TC 97

# International Standard 6373

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION•МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ•ORGANISATION INTERNATIONALE DE NORMALISATION

# Data processing — Programming languages — Minimal BASIC

*Traitement de l'information — Langages de programmation — BASIC minimal*

First edition — 1984-03-15



UDC 681.3.06 : 800.92

Ref. No. ISO 6373-1984 (E)

**Descriptors** : data processing, programming languages, information interchange.



Price based on 33 pages

## Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of developing International Standards is carried out through ISO technical committees. Every member body interested in a subject for which a technical committee has been authorized has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council.

International Standard ISO 6373 was developed by Technical Committee ISO/TC 97, *Information processing systems*, and was circulated to the member bodies in April 1982.

It has been approved by the member bodies of the following countries :

| | | |
|---|---|---|
| Belgium | Germany, F. R. | Romania |
| Canada | Hungary | South Africa, Rep. of |
| China | Ireland | Spain |
| Czechoslovakia | Italy | Sweden |
| Egypt, Arab Rep. of | Japan | Switzerland |
| Finland | Netherlands | USA |
| France | Poland | Yugoslavia |

The member bodies of the following countries expressed disapproval of the document on technical grounds :

Australia
United Kingdom





Printed in Switzerland

# Contents

# Data processing — Programming languages — Minimal BASIC

## 1 Scope and field of application

This International Standard is designed to promote the interchangeability of BASIC programs among a variety of automatic data processing systems.

This International Standard establishes :

a) the syntax of a program written in Minimal BASIC;

b) the formats of data and the precision and range of numeric representations which are acceptable as input to an automatic data processing system being controlled by a program written in Minimal BASIC;

c) the formats of data and the precision and range of numeric representations which can be generated as output by an automatic data processing system being controlled by a program written in Minimal BASIC;

d) the semantic rules for interpreting the meaning of a program written in Minimal BASIC;

e) the errors and exceptional circumstances which shall be detected and also the manner in which such errors and exceptional circumstances shall be handled.

Subsequent International Standards for the same purpose will describe extensions and enhancements to this International Standard. Programs conforming to this International Standard, as opposed to extensions or enhancements of this International Standard, will be said to be written in Minimal BASIC.

Although the BASIC language was originally designed primarily for interactive use, this International Standard describes a language that is not so restricted.

The organization of the International Standard is outlined in annex A. The method of syntax specification used is explained in annex B.

## 2 Conformance

There are two aspects of conformance to this International Standard :

a) conformance by a program written in the language;

b) conformance by an implementation which processes such programs.

A program conforms to this International Standard only when

— each statement contained therein is a syntactically valid instance of a statement specified in this International Standard;

— each statement has an explicitly valid meaning specified herein;

— the totality of statements compose an instance of a valid program which has an explicitly valid meaning specified herein.

An implementation conforms to this International Standard only when

— it accepts and processes programs conforming to this International Standard;

— it reports reasons for rejecting any program which does not conform to this International Standard;

— it interprets errors and exceptional circumstances according to the specifications of this International Standard;

— its interpretation of the semantics of each statement of a standard-conforming program conforms to the specifications in this International Standard;

— its interpretation of the semantics of a standard-conforming program as a whole conforms to the specifications in this International Standard;

— it accepts as input, manipulates, and can generate as output numbers of at least the precision and range specified in this International Standard;

— it is accompanied by a reference manual which clearly defines the actions taken in regard to features which are called "undefined" or "implementation-defined" in this International Standard.

This International Standard does not include requirements for reporting specific syntactic errors in the text of a program.

Implementations conforming to this International Standard may accept programs written in an enhanced language without having to report all constructs not conforming to this International Standard. However, whenever a statement or other program element does not conform to the syntactic rules given herein, either an error shall be reported or the statement or other program element shall have an implementation-defined meaning.

An exception occurs when an implementation recognizes that a program may not perform or is not performing in accordance with this International Standard. All exceptions described in this International Standard shall be reported to the user unless some mechanism provided in an enhancement to this International Standard has been invoked by the user to handle exceptions.

Where indicated, certain exceptions may be handled by the procedures specified in this International Standard; if no procedure is given, or if restrictions imposed by the hardware or the operating environment make it impossible to follow the given procedures, then the exception shall be handled by terminating the program. Enhancements to this International Standard may describe mechanisms for controlling the manner in which exceptions are reported and handled, but no such mechanisms are specified in this International Standard.

This International Standard does not specify an order in which exceptions shall be detected or processed.

## 3 References

ISO 646, *Information processing — 7-bit coded character set for information interchange.*[1)]

ISO 4873, *Information processing — 8-bit coded character set for information interchange.*

ISO 6093, *Information processing — Specification for representation of numeric values in character strings for information interchange.*[2)]

## 4 Definitions

For the purpose of this International Standard, the following definitions apply.

**4.1 BASIC** : A term applied as a name to members of a special class of languages which possess similar syntaxes and semantic meanings; acronym for Beginner's All-purpose Symbolic Instruction Code.

**4.2 batch-mode** : The processing of programs in an environment where no provision is made for user interaction.

**4.3 can** : The word "can" is used in a descriptive sense to indicate that standard-conforming programs are allowed to contain certain constructions and that standard-conforming implementations are required to process such programs correctly.

**4.4 end-of-line** : The character(s) or indicator which identifies the termination of a line. Lines of three kinds may be identified in Minimal BASIC : program *lines*, print lines, and *input-reply* lines. *End-of-lines* may vary between the three cases and may also vary depending upon context. Thus, for example, the *end-of-line* in an *input-reply* may vary on a given system depending on the terminal being used in interactive or batch mode.

Typical examples of *end-of-line* are carriage-return, carriage-return line-feed, and end of record (such as end of card).

**4.5 error** : A flaw in the syntax of a *line* which causes it not to be part of a standard program.

**4.6 exception** : A circumstance arising in the course of executing a *program* which results from faulty data or computations or from exceeding some resource constraint. Where indicated certain exceptions (non-fatal exceptions) may be handled by the specified procedures; if no procedure is given (fatal exceptions) or if restrictions imposed by the hardware or operating environment make it impossible to follow the given procedure, then the exception shall be handled by terminating the *program*.

**4.7 identifier** : A character string used to name a *variable* or a function.

**4.8 interactive mode** : The processing of programs in an environment which permits the user to respond directly to the actions of individual programs and to control the commencement and termination of these programs.

**4.9 keyword** : A character string, usually with the spelling of a commonly used or mnemonic word, which provides a distinctive identification of a statement or a component of a statement of a programming language.

The keywords in Minimal BASIC are : 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, and TO.

**4.10 line** : A single transmission of characters which terminates with an *end-of-line*.

**4.11 machine infinitesimal** : The smallest positive value (other than zero) which can be represented and manipulated by a BASIC implementation.

**4.12 machine infinity** : The positive and negative values of greatest magnitude which can be represented and manipulated by a BASIC implementation. It is not required that manipulations of machine infinity yield non infinite results.

**4.13 may** : The word "may" is used in a permissive sense to indicate that a standard-conforming implementation may or may not provide a particular feature.

**4.14 overflow** : With respect to numeric operations, the condition which exists when a prior operation has attempted to generate a result which exceeds machine infinity.



1) At present at the stage of draft. (Revision of ISO 646-1973.)

2) At present at the stage of draft.

