The Formal Description of Minimal Basic

Based on ANSI document no. X3J2/76-01

by D.L. Reese and J.A.N. Lee

Language Research Laboratory
Virginia Polytechnic Institute
and
State University

Technical Report no. CS76005-R
June, 1976

The purpose of this formal description of Minimal Basic is to provide an operational description which is not subject to the ambiguities of interpretation of natural language specifications. The description is composed of three parts:

1. A syntactic definition of the textural representation of the program, known as the concrete syntax. The concrete syntax used throughout this report corresponds exactly to the syntax and section titles in document no. X3J2/76-01 which has been transmitted by American National Standards Committee - X3J2 to the American National Standards Institute as a draft proposed standard for Minimal Basic.

2. A structural descriptiion of the abstract form of the program known as the abstract syntax. The abstract syntax which is used here corresponds directly with that presented by Neuhold (ref 1)* and Lee (ref 2)*, with the following exceptions, the symbol used for logical implication herein are the pair =>, the symbol for logical (inclusive) or is v, and the symbol for logical and is &. Whereas in previous formal descriptions the two syntaces have been kept separate, in this document the two are interleaved in order to maintain their proximity to each other and to the interpretation.

3. A semantic description of the operational meanings of the elements of the program. The semantic descriptions used here correspond directly with the Vienna Description Language used by Neuhold (ref 1, above) and Lee (ref 2, above) with the following exceptions. To make the conditional expressions more readable the word "otherwise" has been used instead of the truth value "t", where the semantic meaning is actually otherwise.

The section numbers used herein correspond in their first elements to the section numbers in X3J2/76-01. Thus it is possible to interpolate this description into the existing document without modifying the section numbers of the current sections.

---

* ref 1:
Neuhold,E.J., The Formal Description of Programming Languages,
IBM Systems Journal, V10,no.2,1971.

* ref 2:
Lee,J.A.N., Computer Semantics, Van Nostrand Reinhold, New York, 1972.

Section 2. The Abstract Machine

```
is-state = (<s-symtab:is-symtab>,
            <s-text:is-program-statement-list>,
            <s-data-list:is-data-list>,
            <s-kbd-input:is-data-list>,
            <s-kbd-output:is-print-list>,
            <s-line-no-stack:is-line-no-stack>,
            <s-for-stack:is-for-stack>,
            <s-key:is-key>,
            <s-options:is-options>)

is-symtab = ((<s(name):is-attr-data>/for all names in the program))
where
is-attr-data = is-variable-data v is-def-data
```

note: The names include the names of functions both user defined and implementation supplied, parameter names (modified by their concatenation with the name of the function in which they are used to create unique names) and normal variable names.

```
is-key = is-integer
is-data is defined in section 14.2
is-options = (set(<is-option-name:is-option-value>))
is-option-name = "base"
is-option-value = is-base-value
is-base-value = 0 v 1
mode(element) = is-numeric-expression(element) => "numeric"
                is-string-expression(element) => "string"
```

There exists within the interpreter, the need to test for two specific empty elements; the null structure (as used in the empty statement) and the empty list (as used in several instructions to be the initial value of a list which is to be evaluated). Two fundamental predicates provide this feature:
is-null which is satisfied by a null structure, which in turn is the product of applying a selector function to an elementary object, and is-<> which is satisfied by the object which also satisfies the predicate is-list but which contains no elements.
The search function that selects a unique element from a given set of elements according to some defined criteria has the following general form:
(that:i)(is-pred(i))
Where is-pred(i) defines the crireria by which the set is selected from the universe of elements in the state of the machine and the criterion for selecting i. When there is no unique element which satisfies the criteria specified in the that function, the value returned to the referencing expression is undefined. Although there is no object wihtin the machine which can be transmitted from one structure to another, the predicate is-undef, applied to an expression whose result may be undefined, will provide a truth value as appropriate.
The initial state of the machine over which the interpretation is to be performed contains all the elements of the program (in their various forms). The control portion of the state contains a single instruction:

execute-program

When the program halts, the control portion of the state will be empty.

Section 3 Characters and Strings

3.2 Syntax

In this section only, the concrete syntactic forms of the characters and their corresponding abstract forms are listed in pairs for ease of reading. There is no semantic description associated with the individual characters or strings.

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

digit = 0/1/2/3/4/5/6/7/8/9
is-digit = 0 v 1 v 2 v 3 v 4 v 5 v 6 v 7 v 8 v 9

string-character = quote / quoted-string-character
is-string-character = is-quote v is-quoted-string-character

quoted-string-character = ampersand / apostrophe / comma /
                          exclamation-point /
                          unquoted-string-character
is-quoted-string-character = is-ampersand v is-apostrophe v
                             is-comma v is-exclamation-point v
                             is-unquoted-string-character

unquoted-string-character = space / plain-string-character
is-unquoted-string-character = is-space v
                               is-plain-string-character

plain-string-character = letter / digit / asterisk / circumflex /
                         close / colon / dollar / equals /
                         greater-than / less-than / minus /
                         number-sign / open / percent /
                         period / plus / question-mark /
                         semicolon / slant / underline
is-plain-string-character = is-letter v is-digit v is-asterisk v
                            is-circumflex v is-close v is-colon v
                            is-dollar v is-equals v
                            is-greater-than v is-less-than v
                            is-minus v is-number-sign v
                            is-open v is-percent v is-period v
                            is-plus v is-question-mark v
                            is-semicolon v is-slant v is-underline

remark-string = string-character*
is-remark-string = is-string-character-list

quoted-string = quote quoted-string-character* quote
is-quoted-string = is-quoted-string-character-list

unquoted-string = plain-string-character / plain-string-character
                  unquoted-string-character* plain-string-character
is-unquoted-string = is-plain-string-character v
                     is-unquoted-string-character-list
```

