# THE FORMAL DEFINITION OF THE BASIC LANGUAGE

John A. N. Lee
Department of Computer and Information Science
Technical Report 72B-1

University of Massachusetts
Amherst, Massachusetts 01002

# The formal definition of the BASIC language*

J. A. N. Lee

*Department of Computer Science, University of Massachusetts, Amherst, Massachusetts 01002, USA*

**This paper presents a proposal for the formal definition of the Basic Language (Dartmouth College, 1970), based on the method of definition developed by Lucas *et al.* (1968) and as extended by Lee and Wu (1969). This version of the formal definition of the Basic Language does not include consideration of either the semantics or syntax of MAT statements. Similarly, the definitions of some of the more esoteric features of certain implementations have been omitted pending the resolution of the fundamental features of the language.**

**In summary, the definition is divided into three parts:**

1. **The concrete syntax, the definition of the syntactic form of the language as used by the programmer.**
2. **The abstract syntax, the definition of the essential structural form of the language which is to be used as input to the interpreter, and**
3. **The definitions of instructions and functions which are used by the interpreter to 'execute' a Basic program.**

(Received March 1971)

## The method of definition

The block diagram shown in Fig. 1 indicates the relationships between the three parts of the formal definition of a language and the abstract machine which operates over these definitions. The abstract machine consists of three processors named specifically as the analyser, the translator and the interpreter. These processors, the definitions, a set of default conditions, the original text and the data are sufficient to represent the execution of a Basic program. The overall concept is similar to that reported by Lucas for the formal definition of PL/1. However, a distinction has been made between the interpreter (which may be regarded as a state transition function) and the definition of the instructions and functions which are used by the interpreter. Further, the definition of the abstract syntax is used as an input to both the translator and the interpreter. The original Lucas scheme did not relate the abstract syntax to the interpreter. It is felt by the author that this relationship is necessary since the predicates defined in the abstract syntax are used in the definitions of instructions and functions as the propositions in McCarthy conditional expressions.

## The analyser

Using the concrete syntax, the analyser verifies the validity of the concrete text and generates a parsed text which may be represented as either a syntactic tree or a phrase marker of the concrete text. Irrespective of the form of this parsed text, it is required that all of the phrases of the concrete text have been identified and are associated with their syntactic metacomponent symbols. Since the concept of a syntactic analyser is not peculiar to either the language or this method of formal definition, but is a well established process within the state of the art of computing (Cheatham and Sattley, 1964; Ingerman, 1966), such a processor is not described here.

## The translator

The translator which transforms the parsed text into the abstract text (that is, a text which conforms to the abstract syntax and takes into account the default conditions) is peculiar to the language in the Lucas system but has not been developed here since it is the author's opinion that a general translator may be developed which is applicable to all languages.

The default rules for Basic are:

1. All subscriptors of an array variable are assumed to have the limits

lower bound—scalar 0
upper bound—scalar 10

unless the upper bound has been explicitly specified in a DIM statement.

These bounds are inserted into the s-data-component of the state of the machine. Each variable in the program text has a corresponding component in this s-data-component which contains the attributes of that variable. Besides a pointer to the storage component of the state of the machine, in which the current value assigned to the variable is stored, the attributes of an array variable also contain the dimensions of the array. Since the lower bound of all array variables is zero, only the upper bounds are stored in the attribute data of each array variable. These bounds are utilised by the interpreter to determine the location of the current value of each element of the array in the storage part of the state of the machine.

2. Simple variables and array variables with the same name are to be distinguished. Since Basic permits the repetitious use of variable names for a simple variable (requiring only a single cell in the storage part of the machine) or as the name of a whole array, the translator must recognise these two types of elements and give them new distinct names so as to prepare an unambiguous text for processing by the interpreter. In the formal definition presented here, the opening parenthesis of the subscripting expressions is concatenated with the array name to form a new name which is distinct from the simple variable of the same name.

Thus a simple variable A and an array A are given the unique names A and A( respectively.

The abstract syntax of an array variable reference (either to fetch the value or to store a value in that location) conforms to the predicate is-array-variable which contains as its s-name component, the name of the array variable. Similarly, the object which represents a reference to a simple variable in the text of the program, contains a s-name component which contains the name of the variable. Additionally, the state of the machine contains a s-data-component which contains the attribute data pertinent to each variable (simple and array types). Amongst this attribute data is the location in the storage component of the machine of the current value assigned to that variable. In the case of an array variable, this location is the location of the first element in the array.

Within the s-data-component of the state of the machine, the attribute of each variable is selected by a selector of the form

*The work reported here was supported in part by the National Science Foundation, Office of Computing Activities Grant No. GJ-60.

  

s-name(var) where var is the unique name of the variable as generated from the rules outlined above. Thus the attributes of the simple variable A are selected by s-name(A) whereas those of an array A are selected by s-name(A().

3. A one-dimensional array (called a 'list' in the Basic manual, 1970) and a two-dimensional array (called a 'table' in the manual) may not have the same name. Or the number of subscriptors in an array variable reference must equal the number of dimensions specified in its corresponding DIM specification.

This condition cannot be recognised by a syntactic analyser since data is not transferred between recognised components or phrases of the string being analysed. Since the Basic language manual (1970) is very explicit on this point, and since this definition assumes that storage allocation can be performed prior to the 'execution' of the program, this error is detected in the translator.