With respect to string operations, the condition which exists when a prior operation attempts to generate a result which has more characters than can be contained in a string of maximal length, as determined by the language processor.

**4.15 print zone:** A contiguous set of character positions in a printed output line which may contain an evaluated *print-statement* element.

**4.16 rounding:** The process by which a representation of a value with lower precision is generated from a representation of higher precision taking into account the value of that portion of the original number which is to be omitted. For example, rounding X to the nearest integer may be accomplished by INT(X + 0,5).

**4.17 shall:** The word "shall" is used in an imperative sense to indicate that a program is required to be constructed, or that an implementation is required to act, as specified in order to meet the constraints of standard conformance.

**4.18 significant digits:** The contiguous sequence of digits between the high-order non-zero digit and the low-order non-zero digit, without regard for the location of the radix point. Commonly, in a normalized floating point internal representation, only the significant digits of a representation are maintained in the significand.

**4.19 truncation:** The process by which a representation of a value with lower precision is generated from a representation of higher precision by merely deleting the unwanted low order digits of the original representation.

**4.20 underflow:** The condition which exists when a prior operation has attempted to generate a result, other than zero, which is less in magnitude than machine infinitesimal. This term is applied only to numeric operations.



## 5 Characters and strings

### 5.1 General description

The character set for BASIC is contained in the International Reference Version of ISO 646. Strings are sequences of characters and are used in BASIC *programs* as comments (see clause 20), as *string-constants* (see clause 7), or as *data* (see clause 16).

### 5.2 Syntax

The syntax shall be defined as follows:

1) *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

2) *digit* = 0/1/2/3/4/5/6/7/8/9

3) *string-character* = *quotation-mark/quoted-string-character*

4) *quoted-string-character* = *exclamation-mark/number-sign/dollar-sign/percent-sign/ampersand/apostrophe/left-parenthesis/right-parenthesis/asterisk/comma/solidus/colon/semicolon/less-than-sign/equals-sign/greater-than-sign/question-mark/circumflex-accent/underline/unquoted-string-character*

5) *unquoted-string-character* = *space/plain-string-character*

6) *plain-string-character* = *plus-sign/minus-sign/full-stop/digit/letter*

7) *remark-string* = *string-character**

8) *quoted-string* = *quotation-mark/quoted-string-character*/quotation-mark*

9) *unquoted-string* = *plain-string-character/(plain-string character unquoted-string-character* plain-string-character)*

### 5.3 Examples

The following examples are taken from 5.2:

7) ANY CHARACTERS AT ALL (?!*!!) CAN BE USED IN A "REMARK".

8) "SPACES, AND COMMAS, CAN OCCUR IN QUOTED STRINGS."

9) COMMAS CANNOT OCCUR IN UNQUOTED STRINGS.

### 5.4 Semantics

The *letters* shall be the set of uppercase roman letters contained in the ISO 7-bit coded character set in positions 4/1 to 5/10.

The *digits* shall be the set of arabic digits contained in the ISO 7-bit coded character set in positions 3/0 to 3/9.

The remaining *string-characters* shall correspond to the remaining graphic characters in positions 2/0 to 2/15, 3/10 to 3/15, 5/14 and 5/15 of the ISO 7-bit coded character set.

The names of characters are specified in table 1.

The coding of characters is specified in table 2; however, this coding applies only when *programs* and/or input/output data are exchanged by means of coded media.

### 5.5 Exceptions

None.

### 5.6 Remarks

Other characters from the ISO 7-bit coded character set (including control characters) may be accepted by an implementation and may have a meaning for some other processor (such as an editor), but have no prescribed meaning within this International Standard. Programs containing characters other than the *string-characters* described in 5.4 and 5.2 and *end-of-line* characters are not standard-conforming *programs*.

The various kinds of characters and strings described by the syntax correspond to the various uses of strings in a BASIC *program*. *Remark-strings* may be used in *remark-statements* (see clause 20). *Quoted-strings* may be used as *string-constants* (see clause 7). *Unquoted-strings* may be used in addition to *quoted-strings* as *data* elements (see clause 18) without being enclosed in *quotation-marks*; *unquoted-strings* cannot contain leading or trailing *spaces*.

## 6 Programs

### 6.1 General description

BASIC is a line-oriented language. A BASIC *program* is a sequence of *lines*, the last of which shall be an *end-line* and each of which shall contain a keyword. Each *line* shall contain a unique *line-number* which serves as a label for the *statement* contained in that *line*.

### 6.2 Syntax

The syntax shall be defined as follows:

1) *program* = *block* end-line*

2) *block* = *statement-line/for-block*

3) *statement-line* = *line-number statement end-of-line*

4) *line-number* = *digit digit? digit? digit?*

5) *end-of-line* = *[implementation-defined]*

6) *end-line* = *line-number end-statement end-of-line*

7) *end-statement* = END

8) *statement* = *data-statement/def-statement/dimension-statement/gosub-statement/goto-statement/if-then-statement/input-statement/let-statement/on-goto-statement/option-statement/print-statement/randomize-statement/read-statement/remark-statement/restore-statement/return-statement/stop-statement*

9) *line* = *statement-line/end-line/for-line/next-line*

## 6.3 Examples

The following example is taken from 6.2:

6) 999 END

## 6.4 Semantics

A BASIC *program* shall be composed of a sequence of *lines* ordered by *line-numbers*, the last *line* of which shall be an *end-line*. Program *lines* shall be executed in sequential order, starting with the first *line*, until

— some other action is dictated by execution of a control *statement* or *for-block*, or

— a fatal exception occurs, or

— a *stop-statement* or *end-statement* is executed.

The syntax as described generates *programs* which contain no *spaces* other than those occurring in *remark-statements*, in certain *quoted-strings* and *unquoted-strings*, or where the presence of a *space* is explicitly indicated by the metaname *space*.

Special conventions shall be observed regarding *spaces*. With the following exceptions, *spaces* may occur anywhere in a BASIC *program* without affecting the execution of that *program* and may be used to improve the appearance and readability of the *program*.

*Spaces* shall not appear

a) at the beginning of a *line*;

b) within keywords;

c) within the word TAB in a *tab-call*;

d) within *numeric-constants*;

e) within *line-numbers*;

f) within function or *variable* names;

g) within multicharacter *relation* symbols.

In addition, *spaces* which appear in *quoted-strings* and *unquoted-strings* are significant.

All keywords in a *program* shall be preceded by at least one *space* and, if not at the end of a *line*, shall be followed by at least one *space*.

Each *line* shall begin with a *line-number*. The values of the integers represented by the *line-numbers* shall be positive and non-zero; leading zeroes shall have no effect. *Statements* shall occur in ascending *line-number* order.

The manner in which the end of a statement *line* is detected is determined by the implementation; for example, the *end-of-line* may be a carriage-return character, a carriage-return character followed by a line-feed character, or the end of a physical record.

*Lines* in a standard-conforming *program* may contain up to 72 characters; the *end-of-line* indicator is not included within this 72 character limit.

The *end-statement* serves both to mark the physical end of the main body of a *program* and to terminate the execution of the *program* when encountered.

### 6.5 Exceptions

None.

### 6.6 Remarks

Local editing facilities may allow for the entry of statement *lines* in any order and also allow for duplicate *line-numbers* and lines containing only a *line-number*. Such editing facilities usually sort the *program* into the proper order; in the case of duplicate *line-numbers*, the last *line* entered with that *line-number* is retained. In many implementations, a line containing only a *line-number* (without trailing *spaces*) is deleted from the *program*.

