ∆    WE WISH TO WRITE A PROGRAM AND STORE IT INSIDE LCC.  THUS FAR
     ∆    IN THIS CONVERSATION, NONE OF OUR STATEMENTS HAVE BEEN KEPT
     ∆    AFTER BEING EXECUTED, THOUGH LCC HAS SAVED THE VALUES WHICH WE
     ∆    ASSIGNED TO OUR VARIABLES.  WE CAN SAVE STATEMENTS WHICH ARE
     ∆    TO BE CALLED OUT LATER FOR EXECUTION BY GIVING THEM 'STEP
     ∆    NUMBERS' WHICH BOTH IDENTIFY THEM FOR OUR FUTURE USE AND ALLOW
     ∆    LCC TO ORDER THEM PROPERLY.  AS AN EXAMPLE, LET'S WRITE A
     ∆    SIMPLE PROGRAM TO COMPUTE FACTORIALS.

     3.1:  FACT ← 1;

     ∆ THE STEP NUMBER, 3.1, CAN BE SEPARATED INTO TWO PORTIONS, THE
     ∆    INTEGER PORTION, WHICH IS THE 'PART NUMBER', AND THE FRACTIONAL
     ∆    PORTION.  SINCE THE INTEGER PORTION IS 3, THIS STEP IS STORED
     ∆    IN PART 3, AND THE FRACTION INDICATES ITS POSITION RELATIVE TO
     ∆    OTHER STEPS IN PART 3. PART NUMBERS MUST BE BETWEEN 1 AND 9999,
     ∆    AND THE STEP FRACTION MUST BE BETWEEN .0001 AND .9999.  LEADING
     ∆    ZEROS IN THE PART NUMBER AND TRAILING ZEROS IN THE FRACTION MAY
     ∆    BE OMITTED.
     ∆ LET'S GO ON WITH OUR PROGRAM.

     3.2000:  FACT ← FACT * N;  ∆ WE'LL COMPUTE N! AND PUT IT INTO FACT.
     3.3:  IF N = 1 THEN RETURN ;
     3.40:  N ← N - 1 ;
     3.5:  GO TO 3.3;  ∆ WE CAN TRANSFER CONTROL TO A NUMBERED STEP.

     ∆ NOW LET'S SEE WHAT PART 3 LOOKS LIKE.
     DISPLAY PART 3 ;  ∆ THIS WILL TYPE OUT THE STEPS IN PART 3.

3.1:  FACT ← 1;
3.2:  FACT ← FACT * N;  ∆ WE'LL COMPUTE N! AND PUT IT INTO FACT.
3.3:  IF N = 1 THEN RETURN ;
3.4:  N ← N - 1 ;
3.5:  GO TO 3.3;  ∆ WE CAN TRANSFER CONTROL TO A NUMBERED STEP.

     ∆ NOW I'LL GIVE A VALUE TO N AND CALL PART 3.  EXECUTION WILL BEGIN
     ∆    WITH STEP 3.1 AND PROCEED TO SUCCESSIVELY HIGHER NUMBERED STEPS
     ∆    UNLESS WE EXPLICITLY TRANSFER CONTROL WITH A 'GO TO' STATEMENT.
```

```
    N ← 5; PART 3
    TYPE FACT
5
    ∆ HMMM...  THAT'S NOT 5! --- I GUESS I HAVE A BUG.
    ∆ OH, YES; STEP 3.5 SHOULD GO TO 3.2.  I'LL CHANGE IT BY RETYPING
    ∆    STEP 3.5.  THAT WILL ERASE THE OLD STEP AND REPLACE IT BY MY
    ∆    NEW ONE.

    3.5:  GO TO 3.2 ;
    ∆ NOW TRY AGAIN.
    N←5 ; PART 3
    TYPE FACT
120
    ∆ THAT'S BETTER.  LET'S FIX STEP 3.3 SO IT WILL RETURN THE VALUE
    ∆    OF FACT.