```
keyword = BASE / DATA / DEF / DIM / END / FOR / GO / GOSUB /
          GOTO / IF / INPUT / LET / NEXT / ON / OPTION /
          PRINT / RANDOMIZE / READ / REM / RESTORE /
          RETURN / STEP / STOP / SUB / THEN / TO
```

There is no abstract syntax associated with keywords.

Section 4 Programs

4.2.1 Concrete Syntax

```
program = line* end-line
line = line-number statement end-of-line
line-number = digit digit? digit? digit?
end-of-line =  implementation-defined
end-line = line-number end-statement end-of-line
end-statement = END
statement = data-statement / def-statement / dimension-statement /
            for-statement / gosub-statement / goto-statement /
            if-then-statement / input-statement / let-statment /
            next-statement / on-goto-statement / option-statement /
            print-statement / randomize-statement /
            read-statement / remark-statement / restore-statement /
            return-statement / stop-statement
```

4.2.2 Abstract Syntax

In the process of translation from concrete form to abstract form, the declarative statements within the program are replaced by empty statements except in the case of an end statement which is replaced by a stop statement. The data within the declarative statements is recorded elsewhere in the abstract machine. Hence the program in the abstract machine contains only executable statements, it being the task of the syntactic analyzer to ensure that the end statement is physically the last statement in the program.

```
is-program = is-statement-list
is-line = (<s-line-no:is-line-no>,
           <s-st-name:is-st-name>,
           <s-st-body:is-st-body>)
is-line-no = is-integer
is-end-line = is-end-statement
```

In the translation of the concrete text into the abstract form, the end statement is transformed into a stop statement. This transformation reflects the dual use of the end statement both as a text delimiter and as a default termination of execution statement. (see section 10.2.2)
Statements in the program are classified under two broad categories: executable statements and declarative statements. Since declarative statements are not retained by the translator for inclusion in the text over which the interpreter operates, there is no definitive abstract syntax for declarative statements. Hence the only remaining abstract syntactic specification is:

```
is-statement = is-executable-statement
```

Executable statements are further classified into five subclasses to facilitate the description of their semantics.

```
is-executable-statement = is-assignment-statement v
                          is-control-statement v
                          is-input/output-statement v
                          is-empty-statment v
                          is-randomize-statement
```

Further classification of the statements belonging to each subclass

will be given in the appropriate sections.

4.4 Semantics

During the translation of the concrete program into the abstract form used by the interpreter, the program statements are ordered sequentially and any conflicting line numbers are resolved. The following two functions are elementary to the interpreter:
The stmt function selects a statement from the text, given the line number of the statement required:

```
stmt(line-no,text) =
  elem((that:i)(s-line-no(elem(i,text)) = line-no),text)
```

The next function selects the next statement to be executed following the statement just executed, provided that the normal sequence is to be followed:

```
next(line-no) =
  s-line-no.elem((that:i)(s-line-no.elem(i-1,text) = line-no),text)
```

```
execute-program =
                    execute-statement(
                                        s-line-no(head(text)))
```

where text is the s-text component of the state of the machine and represents the program.

```
execute-statement(line-no) =
 is-undef((that:i)(s-line-no.elem(i,text) = line-no)) =>
                                                  error

 is-assignment-statement(stmt(line-no,text)) =>
     exec-assignment-st(
                          stmt(line-no,text))

 is-control-statement(stmt(line-no,text)) =>
     exec-control-st(
                        stmt(line-no,text))

 is-input/output-statement(stmt(line-no,text)) =>
     exec-i/o-st(
                    stmt(line-no,text))

 is-randomize-statement(stmt(line-no,text)) =>
    exec-randomize-st(
                         stmt(line-no,text))

 is-empty-statement(stmt(line-no,text)) =>
     exec-statement(
                       next(line-no,text))
```

note: Although the Minimal Basic does not contain a provision for empty (or null) statements, the semantics here include a check for an empty statement to take care of the case(s) of executing program lines that originally contained declaration statements which were removed by the translator.

The semantics associated with each specific statement type are given in the section appropriate to the statement.

Section 5 Constants

5.2.1 Concrete Syntax

```
constant = numeric-constant / string-constant
numeric-constant = sign? numeric-rep
sign = plus / minus
numeric-rep = significand exrad?
significand = integer period? / integer? fraction
integer = digit digit*
fraction = period digit digit*
exrad = E sign? integer
string-constant = quoted-string
```

5.2.2 Abstract Syntax

Irrespective of the external (textural) form of a constant or of a value generated by the execution of the program, there exists only one internal representation of the number which conforms to the specification is-implementation-defined.
There exists one specific numeric representation which needs to be distinguished; that is the representation of integers. Since the subscripting and dimensioning of arrays and the numbering of lines is possible only by the use of integers, the specification is-integer is elementary to the definition system.
The specification is-integer defines a set of objects which is contained within the set of objects defined by is-numeric-rep.