## 7 Constants

### 7.1 General description

Constants can denote both scalar numeric values and string values.

A *numeric-constant* is a decimal representation in positional notation of a number. There are four general syntactic forms of *numeric-constants*:

a) implicit point representation: sd ... d

b) explicit point unscaled representation: sd ... drd ... d

c) explicit point scaled representation: sd ... drd ... dEsd ... d

d) implicit point scaled representation: sd ... dEsd ... d

where

d is a deciumal *digit*;

r is a *full-stop*;

s is an optional *sign*;

E is the explicit character E.

A *string-constant* is a character string enclosed in *quotation-marks* (see clause 5).

### 7.2 Syntax

The syntax shall be defined as follows:

1) *numeric-constant* = *sign? numeric-rep*

2) *sign* = *plus-sign/minus-sign*

3) *numeric-rep* = *significand exrad?*

4) *significand* = *(integer full-stop?)/(integer? fraction)*

5) *integer* = *digit digit**

6) *fraction* = *full-stop digit digit**

7) *exrad* = *E sign? integer*

8) *string-constant* = *quoted-string*

### 7.3 Examples

The following examples are taken from 7.2:

1) −21.

3) 1E10
   5E-1
   .4E + 1

4) 500
   1

6) .255

8) "XYZ"
   "X − 3B2"
   "1E10"

### 7.4 Semantics

The value of a *numeric-constant* is the number represented by that constant. "E" stands for "times ten to the power"; if no *sign* follows the symbol *E*, then a *plus-sign* is understood. *Spaces* shall not occur in *numeric-constants*.

A *program* may contain numeric representations which have an arbitrary number of *digits*, though implementations may round the values of such representations to an implementation-defined precision of not less than six significant decimal digits.

*Numeric-constants* may also have an arbitrary number of *digits* in the *exrad*, though non-zero constants whose magnitude is outside an implementation-defined range may be treated as exceptions. It is recommended that the implementation-defined range for *numeric-constants* be approximately 1E − 38 to 1E + 38 or larger. Constants whose magnitudes are less than machine infinitesimal shall be replaced by zero, while constants whose magnitudes are larger than machine infinity shall be reported as causing an overflow.

A *string-constant* has as its value the string of all characters between the *quotation-marks*; *spaces* shall not be ignored. The length of a *string-constant*, i.e. the number of characters contained between the *quotation-marks*, is limited only by the length of a *line*.

### 7.5 Exceptions

The evaluation of a *numeric-constant* causes an overflow (non-fatal; the recommended recovery procedure is to supply machine infinity with the appropriate sign, report it and continue).

### 7.6 Remarks

Since this International Standard does not require that strings with more than 18 characters be assignable to *string-variables* (see clause 8), conforming *programs* can use *string-constants* with more than 18 characters only as elements in a *print-list*.

It is recommended that implementations report constants whose magnitudes are less than machine infinitesimal as underflows and continue.



## 8 Variables

### 8.1 General description

*Variables* in BASIC are associated with either numeric or string values and, in the case of *numeric-variables*, may be either simple *variables* or references to elements of one or two dimensional arrays; such references are called subscripted *variables*.

Simple-numeric-variables shall be named by a *letter* followed by an optional digit.

Subscripted *numeric-variables* shall be named by a *letter* followed by one or two *numeric-expressions* enclosed within parentheses.

*String-variables* shall be named by a letter followed by a *dollar sign*.

explicit declarations of variable types are not required; a *dollar-sign* serves to distinguish *string-variables* from *numeric-variables*, and the presence of a *subscript* distinguishes a subscripted *variable* from a simple one.

## 8.2 Syntax

The syntax shall be defined as follows:

1) *variable* = *numeric-variable/string-variable*
2) *numeric-variable* = *simple-numeric-variable/numeric-array-element*
3) *simple-numeric-variable* = *letter digit?*
4) *numeric-array-element* = *numeric-array-name subscript*
5) *numeric-array-name* = *letter*
6) *subscript* = *left-parenthesis numeric-expression (comma numeric-expression)? right parenthesis*
7) *string-variable* = *letter dollar-sign*

## 8.3 Examples

The following examples are taken from 8.2:

3) X
   A5
4) V(3)
   W(X,X + Y/2)
7) S$

## 8.4 Semantics

At any instant in the execution of a *program*, a *numeric-variable* is associated with a single numeric value and a *string-variable* is associated with a single string value. The value associated with a *variable* may be changed by the execution of *statements* in the *program*.

The length of the character string associated with a *string-variable* can vary during the execution of a *program* from a length of zero characters (signifying the null or empty string) to 18 characters.

*Simple-numeric-variables* and *string-variables* are declared implicitly through their appearance in the *program*.

A subscripted *variable* refers to the element in the one or two-dimensional array selected by the value(s) of the subscript(s). The value of each subscript is rounded (see 4.16) to the nearest integer. Unless explicitly declared in a *dimension-statement*, subscripted *variables* are implicitly declared by their first appearance in a *program*. In this case, the range of each subscript is from zero to ten inclusive, unless the presence of an *option-statement* indicates that the range is from one to ten inclusive. *Subscript expressions* shall have values within the appropriate range (see clause 19).

The same *letter* shall not be the name of both a simple *variable* and an array, nor the name of both a one-dimensional and a two-dimensional array.

There is no relationship between a *numeric-variable* and a *string-variable* whose names agree except for the *dollar-sign*.

At the initiation of execution the values associated with all *variables* shall be implementation-defined.

## 8.5 Exceptions

An integer obtained as the value of a subscript expression is not in the range of the explicit or implicit dimensioning bounds (fatal).

## 8.6 Remarks

Since initialization of *variables* is not specified, and hence may vary from implementation to implementation, *programs* that are intended to be transportable should explicitly assign a value to each *variable* before any *expression* involving that *variable* is evaluated.

There are many commonly used alternatives for associating implementation-defined initial values with *variables*; it is recommended that all *variables* are recognizably undefined in the sense that an exception will result from any attempt to access the values of any *variable* before that *variable* is explicitly assigned a value.

# 9 Expressions

## 9.1 General description

*Expressions* shall be either *numeric-expressions* or *string-expressions*.

*Numeric-expressions* may be constructed from *variables*, constants, and function references using the operations of addition, substraction, multiplication, division, and involution (raising to a power).

*String-expressions* are composed of either a *string-variable* or a *string-constant*.

## 9.2 Syntax

The syntax shall be defined as follows:

1) *expression* = *numeric-expression/string-expression*
2) *numeric-expression* = *sign? term (sign term)**
3) *term* = *factor (multiplier factor)**
4) *factor* = *primary (circumflex-accent primary)**
5) *multiplier* = *asterisk/solidus*
6) *primary* = *numeric-variable/numeric-rep/numeric-function-ref/ (left-parenthesis numeric-expression right-parenthesis)*
7) *numeric-function-ref* = *numeric-function-name argument-list?*
8) *numeric-function-name* = *numeric-defined-function/numeric-supplied-function*
9) *argument-list* = *left-parenthesis argument right-parenthesis*
10) *argument* = *numeric-expression*
11) *string-expression* = *string-variable/string-constant*

## 9.3 Examples

The following examples are taken from 9.2:

2) 3*X − Y^2
   A(1) + A(2) + A(3)
   −X/Y
4) 2^(−X)
6) SQR(X^2 + Y^2)

## 9.4 Semantics

The formation and evaluation of *numeric-expressions* shall follow the normal algebraic rules. The symbols *circumflex-accent*, *asterisk*, *solidus*, *plus-sign*, and *minus-sign* shall represent the operations of involution, multiplication, division, addition, and substraction, respectively. Unless parentheses dictate otherwise, involutions shall be performed first, then multiplications and divisions, and finally additions and substractions. In the absence of parentheses, operations of the same precedence shall be associated to the left.