In formalising other languages, special default or error conditions may be substituted, but depending on the manner of storage allocation, the processor in which the default or error may be handled may vary between the translator and the interpreter.

In FORTRAN as defined in the American National standards, for example, storage allocation is static and thus these conditions may be recognised in the translator. However, the default condition in FORTRAN would be to replace a non-existent subscript expression by the integer value 1, but to indicate an error if there are too many subscripts.

The storage part of the state of the machine consists simply of a list of internal values. During the translation of the program from its parsed text form to the abstract text form, all simple and array variables are assigned locations within this list and the internal representation of scalar zero is assigned to each element of that list.

4. The evaluated subscripts of an array variable are the integer parts of the expressions which form the subscripts. This is a feature of most implementations of the Basic language. In the formal definition presented here, it is assumed that the translator will insert the INT function into all subscript expressions and thus the interpreter instruction definitions do not have to contain a special function to extract the integer part of the evaluated subscript expression.

In terms of concrete text, the translator would transform the reference X(N/3 + K) into the reference X(INT(N/3 + K)).

5. All variables or elements of arrays are assumed to have initial values of scalar 0 prior to interpretation of the program. This feature has been included in this definition since most known compilers set storage to zero before execution of the program. This assumption also requires that the storage portion of the machine be statically defined. The questions of static or dynamic storage, or predefined values of the elements are among the problems which must be resolved during the process of standardisation of the Basic language. However, with the assumption of static storage assignment and predefined values of all elements of storage, the translator may organise the 'memory' thus relieving the interpreter of many problems.

6. In a FOR statement, the value of the increment in STEP clause is assumed to be +1 unless specified otherwise explicitly.

7. Any statement consisting only of a line number is assumed to be a null statement having no effect on either the flow of control of the program or the values of any variables or elements of arrays in the storage part of the machine.

Except in the case of the null statement, each statement in the concrete text of the program is represented in the abstract text by an object which contains s-line-no and s-st-name components, along with components which are peculiar to that statement.

In the case of a null statement, only the s-line-no component is present, which ensures that the text of the program contains a statement with the line number of the null statement. The representation of a null by the null object would not meet this requirement.

This default has been included instead of merely deleting the statement from the program to allow programmers to delete a statement from a program without requiring them to alter the line numbers in control statements which refer to the statement being deleted.

For example, the programs

```
010 X = X                    010
 . . .                        . . .
070 GO TO 010                070 GO TO 010
```

are equivalent.

This default is typical of the environment in which programs written in this language are expected to operate. That is, an environment of remote console, an interactive time-sharing.

Similarly, REM statements (remarks or comments) are converted into null statements by the translator, since control statements may transfer control to a REM statement.

8. Any non-null statement which does not contain a key word (always identified in Basic by the first three characters of the statement) is assumed initially to be a LET statement. This default is taken into account in the analyser rather than the translator since the definition of a LET statement permits this default. The set of acceptable key words is implicitly defined in the concrete syntax.

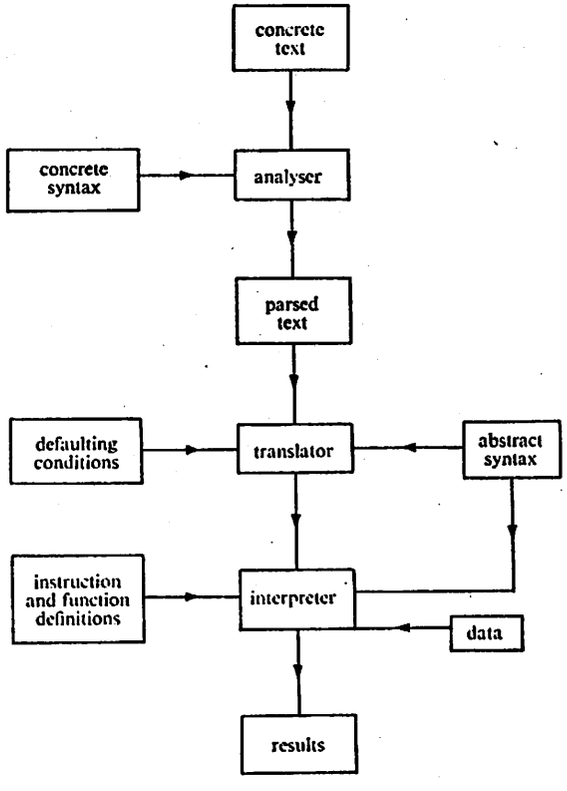


**Fig. 1. The abstract definition machine**

### The interpreter

The third processor in the abstract machine operates on the abstract text according to instructions and functions defined for the language. The state of the interpreter contains the abstract text of the program and an instruction stack from which the interpretation of the abstract text is directed. Instructions may consist of two types—self replacing and value returning. The

former instruction type replaces itself in the stack by one of its defined groups, chosen according to the condition currently existing in the state of the abstract machine. The execution of a value returning instruction causes changes in the state of the machine and the deletion of the instruction from the stack.

Initially the instruction stack contains the single instruction:

execute program(s-text($\xi$))

## Appendix 1. Concrete syntax

The meta language used in this concrete syntax definition is a modification of Backus Naur Form. Repetitive concatenations of objects are indicated by the notation $\{\ldots\}_i^j$ where $i$ is the minimum number of repetitions required and $j$ is the maximum number of repetitions permitted. Where either index is represented by a variable, the domain of the variable is the meta expression.