```
is-constant = is-numeric-constant v is-string-constant
is-string-constant = is-quoted-string
is-numeric-constant = is-numeric-rep
is-numeric-rep = is-implementation-defined
is-integer = is-implementation-defined
```

Section 6 Variables

6.2.1 Concrete Syntax

```
variable = numeric-variable / string-variable
numeric-variable = simple-numeric-variable / numeric-array-element
simple-numeric-variable = letter digit?
numeric-array-element = numeric-array-name subscript
numeric-array-name = letter
subscript = open numeric-expression (comma numeric-expression )?
            close
string-variable = letter dollar
```

6.2.2 Abstract Syntax

```
is-variable = is-numeric-variable v is-string-variable
```

As variables are initially encountered by the translator, they are recorded in the s-symtab component of the abstract machine. The data contained in this symbol table conforms to the specification:

```
is-variable-data = (<s-name:is-variable>,
                    <s-attributes:is-type>,
                    <s-value:is-value>)
where
is-type = (<s-mode:is-"numeric" v is-"string">,
           <s-dimensions:is-dimension-list >)
is-dimension is defined in section 15.2.2
is-value = is-numeric-rep v is-string-character-list
```

(note: The values associated with all variables initially conform to the specification is-implementation-defined.)

Where the attributes are specified to be numeric (that is, is-"numeric") then the s-value-component must conform to the specification is-numeric-rep. Conversely, if the type is string, then the value conforms with the specification is-string-character-list. When an array element is encountered by the translator for which there was not a prior dimensioning declaration (a DIM statement), a set of dimensioning attributes are developed and are recorded in the symbol table. The upper limit of dimensions is established to be 10 in these cases and the lower limit is either 0 or the value set by the previous execution of an option base statement. (see section 15)

```
is-numeric-variable = is-simple-numeric-variable v
                      is-numeric-array-element
is-simple-numeric-variable specifies an elementary object which is
                           the transformation of the concrete
                           syntactic string which is defined by the
                           component simple-numeric-variable.
is-numeric-array-element = (<s-array-name:is-array-name>,
                            <s-subscripts:is-subscript-list>)
is-array-name specifies an elementary object which is the
              transformed form of the syntactic element specified by
              the rule for numeric-array-name and which is
```

distinguishable from the same identifier used for a simple-numeric-variable. The translator, in preparing the abstract form of the program, ensures that array variables are either one or two dimensional, but not both, and rejects those statements which do not conform to this specification.

is-subscript = is-numeric-expression

is-string-variable specifies an elementary object which is the transformed form of the syntactic element specified by the rule for string-variable.

6.4 Semantics

Variables occur in two typical situations within a basic program; either as the destination of a datum or as the source of a datum for use by the program. In the semantic model used here, where the variable name is associated with either a simple numeric variable or a string variable (there are no string arrays defined), the variable name is the key to the storage location of the value. In the case of a numeric array element, the parameters needed to ascertain the location of the associated value are the array name, the evaluated subscripts, and the declared dimensions of the array. The two semantic operations described here correspond to the normal operations of data retrieval (fetch) and data storage (store).

```
fetch(variable) =
   is-numeric-array-element(variable) =>
      fetch-element(variable)
   otherwise =>
      PASS:(s-value.s(variable)(symtab))

store(variable,value) =
   is-numeric-array-element(variable) =>
      store-element(variable,value)
   otherwise =>
      (is-numeric-rep(value) &
         is-"numeric"(s-mode.s-attributes.s(variable)(symtab)) v
       is-string-constant(value) &
         is-"string"(s-mode.s-attributes.s(variable)(symtab)) =>
        s-value.s(variable)(symtab):value
       otherwise => error )
fetch-element(element) =
   pass(get(check-bounds(subscript-list),
            s-attributes.s(s-array-name(element))(symtab),
            s-value.s(s-array-name(element))(symtab) ) );
   subscript-list:eval-subs(element)
```

Where the function get, with the arguments:
evaluated subscript list,
dimension list,
and array element value list,
is implementation defined and returns the correct array element value or a subscript out of range error. The function check-bounds is a special implementation defined function which matches the implicit or explicit bounds for the dimensions of the array with the rounded subscripts in the array reference and reports conflicts as errors.

```
eval-subs(element) =
   pass(round(subscript-list) );
   subscript-list:eval-exp-list(s-subscripts(element),<>)
```

Where the function round is an implementation defined function which transforms the elements in the subscript list to conform to the specification is-integer.

```
eval-exp-list(exp-list,out-list) =
   is-<>(exp-list) => PASS:out-list
   otherwise =>
     eval-exp-list(tail(exp-list),
                   cat(out-list,list(exp)));
         exp:eval-expression(head(exp-list))
store-element(element,value) =
   put(check-bounds(subscript-list),
       s-attributes.s(s-array-name(element))(symtab),value);
   subscript-list:eval-subs(element)
```

Where the instruction put is implementation defined and performs a mutate operation on the s-symtab-component of the machine which reflects the new array element value.

Section 7 Expressions

7.2.1 Concrete Syntax

```
expression = numeric-expression / string-expression
numeric-expression = sign? term (sign term )*
term = factor ( multiplier factor )*
factor = primary (involution primary )*
multiplier = asterisk / slant
involution = circumflex
primary = numeric-variable / numeric-rep / numeric-function-ref /
          open numeric-expression close
numeric-function-ref = numeric-function-name argument-list?
numeric-function-name = numeric-defined-function /
                        numeric-supplied-function
argument-list = open argument close
argument = numeric-expression
string-expression = string-variable / string-constant
```

7.2.2 Abstract Syntax

```
is-expression = is-numeric-expression v is-string-expression
is-numeric-expression = is-infix-expression v
                        is-prefix-expression v
                        is-numeric-variable v
                        is-numeric-constant v
                        is-function-reference
is-infix-expression = (<s-operator:is-infix-operator>,
                       <s-operand-1:is-numeric-expression>,
                       <s-operand-2:is-numeric-expression>)
is-prefix-expression = (<s-operator:is-prefix-operator>,
                        <s-operand:is-numeric-expression>)
is-infix-operator = is-plus v is-minus v is-asterisk v
                    is-slant v is-circumflex
is-prefix-operator = is-plus v is-minus
```

The transformation from concrete textural form to abstract form converts expressions from their truly syntactic form into a form which is representative of the order of evaluation of the individual terms and factors. In the abstract form parentheses have been eliminated entirely.