A-B-C shall be interpreted as (A-B)-C, A^B^C as (A^B)^C, A/B/C as (A/B)/C and −A^B as −(A^B).

If an underflow occurs in the evaluation of a *numeric-expression*, then the value generated by the operation which resulted in the underflow shall be replaced by zero.

O^O is defined as 1.

When the order of evaluation of an *expression* is not constrained by the use of parentheses, and if the mathematical use of operators is associative, commutative, or both, then full use of these properties may be made in order to revise the order of evaluation of the *expression*.

In a function reference, the number of *arguments* supplied shall be equal to the number of *parameters* required by the definition of the function.

A function reference is a notation for the invocation of a predefined algorithm, into which the *argument* value, if any, is substituted for the *parameter* (see clauses 10 and 11) which is used in the function definition. All functions referenced in an *expression* shall either be implementation-supplied or be defined in a *def-statement*. The result of the evaluation of the function, achieved by the execution of the defining algorithm, is a scalar numeric value which replaces the function reference in the *expression*.

The value of a *string-expression* is the value of the *string-variable* or the *string-constant* which constitutes that *string-expression*.

### 9.5 Exceptions

Evaluation of an *expression* results in a division by zero (nonfatal; the recommended recovery procedure is to supply machine infinity with the sign of the numerator, report it and continue).

Evaluation of an *expression* results in an overflow (nonfatal; the recommended recovery procedure is to supply machine infinity with the algebraically correct sign, report it and continue).

Evaluation of the operation of involution results in a negative number being raised to a non-integral power (fatal).

Evaluation of the operation of involution results in zero being raised to a negative power (nonfatal; the recommended recovery procedure is to supply positive machine infinity, report it and continue).

### 9.6 Remarks

The accuracy with which the evaluation of an *expression* takes place will vary from implementation to implementation. While no minimum accuracy is specified for the evaluation of *numeric-expressions*, it is recommended that implementations maintain at least six significant decimal digits of precision.

The method of evaluation of the operation of involution may depend upon whether or not the exponent is an integer. If it is, then the indicated number of multiplications may be performed; if it is not, then the operation may be evaluated using the LOG and EXP functions (see clause 10).

It is recommended that implementations report underflow as an exception and continue.

0^0 is defined as 1 to allow commonly occurring formulas such as

P^R * (1 – P)^(N – R),

which appears in the formula for binomial probabilities, to be evaluated properly when P and R are both equal to 0.

## 10 Implementation-supplied functions

### 10.1 General description

Predefined algorithms are supplied by the implementation for the evaluation of commonly used numeric functions.

### 10.2 Syntax

The syntax shall be defined as follows:

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

### 10.3 Examples

None.

### 10.4 Semantics

The values of the implementation-supplied functions, as well as the number of *arguments* required for each function, are described in table 1. In all cases, X stands for a *numeric-expression*.

**Table 1 — Values of implementation-supplied functions**

| Function | Function value |
|---|---|
| ABS(X) | The absolute value of X. |
| ATN(X) | The arctangent of X in radians, i.e., the angle whose tangent is X. The range of the function is $-(pi/2) < ATN(X) < (pi/2)$ where pi is the ratio of the circumference of a circle to its diameter. |
| COS(X) | The cosine of X, where X is in radians. |
| EXP(X) | The exponential of X, i.e., the value of the base of natural logarithms raised to the power X; if EXP(X) is less than machine infinitesimal, then its value shall be replaced by zero. |
| INT(X) | The largest integer not greater than X; for example INT(1.3) = 1 and INT(−1.3) = −2. |
| LOG(X) | The natural logarithm of X; X shall be greater than zero. |
| RND | The next pseudo-random number in an implementation-supplied sequence of pseudo-random numbers uniformly distributed in the range $0 <= RND < 1$ (see also clause 21). |
| SGN(X) | The sign of X: −1 if X < 0; 0 if X = 0, and + 1 if X > 0. |
| SIN(X) | The sine of X, where X is in radians. |
| SQR(X) | The non-negative square root of X; X shall be non-negative. |
| TAN(X) | The tangent of X, where X is in radians. |

### 10.5 Exceptions

The value of the argument of the LOG functions is zero or negative (fatal).

The value of the argument of the SQR function is negative (fatal).

The magnitude of the value of the exponential or tangent function is larger than machine infinity (nonfatal; the recommended recovery procedure is to supply machine infinity with the appropriate sign, report it and continue).

### 10.6 Remarks

The RND function in the absence of a *randomize-statement* (see clause 21) shall generate the same sequence of pseudo-random numbers each time a *program* is run. This convention is chosen so that programs employing pseudo-random numbers can be executed several times with the same result.

It is recommended that, if the value of the exponential function is less than machine infinitesimal, implementations report this as an underflow and continue.

This International Standard requires that overflow be reported only for the final values of *numeric-supplied-functions*; exceptions which occur in the evaluation of these functions need not be reported, though implementations shall take appropriate actions in the event of such exceptions to ensure the accuracy of the final values.

## 11 User-defined functions

### 11.1 General description

In addition to the implementation-supplied functions (see clause 10), BASIC allows the programmer to provide single-line definitions for functions within a *program*.

The general syntactic form of *statements* for defining functions is

DEF FNx = *expression*

or

DEF FNx (*parameter*) = *expression*

where x is a single *letter* and a *parameter* is a simple *numeric-variable*.

### 11.2 Syntax

The syntax shall be defined as follows:

1) *def-statement* = DEF *numeric-defined-function* *parameter-list?* *equals-sign* *numeric-expression*
2) *numeric-defined-function* = FN *letter*
3) *parameter-list* = *left-parenthesis* *parameter* *right-parenthesis*
4) *parameter* *simple-numeric-variable*

### 11.3 Examples

The following examples are taken from 11.2:

1) DEF FNF(X) = X^4 − 1
   DEF FNA(X) = A*X + B
   DEF FNP = 3.14159

### 11.4 Semantics

A function definition specifies the means of evaluating a function in terms of the value of an *expression* involving the *parameter*, if any, appearing in the *parameter-list* and possibly other variables or constants. If the function definition does not contain a *parameter-list*, then references to the function (i.e., *numeric-function-refs* in *expressions* involving the function), shall not contain *argument-lists*. If a function definition does contain a *parameter-list*, then references to the function shall contain *argument-lists*, in which case the *expression* in the *argument-list* is evaluated and its value assigned to the *parameter* in the *parameter-list* for the function definition. The *expression* in the function definition is then evaluated, and this value is assigned as the value of the function.

The *parameter* appearing in the *parameter-list* of a function definition is local to that definition, i.e., it is distinct from any *variable* with the same name outside of the function definition. *Variables* in the *numeric-expression* which do not appear in the *parameter-list* are the *variables* of the same names outside the function definition.

A function definition shall occur in a lower numbered *line* than that of the first reference to the function. The *expression* in a *def-statement* is not evaluated unless the defined function is referenced.

