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) 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 subscripts of an array variable are assumed to have the limits

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

lower bound—scalar 0
upper bound—scalar 10

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.

1. 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 subscribing 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

\[
\text{selector} \left[ \text{variable} \right] = \text{attribute}.
\]
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 subscripts 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
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 system. 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.

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

---

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

---

The Computer Journal
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(£))
```

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 \} where i is the
minimum number of repetitions required and j is the maximum
number of repetitions permitted. Where either index is represen-
ted by a variable, the domain of the variable is the meta
expression.

1. \langle statement \rangle := \langle if \ no \ \rangle | \langle let \ st \rangle | \langle read \ st \rangle | \langle data \ st \rangle |
2. \langle goto \ st \rangle | \langle on \ st \rangle | \langle if \ st \rangle | \langle print \ st \rangle |
3. \langle def \ st \rangle | \langle def \ next \ st \rangle | \langle def \ on \ st \rangle | \langle def \ if \ st \rangle |
4. \langle rem \ st \rangle | \langle null \ st \rangle | \langle fn-end \ st \rangle |
5. \langle print \ st \rangle | \langle read \ st \rangle | \langle data \ st \rangle | \langle def \ st \rangle |
6. \langle relation \rangle := = | > = | < = | > | < | < > |

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

```
\langle def \ next \ st \rangle := NEXT {\langle simple \ variable \rangle | \langle function \ variable \rangle}
```

The specification of the punctuation of the elements of the
PRINT statement cannot be expressed in a context-free-gram-
ar 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 punc-
uation in a print string:

```
\langle message \rangle := \langle message \rangle {< punct >} \langle message \rangle
```

Appendix 2. Abstract syntax

1. is-\langle s-text \rangle = is-\langle s-text \rangle
2. is-\langle s-key \rangle = is-\langle s-key \rangle
3. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
4. is-\langle def \ head \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
5. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
6. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
7. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
8. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
9. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
10. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
11. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
12. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
13. is-\langle def \ head \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
14. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
15. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
16. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
17. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
18. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
19. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
20. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
21. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
22. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
23. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
24. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
25. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
26. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
27. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
28. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
29. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
30. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
31. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
32. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
33. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
34. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
35. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
36. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
37. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
38. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
39. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
40. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
41. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
42. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
43. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
44. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
45. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
46. is-\langle def \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
47. is-\langle def \ next \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
48. is-\langle def \ on \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
49. is-\langle def \ if \ st \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)
50. is-\langle def \ group \rangle = (\langle s-def \ name \rangle, \langle s-def \ name \rangle)

\[
\text{is-def-statement} = \text{is-let-st} \vee \\
\text{is-read-st} \vee \\
\text{is-data-st} \vee \\
\text{is-print-st} \vee \\
\text{is-goto-st} \vee \\
\text{is-on-st} \vee \\
\text{is-if-st} \vee \\
\text{is-next-st} \vee \\
\text{is-rem-st} \vee \\
\text{is-null-st} \vee \\
\text{is-fn-end-st} \vee \\
\text{is-for-st}
\]
21. is-fnend-st = ( <s-line-no: is-line-no>, <s-st-name: FNEND>, <s-function-name: is-function-name>, <s-relation: is-relation>, <s-expression: is-expression>, <s-operand-1: is-function-operand-1>, <s-operand-2: is-function-operand-2> )
22. is-variable = is-simple-variable v is-array-variable v is-function-variable
23. is-function-variable = ( <s-name: is-function-name>, <s-subscript-1: is-subscript-1>, <s-subscript-2: is-subscript-2> )
24. is-simple-variable = ( <s-name: is-simple-name> )
25. is-array-variable = ( <s-name: is-array-name>, <s-index: is-simple-variable v is-function-variable> )
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-operator: is-prefix-operator>, <s-operand: is-expression> )
29. is-infix-operator = is-'+' v is-'—' v is-'*' v is-'/' v is-'f'
30. is-prefix-operator = is-'+' v is-'—'
31. is-function-ref = ( <s-name: is-function-name> )
32. is-dummy-parameter = ( <s-name: is-d-function-name>, <s-def: is-function-name>, <s-var: is-intg> )
33. is-set-st = ( <s-line-no: is-line-no>, <s-st-name: LET>, <s-name: is-name>, <s-destination: is-line-no> )
34. is-null-st = ( <s-line-no: is-line-no>, <s-variable: is-aloadable>, <s-expression: is-expression> )
35. is-rem-st = ( <s-line-no: is-line-no>, <s-st-name: REM>, <s-name: is-name> )
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-expression: is-expression> )
40. is-if-st = ( <s-line-no: is-line-no>, <s-st-name: IF>, <s-machine: is-mintg>, <s-boolean: is-boolean>, <s-index: variable> )
41. is-boolean = ( <s-exp-1: is-expression>, <s-exp-2: is-expression>, <s-relation: is-relation>, <s-boolean: is-boolean> )
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-aloadable-list> )
48. is-aloadable = 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-‘;’
53. is-char = is-collection
54. is-simple-name = A( is-simple-variable )
55. is-array-name = A( is-letter )
56. is-function-name = A( is-function-name )
57. is-library-function = A( is-library-function )
58. is-user-function = A( is-user-function )
59. is-constant = A( is-intg )
60. is-line-no = A( is-line-no )
61. is-relation = A( is-relation )
62. is-intg = A( is-intg )

The two predicates are ext-value and is-int-value are implemented as an attribute of the representation of external and internal values respectively.

Appendix 3. Interpreter of the Basic machine instructions and functions

1. execute-program(text) =
2. execute-statement(line-no, text, line-no-stack) =
3. execute-statement(line-no, line-no-stack) =
4. execute-statement(next-line-no(text, 0), text, < > ) =
5. execute-statement(next-line-no(text, line-no), < > ) =
6. execute-statement(next-line-no(text, line-no), text) =