    ALTER STEP 3.3 : 'RETURN' ← 'RETURN FACT'

    ∆ THAT CHANGED THE TEXT OF STEP 3.3 BY SUBSTITUTING ONE STRING FOR
    ∆    ANOTHER.  THE KEYWORD 'STEP' WAS OPTIONAL IN THIS ALTER
    ∆    STATEMENT, AND I COULD HAVE USED A ',' IN PLACE OF THE ':'.
    DISPLAY STEPS 3.3 TO 3.5;  ∆ LET'S CHECK THE TAIL END OF OUR PART.

3.3:  IF N = 1 THEN RETURN FACT ;
3.4:  N ← N - 1 ;
3.5:  GO TO 3.2 ;

    ∆ LOOKS O.K.  A FURTHER WORD ABOUT THAT DISPLAY STATEMENT -- IN
    ∆    SPECIFYING A GROUP OF ONE OR MORE STEPS OR PARTS, THE KEYWORDS
    ∆    'STEP' AND 'STEPS' ARE EQUIVALENT EVERYWHERE IN LCC, AS ARE
    ∆    'PART' AND 'PARTS'.  MOREOVER, IN MOST CASES, SUCH AS THIS
    ∆    ONE, THE KEYWORD 'STEP' MAY BE OMITTED.  THUS I COULD JUST
    ∆    AS WELL HAVE SAID
    ∆                    DISPLAY STEP 3.3 TO 3.5
    ∆    OR              DISPLAY 3.3 TO 3.5
    ∆ NOW I'LL TRY PART 3 AGAIN.
    N←6;PART 3
720
    N←10;PART 3
3628800
    N←0;PART 3;
!
ATTN AT 3.2
       ∆ THAT WENT INTO A LOOP, AND I HAD TO HIT THE 'ATTN' KEY TO GET
       ∆    OUT OF IT.  I GUESS THE PROGRAM IS STILL BUGGY.
       ∆ I'LL THINK ABOUT IT.   *   *   *   *   *
       TYPE FACT,N ;  ∆ I WONDER WHAT MY VARIABLES ARE NOW?
0
-15
       ∆ OH, I SEE -- PART 3 WON'T WORK FOR ANY VALUES LESS THAN 1.
       ∆    I'LL FIX IT BY ADDING ANOTHER STATEMENT.
       3.15:  IF N ≤ 0 THEN RETURN FACT ;
       N ← 0; PART 3;   ∆ TRY AGAIN.
1
```

```
        ∆ THAT'S MUCH BETTER.  NOTE, HOWEVER, THAT I STILL HAVEN'T GOTTEN
        ∆    OUT OF MY ORIGINAL LOOP (YOU CAN TELL BY THE INDENTATION - 7
        ∆    SPACES INSTEAD OF 4).  I CAN SAY 'GO', WHICH WILL GO ON FROM
        ∆    THE POINT WHERE I HIT 'ATTN', BUT THAT WON'T DO MUCH GOOD.
        ∆    I'LL TRY IT ANYWAY TO SHOW YOU.
        GO
!
ATTN AT 3.2
        ∆ YOU SEE, I'M BACK IN THE LOOP AGAIN.  TO GET OUT, I'LL FORCE AN
        ∆    END TO PART 3 BY GOING TO STEP 3.15.
        GO TO 3.15
0
    ∆ FACT STILL HAS THE VALUE OF ZERO BECAUSE IT WAS ERRONEOUSLY
    ∆    MULTIPLIED BY THE ZERO VALUE OF N.  NOTE ALSO THAT N HAS BEEN
    ∆    COUNTED DOWN AGAIN BY THE LOOP.
    TYPE N
-5
    ∆ WE CAN HAVE PART 3 ASK US FOR A VALUE OF N BY USING A REQUEST
    ∆    STATEMENT.
    3.05:  ?N
    PART 3
AT 3.05   N ←5;  ∆ I'LL SET N TO 5.
120
    ∆ WE CAN INCLUDE OUR OWN MESSAGE IN THE REQUEST BY PUTTING A STRING
    ∆    BETWEEN THE QUESTION MARK AND THE VARIABLE NAME (N).
    3.05:  ? 'TYPE N FOR N!' N
    PART 3
TYPE N FOR N! 4
24

    ∆ WE CAN USE ANOTHER PART TO CALL PART 3 REPEATEDLY.  WE'LL USE
    ∆    PART 25.  LET'S USE A 'NUMBER' STATEMENT TO GENERATE THE STEP
    ∆    NUMBERS AUTOMATICALLY.