If the execution of a *program* reaches a *line* containing a *def-statement*, then it shall proceed to the next *line* with no other effect.

A function definition may refer to other defined functions, but not to the function being defined. A function shall be defined only once in a *program*.

### 11.5 Exceptions

None.

### 11.6 Remarks

None.

## 12 Let statement

### 12.1 General description

A *let-statement* provides for the assignment of the value of an *expression* to a *variable*. The general syntactic form of the *let-statement* shall be

LET *variable* = *expression*

### 12.2 Syntax

The syntax shall be defined as follows:

1) *let-statement* = *numeric-let-statement*/*string-let-statement*
2) *numeric-let-statement* = LET *numeric-variable equals-sign numeric-expression*
3) *string-let-statement* = LET *string-variable equals-sign string-expression*

### 12.3 Examples

The following examples are taken from 12.2:

2) LET P = 3.14159
   LET A(X,3) = SIN(Y)*Y + 1
3) LET A$ = "ABC"
   LET A$ = B$

### 12.4 Semantics

The *expression* is evaluated (see clause 9) and its value is assigned to the *variable* to the left of the *equals-signs*.

### 12.5 Exceptions

An assignment of a *string-expression* to a *string-variable* results in a string overflow (fatal).

### 12.6 Remarks

None.

## 13 Control statements

### 13.1 General description

Control *statements* allow for the interruption of the normal sequence of execution of *statements* by causing execution to continue at a specified *line*, rather than at one with the next higher *line-number*.

The *goto-statement*

GO TO *line-number*

allows for an unconditional transfer.

The *if-then-statement*

IF exp1 rel exp2 THEN *line-number*

where "exp1" and "exp2" are *expressions* and "rel" is a relational operator, allows for a conditional transfer.

The *gosub-statement* and *return-statement*

GOSUB *line-number*
RETURN

allow for subroutine calls.

The *on-goto-statement*

ON *expression* GO TO *line-number*, ..., *line-number*

allows control to be transferred to a selected line.

The *stop-statement*

STOP

allows for *program* termination.

## 13.2 Syntax

The syntax shall be defined as follows:

1) *goto-statement* = GO *space** TO *line-number*
2) *if-then-statement* = IF *relational-expression* THEN *line-number*
3) *relational-expression* = *(numeric-expression relation numeric-expression)/ (string-expression equality-relation string-expression)*
4) *relation* = *equality-relation//less-than-sign/greater-than-sign/not-less/not-greater*
5) *equality-relation* = *equals-sign/not-equals*
6) *not-less* = *greater-than-sign equals-sign*
7) *not-greater* = *less-than-sign equals-sign*
8) *not-equals* = *less-than-sign greater-than-sign*
9) *gosub-statement* = GO *space** SUB *line-number*
10) *return-statement* = RETURN
11) *on-goto-statement* = ON *numeric-expression* GO *space** TO *line-number (comma line-number)**
12) *stop-statement* = STOP

## 13.3 Examples

The following examples are taken from 13.2:

1) GO TO 999
   GOTO 999
2) IF X > Y + 83 THEN 200
   IF A$ < > B$ THEN 550
9) GO SUB 1000
   GOSUB 1000
11) ON L + 1 GO TO 300, 400, 500
12) STOP

## 13.4 Semantics

A *goto-statement* indicates that execution of the *program* shall be continued at the *line* with the specified *line-number*.

If the value of the *relational-expression* in an *if-then-statement* is true, then execution of the *program* shall be continued from the *line* with the specified *line-number*; if the value of the *relational-expression* is false, then execution shall be continued in sequence, i.e. with the *line* following that containing the *if-then-statement*.

The relation "less than or equal to" shall be denoted by < =. Similarly, "greater than or equal to" shall be denoted by > =, while "not equal to" shall be denoted by < >.

The relation of equality holds between two strings if and only if the two strings have the same length and contain identical sequences of characters.

The execution of the *gosub-statement* and the *return-statement* can be described in terms of a stack of *line-numbers* (but may be implemented in some other fashion). Prior to execution of the first *gosub-statement* by the *program*, this stack is empty. Each time a *gosub-statement* is executed, the *line-number* of the *gosub-statement* is placed on top of the stack and execution of the *program* is

continued at the *line* specified in the *gosub-statement*. Each time a *return-statement* is executed, the *line-number* on top of the stack is removed from the stack and execution of the *program* is continued at the *line* following the one with that *line-number*.

It is not necessary that equal numbers of *gosub-statements* and *return-statements* be executed before termination of the *program*.

The *expression* in an *on-goto-statement* shall be evaluated and rounded to obtain an integer, whose value is then used to select a *line-number* from the list following the GOTO (the *line-numbers* in the list are indexed from left to right, starting with 1). Execution of the *program* shall continue at the *line* with the selected *line-number*.

All *line-numbers* in control-statements shall refer to *lines* in the *program*.

The *stop-statement* causes termination of the *program*.

### 13.5 Exceptions

An attempt is made to execute a *return-statement* without having executed a corresponding *gosub-statement* (fatal).

The integer obtained as the value of an *expression* in an *on-goto-statement* is less than one or greater than the number of *line-numbers* in the list (fatal).

### 13.6 Remarks

None.

## 14 For and next statements

### 14.1 General description

The *for-statement* and *next-statement* provide for the construction of loops. The general syntactic form of the *for-statement* and *next-statement* is

FOR v = *initial-value* TO *limit* STEP *increment*.

NEXT v

where "v" is a *simple-numeric-variable* and the *initial-value*, *limit*, and *increment* are *numeric-expressions*; the clause "STEP *increment*" is optional.

### 14.2 Syntax

The syntax shall be defined as follows:

1) *for-block* = *for-line for-body*
2) *for-body* = *block next-line*
3) *for-line* = *line-number for-statement end-of-line*
4) *next-line* = *line-number next-statement end-of-line*
5) *for-statement* = FOR *control-variable equals-sign initial-value* TO *limit (*STEP *increment)?*
6) *control-variable* = *simple-numeric-variable*
7) *initial-value* = *numeric-expression*
8) *limit* = *numeric-expression*
9) *increment* = *numeric-expression*
10) *next-statement* = NEXT *control-variable*

### 14.3 Examples

The following examples are taken from 14.2:

```
1)  100 FOR I = 1 TO 10
    (other blocks or lines)
    200 NEXT I
5)  FOR I = A TO B STEP -1
10) NEXT C7
```

### 14.4 Semantics

The *for-statement* and the *next-statement* are defined in conjunction with each other. The physical sequence of *statements* beginning with a *for-statement* and continuing up to and including the first *next-statement* with the same *control-variable* is termed a *for-block*. *For-blocks* may be physically nested, i.e., one may contain another, buth they shall not be interleaved, i.e., a *for-block* which contains a *for-statement* or a *next-statement* shall contain the entire *for-block* begun or ended by that *statement*.

Furthermore, physically nested *for-blocks* shall not use the same *control-variable*.

In the absence of a STEP clause in a *for-statement*, the increment is assumed to be + 1.