1. <statement> : = <line no> [<let st> | <read st> | <data st> | <print st> | <goto st> | <on st> | <if st> | <for st> | <next st> | <dim st> | <def st> | <gosub st> | <return st> | <restore st> | <stop st> | <rem st>]$_0^1$
2. <terminal st> : = <line no> <end st>
3. <program> : = [<statement>]$_0^n$ <terminal st>
4. * <line no> : = [<digit>]$_1^5$ [<blank>]$_0^1$
5. <expression> : = <multiply factor> [<prefix op> <expression> | <expression> [+|−]$_1^1$ <multiply factor>
6. <multiply factor> : = <multiply factor> [*|/]$_1^1$ <involution factor> | <involution factor>
7. <prefix op> : = +|−
8. <involution factor> : = <term> | <term> ↑ <term>
9. <term> : = <constant> | <variable> | <function ref> | (<expression>)
10. <variable> : = <simple variable> | <subscripted variable>
11. <simple variable> : = <letter> [<digit>]$_0^1$
12. <subscripted variable> : = <letter> (expression> {,<expression>}$_0^1$)
13. † <integer> : = [<digit>]$_1^9$
14. <fraction> : = .[<digit>]$_1^9$
15. <decimal> : = [<digit>]$_1^{\prime 9}$ . [<digit>]$_0^{9-\prime}$
16. <exponent> : = E [<sign>]$_0^1$ [<digit>]$_1^2$
17. <constant> : = <number> [<exponent>]$_0^1$
18. <number> : = <integer> | <fraction> | <decimal>
19. <digit> : = 0|1|2|3|4|5|6|7|8|9
20. <sign> : = +|−
21. <function ref> : = <function name> (<expression> {,<expression>}$_0^n$)
22. <function name> : = <library function> | <user function>
23. <user function> : = FN <letter>
24. <library function> : = SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT|RND
25. <dim st> : = DIM <array dim> {,<array dim>}$_0^n$
26. <array dim> : = <letter> (<integer> [,<integer>]$_0^1$)
27. <let st> : = [LET]$_0^1$ <variable> = <expression>
28. <signed number> : = [<sign>]$_0^1$ <constant>
29. <data list> : = <signed number> {,<signed number>}$_0^n$
30. <data statement> : = DATA {data list>}$_0^1$
31. <restore st> : = RESTORE
32. <goto st> : = GO TO <line no>
33. <on st> : = ON <expression> GO TO <line no> [,<line no>]$_1^n$
34. <gosub st> : = GOSUB <line no>
35. <return st> : = RETURN
36. <if st> : = IF <boolean expression> THEN <line no>
37. <boolean expression> : = <expression> <relation> <expression>
38. <relation> : = =|>|=|<=|>|<|<>
39. <for st> : = FOR <simple variable> = <expression> TO <expression> [STEP <expression>]$_0^1$
40. <next st> : = NEXT <simple variable>
41. <rem st> : = REM [<character>]$_0^n$
42. <stop st> : = STOP
43. <end st> : = END
44. <read st> : = READ <variable> {,<variable>}$_0^n$
45. <print st> : = PRINT [<print string>]$_0^1$ {<punct>}$_0^1$
46. <punct> : = ,|;
47. <message> : = "[<char>]$_0^n$"
48. <char> : = <letter> | <digit> | <special char>
49. <print string> : = <message> | <expression> | [<print string>]$_1^n$
50. <def st> : = DEF <blank> FN <letter> (<simple variable> {,<simple variable>}$_0^n$) = <expression> | <def block>
51. <def head> : = DEF <blank> FN <letter> (<simple variable> {,<simple variable>}$_0^n$)
52. <def end> : = FNEND
53. <function variable> : = FN <letter>
54. <def variable> : = <simple variable> | <subscripted variable> | <function variable>
55. <def statement> : = <line no> [<def let st> | <def read st> | <def print st> | <go to st> | <def on st> | <def if st> | <def for st> | <def next st> | <gosub st> | <return st> | <restore st> | <stop st> | <rem st>]$_0^1$
56. <def block> : = <line no> <def head> [<def statement>]$_0^n$ <line no> <def end>
57. <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
58. <special char> : = +|−|*|/|=|)|(|>|<|·|,|;| <blank> |$|↑|?

The statements defined by the meta component names of the form <def n st> have the same structure as the object <n st> except that <simple variable> may be replaced by <function variable> in all components. For example:

<def next st> : = NEXT [<simple variable> | <function variable>]$_1^1$

The specification of the punctuation of the elements of the PRINT statement cannot be expressed in a context-free-grammar since the quotation marks surrounding a literal string can be recognised as a separator. Thus the symbol ',' may be omitted in certain instances. The following four context-sensitive productions specify the permitted forms of punctuation in a print string:

<message> <expression> : = <message> [<punct>]$_0^1$ <expression>
<expression> <message> : = <expression> [<punct>]$_0^1$ <message>
<message> <message> : = <message> [<punct>]$_0^1$ <message>
<expression> <expression> : = <expression> <punct> <expression>