```
is-function-reference = (<s-function-name:is-function-name>,
                         <s-argument-list:is-numeric-expression-list>)
is-function-name = is-predefined-name v is-user-defined-name
```

It is important to note that the sets defined by is-user-defined-name and is-predefined-name are disjoint.

```
is-string-expression = is-string-variable v is-string-constant
```

7.4 Semantics

```
eval-expression(expression) =
   is-infix-expression(expression) =>
   int-infix-exp(a,b,s-operator(expression));
```

```
      a:eval-expression(s-operand-1(expression)),
      b:eval-expression(s-operand-2(expression))

   is-prefix-expression(expression) =>
   int-prefix-exp(a,s-operator(expression));
      a:eval-expression(s-operand(expression))

   is-numeric-variable(expression) =>
      fetch(expression)

   is-string-variable(expression) => error

   is-numeric-constant(expression) =>
      PASS:expression

   is-string-constant(expression) => error

   is-function-reference(expression) =>
   int-function-reference(
                          expression)
where
int-function-reference(ref) =
   int-fn-ref(s-function-name(ref),value-list);
      value-list:eval-exp-list(s-argument-list(ref),<>)
where
int-fn-ref(name,arg-list) =
   is-user-defined-name(name) =>
      int-user-fn(s(name)(symtab),arg-list)
   is-predefined-name(name) &
   ((is-"log"(name) & arg-list ≤ 0 ) v
   (is-"sqr"(name) & arg-list < 0 )) => error
   is-predefined-name(name) => name(arg-list)

the instrucion int-user-fn is defined in section 16.4

eval-string-expression(expression) =
   is-string-constant(expression) => PASS:expression
   is-string-variable(expression) => fetch(expression)
```

Section 8 Implementation-supplied Functions

8.2.1 Concrete Syntax

```
numeric-supplied-function = ABS / ATN / COS / EXP / INT / LOG /
                            RND / SGN / SIN / SQR / TAN
```

8.2.2 Abstract Syntax

```
is-predefined-name = abs v atn v cos v exp v int v log v
                     rnd v sgn v sin v sqr v tan
```

8.4 Semantics

Implementation-supplied functions are assumed to exist as mathematical functions whose values are obtained by the reference.

Section 9 The LET statement

9.2.1 Concrete Syntax

```
let-statement = numeric-let-statement / string-let-statement
numeric-let-statement = LET numeric-variable equals
                        numeric-expression
string-let-statement = LET string-variable equals
                       string-expression
```

9.2.2 Abstract Syntax

```
is-assignment-statement = is-numeric-let-statement v
                          is-string-let-statement
is-numeric-let-statment = (<s-st-name:is-"let">,
                           <s-line-no:is-line-no>,
                           <s-st-body:is-numeric-let-body>)
is-numeric-let-body = (<s-numeric-variable:is-numeric-variable>,
                       <s-expression:is-expression>)
is-string-let-statement = (<s-st-name:is-"let">,
                           <s-line-no:is-line-no>,
                           <s-st-body:is-string-let-body>)
is-string-let-body = (<s-string-variable:is-string-variable>,
                      <s-expression:is-string-expression>)
```

9.4 Semantics

```
exec-assignment-st(stmt) =
   is-numeric-let-statement(stmt) =>
      exec-numeric-let(stmt)
   is-string-let-statement(stmt) =>
      exec-string-let(stmt)
where
exec-numeric-let(stmt) =
   execute-statement(next(s-line-no(stmt)));
      store(s-numeric-variable.s-st-body(stmt),value);
         value:eval-expression(
                                 s-expression.s-st-body(stmt))
exec-string-let(stmt) =
 execute-statement(next(s-line-no(stmt)));
      store(s-string-variable.s-st-body(stmt),value);
         value:eval-string-exp(
                                 s-expression.s-st-body(stmt))
```

Section 10 Control Statements

10.2.1 Concrete Syntax

```
goto-statement = GO space* TO line-number
if-then-statment = IF relational-expression THEN line-number
relational-expression = numeric-expression relation
                        numeric-expression / string-expression
                        equality-relation string-expression
relation = equality-relation / less-than / greater-than /
           not-less / not-greater
equality-relation = equals / not-equals
not-less = greater-than equals
not-greater = less-than equals
not-equals = less-than greater-than
gosub-statement = GO space* SUB line-number
return-statement = RETURN
on-goto-statement = ON numeric-expression GO space* TO
                    line-number ( comma line-number )*
stop-statement = STOP
```

10.2.2 Abstract Syntax

```
is-control-statement = is-go-to-statement v
                       is-if-statement v
                       is-go-sub-statement v
                       is-return-statement v
                       is-on-statement v
                       is-stop-statement v
                       is-for-statement v
                       is-next-statement
is-go-to-statement = (<s-st-name:is-"goto">,
                      <s-line-no:is-line-no>,
                      <s-st-body:is-line-no>)
is-if-statement = (<s-st-name:is-"if">,
                   <s-line-no:is-line-no>,
                   <s-st-body:(<s-rel-exp:is-relational-expression>,
                               <s-destination:is-line-no>) >)
is-go-sub-statement = (<s-st-name:is-"goto">,
                       <s-line-no:is-line-no>,
                       <s-st-body:is-line-no>)
is-return-statement = (<s-st-name:is-"return">,
                       <s-line-no:is-line-no>)
is-on-statement = (<s-st-name:is-"on">,
                   <s-line-no:is-line-no>,
                   <s-st-body:(<s-expression:is-numeric-expression>,
                               <s-destination:is-line-no-list>) >)
is-stop-statement = (<s-st-name:is-"stop">,
                     <s-line-no:is-line-no>)
```

The abstract syntax for the for-statement and the next-statement is given in the next section together with the concrete syntax.