The action of the *for-statement* and the *next-statement* is defined in terms of other *statements*, as follows:

```
FOR v = initial-value TO limit STEP increment
(block)
NEXT v
```

is equivalent to:

```
line 1  LET own 1 = limit
        LET own 2 = increment
        LET v = initial-value

line 1  IF (v-own 1) * SGN (own 2) > 0 THEN line 2
        (block)
        LET v = v + own 2
        GOTO line 1

line 2  REM continued in sequence
```

Here v is any *simple-numeric-variable*, own 1 and own 2 are variables associated with the particular *for-block* and not accessible to the programmer, and line 1 and line 2 are line-numbers associated with the particular *for-block* and not accessible to the programmer. The variables own 1 and own 2 are distinct from similar variables associated with other *for-blocks*.

A *program* shall not transfer control into a *for-body* by any *statement* other than a *return-statement* (see clause 13).

### 14.5 Exceptions

None.

### 14.6 Remarks

Where arithmetic is approximate (as with decimal fractions in a binary machine), the loop will be executed within the limits of machine arithmetic. No presumptions about approximate achievement of the end test are made. It is noted that in most ordinary situations where machine arithmetic is truncated (rather than rounded), such constructions as

```
FOR X = 0 TO 1 STEP 0.1
```

will work as the user expects, even though 0.1 is not representable exactly in a binary machine. If this is indeed the case, then the construction

```
FOR X = 1 TO 0 STEP −0.1
```

will probably not work as expected.

As specified above, the value of the *control-variable* upon exit from a *for-block* via its *next-statement* is the first value not used; if the exit is via a control *statement*, the *control-variable* retains the value it has when the control *statement* is executed.

The variables "own 1" and "own 2" associated with a *for-block* are assigned values only upon entry to the *for-block* through its *for-statement*.

## 15 Print statement

### 15.1 General description

The *print-statement* is designed for the generation of tabular output in a consistent format.

The general syntactic form of the *print-statement* is

```
PRINT item p item p ... p item
```

where each item is an *expression*, a *tab-call*, or null, and each punctuation mark p is either a *comma* or a *semicolon*.

### 15.2 Syntax

The syntax shall be defined as follows:

1) *print-statement* = PRINT *print-list*
2) *print-list* = *(print-item? print-separator)* print-item?*
3) *print-item* = *expression/tab-call*
4) *tab-call* = TAB *left-parenthesis numeric-expression right-parenthesis*
5) *print-separator* = *comma/semicolon*

### 15.3 Examples

The following example is taken from 15.2:

1)
```
PRINT X
PRINT X, Y
PRINT X, Y, Z
PRINT ,,,X
PRINT
PRINT "X EQUALS", 10
PRINT X; (Y + Z)/2
PRINT TAB(10); A$; "IS DONE."
```

### 15.4 Semantics

The execution of a *print-statement* generates a string of characters for transmission to an external device. This string of characters is determined by the successive evaluation of each *print-item* and *print-separator* in the *print-list*.

*Numeric-expressions* shall be evaluated to produce a string of characters consisting of a leading *space* if the number is positive, or a leading *minus-sign* if the number is negative, followed by the decimal representation of the absolute value of the number and a trailing *space*. The possible decimal representations of a number are the same as those described for *numeric-constants* in clause 7 and are used as follows.

Each implementation shall define two quantities, a significance-width d to control the number of significant decimal *digits* printed in numeric representations, and an exrad-width e to control the number of *digits* printed in the *exrad* component of a numeric representation. The value of d shall be at least six and the value of e shall be at least two.

Each number that can be represented exactly as an *integer* with d or fewer decimal *digits* is output using the implicit point unscaled representation.

All other numbers shall be output using either explicit point unscaled representation or explicit point scaled representation. Numbers which can be represented with d or fewer *digits* in the unscaled representation no less accurately than they can in the scaled representation shall be output using the unscaled representation. For example, if d = 6, then 10^(−6) is output as .000001 and 10^(−7) is output as 1.E−7.

Numbers represented in the explicit point unscaled representation shall be output with up to d significant decimal *digits* and a *full-stop*; trailing zeroes in the fractional part may be omitted. A number with a magnitude less than 1 shall be represented with no *digits* to the left of the *full-stop*. This form requires up to d + 3 characters counting the *sign*, the *full-stop* and the trailing *space*.

Numbers represented in the explicit point scaled representation shall be output in the format

*significand* E *sign integer*

where the value x of the *significand* is in the range 1 < = x < 10 and is to be represented with exactly d *digits* of precision, and where the *exrad* component has one to e *digits*. Trailing zeroes may be omitted in the fractional part of the *significand* and leading zeroes may be omitted from the *exrad*. A *full-stop* shall be printed as part of the *significand*. This form requires up to d + e + 5 characters counting the two *signs*, the *full-stop*, the "E" and a trailing *space*.

*String-expressions* shall be evaluated to generate the corresponding string of characters.

The evaluation of the *semicolon* separator shall generate the null string, i.e., a string of zero length.

The evaluation of a *tab-call* or a *comma* separator depends upon the string of characters already generated by the current or previous *print-statements*. The "current line" is the (possibly empty) string of characters generated since the last *end-of-line* was generated. The "margin" is the number of characters, excluding the *end-of-line* character, that can be output on one line and is defined by the implementation. The "columnar position" of the current line is the print position that will be occupied by the next character output to that line; print positions are numbered consecutively from the left, starting with position one.

Each print-line is divided into a fixed number of print zones, where the number of zones and the length of each zone is implementation-defined. All print zones, except possibly the last one on a line, shall have the same length. This length shall be at least d + e + 6 characters in order to accommodate the printing of numbers in explicit point scaled representation as described above and to allow the *comma* separator to move the printing mechanism to the next zone as described below.

The purpose of the *tab-call* is to set the columnar position of the current line to the specified value prior to printing the next *print-item*. More precisely, the argument of the *tab-call* is evaluated and rounded to the nearest integer n. If n is less than one, an exception occurs. If n is greater than the margin m, then n is reduced by an integral multiple of m so that it is in the range 1 < = n < = m; i.e., n is set equal to

n − m *INT((n − 1)/m).

If the columnar position of the current line is less than or equal to n, then *spaces* are generated, if necessary, to set the columnar position to n; if the columnar position of the current line is greater than n, then an *end-of-line* is generated followed by n − 1 *spaces* to set the columnar position of the new current line to n.

The evaluation of the *comma print-separator* depends upon the columnar position. If this position is neither in the last print zone on a line nor beyond the margin, then one or more *spaces* are generated to set the columnar position to the beginning of the next print zone on the line. If the initial columnar position is in the last print zone on a line, then an *end-of-line* is generated. Finally, if the initial columnar position is beyond the margin (as it would be if evaluation of the last *print-item* exactly filled the line), then an *end of line* is generated followed by enough *spaces* to fill the first print zone on the new line.

Whenever the columnar position is greater than one and the evaluation of the next *print-item* would cause that position to exceed the margin by more than one, then an *end-of-line* is generated prior to the characters generated by that *print-item*.

During the evaluation of a *print-item*, whenever the generation of a character would cause the columnar position to exceed the margin by more than one, an *end-of-line* is generated before that character, resetting the columnar position to one.

When evaluation of a *print-list* is completed, if that *print-list* does not end with a *print-separator*, then a final *end-of-line* is generated; otherwise, no such final *end-of-line* is generated.

### 15.5 Exceptions

The evaluation of a *tab-call* argument generates a value less than one (nonfatal; the recommended recovery procedure is to supply one and continue).

### 15.6 Remarks

The *comma* used as a *print-separator* allows the programmer to tabulate the printing mechanism to fixed tab settings at the end of each print zone.