## Appendix 2. Abstract syntax

1. is-$\xi$ = (<s-text: is-text>, <s-stg: is-int-value-list>, <s-input: is-input>, <s-attr: is-attr>, <s-output: is-output>, <s-for-stack: is-for-stack>, <s-functions: is-def-group>)
2. is-text = ({<s-line(i): is-statement> | i ∈ line-no-set})
3. is-statement = is-null-st v is-end-st v is-read-st v is-print-st v is-goto-st v is-gosub-st v is-return-st v is-if-st v is-next-st v is-let-st v is-restore-st v is-for-st
4. is-input = (<s-key: is-intg>, <s-data: is-ext-value-list>)
5. is-attr = ({<s-name(var): is-variable-data> | var ∈ variable-name-set})
6. is-variable-data = is-simple-variable-data v is-function-variable-data v is-vector-data v is-table-data
7. is-function-variable-data = (<s-element: is-location>)
8. is-simple-variable-data = (<s-element: is-location>)
9. is-vector-data = (<s-upper-bound: is-dimension-1>, <s-element: is-location>)
10. is-table-data = (<s-upper-bound: is-dimension-1>, <s-element: (<s-upper-bound: is-dimension-2> <s-element: is-location>)>)
11. is-dimension-1 = is-intg
12. is-dimension-2 = is-intg
13. is-location = is-intg
14. is-output = is-ext-value v is-literal v is-punct v is-'CR'
15. is-for-stack = is-for-element-list
16. is-for-element = (<s-line-no: is-line-no>, <s-index: is-simple-variable v is function-variable> <s-limit: is-int-value>, <s-increment: is-int-value>)
17. is-def-group = ({<s-def(name): is-def> | name ∈ def-name-set})
18. is-def = (<s-par: is-int-value-list>, <s-exp: is-expression v is-def-block>)
19. is-def-block = ({<s-line(i): is-def-statement> | i ∈ line-no-set})
20. is-def-statement = is-let-st v is-read-st v is-print-st v is-goto-st v is-on-st v is-if-st v is-for-st v is-next-st v is-gosub-st v is-return-st v is-restore-st v is-stop-st v is-rem-st v is-null-st v is fnend-st

*Depending on the implementation, a blank character is optionally the trailing character to a line number. Otherwise blanks may be added to statements at will.

†In the following definitions, the maximum number of digits has been set in conformance with the implementation at the author's institution.

```
21. is-fnend-st = (<s-line-no: is-line-no>,
                   <s-st-name: FNEND>
                   <s-function-name: is-function-name>)
22. is-variable = is-simple-variable v is-array-variable v
                  is-function-variable
23. is-function-variable = (<s-name: is-function-name>)
24. is-simple-variable = (<s-name: is-simple-name>)
25. is-array-variable = (<s-name: is-array-name>,
                         <s-subscript-1: is-expression>,
                         <s-subscript-2: is-expression v is-Ω>)
26. is-expression = is-infix-expression v is-prefix-expression
                    v is-variable v is-constant
                    v is-function-ref v is-dummy-parameter
27. is-infix-expression = (<s-operand-1: is-expression>,
                           <s-operand-2: is-expression>,
                           <s-operator: is-infix-operator>)
28. is-prefix-expression = (<s-operand: is-expression>,
                            <s-operator: is-prefix-operator>)
29. is-infix-operator = is-'+' v is-'−' v is-'*'
                        v is-'/' v is-'↑'
30. is-prefix-operator = is-'+' v is-'−'
31. is-function-ref = (<s-name: is-function-name>,
                       <s-arg: is-expression-list>)
32. is-dummy-parameter = (<s-name: (<s-def: is-function-name>
                                   <s-var: is-intg>)>)
33. is-let-st = (<s-line-no: is-line-no>,
                 <s-st-name: LET>,
                 <s-variable: is-loadable>,
                 <s-expression: is-expression>)
34. is-null-st = (<s-line-no: is-line-no>)
35. is-rem-st = (<s-line-no: is-line-no>,
                 <s-st-name: REM>)
36. is-stop-st = (<s-line-no: is-line-no>,
                  <s-st-name: STOP>)
37. is-end-st = (<s-line-no: is-line-no>,
                 <s-st-name: END>)
38. is-goto-st = (<s-line-no: is-line-no>,
                  <s-st-name: GOTO>,
                  <s-destination: is-line-no>)
39. is-on-st = (<s-line-no: is-line-no>,
                <s-st-name: ON>,
                <s-exp: is-expression>,
                <s-line-no-list: is-line-no-list>)
40. is-if-st = (<s-line-no: is-line-no>,
                <s-st-name: IF>,
                <s-boolean: is-boolean>,
                <s-destination: is-line-no>)
41. is-boolean = (<s-exp-1: is-expression>,
                  <s-exp-2: is-expression>,
                  <s-relation: is-relation>)
42. is-gosub-st = (<s-line-no: is-line-no>,
                   <s-st-name: GOSUB>,
                   <s-destination: is-line-no>)
43. is-return-st = (<s-line-no: is-line-no>,
                    <s-st-name: RETURN>)
44. is-for-st = (<s-line-no: is-line-no>,
                 <s-st-name: FOR>,
                 <s-index: is-simple-variable v is-function-variable>,
                 <s-initial: is-expression>,
                 <s-limit: is-expression>,
                 <s-increment: is-expression>)
45. is-next-st = (<s-line-no: is-line-no>,
                  <s-st-name: NEXT>,
                  <s-index: is-simple-variable>)
46. is-restore-st = (s-line-no: is-line-no>,
                     <s-st-name: RESTORE>)
47. is-read-st = (<s-line-no: is-line-no>,
                  <s-st-name: READ>,
                  <s-load: is-loadable-list>)
48. is-loadable = is-variable
49. is-print-st = (<s-line-no: is-line-no>,
                   <s-st-name: PRINT>,
                   <s-output: is-print-list>)
50. is-print = is-expression v is-literal v is-punct
51. is-literal = is-char-list
52. is-punct = is-',' v is-';'
```