```
is-relational-expression = (<s-relation:is-relation>,
                            <s-operand-1:is-expression>,
                            <s-operand-2:is-expression>)
```

```
is-relation = is-less-than v
              is-greater-than v
              is-less-than-or-equal-to v
              is-greater-than-or-equal-to v
              is-equals v
              is-not-equal-to
```

is-less-than-or-equal-to specifies an elementary object which is the transformation of the syntactic element specified by the rule for not-greater.
is-greater-than-or-equal-to specifies an elementary object which is the transformation of the syntactic element specified by the rule for not-less.
is-not-equal-to specifies an elementary object which is the transformation of the syntactic element specified by the rule for not-equals.

10.4 Semantics

Successful completion of the function next, which is invoked as part of the instruction execute-statement, guarantees the existance of the line numbers used as destinations of the control statements.

```
exec-control-st(stmt) =
   is-go-to-statement(stmt) =>
      exec-go-to-st(stmt)
   is-if-statement(stmt) =>
      exec-if-st(stmt)
   is-go-sub-statement(stmt) =>
      exec-go-sub-st(stmt)
   is-return-statement(stmt) =>
      exec-return-st(stmt)
   is-on-statement(stmt) =>
      exec-on-st(stmt)
   is-stop-statement(stmt) => null
   is-for-statement(stmt) =>
      exec-for-st(stmt)
   is-next-statement(stmt) =>
      exec-next-st(stmt)
where:
exec-go-to-st(stmt) =
   execute-statement(
                     s-st-body(stmt))

exec-if-st(stmt) =
   if-branch(next(s-line-no(stmt)),
                  s-destination.s-st-body(stmt),truth-value);
      truth-value:eval-rel-exp(
                              s-rel-exp.s-st-body(stmt))
where:
if-branch(next-line,branch,truth-value) =
   is-"true"(truth-value) =>
      execute-statement(branch)
   otherwise =>
      execute-statement(next-line)

eval-rel-exp(rel-exp) =
```

```
    mode(s-operand-1(rel-exp)) ≠ mode(s-operand-2(rel-exp)) =>
       error
    otherwise =>
    is-string-expression(s-operand-1(rel-exp)) =>
       eval-string-rel(a,b,
                              s-rel(rel-exp));
          a:eval-string-exp(
                              s-operand-1(rel-exp)),
          b:eval-string-exp(
                              s-operand-2(rel-exp))
     is-numeric-expression(operand-1(rel-exp)) =>
       eval-numeric-rel(a,b,
                              s-rel(rel-exp));
          a:eval-expression(
                              s-operand-1(rel-exp)),
          b:eval-expression(
                              s-operand-2(rel-exp)) )
where the function mode is defined in section 2.

eval-string-rel(op-1,op-2,rel) =
   not(is-equals(rel) v is-not-equal-to(rel)) => error
   otherwise =>
      is-equals(rel) & (op-1 = op-2) v
      is-not-equal-to(rel) & (op-1 ≠ op-2) =>
         PASS:("true")
      otherwise =>
         PASS:("false")

eval-numeric-rel(op-1,op-2,rel) =>
   is-equals(rel) & (op-1 = op-2) v
   is-less-than(rel) & (op1 < op-2) v
   is-greater-than(rel) & (op-1 > op-2) v
   is-less-than-or-equal-to(rel) & (op-1 ≤ op-2) v
   is-greater-than-or-equal-to(rel) & (op-1 ≥ op-2) v
   is-not-equal-to(rel) & (op-1 ≠ op-2) =>
      PASS:("true")
   otherwise =>
      PASS:("false")

exec-go-sub-st(stmt) =
   execute-statement(s-st-body(stmt));
      save(next(s-line-no(stmt)))
where:
save(line-no) =
   s-line-no-stack:cat(list(line-no),line-no-stack)

where line-no-stack = s-line-no-stack(state)

exec-return-st(stmt) =
   execute-statement(re-entry-pt);
   re-entry-pt:unsave

where:
unsave =
   is-<>(line-no-stack) => error
   otherwise =>
```

```
      PASS:head(line-no-stack)
      s-line-no-stack:tail(line-no-stack)

exec-on-st(stmt) =
   multi-branch(s-destination.s-st-body(stmt),value);
      value:eval-round(s-expression.s-st-body(stmt))

where:
eval-round(value) =
   pass(round(val));
      val:eval-expression(value)

multi-branch(line-no-list,value) =
   value < 1 v value > length(line-no-list) => error
   otherwise =>
      execute-statement(
                        elem(value,line-no-list)
```

Section 11 FOR and NEXT statements

11.2.1 Concrete Syntax

```
for-statement = FOR control-variable equals initial-value TO
                limit  ( STEP increment )?
control-variable = simple-numeric-variable
initial-value = numeric-expression
limit = numeric-expression
increment = numeric-expression
next-statement = NEXT control-variable
```

11.2.2 Abstract Syntax

The overall concrete syntax of a for-next block requires that the pair of for and next statements be so physically arranged in the program that the next statement physically follows the for statement and has the same control variable. Further, any other for-next blocks must be completely contained within the enclosing block. *

```
is-for-statement = (<s-st-name:is-"for">,
                    <s-line-no:is-line-no>,
                    <s-st-body:(<s-index:is-numeric-variable>,
                                <s-limits:is-for-limits>) >)
is-for-limits = is-numeric-expression-list
```

In the abstract form, the limits are in the order initial-value, limit-value, and increment-value. if the increment-value has not been specified in the original text, the translator supplies the value of unity.