A completely empty *print-list* will generate an *end-of-line*, thereby completing the current line of output. If this line contained no characters, then a blank line results.

A print line on a typical terminal might be divided into five print zones of fifteen print positions each.

The character string generated by printing the value of a *numeric-expression* contains a single trailing *space*. If the generation of that *spece* would cause the columnar position to exceed the margin by more than one, then implementations may choose not to generate that *space*, thereby allowing the number to be printed in the final print zone on a line.

## 16 Input statement

### 16.1 General description

*Input-statements* provide for user interaction with a running *program* by allowing *variables* to be assigned values that are supplied by a source external to the *program*. The *input-statement* enables the entry of mixed string and numeric *data*, with *data* items being separated by *commas*. The general syntactic form of the *input-statement* is

INPUT *variable*, ... , *variable*.

### 16.2 Syntax

The syntax shall be defined as follows:

1) *input-statement* = INPUT *variable-list*
2) *variable-list* = *variable (comma variable)**
3) *input-prompt* = *[implementation-defined]*
4) *input-reply* = *input-list end-of-line*
5) *input-list* = *padded-datum (comma padded-datum)**
6) *padded-datum* = *space*datum space**
7) *datum* = *quoted-string/unquoted-string*

### 16.3 Examples

The following examples are taken from 16.2:

1) INPUT X
   INPUT X, A$, Y(2)
5) 2, SMITH, −3
   25, 0, −15
7) 3.14159

### 16.4 Semantics

After validation of an *input-reply* supplied during execution of a *program*, an *input-statement* causes the *variables* in the *variable-list* to be assigned, in order, values from the *input-reply*. In the interactive mode, the user of the *program* is informed of the need to supply *data* by the output of an *input-prompt*. In the batch mode, the *input-reply* is requested from the external source by an implementation-defined means. Execution of the *program* is suspended until a valid *input-reply* has been supplied.

The type of each *datum* in the *input-reply* shall correspond to the type of the *variable* to which it is to be assigned; i.e., *unquoted-strings* which are *numeric-constants* shall be supplied as input for *numeric-variables*, and either *quoted-strings* or *unquoted-strings* shall be supplied as input for *string-variables*.

*Subscript expressions* in the *variable-list* are evaluated after values have been assigned to the *variables* preceding them (i.e., to the left of them) in the *variable-list*.

No assigment of values in the *input-reply* shall take place until the *input-reply* has been validated with respect to the type of each *datum*, the number of input items, and the allowable range for each *datum*.

### 16.5 Exceptions

The type of *datum* does not match the type of the *variable* to which it is to be assigned (nonfatal; the recommended recovery procedure is to request that the *input-reply* be resupplied).

There is insufficient *data* in the *input-list* (nonfatal; the recommended recovery procedure is to request that the *input-reply* be resupplied).

There is too much *data* in the *input-list* (nonfatal; the recommended recovery procedure is to request that the *input-reply* be resupplied).

The evaluation of a numeric *datum* causes an overflow (nonfatal; the recommended recovery procedure is to request that the *input-reply* be resupplied).

The assignment of a *datum* to a *string-variable* results in a string overflow (nonfatal; the recommended recovery procedure is to request that the *input-reply* be resupplied.

### 16.6 Remarks

This International Standard requires that users in the interactive mode always be given the option of resupplying erroneous *input-replies*. This International Standard does not require an implementation to provide facilities for correcting erroneous *input-replies*, though such facilities may be provided.

It is recommended that the *input-prompt* consists of a *question-mark* followed by a single *space*.

This International Standard does not require an implementation to output the *input-reply*.

It is recommended that implementations report an invalid reply as an exception and allow the *input-reply* to be resupplied.

If the response to input for a *string-variable* is an *unquoted-string*, leading and trailing *spaces* are ignored (see clause 5). If the response to such input is a *quoted-string*, then all *spaces* are significant (see clause 7).

## 17 Read and restore statements

### 17.1 General description

The *read-statement* provides for the assignment of values to *variables* from a sequence of data created from *data-statements* (see clause 18). The *restore-statement* allows the *data* in the *program* to be reread. The general syntactic forms of the read and restore *statements* are

READ *variable*, ..., *variable*

RESTORE

### 17.2 Syntax

The syntax shall be defined as follows:

1) *read-statement* = READ *variable-list*
2) *restore-statement* = RESTORE

### 17.3 Examples

The following examples are taken from 17.2:

1) READ X, Y, Z
   READ X(1), A$
2) RESTORE

### 17.4 Semantics

The *read-statement* causes *variables* in the *variable-list* to be assigned values, in order, from the sequence of *data* (see clause 18). A conceptual pointer is associated with the data sequence. At the initiation of execution of a *program*, this pointer points to the first *datum* in the data sequence. Each time a *read-statement* is executed, each *variable* in the *variable-list* in sequence is assigned the value of the *datum* indicated by the pointer and the pointer is advanced to point beyond that *datum*.

The *restore-statement* resets the pointer for the data sequence to the beginning of the sequence, so that the next *read-statement* executed will read *data* from the beginning of the sequence once again.

The type of a *datum* in the data sequence shall correspond to the type of a *variable* to which it is to be assigned; i.e., *numeric-variables* require *unquoted-strings* which are *numeric-constants* as data and string-variables require quoted-strings or *unquoted-strings* as *data*. An *unquoted-string* which is a valid numeric representation may be assigned to either a *string-variable* or a *numeric-variable* by a *read-statement*.

If the evaluation of a numeric *datum* causes an underflow, then its value shall be replaced by zero.

*Subscript expressions* in the *variable-list* are evaluated after values have been assigned to the *variables* preceding them (i.e., to the left of them) in the list.

### 17.5 Exceptions

The *variable-list* in a *read-statement* requires more *data* than are present in the remainder of the data sequence (fatal).

An attempt is made to assign a *quoted-string* or an *unquoted-string* which is not a valid representation for *numeric-constant* to a *numeric-variable* (fatal).

The evaluation of a numeric *datum* causes an overflow (nonfatal; the recommended recovery procedure is to supply machine infinity with the appropriate sign, report it and continue).

The assignment of a *datum* to a *string-variable* results in a string overflow (fatal).

### 17.6 Remarks

It is recommended that implementations report an underflow as an exception and continue.

If the evaluation of a numeric *datum* causes an underflow, then its value shall be replaced by zero.

*Subscript expressions* in the *variable-list* are evaluated after values have been assigned to the *variables* preceding them (i.e., to the left of them) in the list.

## 18 Data statement

### 18.1 General description

The *data-statement* provides for the creation of a sequence of representations for *data* elements for use by the *read-statement*. The general syntactic form of the *data-statement* is

DATA *datum*, ... , *datum*

*where each datum* is either an *unquoted-string* (which may represent a *numeric-constant*) or a *quoted-string*.

### 18.2 Syntax

The syntax shall be defined as follows:

1) *data-statement* = DATA *data-list*
2) *data-list* = *datum (comma datum)**

### 18.3 Example

The following example is taken from 18.1:

1) DATA 3.14159, PI, 5E-10, «,»

### 18.4 Semantics

*Data* from the totality of *data-statements* in the *program* are collected into a single data sequence. The order in which *data* appear textually in the totality of all *data-statements* determines the order of the *data* in the data sequence.

If the execution of a *program* reaches a *line* containing a *data-statement*, then it shall proceed to the next *line* with no other effect.