The following predicates are not defined explicitly in this section since the objects which satisfy the predicates are transformations of objects defined by the concrete system. If we define the function $A$ as the transformation function which converts a character string in the concrete text into a unique elementary object in the qualified text, then we may define these predicates in terms of concrete syntax components.

```
53. is-char = A(<char>)
54. is-simple-name = A(<simple variable>)
55. is-array-name = A(<letter>)
56. is-function-name = A(<function name>)
57. is-library-function = A(<library function>)
58. is-user-function = A(<user function>)
59. is-constant = A(<constant>)
60. is-line-no = A(<line no>
61. is-relation = A(<relation>)
62. is-intg = A(<integer>)
```

The two predicates is-ext-value and is-int-value are implementation defined, and represent the attributes of the representation of external and internal values respectively.

## Appendix 3. Interpreter of the Basic machine instructions and functions

```
1. execute-program(text) =
     execute-statement(next-line-no(text, 0), text, < >)
2. execute-statement(line-no, text, line-no-stack) =
     is-Ω(line-no) v is-Ω(s-line(line-no)(text)) → error
     is-Ω(s-st-name's-line(line-no)(text)) v
     is-REM(s-st-name's-line(line-no)(text)) →
       execute-statement(next-line-no(text, line-no), text, line-no-stack)
     is-END(s-st-name's-line(line-no)(text)) v
     is-STOP(s-st-name's-line(line-no)(text)) → null
     is-GOTO(s-st-name's-line(line-no)(text)) →
       execute-statement(s-destination's-line(line-no)(text),
                         text, line-no-stack)
     is-GOSUB(s-st-name's-line(line-no)(text)) →
       execute-statement(s-destination's-line(line-no)(text), text,
                         <next-line-no(text, line-no)> ∩line-no-stack)
     is-RETURN(s-st-name's-line(line-no)(text)) →
       execute-statement(head(line-no-stack), text, tail(line-no-stack))
     is-FOR(s-st-name's-line(line-no)(text)) →
       execute-for-st(s-line(line-no)(text), text, line-no-stack)
     is-NEXT(s-st-name's-line(line-no)(text)) →
       execute-next-st(s-for-stack(ξ): text, line-no, line-no-stack)
     is-LET(s-st-name's-line(line-no)(text)) →
       store(s-stg(ξ); location, value);
         location: get-location(s-variable's-line(line-no)(text)),
           value: evaluate-expression(s-expression's-line(line-no)(text))
     is-READ(s-st-name's-line(line-no)(text)) →
       execute-read-st(s-stg(ξ); line-no,
                       s-input(ξ), text, line-no-stack)
     is-RESTORE(s-st-name's-line(line-no)(text)) →
       restore-key(s-key's-input(ξ):)
     is-PRINT(s-st-name's-line(line-no)(text)) →
       execute-print-st(line-no, text, line-no-stack)
     is-IF(s-st-name's-line(line-no)(text)) →
       execute-statement(c, text, line-no-stack);
       c: compare-relation(a, b,
            s-relation's-boolean's-line(line-no)(text),
              s-destination's-line(line-no)(text),
                next-line-no(text, line-no));
          a: evaluate-expression(s-exp-1's-boolean's-line(line-no)(text)),
          b: evaluate-expression(s-exp-2's-boolean's-line(line-no)(text))
     is-FNEND(s-st-name's-line(line-no)(text)) →
       pass(result);
         result: fetch(s-stg(ξ), location);
           location: get-location(s-function-name's-line(line-no)(text))
     is-ON(s-st-name's-line(line-no)(text)) →
       int-on-st(a, s-line-no-list's-line(line-no)(text),
                 text, line-no-stack);
           a: evaluate-expression(s-exp's-line(line-no)(text))
3. execute-for-st(for-st, text, line-no-stack) =
     not-{}(s-index'elem(i, s-for-stack(ξ)) = s-index(for-st)) → error
     T → execute-statement(a, text, line-no-stack);
           a: compare(s-for-stack(ξ); init,
                      least(those-next(s-index(for-st),
                                       s-line-no(for-st)), text);
                store(s-stg(ξ); location, init);
                  stack(s-for-stack(ξ); s-line-no(for-st),
                        s-index(for-st), limit, inc);
                    location: get-location(s-index(for-st)),
                    init: evaluate-expression(s-initial(for-st)),
                    limit: evaluate-expression(s-limit(for-st)),
                    inc: evaluate-expression(s-increment(for-st))
4. stack(for-stack; line-no, variable, limit, inc) =
     I: <μ0(<s-line-no: line-no>,
            <s-index: variable>,
            <s-limit: limit>,
            <s-increment: inc>) < ∩for-stack
5. execute-next-st(for-stack; text, line-no, line-no-stack) =
     is-< > (for-stack) → error
     s-index's-line(line-no)(text) ≠ s-index(head(for-stack)) →
       execute-next-st(tail(for-stack); text, line-no, line-no-stack)
     T → execute-statement(b, text, line-no-stack);
         b: compare(for-stack; a, line-no);
            a: increment(for-stack)
6. increment(for-stack) =
       pass(x);
         store(s-stg(ξ); location, x);
           x: int-infix-expression(s-increment(head(for-stack)),
                                   value, +);
             value: fetch(s-stg(ξ), location);
               location: get-location(s-index(head(for-stack)))
```

```
7. compare(for-stack; value, line-no) =
     signum(s-increment(head(for-stack))) ×
       (value-s-limit(head(for-stack))) > 0 →
         PASS: next-line-no(s-text(ξ), line-no)
               1: tail(for-stack)
       T → PASS: next-line-no(s-text(ξ), s-line-no(head(for-stack)))
8. fetch(stg, location) =
     PASS: elem(location, stg)
9. store(stg; location, value) =
     elem(location): value
10. get-location(variable) =
     is-simple-variable(variable) →
