**ANSI ®**
**X3J18-199X**

American National Standard
for Information Systems –

# Programming Language REXX

## Contents

## Foreword (This foreword is not part of American National Standard X3J18-199X.)

### Purpose

This standard provides an unambiguous definition of the programming language Rexx. Its purpose is to facilitate portability of Rexx programs for use on a wide variety of computer systems.

### History

The computer programming language Rexx was designed by Mike Cowlishaw to satisfy the following principal aims:

– to provide a highly readable command programming language for the benefit of programmers and program readers, users and maintainers;

– to incorporate within this language program design features such as natural data typing and control structures which would contribute to rapid, efficient and accurate program development;

– to define a language whose implementations could be both reliable and efficient on a wide variety of computing platforms.

However, it has become apparent that Rexx has attributes that go far beyond these original goals. Given this retrospect, we would now add a fourth goal:

– to provide a programming language that is appropriate for a wide variety of uses of both systems and applications software.

Rexx is being used increasingly in the commercial and educational sectors; this standard is primarily a consequence of the growing commercial interest in Rexx and the need to promote the portability of Rexx programs between data processing systems. In drafting this standard the continued stability of Rexx has been a prime objective.

In June, 1990, a proposal to develop an X3 Standard for the Rexx programming language was submitted to the X3 Secretariat/CBEMA. This proposal called for the establishment of a new X3 Technical Committee for Rexx.

In November, 1990, X3 announced the formation of a new technical committee, X3J18, to develop an American National Standard for Rexx. The first meeting of this committee as X3J18 was held in January, 1991.

### Committee lists

This standard was prepared by the Technical Development Committee for Rexx, X3J18.

There are two annexes in this standard. Both annexes are informative and are not considered part of this standard.

Suggestions for improvement of this standard will be welcome. They should be sent to the

Information Technology Industry Council, 1250 Eye Street, NW, Washington DC 20005-3922.

This standard was processed and approved for submittal to ANSI by the Accredited Standards Committee on Information Processing Systems, X3. Committee approval of this standard does not necessarily imply that all committee members voted for its approval. At the time it approved this standard, the X3 Committee had the following members:

*To be inserted*

The people who contributed to Technical Committee X3J18 on REXX, which developed this standard, include:

Brian Marks, Chair
Neil Milsted, Vice-Chair

| | |
|---|---|
| Thomas Brawn | Linda Littleton |
| Ian Collier | Reed Meseck |
| Mike Cowlishaw | William Potvin |
| Cathie Dager | David Robin |
| Charles Daney | Mike Sinz |
| Chip Davis | Ed Spire |
| Dave Gomberg | Bernie Style |
| Linda Suskind Green | Keith Watts |
| Klaus Hansjakob | Bebo White |
| Bill Hawes | Dean Williams |
| Luc Lafrance | |

# 0 Introduction

This standard provides an unambiguous definition of the programming language REXX.

# 1 Scope, purpose, and application

## 1.1 Scope

This standard specifies the semantics and syntax of the programming language REXX by specifying requirements for a conforming language processor. The scope of this standard includes

– the syntax and constraints of the REXX language;

– the semantic rules for interpreting REXX programs;

– the restrictions and limitations that a conforming language processor may impose;

– the semantics of configuration interfaces.

– the syntax of the configuration interfaces.

This standard does not specify

– the mechanism by which REXX programs are transformed for use by a data-processing system;

– the mechanism by which REXX programs are invoked for use by a data-processing system;

– the mechanism by which input data are transformed for use by a REXX program;

– the mechanism by which output data are transformed after being produced by a REXX program;

– the encoding of REXX programs;

– the encoding of data to be processed by REXX programs;

– the encoding of output produced by REXX programs;

– the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular language processor;

– all minimal requirements of a data-processing system that is capable of supporting a conforming language processor;

## 1.2 Purpose

The purpose of this standard is to facilitate portability of REXX programs for use on a wide variety of configurations.

## 1.3 Application

This standard is applicable to REXX language processors.

## 1.4 Recommendation

It is recommended that before detailed reading of this standard, a reader should first be familiar with the REXX language, for example through reading one of the books about REXX. It is also recommended that Annex A and Annex B should be read in conjunction with this standard.