```
is-next-statement = (<s-st-name:is-"next">,
                     <s-line-no:is-line-no>,
                     <s-st-body:is-numeric-variable>)
```

11.4 Semantics

```
exec-for-st(stmt) =
   test-range(limits,stmt);
```

---

*

The Backus-Naur form of syntactic specification is not capable of expressing these contextural requirements. Whilst other forms of specification have been proposed to control such relationships, there is no commonly accepted specification form. Thus we assume that the translator, in the process of transforming the text from concrete to abstract form, tests for these situations and rejects programs which do not conform.
For proposed forms of contextural specification see:

Lee, J.A.N. and Dorocak, J., Conditional Syntactic Specification, proc. ACM National Conference, Atlanta, Ga., 1973.
Ledgard, H.F., Production Systems, Communications of the ACM, Volume 17, Number 2, February 1974.

```
      store(s-index.s-st-body(stmt),elem(1,limits));
      limits:eval-limits(s-limits.s-st-body(stmt),<>)
         unstack(a);
         a:check(s-index.s-st-body(stmt))
where:
test-range(limits,stmt) =
   (own1 -own2) x sgn(own3) > 0 =>
   execute-statement(next(s-line-no(
            elem(least(those:i)
            (is-next-statement(elem(i,text)) &
             s-line-no(elem(i,text)) > s-line-no(stmt) &
             s-st-body(elem(i,text)) = s-index.s-st-body(stmt)) ),
             text)))
   otherwise =>

   execute-statement(next(s-line-no(stmt)));
      stack(list(own1,own2,own3,next(s-line-no(stmt)),
                 s-index.s-st-body(stmt)))

   (note: own1 = elem(1,limits),
          own2 = elem(2,limits),
          own3 = elem(3,limits) )

stack(for-list) =
   s-for-stack(state):cat(list(for-list),for-stack)
```

note: for-stack is the name of the s-for-stack-component of the state of the basic machine.

the elementary function least is defined as follows:

```
   least(set) =
      crd(set) = 0 => error
      crd(set) = 1 => (that:i)(i is-an-element-of set)
      otherwise => (that:i)((for all j is-an-element-of set) &
                            (i ≤ j))
```

Where the elementary function crd returns the cardinality of the set passed as the argument.

```
unstack(value) =
   value = 0 => null
   otherwise =>
   unstack(value-1);
      unstack-one

unstack-one =
   s-for-stack:tail(for-stack)

check(var) =
   is-undef((that:i)(elem(5).elem(i)(for-stack) = var)) =>
      PASS:0
   otherwise =>
      PASS:(that:i)(elem(5).elem(i)(for-stack) = var)

eval-limits(exp-list,out-list) =
   eval-exp-list(exp-list,out-list)
```

```
exec-next-st(stmt) =
 test-done(limits,index-value,next(s-line-no(stmt)),a);
   limits:get-stack-limits(a);
          a:check(s-st-body(stmt))
   index-value:fetch(s-st-body(stmt))

where:
test-done(limits,value,next,stack-pointer) =
   (value - own2) x sgn(own3) > 0 =>
   execute-statement(next);
      unstack(stack-pointer)
   otherwise =>
   execute-statement(block-head);
      store(s-st-body(stmt),value + own3 )

get-stack-limits(value) =
   value = 0 => error
   otherwise =>
   PASS:elem(value)(for-stack)

   (note: own1 = elem(1,limits),
          own2 = elem(2,limits),
          own3 = elem(3,limits),
          block-head = elem(4,limits) )
```

****

In conforming to the specifications proposed by the X3J2 committee, the following interpretations are used in this document:

its next statement is defined to be any next statement in the program sequence which has a higher line-number and the same control variable as some existant for statement. This implies that it would be allowable to branch from the middle of a for-next block into the middle of another for-next block having the same control variable and return successfully to the original for-next block upon executing a next statement. While executing a next statement, if the interpreter finds that the associated for-block is to be terminated, all nested for-blocks are also terminated as part of this process. Executing a next statement without first having executed a for statement with the same control variable produces an error.

****

Section 12 The PRINT statement

12.2.1 Concrete Syntax

```
print-statement = PRINT print-list?
print-list = ( print-item? print-separator )* print-item?
print-item = expression / tab-call
tab-call = TAB open numeric-expression close
print-separator = comma / semicolon
end-of-print-line =  implementation defined
```

12.2.2 Abstract Syntax

```
is-input/output-statement = is-print-statement v
                            is-input-statement v
                            is-restore-statement v
                            is-read-statement
is-print-statement = (<s-st-name:is-"print">,
                      <s-line-no:is-line-no>,
                      <s-st-body:is-print-list>)
is-print = is-expression v is-punctuation v is-tab-call
is-punctuation = is-comma v is-semicolon v is-carriage-return
is-tab-call = (<s-fn-name:is-"tab">,
               <s-argument:is-numeric-expression>)
```

A carriage-return is added to a print list by the translator when the terminating print element is not other punctuation; that is, is not either a comma or a semicolon. This also includes the empty print list.

12.4 Semantics

```
exec-i/o-st(stmt) =
   is-print-statement(stmt) =>
      exec-print-st(stmt)
   is-input-statement(stmt) =>
      exec-input-st(stmt)
   is-restore-statement(stmt) =>
      exec-restore-st(stmt)
   is-read-statement(stmt) =>
      exec-read-st(stmt)

exec-print-st(stmt) =
   execute-statement(next(s-line-no(stmt)));
      print(s-st-body(stmt))
where:
print(out-list) =
   is-<>(out-list) => null
   is-string-expression(head(out-list)) =>
     print(tail(out-list));
         output(value);
         value:eval-string-exp(head(out-list))
   is-punctuation(head(out-list)) =>
      print(tail(out-list));
         output(head(out-list))
   is-numeric-expression(head(out-list)) =>
```