       get-loc-1(s-name(s-name(variable)) 's-attr(ξ))
       T → get-loc-2(s-name(s-name(variable)) 's-attr(ξ), variable)
11. get-loc-1(symtab) =
     PASS: s-element(symtab)
12. get-loc-2(symtab, variable) =
     is-eo(s-element(symtab)) →
       (not-Ω(s-subscript-2(variable)) → error
       T → map-3(s-upper-bound(symtab), s-element(symtab),
                                                  subscript);
           subscript: evaluate-expression(s-subscript-1(variable)) )
     T → (is-Ω(s-subscript-2(variable)) → error
          T → map-4(s-upper-bound(symtab),
                    s-upper-bound's-element(symtab),
                      s-element's-element(symtab),
                        subscript-1, subscript-2);
       subscript-1: evaluate-expression(s-subscript-1(variable)),
       subscript-2: evaluate-expression(s-subscript-2(variable)) )
13. map-3(dim-1, location, subscript) =
     PASS: map-1(dim-1, location, subscript)
14. map-4(dim-1, dim-2, location, subscript-1, subscript-2) =
     PASS: map-2(dim-1, dim-2, location, subscript-1, subscript-2)
15. evaluate-expression(expression) =
     is-infix-expression(expression) →
       int-infix-expression(a, b, s-operator(expression));
         a: evaluate-expression(s-operand-1(expression)),
         b: evaluate-expression(s-operand-2(expression))
     is-prefix-expression(expression) →
       int-prefix-expression(a, s-operator(expression));
         a: evaluate-expression(s-operand(expression))
     is-variable(expression) →
       fetch(s-stg(ξ), location);
       location: get-location(expression)
     is-constant(expression) →
       pass(internal-rep(expression))
     is-function-ref(expression) →
       int-function-ref(expression)
     is-dummy-parameter(expression) →
       pass(elem(s-var's-name(expression)) 's-par'
            s-def(s-def's-name(expression)) 's-functions(ξ))
16. int-infix-expression(op-1, op-2, opr) =
     is-'+'(opr) → PASS: op-1 + op-2
     is-'−'(opr) → PASS: op-1 − op-2
     is-'*'(opr) → PASS: op-1 × op-2
     is-'/'(opr) → PASS: op-1 ÷ op-2
     is-'↑'(opr) → PASS: e^(op-2 × ln(op-1))
17. int-prefix-expression(opd, opr) =
     is-'+'(opr) → PASS: opd
     is-'−'(opr) → PASS: −opd
18. compare-relation(exp-1, exp-2, rel, destination, next-line) =
     is-'='(rel) & (exp-1 = exp-2) v
     is-'<'(rel) & (exp-1 < exp-2) v
     is-'>'(rel) & (exp-1 > exp-2) v
     is-'<='(rel) & (exp-1 ≤ exp-2) v
     is-'>='(rel) & (exp-1 ≥ exp-2) v
     is-'<>'(rel) & (exp-1 ≠ exp-2) → PASS: destination
     T → PASS: next-line
19. int-on-st(value, list, text, line-no-stack) =
     execute-statement(elem(compress(value, length(list)), list),
                       text, line-no-stack)
20. int-function-ref(exp) =
     is-library-function(s-name(exp)) &
       length(s-arg(exp)) = 1 →
         int-library-function(s-name(exp), x);
           x: evaluate-expression(head(s-arg(exp)))
     is-user-function(s-name(exp)) →
     int-function-def(s-exp's-def(s-name(exp))'s-functions(ξ));
         evaluate-args(s-par's-def(s-name(exp))'s-functions(ξ);
                       s-arg(exp), 1)
21. int-library-function(name,arg) =
     pass(name(arg))
22. int-function-def(block) =
     is-expression(block) →
       evaluate-expression(block)
     is-def-block(block) → execute-program(block)
23. evaluate-args(parameter-list; argument-list, n) =
     n > length(argument-list) → null
     T → evaluate-args(parameter-list; argument-list, n + 1);
           store-par(elem(n, parameter-list); a);
             a: evaluate-expression(elem(n, argument-list))
24. store-par(par; a) =
     1: a
25. execute-read-st(stg; line-no, input, text, line-no-stack) =
     execute-statement(next-line-no(text, line-no), text, line-no-stack);
     read(stg; input, s-line(line-no)(text), 1)
26. read(stg; input, read-st, n) =
     n > length(s-load(read-st)) → null
     T → read(stg; input, read-st, n + 1);
           store(stg; location, value);
             location: get-location(elem(n, s-load(read-st))),
             value: get-data-value(s-key's-input(ξ); s-date(input))
27. get-data-value(key; data-list) =
     key > length(data-list) → error
     T → PASS: internal-rep(elem(key, data-list))
               1: key + 1
28. restore-key(key;) =
     1: 1
29. execute-print-st(line-no, text, line-no-stack) =
     execute-statement(next-line-no(text, line-no), text, line-no-stack);
     print(s-output(ξ); s-output's-line(line-no)(text), 1)
30. print(output; print-st, n) =
     n > length(print-st) → add-to-output(output; 'CR')
     n = length(print-st) & is-punct(elem(n, print-st)) →
         add-to-output(output; elem(n, print-st))
     is-expression(elem(n, print-st)) →
         print(output; print-st, n + 1);
           add-to-output(output; a);
             a: evaluate-expression(elem(n, print-st))
     T → print(output; print-st, n + 1);
           add-to-output(output; elem(n, print-st))
31. add-to-output(output; element) =
     is-int-value(element) → 1: output ∩ <external-rep(element)>
     T → 1: output ∩ <element>
32. least(list) =
     is-< >(list) → error
     length(list) = 1 → head(list)
     head(list) < head(tail(list)) →
                  least(<head(list)> ∩ tail(tail(list)))
     T → least(tail(list))
33. those-next(index, line-no, text) =
     <(τi)(s-st-name's-line(i)(text) = NEXT &
       s-index's-line(i)(text) = index &
       i > line-no)>
34. compress(value, list-length) =
     value < 1 → 1
     value > list-length → list-length
     T → value
35. next-line-no(text, i) =
     i > (τj)(s-st-name's-line(j)(text) = END) → error
     is-Ω(s-line(i)(text)) → next-line-no(text, i + 1)
     T → i
```