# 2 Normative references

There are no standards which constitute provisions of this American National Standard.

# 3 Definitions and document notation

## 3.1 Definitions

**3.1.1 application programming interface:** A set of functions which allow access to some REXX facilities from non-REXX programs.

**3.1.2 arguments:** The expressions (separated by commas) between the parentheses of a function call or following the name on a CALL instruction. Also the corresponding values which may be accessed by a function or routine, however invoked.

**3.1.3 built-in function:** A function (which may be called as a subroutine) that is defined in section 9 of this standard and can be used directly from a program.

**3.1.4 character string:** A sequence of zero or more characters.

**3.1.5 clause:** A section of the program, ended by a semicolon. The semicolon may be implied by the end of a line or by some other constructs.

**3.1.6 coded:** A coded string is a string which is not necessarily comprised of characters.

Coded strings can occur as arguments to a program, results of external routines and commands, and the results of some built-in functions, such as D2C.

**3.1.7 command:** A clause consisting of just an expression is an instruction known as a command. The expression is evaluated and the result is passed as a command string to some external environment.

**3.1.8 condition:** A specific event, or state, which can be trapped by CALL ON or SIGNAL ON.

**3.1.9 configuration:** Any data-processing system, operating system and software used to operate a language processor.

**3.1.10 conforming language processor:** A language processor which obeys all the provisions of this standard.

**3.1.11 construct:** A named syntax grouping, for example "expression", "do_specification".

**3.1.12 default error stream:** An output stream, determined by the configuration, on which error messages are written.

**3.1.13 default input stream:** An input stream having a name which is the null string. The use of this stream may be implied.

**3.1.14 default output stream:** An output stream having a name which is the null string. The use of this stream may be implied.

**3.1.15 direct symbol:** A symbol which, without any modification, names a variable in a variable pool.

**3.1.16 dropped:** A symbol which is in an unitialized state, as opposed to having had a value assigned to it, is described as dropped. The names in a variable pool have an attribute of 'dropped' or 'not-dropped'.

**3.1.17 encoding:** The relation between a character string and a corresponding number. The encoding of character strings is determined by the configuration.

**3.1.18 end-of-line:** An event that occurs during the scanning of a source program. Normally the end-of-lines will relate to the lines shown if the configuration lists the program. They may, or may not, correspond to characters in the source program.

**3.1.19 environment:** The context in which a command may be executed. This is comprised of the environment name, details of the resource that will provide input to the command, and details of the resources that will receive output of the command.

**3.1.20 environment name:** The name of an external procedure or process that can execute commands. Commands are sent to the current named environment, initially selected externally but then alterable by using the ADDRESS instruction.

**3.1.21 error number:** A number which identifies a particular situation which has occurred during processing. The message prose associated with such a number is defined by this standard.

**3.1.22 exposed:** Normally, a symbol refers to a variable in the most recently established variable pool. When this is not the case the variable is referred to as an exposed variable.

**3.1.23 expression:** The most general of the constructs which can be evaluated to produce a single string value.

**3.1.24 external data queue:** A queue of strings that is external to REXX programs in that other programs may have access to the queue whenever REXX relinquishes control to some other program.

**3.1.25 external routine:** A function or subroutine that is neither built-in nor in the same program as the CALL instruction or function call that invokes it.

**3.1.26 external variable pool:** A named variable pool supplied by the configuration which can be accessed by the VALUE built-in function.

**3.1.27 function:** Some processing which can be invoked by name and will produce a result. This term is used for both REXX functions (see section 7.5) and functions provided by the configuration (see section 5).

**3.1.28 identifier:** The name of a construct.

**3.1.29 implicit variable:** A tailed variable which is in a variable pool solely as a result of an operation on its stem. The names in a variable pool have an attribute of 'implicit' or 'not-implicit'.

**3.1.30 instruction:** One or more clauses that describe some course of action to be taken by the language processor.

**3.1.31 internal routine:** A function or subroutine that is in the same program as the CALL instruction or function call that invokes it.

**3.1.32 keyword:** This standard specifies special meaning for some tokens which consist of letters and have particular spellings, when used in particular contexts. Such tokens, in these contexts, are keywords.