```
      print(tail(out-list));
         output(value);
         value:eval-expression(head(out-list))
   is-tab-call(head(out-list)) =>
      print(tail(out-list));
         output(tab(value));
         value:eval-round(s-argument.head(out-list))
```

The tab function is an elementary implementation defined function which returns a character string composed of spaces and/or a carriage return which conforms to the specifications set by the X3J2 committee. If the evaluated value of the argument to the tab function is less than 1, the value of unity is supplied by the translator.

```
output(element) =
   s-kbd-output:cat(s-kbd-output(state),list(element))
```

The semantics of the print statement are such that the output is assumed to be an unbounded list of output elements; these elements are composed of the following three types:

numeric value representations
character strings and
punctuation marks.

The punctuation marks are accepted by an implementation defined peripheral processor, so that the eventual output conforms to the specifications of the standard. Included in the set of punctuation marks is the carriage return. Normally, a carriage return is output at the completion of the output from a single print statement; this is accomplished by the translator inserting the carriage return into each abstract form of the print statement as an end of statement marker. This character conforms with the abstract syntactic specification is-punctuation and therefore is output by the interpreter.

Section 13 The INPUT statement

13.2.1 Concrete Syntax

```
input-statement = INPUT variable-list
variable-list = variable ( comma variable )*
input-prompt =  implementation defined
input-reply = data-list end-of-input-reply
data-list = datum ( comma datum )*
datum = constant / unquoted-string
end-of-input-reply =  implementation defined
```

13.2.2 Abstract Syntax

```
is-input-statement = (<s-st-name:is-"input">,
                      <s-line-no:is-line-no>,
                      <s-st-body:is-variable-list>)
is-input-prompt =  implementation defined
```

13.4 Semantics

```
exec-input-st(stmt) =
   execute-statement(next(s-line-no(stmt)));
      input(s-st-body(stmt),s-kbd-input(state));
        output(input-prompt)
```

Where "input-prompt" specifies the character which corresponds to the specification is-input-prompt.

```
where:
input(input-list,value-list) =
   is-<>(input-list) &
   not-<>(value-list) =>
      output(input-prompt)
   otherwise => null
   is-<>(value-list) =>
      input(input-list,s-kbd-input(state));
         output(input-prompt)
   otherwise =>
      input(tail(input-list),tail(value-list));
         store(head(input-list),head(value-list))
```

Note that although the specifications for the input statement include a requirement that the mode (numeric or string) of elements of the input list and the data be equal, the semantic description of this check is contained in the interpretation of the store instruction. The requirements for the form of the input reply, including numeric and string conversion errors are handled by the translator in forming the s-kbd-input component of the machine.

Section 14 The DATA, READ, and RESTORE statements

14.2.1 Concrete Syntax

```
data-statement = DATA data-list
read-statement = READ variable-list
restore-statement = RESTORE
```

14.2.2 Abstract Syntax

```
is-read-statement = (<s-st-name:is-"read">,
                     <s-line-no:is-line-no>,
                     <s-st-body:is-variable-list>)
is-restore-statement = (<s-st-name:is-"restore>,
                        <s-line-no:is-line-no>)
```

The data statement is a non-executable statement in the program. Non-executable statements are replaced, in the text of the program, by empty statements (with the exception of the END statement, see section 4), since their contents are not to be interpreted but instead direct the interpretation of the program.
The abstract syntax for the empty statement is:

```
is-empty-statement = (<s-st-name:is-"empty">,
                      <s-line-no:is-line-no>)
```

The elements of the data statement are not included in the abstract text of the machine. Instead, the data elements are contained in a special component (the s-data-list component of the abstract machine) which conforms to the specification:
is-data = is-constant v is-unquoted-string
Further, a special pointer is maintained in a register known as the s-key-component of the abstract machine, which signifies the next available data element to be read. This register conforms to the specification is-integer. Initially, the contents of the register are set to unity, indicating that the first element in the data list component is the next available data element to be read by a read statement. The restore statement also operates over this s-key-component.

14.4 Semantics

The interpretation of the empty statement is specified in section 4.4 .

```
exec-read-st(stmt) =
   execute-statement(next(s-line-no(stmt)));
      perform-read(s-st-body(stmt))
where:
perform-read(var-list) =
   is-<>(var-list) => null
   otherwise =>
      perform-read(tail(var-list));
         update-key(key+1);
            store(head(var-list,elem(key,data-list))
```

Where data-list refers to the s-data-list component of the state and represents the set of data elements which were derived from the set of data statements in the program.