References

ANON (1970). *BASIC*, Fifth Edition. Dartmouth College, Hanover, New Hampshire.

CHLATHAM, T., and SATTLEY, K. (1964). Syntax Directed Compiling, *Proc. S.J.C.C.*, pp. 31-57.

INGERMAN, P. Z. (1966). *A Syntax Oriented Translator*, New York: Academic Press.

LEE, J. A. N., and WU, D. (1969). The Vienna Definition Language: A Generalisation of Instruction Definitions. SIGPLAN Symposium on Programming Language Definition.

LUCAS, P. *et al.* (1968). Method and Notation for the Formal Definition of Programming Languages, Technical Report TR25.087. IBM Laboratory, Vienna.

COMPUTER AND INFORMATION SCIENCE

TECHNICAL NOTES

The following TECHNICAL NOTES are now available at the Computer and Information Science Department, Graduate Research Center except those marked with an asterisk.

| | |
|---|---|
| *TN/CS/00001 (Out of Date) | Current Research Toward the Standardization and Formal Definition of PL/I by John A.N. Lee (April 1,1968). |
| *TN/CS/00002 | Discrete Markov Chains: An Heuristic Approach by Sue N. Stidham (July 1,1968). |
| *TN/CS/00003 | The Domelki Syntactic Analysis Algorithm by Susan L. Gerhart (August 28,1968). |
| *TN/CS/00004 | A Survey of Hashing Techniques by John A.N. Lee (September 15,1968). |
| TN/CS/00005 | The Recognition and Use of Null Elements in a Syntax Directed Translator by John A.N. Lee (September 19,1968). |
| *TN/CS/00006 (Revision is TN/CS/00020) | SYNFUL, A Proposed General Purpose Translator System by John A.N. Lee (October 1,1968). |
| TN/CS/00007 | Sorting Almost Ordered Arrays by Caxton C. Foster (November 18,1968). |
| *TN/CS/00008 (Out of Date) | Vienna Definition Language--Semantics by John A.N. Lee (December 13,1968) |
| *TN/CS/00009 | An Examination of Two Hash Transforms by Caxton C. Foster (May 9,1969). |
| TN/CS/00010 | Multiplexing Without Tears by Caxton C. Foster (May 30,1969). |
| *TN/CS/00011 (Out of Date) | Vienna Definition Language--A Generalization of Instruction Definitions by John A.N. Lee and Delmore Wu (April,1969). |
| *TN/CS/00012 | An Unclever Time-Sharing System by Caxton C. Foster (October,1969). |
| *TN/CS/00013 | The Formal Definition of the Basic Language by John A.N. Lee (April,1970) (Also published in Computer Journal and as Technical Report 72B-1) |
| TN/CS/00014 | A Debugging Aid by Caxton C. Foster and Hugh C. Schulz (January 16,1970). |
| *TN/CS/00015 | Conditional Interpretation of Operation Codes by Caxton C. Foster and Robert H. Gonter (February,1970). |
| TN/CS/00016 | Some Simple Algorithms for Content Addressable Memories by Caxton C. Foster (July,1970). |
| TN/CS/00017 | A Data Distributor--The Sprinkler System by Caxton C. Foster (July,1970). |

*TN/CS/00018 An Annotated Bibliography on Syntax-Directed Translation by John A.N. Lee, Taveta K. Bogert, and Helen Gigley (July,1970).

TN/CS/00019 Chargoggaggoggmanchaugagoggchaubunagungamaug--A Novel Multiply-by-Three Circuit by Caxton C. Foster, Edward Riseman, Fred Stockton, and Conrad Wogrin (September,1970).