    NUMBER AS 25 BY .1

25.1:  PART 3
25.2:  ? '   TYPE 1 TO GO ON, 0 TO STOP   ' FLAG;
25.3:IF FLAG = 1 THEN GO TO 25.1;
25.4:
    ∆ THE AUTOMATIC NUMBERING IS TURNED OFF BY PRESSING THE RETURN KEY
    ∆    IMMEDIATELY AFTER THE STEP NUMBER IS TYPED TO US.
    PART 25;  ∆ NOW CALL OUR PROGRAM.
TYPE N FOR N!  1
1
   TYPE 1 TO GO ON, 0 TO STOP     1
TYPE N FOR N!  6
720
   TYPE 1 TO GO ON, 0 TO STOP     1
TYPE N FOR N!  0
1
   TYPE 1 TO GO ON, 0 TO STOP     1
TYPE N FOR N!  8
40320
```

```
   TYPE 1 TO GO ON, 0 TO STOP     1
TYPE N FOR N!  2.4
!
ATTN AT 3.4
       ∆ OH, OH -- I'M IN A LOOP AGAIN.  I'LL PLANT A 'PAUSE' STATEMENT
       ∆    INSIDE IT TO SEE WHAT IS HAPPENING.
       3.21: PAUSE ; ∆ THIS WILL GIVE ME CONTROL AFTER STEP 3.2 IS DONE.
       GO;  ∆ NOW I'LL GO ON WITH THE LOOP.
PAUSE AT 3.21
       TYPE FACT,N;  ∆ I'LL TAKE A LOOK AT THE VARIABLES.
15604.49567
-7.6
       GO ; ∆ IF I SAY GO, THE PROGRAM WILL GO THROUGH THE LOOP AGAIN.
PAUSE AT 3.21
       TYPE FACT,N
-134198.6628
-8.6
       ∆ AS YOU CAN SEE, OUR PROGRAM DOESN'T WORK FOR NON-INTEGERS.
       ∆    LET'S FIX IT BY TRUNCATING N WHEN WE ENTER PART 3.
       3.06,  N ← ⌊N;
       ∆ NOW TO GET RID OF THE PAUSE STATEMENT.  I'LL USE A 'DELETE'
       ∆    STATEMENT, WHICH WILL ERASE IT.
       DELETE STEP 3.21
       GO;  ∆ LET'S GO ON.
ERROR GO10 STEP 3.21 CHANGED; GO CANNOT BE USED
       ∆ OH,OH -- I FORGOT THAT I CAN'T CONTINUE NORMALLY AFTER I DELETE
       ∆    AN ACTIVE STEP.  THERE ARE A NUMBER OF WAYS TO RECOVER FROM
       ∆    THIS SITUATION, BUT THE SIMPLEST IS TO START OVER.  TO DO
       ∆    THAT WE HAVE TO GET OUT OF THE CURRENT PART CALLS, AND THE
       ∆    EASIEST WAY IS TO EXECUTE AN 'EXIT ALL' STATEMENT, WHICH
       ∆    WILL TAKE US BACK TO THE ORIGINAL USER STATE.  REMEMBER THAT
       ∆    CURRENTLY WE ARE IN PART 3, WHICH WAS CALLED FROM PART 25,
       ∆    WHICH WAS CALLED BY ME, SO OUR CONTROL NESTING DEPTH IS 2
       ∆    (I COULD THUS USE TWO SIMPLE 'EXIT' STATEMENTS INSTEAD OF
       ∆    THE 'EXIT ALL').
       ∆ IF WE AREN'T SURE WHAT OUR CURRENT CONTROL STATE IS, WE CAN
       ∆    FIND OUT BY MEANS OF A 'DISPLAY RETURN STEPS' STATEMENT,
       ∆    WHICH WILL LIST THE STEPS CURRENTLY BEING EXECUTED.  LET'S
       ∆    SEE WHAT OUR STATUS IS NOW.