```
   key = s-key(state)

update-key(value) =
   s-key > length(data-list) => error
   otherwise => s-key:value

exec-restore-st(stmt) =
   execute-statement(next(s-line-no(stmt)));
      update-key( 1 )
```

Section 15 Array Declarations

15.2.1 Concrete Syntax

```
dimension-statement = DIM array-declaration ( comma
                      array-declaration )*
array-declaration = numeric-array-name open bounds close
bounds = integer ( comma integer )?
option-statement = OPTION BASE ( 0 / 1 )
```

15.2.2 Abstract Syntax

The translator does not include elements of dim statements within the textural part of the abstract machine; instead, the dim statement is replaced by an empty statement (see section 4) with the same line number; the data relative to arrays is recorded within the symbol table. Within that table, the array attributes are recorded in accordance with the specification:

```
is-variable-data which is defined in section 6.
is-dimension = (<s-upper-bound:is-integer>,
                <s-lower-bound:is-integer>)
```

Where an array is not explicitly dimensioned by occurence in a dim statement, the s-upper-bound.s-dimensions.s-attributes component is set to contain the value(s) 10, and the s-lower-bound.s-dimensions.s-attributes component is set to contain the value(s) 0. During the transformation from concrete to abstract form, the translator ensures that all arrays contained in the program are declared explicitly in a dim statement only once. Similarly, an option-base statement if present in the program is replaced by the empty statement and the information contained in the body of the statement is recorded by the translator in the s-options-component of the machine. In setting the lower bounds for arrays which are implicitly dimensioned, the translator utilizes the value supplied to it by the option-base statement if one has been encountered or uses the value of 0 if no option-base statement is present in the program. It is the task of the syntactic analyzer to ensure that the option-base statement occurs on a physically lower numbered line number than any dim statement in the program and that dim statements occur on a physically lower numbered line number than the first reference to any array which is declared in a dim statement. If during the transformation of a dim statement, the translator detects that the lower bound declared in the dim statement is lower than the bound declared through an option-base statement, an error is produced and the statement is rejected.

Section 16 User Defined Functions

16.2.1 Concrete Syntax

```
def-statement = DEF numeric-defined-function parameter-list?
                equals numeric-expression
numeric-defined-function = FN letter
parameter-list = open parameter close
parameter = simple-numeric-variable
```

16.2.2 Abstract Syntax

The translator does not include elements of def statements within the textural part of the machine. Instead, the def statement is replaced by an empty statement (see section 4) with the same line number and the information relative to the evaluation of the function is recorded within the symbol table. Within that table the definition is recorded in accordance with the specification:

```
is-def-attr = (<s-name:is-function-name>,
               <s-attributes:is-parameter-list>,
               <s-value:is-def-expression>)
is-def-expression = is-def-infix-expression v
                    is-def-prefix-expression v
                    is-parameter v
                    is-numeric-constant v
                    is-def-function-reference
is-def-infix-expression = (<s-operator:is-infix-operator>,
                           <s-operand-1:is-def-expression>,
                           <s-operand-2:is-def-expression>)
is-def-prefix-expression = (<s-operator:is-prefix-operator>,
                            <s-operand-1:is-def-expression>)
is-parameter = (<s-function-name:is-user-defined-name>,
                <s-local-name:is-simple-numeric-variable>)
```

The transformation from the concrete syntactic form to the abstract form of a parameter, appends the name of the user defined function to the the parameter so as to distinguish both between similarly named parameters in differing functions and between the parameter and the global ariable of the same name. This distinction is apparent in the original text only by use of the context.
is-user-defined-name specifies an elementary object which is the transformation of the syntactic element specified by the rule for numeric-defined-function.

```
is-def-function-reference = (<s-function-name:is-function-name>,
                             <s-argument-list:is-def-expression-
                                                list>)
```

In building the symbol table, the translator ensures that only one definition is given for each user defined function and that such a definition does not contain any function which references itself inside its own definition (recursive functions are not allowed).

16.4 Semantics

```
int-user-fn(defn,arg-list) =
   is-null(defn) v
```

```
   length(arg-list) ≠ length(s-attributes(defn)) => error
   otherwise =>
      eval-def-exp(s-value(defn));
        pass-args(s-attributes(defn),arg-list)
where:
pass-args(par-list,arg-list) =
   is-<>(par-list) => null
   otherwise =>
      pass-args(tail(par-list),tail(arg-list));
         store(head(par-list),head(arg-list))

eval-def-exp(exp) =
   is-infix-def-expression =>
      int-infix-def-exp(
                        a,b,s-operator(exp));
         a:eval-def-exp(s-operand-1(exp),
         b:eval-def-exp(s-operand-2(exp))
   is-prefix-def-expression(exp) =>
      int-prefix-def-exp(
                              a,s-operator(exp));
         a:eval-def-exp(s-operand-1(exp))
   is-numeric-variable(exp) =>
      fetch(exp)
   is-string-variable(exp) => error
   is-numeric-constant(exp) =>
      PASS:exp
   is-string-constant(exp) => error
   is-def-function-reference(exp) =>
      int-def-fn-ref(exp)
   is-parameter(exp) =>
      fetch(exp)

int-def-fn-ref(ref) =
   int-fn-ref(s-function-name(ref),value-list);
      value-list:eval-def-exp-list(
                                 s-argument-list(ref),<>)
eval-def-exp-list(exp-list,out-list) =
   is-<>(exp-list) => PASS:out-list
   otherwise =>
      eval-def-exp-list(
                        tail(exp-list),cat(out-list,list(exp)));
         exp:eval-def-exp(head(exp-list))
```

Section 17 The RANDOMIZE statement

17.2.1 Concrete Syntax

randomize-statement = RANDOMIZE

17.2.2 Abstract Syntax

```
is-randomize-statement = (<s-st-name:is-"randomize">,
                          <s-line-no:is-line-no>)
```

17.4 Semantics

```
exec-randomize-st(stmt) =
   execute-statement(next(s-line-no(stmt)));
      randomize
```

Where the instruction randomize is an implementation defined instruction whose execution generates a new, unpredictable starting point for the list of pseudo-random numbers used by the rnd function.

Section 18 The REMARK statement

18.2.1 Concrete Syntax

remark-statement = REM remark-string

18.2.2 Abstract Syntax

In the translation of the concrete text into the abstract form, the remark statement is transformed into an empty statement. (see section 4)