TN/CS/00020 SYNFUL--A General Purpose Translator System and Extensive Modification of Technical Note #00006 by John A.N. Lee and Helen Gigley, edited by Taveta K. Bogert (November,1970).

TN/CS/00021 Maintenance Manual for the UMASS Timesharing Version of SYNFUL by Ronald Lautmann (November,1970).

*TN/CS/00022 Microprogramming: A Design Alternative by Michael J. Sullivan (December,1970).

TN/CS/00023 A Simulated Associative Memory by Caxton C. Foster (December,1970).

TN/CS/00024 A Five Tape Algorithm for the Instant Playback Problem by Caxton C. Foster (December,1970).

TN/CS/00025 A Comparison of Simulation Languages by Albert W. Zukatis (January,1971).

TN/CS/00026 A Stack Oriented Computer by Caxton C. Foster (April,1971).

TN/CS/00027 Certain Formal Properties of the Vienna Definition Language by John A.N. Lee.

*TN/CS/00028 When the Chips are Down by Caxton C. Foster (January,1972).

TN/CS/00029 RAUPEDATA-11 -- A Sophisticated Debugging Program for the PDP-11 by Edward G. Fisher (February,1972).

TN/CS/00030 A Tutorial on Cobol Extensions to Handle Data Bases: The Data Base Group Report by Robert W. Taylor (February,1972).

*TN/CS/00031 System Design: Process Models by Richard H. Eckhouse, Jr. (April,1972).

TN/CS/00032 Automated Accounting Systems by Richard H. Eckhosue, Jr. (April,1972).

TN/CS/00033 A Generalization of AVL Trees by Caxton C. Foster (June,1972).

TN/CS/00034 Conditional Syntactic Specification by J. Dorocak and John A.N. Lee (September,1972).

TN/CS/00035 A Formal Definition of Mini Language Number 7,"Dynamic Type Checking" by Edward G. Fisher (October,1972).

TN/CS/00036 Data Administration, Data Independence, and the DBTG Report by Robert W. Taylor (October 1973).

COMPUTER AND INFORMATION SCIENCE

TECHNICAL REPORTS

The following TECHNICAL REPORTS are now available at the Computer and Information Science Department, Graduate Research Center, except those marked with an asterisk.

70C-2 Organizational Principles for Embryological and Neurophysiological Processes by Michael A. Arbib.

70C-3 On the Likely Evolution of Communicating Intelligence on Other Planets by Michael A. Arbib (August 1, 1970, revised December 30, 1970.)

70C-4 Contextual Error Detection by Roger W. Ehrich and Edward M. Riseman.

70C-5 Transformations and Somatotopy in Perceiving Systems by Michael A. Arbib.

70C-6 Organizational Principles for Theoretical Neurophysiology by Michael A. Arbib, (August 15, 1971).

71B-1 The Definition and Validation of the Radix Sorting Technique by John A. N. Lee, (January, 1972).

71B-2 Two Papers on Group Machines by Michael A. Arbib.

71B-4 Automata with Ranked State Sets by Dieter Schütt.

71C-6 Machines in a Category by M. A. Arbib and E. G. Manes.

72A-1 A Study of the Constraints upon the Parallel Dispatching and Execution of Machine Code Instructions by Caxton C. Foster and Edward M. Riseman.

72B-1 The Formal Definition of the Basic Language by John A. N. Lee.

72B-2 Decomposable Machines and Simple Recursion by Michael A. Arbib and E. Manes.

72C-1 A Contextual Postprocessing System for Error Detection and Correction in Character Recognition by Edward M. Riseman and A. Hanson (October 1972).

73B-1 Adjoint Machines, State-Behavior Machines, and Duality by Michael A. Arbib and Ernest G. Manes (January 1973).

73B-2 Natural State Transformations by Suad Alagić (February 1973). (Revised Nov.1973)

73C-3 A Model of Posited Decisionary and Learning Mechanisms in Mammalian CA-3 Hippocampus by William Kilmer, T. McLardy, and M. Olinski (February 1973).

73C-4 Four Faces of Hal: Using Artificial Intelligence Techniques in Computer-Assisted Instruction by Howard A. Peelle and Edward M. Riseman (March, 1973).

73C-5 System Design of an Integrated Pattern Recognition System, or How to Get the Best Mileage out of your Used Pattern Classifier by A.R. Hanson and E.M. Riseman, (June 1973).

73C-6 Neural Models of Spatial Perception and the Control of Movement, by Michael A. Arbib, C. Curt Boylls & Parvati Dev (June 1973).

73B-3 Foundations of System Theory, I by Michael A. Arbib and Ernest G. Manes, (July, 1973.)

73A-1 ULD and a Description of the PDP-8, by John A. N. Lee (September 1973).

73B-4 Time-Varying Systems, by Michael A. Arbib and Ernest G. Manes (November 1973).

73C-7 Model of a Plausible Learning Scheme for CA3 Hippocampus, by William Kilmer and Melanie Olinski, (Nov., 1973).

73B-5 Algebraic Aspects of Algol 68, by Suad Alagić (November 1973).

73C-8 Biology of Decisionary and Learning Mechanisms in Mammalian CA3-Hippocampus, by William Kilmer (November 1973).

73C-9 Eye Movements and Visual Perception: A "Two Visual System" Model by Richard L. Didday and Michael A. Arbib (December 1973).

73A-1.1 Basic Specifications, by John A.N. Lee, Steven R. Beckhardt, and Arthur I. Karshmer (July 1973).