7. execute-statement(next-line-no(text, line-no), text, line-no-stack) =
8. execute-statement(next-line-no(text, line-no), text, tail(line-no-stack)) =
9. execute-statement(next-line-no(text, line-no), text, line-no-stack) =
10. execute-statement(next-line-no(text, line-no), text, line-no-stack) =

Next page:
7. compare(for-stack; value, line-no) =
   signum(s-increment(head(for-stack))) ×
   (value - limit(head(for-stack))) <→
   PASS: next-line(no-s-text(f), s-line-no(head(for-stack)))
   T → tail(for-stack)
8. fetch(stg, location, value) →
   PASS: elem(location, stg)
9. store(stg; location, value) =
   elem(location) = value
10. get-location(variable) =
    is-simple-variable(variable) →
    get-loc-l(s-name(s-name(variable)) • s-attr(f))
    is-function-ref(expression) =
    is-constant(expression) →
    is-variable(expression) →
    is-prefix-expression(expression) →
    is-function-ref(expression) =
    is-library-function(s-name(express)) &
    length(s-arg(express)) = 1 →
    is-function-def(block) →
    is-function-def(block) →
    is-library-function(name, arg) =
    is-user-function(s-name(express)) -9-
    if-condition(value) =
    is-constant(expression) →
    is-variable(expression) →
    is-prefix-expression(expression) →
    is-function-def(block) =
    is-expression(block) →
    evaluate-expression(block)
11. get-loc-1(symtab) →
    is-eo(s-element(symtab)) →
    PASS: s-element(symtab)
T → (is-f2(s-subscript-2(variable)) → error
12. get-loc-2(symtab, variable) =
    (not(∀(s-subscript-2(variable))) →
    T → map-3(s-upper-bound(symtab), s-element(symtab),
    subscript)
    P = evaluate-expression(s-subscript-1(variable))
    T → PASS: next-line-no(s-text(f), line-no)
    is-nil(value) →
    is-integer(value) →
    T = value
13. map-3(dim-1, location, subscript) →
    is-nil(value) →
    T = value
14. map-4(dim-1, dim-2, location, subscript-1, subscript-2) =
    is-nil(value) →
    T = value
15. evaluate-expression(expression) →
    is-nil(expression) →
    T = value
    is-inf-expression(expression) →
    int-inf-expression(a, b, s-operator(expression)) →
    a: evaluate-expression(s-operand(expression)),
    b: evaluate-expression(s-operand-2(expression))
    is-integer(expression) →
    fetch(s-stg(f), location);
    location: get-location(elem(expression))
    is-constant(expression) →
    pass(internal-rep(expression))
    is-function-ref(expression) →
    int-function-ref(expression)
    is-dummy-parameter(expression) →
    pass(elem(var-name(expression))) • s-par
    s-def(s-def-name(expression)) • s-funcs(f)
16. int-inf-expression(op1, op2, opr) =
    is-left(list) →
    error
    length(list) = 1 →
    head(list)
    head(list) =
    least<head(head(list)) × tail(tail(list))
    T → least(tail(list))
    is-nil(expression) →
    pass(op1)
    is-nil(expression) →
    pass(op2)
17. int-prefix-expression(opd, opr) →
    is-nil(expression) →
    pass(opd)
    is-nil(expression) →
    pass(opr)
18. compare-relation(exp-1, exp-2, rel, destination, next-line) =
    is-nil(expression) →
    (rel = exp-1 = exp-2) →
    is-nil(expression) →
    (rel < exp-1 < exp-2) →
    is-nil(expression) →
    (rel = exp-1 = exp-2) →
    is-nil(expression) →
    (rel = exp-1 < exp-2) →
    is-nil(expression) →
    (rel = exp-1 = exp-2) →
    is-nil(expression) →
    (rel = exp-1 < exp-2) →
    is-nil(expression) →
    (rel = exp-1 = exp-2) →
    is-nil(expression) →
    (rel = exp-1 < exp-2) →
    (rel = exp-1 = exp-2) →
    T → pass: next-line
19. if-condition(value, line-no-stack) =
    execute-statement(elem(compress(value, length(list)), list),
    line-no-stack)
20. int-function-ref(exp) =
    is-library-function(s-name(express)) &
    length(s-arg(express)) = 1 →
    int-library-function(s-name(express), x); x:
    evaluate-expression(head(s-arg(express)))
    is-user-function(s-name(express)) →
    int-function-def(s-express) →
    evaluate-args(s-var(s-name(express)) • s-funcs(f))
    s-arg(express), 1)
21. int-library-function(name, arg) =
    pass(name, arg)
22. int-function-def(block) =
    s-expression(block) →
    evaluate-expression(block)
23. evaluate-args(parameter-list; argument-list, n) =
    n = length(argument-list) →
    null
    T → evaluate-arg(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) =
    I: a
25. execute-read-st(stg; line-no, input, line-no-stack) =
    execute-statement(next-line(no-text, line-no), text, line-no-stack)
    read(stg; input, s-line-no(no-text), 1)
26. read(stg; input, read-st, n) =
    n = length(s-load(read-st), n) →
    null
    T → read(stg; input, read-st, n + 1)
    store(stg; line-no, value);
    location: get-location(elem(n, s-load(read-st))),
    value: get-data-value(s-key(s-input(f)); s-date(input))
27. get-data-value(key, data-list) =
    key > length(data-list) →
    error
    T → PASS: internal-rep(elem(key, data-list))
    I: key + 1
28. restore-key(key) =
    I: 1
29. execute-print-st(stg, line-no, line-no-stack) =
    execute-statement(next-line(no-text, line-no), color-
    line-no, text, line-no-stack)
    print(s-output(f); s-outpufs-line(line-no)(text), 1)
30. print(output; print-st, n) =
    n = length(print-st) →
    add-to-output(output, 'CR')
    n = length(print-st) →
    add-to-output(output, elem(n, print-st))
    print(output; print-st, n, n + 1)
    add-to-output(output, a; a,
    print(stg, print-st, n + 1)
31. add-to-output(output; element) =
    is-integer(element) →
    I: output n < element →
    error
    print(output; print-st, n + 1)
    T ← print(output; print-st, n + 1)
32. least(list) =
    is-nil(list) →
    error
    length(list) = 1 →
    head(list)
    head(list) <
    least < head(tail(tail(list)))
    T ← least(tail(tail(list)))
33. those-next(index, line-no, text) =
    is-nil(expression) →
    error
    s-index(s-line(index, text)) =
    index &
    i > line-no
34. compress(value, list-length) =
    value < 1 →
    value
    value > list-length →
    list-length
    T ← value
35. next-line-no(text, i) =
    i = (1) • st-name(s-line(i)) = END →
    error
    is-nil(expression) →
    next-line-no(text, i + 1)
    T ← i

References
Programming Language Definition.
Laboratory, Vienna.