**3.1.33 label:** A clause that consists of a single symbol or a literal followed by a colon.

**3.1.34 language processor:** Compiler, translator or interpreter working in combination with a configuration.

**3.1.35 notation function:** A function with the sole purpose of providing a notation for describing semantics, within this standard. No REXX program can invoke a notation function.

**3.1.36 null clause:** A clause which has no tokens.

**3.1.37 null string:** A character string with no characters, that is, a string of length zero.

**3.1.38 production:** The definition of a construct, in Backus-Naur form.

**3.1.39 return code:** A string that conveys some information about the command that has been executed. Return codes usually indicate the success or failure of the command but can also be used to represent other information.

**3.1.40 routine:** Some processing which can be invoked by name.

**3.1.41 state variable:** A component of the state of progress in processing a program, described in this standard by a named variable. No REXX program can directly access a state variable.

**3.1.42 stem:** If a symbol naming a variable contains a period which is not the first character, the part of the symbol up to and including the first period is the stem.

**3.1.43 stream:** Named streams are used as the sources of input and the targets of output. The total semantics of such a stream are not defined in this standard and will depend on the configuration. A stream may be a permanent file in the configuration or may be something else, for example the input from a keyboard.

**3.1.44 string:** For many operations the unit of data is a string. It may, or may not, be comprised of a sequence of characters which can be accessed individually.

**3.1.45 subcode:** The decimal part of an error number.

**3.1.46 subroutine:** An internal, built-in, or external routine that may or may not return a result string and is invoked by the CALL instruction. If it returns a result string the subroutine can also be invoked by a function call, in which case it is being called as a function.

**3.1.47 symbol:** A sequence of characters used as a name, see section 6.2.2. Symbols are used to name variables, functions, etc.

**3.1.48 tailed name:** The names in a variable pool have an attribute of 'tailed' or 'non-tailed'. Otherwise identical names are distinct if their attributes differ. Tailed names are normally the result of replacements in the tail of a symbol, the part that follows a stem.

**3.1.49 token:** The unit of low-level syntax from which high-level constructs are built. Tokens are literal strings, symbols, operators, or special characters.

**3.1.50 trace:** A description of some or all of the clauses of a program, produced as each is executed.

**3.1.51 trap:** A function provided by the user which replaces or augments some normal function of the language processor.

**3.1.52 variable pool:** A collection of the names of variables and their associated values.

## 3.2 Document notation

### 3.2.1 REXX Code

Some REXX code is used in this standard. This code shall be assumed to have its private set of variables. Variables used in this code are not directly accessible by the program to be processed. Comments in the code are not part of the provisions of this standard.

### 3.2.2 Italics

Throughout this standard, except in Rexx code, references to the constructs defined in section 6 are italicized.

## 4 Conformance

### 4.1 Conformance

A conforming language processor shall not implement any variation of this standard except where this standard permits. Such permitted variations shall be implemented in the manner prescribed by this standard and noted in the documentation accompanying the processor.

A conforming processor shall include in its accompanying documentation

- a list of all definitions or values for the features in this standard which are specified to be dependent on the configuration.
- a statement of conformity, giving the complete reference of this standard (ANSI X3J18-199x) with which conformity is claimed.

### 4.2 Limits

Aside from the items listed here (and the assumed limitation in resources of the configuration), a conforming language processor shall not put numerical limits on the content of a program.

Where a limit expresses the limit on a number of digits, it shall be a multiple of three. Other limits shall be one of the numbers one, five or twenty five, or any of these multiplied by some power of ten.

Limitations that conforming language processors may impose are:

- NUMERIC DIGITS values shall be supported up to a value of at least nine hundred and ninety nine.
- Exponents shall be supported. The limit of the absolute value of an exponent shall be at least as large as the largest number that can be expressed without an exponent in nine digits.
- String lengths shall be supported. The limit on the length shall be at least as large as the largest number that can be expressed without an exponent in nine digits.
- String literal length shall be supported up to at least two hundred and fifty.
- Symbol length shall be supported up to at least two hundred and fifty.