       DISPLAY RETURN STEPS
 ***
3.21
25.1
***
       ∆ THE '***' INDICATES AN IMMEDIATE STEP, WHICH IMPLIES THAT WE,
       ∆    RATHER THAN A SAVED PROGRAM STEP, ARE IN THE CONTROL CHAIN.
       ∆    NOTE THAT WE ARE IN THE LIST TWICE; WE ARE IN CONTROL NOW
       ∆    (TOP ENTRY) AND WE CALLED PART 25, WHICH WOULD NORMALLY
       ∆    RETURN CONTROL TO US (BOTTOM ENTRY).  THE 'EXIT ALL',
       ∆    HOWEVER, ISN'T NORMAL; IT ERASES THE CONTROL CHAIN SO THAT
       ∆    CONTROL REVERTS TO THE ORIGINAL GLOBAL STATE WHERE ONLY A
       ∆    SINGLE '***' WOULD BE DISPLAYED.
       ∆ THE AMOUNT OF INDENTATION WHICH IS DONE BEFORE LCC GIVES UP
```

```
        ∆     CONTROL TO LET US TYPE A STATEMENT DEPENDS ON THE NUMBER OF
        ∆     TIMES WE ARE THEN IN THE CONTROL CHAIN, WHICH IS THE NUMBER
        ∆     OF '***' ENTRIES IN THE 'DISPLAY RETURN' LIST.  INITIALLY WE
        ∆     ARE ON USER LEVEL 1 (IN THE CHAIN ONCE) AND LCC WILL INDENT
        ∆     4 SPACES. FOR USER LEVEL 2, INDENTATION WILL BE 7, FOR LEVEL
        ∆     3 IT WILL BE 10, FOR LEVEL 4 IT WRAPS AROUND TO 1.  THERE-
        ∆     AFTER, FOR HIGHER NESTING LEVELS THE INDENTATION WILL FOLLOW
        ∆     THE SEQUENCE
        ∆                        4, 7, 10, 1, 4, 7, 10, 1, ...
        ∆ LET'S GO ON.

       EXIT ALL
    DISPLAY PART 3; ∆ LET'S SEE WHAT PART 3 LOOKS LIKE.

3.05:  ? 'TYPE N FOR N!' N
3.06:  N ← ⌊N;
3.1:  FACT ← 1;
3.15:  IF N ≤ 0 THEN RETURN FACT ;
3.2:  FACT ← FACT * N;  ∆ WE'LL COMPUTE N! AND PUT IT INTO FACT.
3.3:  IF N = 1 THEN RETURN FACT ;
3.4:  N ← N - 1 ;
3.5:  GO TO 3.2 ;

    PART 25 ; ∆ LOOKS FINE. NOW IT SHOULD WORK FOR ALL REAL VALUES OF N.
TYPE N FOR N!  2.4
2
   TYPE 1 TO GO ON, 0 TO STOP     1
TYPE N FOR N!  -34.8
1
   TYPE 1 TO GO ON, 0 TO STOP     0
    ∆ THAT'S ENOUGH OF THAT.  WE CAN NOW SAVE PART 3 ON A FILE FOR USE
    ∆    DURING SOME FUTURE INTERACTION SESSION.  I'LL PUT IT ON THE
    ∆    FILE 'FACT3'.
    SAVE PART 3 AS FILE 'FACT3'
    ∆ THAT CREATED A NEW FILE NAMED 'FACT3' AND STORED THE TEXT FROM
    ∆    PART 3 ON IT.  THE TEXT OF PART 3 WILL BE RESTORED IF WE LOAD
    ∆    'FACT3' (USING A 'LOAD' STATEMENT) DURING A FUTURE CONVERSATION
    ∆    WITH LCC.
    OFF;  ∆ LET'S LOG OFF AND END THIS SESSION.
ON LCC FROM 16:35:48 TO 17:17:12
CPU TIME USED: 00:00:06:86
```