### 18.5 Exceptions

None.

### 18.6 Remarks

Errors in the data-list may generate exceptions during the READ and RESTORE operations, see 17.5.

## 19 Array declarations

### 19.1 General description

The *dimension-statement* is used to reserve space for arrays of one or two dimensions. The *option-statement* is used to define the lower bound for all array subscripts. By use of the *option-statement*, the subscripts of all arrays may be declared to have a lower bound of zero or one. Unless declared otherwise in a *dimension-statement*, all array subscripts shall have an upper bound of ten. Thus the default space allocation reserves space for 10 or 11 elements in one-dimensional arrays and 100 or 121 elements in two-dimensional arrays, depending on the setting of the lower bound. By use of a *dimension-statement*, the subscripts of an array may be declared to have an upper bound other than 10.

The general syntactic form of the *dimension-statement* is

DIM declaration, ..., declaration

where each declaration has the form

letter (integer)

or

letter (integer, integer).

The general syntactic form of the option-statement is

OPTION BASE n

where n shall be either 0 or 1.

### 19.2 Syntax

The syntax shall be defined as follows:

1) *dimension-statement* = DIM *array-declaration(comma array-declaration)**
2) *array-declaration* = *numeric-array-name left-parenthesis bounds right-parenthesis*
3) *bounds* = *integer (comma integer)?*
4) *option-statement* = OPTION BASE *(0/1)*

### 19.3 Examples

The following examples are taken from 19.2:

1) DIM A(6), B(10, 10)
4) OPTION BASE 1

### 19.4 Semantics

Each *array-declaration* occurring in a *dimension-statement* declares the array named to be either one- or two-dimensional, according to whether one or two bounds are listed for the array. In addition, the *bounds* specify the maximum values that *subscript expressions* for the array can have.

The declaration for an array, if present at all, shall occur in a lower numbered *line* than any reference to an element of that array. Arrays that are not declared in any *dimension-statement* are declared implicitly to be one- or two-dimensional according to their use in the *program*, and to have subscripts with a maximum value of ten (see clause 8).

The *option-statement* declares the minimum value for all array subscripts; if no *option-statement* occurs in a *program*, this minimum is zero. An *option-statement*, if present at all, must occur in a lower numbered *line* than any *dimension-statement* or any reference to an element of an array. If an *option-statement* specifies that the lower bound for array subscripts is one, then no *dimension-statement* in the *program* may specify an upper bound of zero. A *program* may contain only one *option-statement*.

If the execution of a *program* reaches a *line* containing a *dimension-statement* or an *option-statement*, then it shall proceed to the next *line* with no other effect.

An array can be explicitly dimensioned only once.

### 19.5 Exceptions

None.

### 19.6 Remarks

None.

## 20 Remark statement

### 20.1 General description

The *remark-statement* allows program annotation.

### 20.2 Syntax

The syntax shall be defined as follows:

1) *remark-statement* = REM *remark-string*

### 20.3 Example

The following example is taken from 20.2:

1) REM FINAL CHECK

### 20.4 Semantics

If the execution of a *program* reaches a *line* containing a *remark-statement*, then it shall proceed to the next *line* with no other effect.

### 20.5 Exceptions

None.

### 20.6 Remarks

None.

## 21 Randomize statement

### 21.1 General description

The *randomize-statement* overrides the implementation-predefined sequence of pseudo-random numbers as values for the RND function, allowing different (and unpredictable) sequences each time a given *program* is executed.

### 21.2 Syntax

The syntax shall be defined as follows :

1) *randomize-statement* = RANDOMIZE

### 21.3 Example

The following example is taken from 21.2 :

1) RANDOMIZE

### 21.4 Semantics

Execution of the *randomize-statement* shall generate a new unpredictable starting point for the list of pseudo-random numbers used by the RND function (see clause 10).

### 21.5 Exceptions

None.

### 21.6 Remarks

In the case of implementations which do not have access to a randomizing device such as a real-time clock, the *randomize-statement* may be implemented by means of an interaction with the user.

Table 2 — BASIC character set

| Name | Graphic |
|---|---|
| Space | |
| Exclamation-mark | ! |
| Quotation-mark | " |
| Number-sign | # |
| Dollar-sign | $ |
| Percent-sign | % |
| Ampersand | & |
| Apostrophe | ' |
| Left-parenthesis | ( |
| Right-parenthesis | ) |
| Asterisk | * |
| Plus-sign | + |
| Comma | , |
| Minus-sign | - |
| Full-stop | . |
| Solidus | / |
| Digits | 0-9 |
| Colon | : |
| Semicolon | ; |
| Less-than-sign | < |
| Equals-sign | = |
| Greater-than-sign | > |
| Question-mark | ? |
| Letters | A-Z |
| Circumflex-accent | ^ |
| Underline | _ |

**Table 3 — BASIC code** (from ISO 646, 7-bit Coded Character Set)

| $b_7$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| $b_6$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $b_5$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $b_4$ $b_3$ $b_2$ $b_1$ | | | | | | | | |
| 0 0 0 0 0 | NUL | TC$_1$ (DLE) | SP | 0 | @ | P | ` | p |
| 0 0 0 1 1 | TC$_1$ (SOH) | DC$_1$ | ! | 1 | A | Q | a | q |
| 0 0 1 0 2 | TC$_2$ (STX) | DC$_2$ | " | 2 | B | R | b | r |
| 0 0 1 1 3 | TC$_3$ (ETX) | DC$_3$ | # | 3 | C | S | c | s |
| 0 1 0 0 4 | TC$_4$ (EOT) | DC$_4$ | ¤ | 4 | D | T | d | t |
| 0 1 0 1 5 | TC$_5$ (ENQ) | TC$_5$ (NAK) | % | 5 | E | U | e | u |
| 0 1 1 0 6 | TC$_6$ (ACK) | TC$_6$ (SYN) | & | 6 | F | V | f | v |
| 0 1 1 1 7 | BEL | TC$_7$ (ETB) | ' | 7 | G | W | g | w |
| 1 0 0 0 8 | FE$_0$ (BS) | CAN | ( | 8 | H | X | h | x |
| 1 0 0 1 9 | FE$_1$ (HT) | EM | ) | 9 | I | Y | i | y |
| 1 0 1 0 10 | FE$_2$ (LF) | SUB | * | : | J | Z | j | z |
| 1 0 1 1 11 | FE$_3$ (VT) | ESC | + | ; | K | [ | k | { |
| 1 1 0 0 12 | FE$_4$ (FF) | IS$_4$ (FS) | , | < | L | \ | l | \| |
| 1 1 0 1 13 | FE$_5$ (CR) | IS$_3$ (GS) | - | = | M | ] | m | } |
| 1 1 1 0 14 | SO | IS$_2$ (RS) | . | > | N | ^ | n | ~ |
| 1 1 1 1 15 | SI | IS$_1$ (US) | / | ? | O | _ | o | DEL |



NOTES

1 In the ISO 7-bit and 8-bit code tables two characters are allocated to position 2/4, namely $ and ¤. In any version of the codes a single character is to be allocated to this position. The character of the 7-bit or of the 8-bit coded character set which corresponds to the character $ of the Minimal BASIC character set is either $ or ¤ (¤ in the International Reference Version).

The same applies to position 2/3 for the characters £ and #, the latter being the character of the International Reference Version.

2 Shaded characters are not part of the BASIC character set.