# 5 Configuration

Any implementation of this standard will be functioning within a configuration. In practice, the boundary between what is implemented especially to support REXX and what is provided by the system will vary from system to system. This section describes what they shall together do to provide the configuration for the REXX language processing which is described in this standard.

## 5.1 Notation

The interface to the configuration is described in terms of functions. The notation for describing the interface functionally uses the name given to the function, followed by any arguments. This does not constrain how a specific implementation provides the function, nor does it imply that the order of arguments is significant for a specific implementation.

The names of the functions are used throughout this standard; the names used for the arguments are used only in this section and section 7.5.4.

The name of a function refers to its usage. A function whose name starts with

– `Config_` is used only from the language processor when processing programs;

– `API_` is part of the application programming interface and is accessible from programs which are not written in the REXX language;

– `Trap_` is not provided by the language processor but may be invoked by the language processor.

As its result each function shall return a completion Response. This is a string indicating how the function behaved. The completion response may be the character 'N' indicating the normal behavior occurred; otherwise the first character is an indicator of a different behavior and the remainder shall be suitable as a human-readable description of the function's behavior.

This standard defines any additional results from `Config_` functions as made available to the language processor in variables. This does not constrain how a particular implementation should return these results.

### 5.1.1 Notation for completion response and conditions

As alternatives to the normal indicator 'N', each function may return a completion response with indicator 'X' or 'S'; other possible indicators are described for each function explicitly. The indicator 'X' means that the function failed because resources were exhausted. The indicator 'S' shows that the configuration was unable to perform the function.

Certain indicators cause conditions to be raised. The possible raising of these conditions is implicit in the use of the function; it is not shown explicitly when the functions are used in this standard.

The implicit action is

```
call #Raise 'SYNTAX', Message, Description
```

where:

#Raise raises the condition, see section 8.4.1.

Message is determined by the indicator in the completion response. If the indicator is 'X' then Message is 5.1. If the indicator is 'S' then Message is 48.1.

Description is the description in the completion response.

The 'SYNTAX' condition 5.1 can also be raised by any other activity of the language processor.

## 5.2 Processing initiation

The processing initiation interface consists of a function which the configuration shall provide to invoke the language processor.

### 5.2.1 API_Start

Syntax:

**`API_Start(How, Source, Environment, Arguments, Streams, Traps)`**

where:

How is one of 'COMMAND', 'FUNCTION', or 'SUBROUTINE' and indicates how the program is invoked.

Source is an identification of the source of the program to be processed.

Environment is the initial value of the environment to be used in processing commands. This has components for the name of the environment and how the input and output of commands is to be directed.

Arguments is the initial argument list to be used in processing. This has components to specify the number of arguments, which arguments are omitted, and the values of arguments that are not omitted.

Streams has components for the default input stream to be used and the default output stream to be used.

Traps is the list of traps to be used in processing (see section 5.12). This has components to specify whether each trap is omitted or not.

Semantics:

This function starts the execution of a REXX program.

If the program was terminated due to a RETURN or EXIT instruction without an expression the completion response is 'N'.

If the program was terminated due to a RETURN or EXIT instruction with an expression the indicator in the completion response is 'R' and the description of the completion response is the value of the expression.

If the program was terminated due to an error the indicator in the completion response is 'E' and the description in the completion response comprises information about the error that terminated processing.

## 5.3 Source programs and character sets

The configuration shall provide the ability to access source programs (see section 5.4.1).

Source programs consist of characters belonging to the following categories:

– syntactic_characters;

– extra_letters;

– other_blank_characters;

– other_negators;

– other_characters.

A character shall belong to only one category.

### 5.3.1 Syntactic_characters

The following characters represent the category of characters called syntactic_characters, identified by their names. The glyphs used to represent them in this document are also shown. Syntactic_characters shall be available in every configuration:

| | | |
|---|---|---|
| – | & | ampersand; |
| – | ' | apostrophe, single quotation mark, single quote; |
| – | * | asterisk, star; |
| – | | blank, space; |
| – | A-Z | capital letters A through Z; |
| – | : | colon; |
| – | , | comma; |
| – | 0-9 | digits zero through nine; |
| – | = | equal sign; |
| – | ! | exclamation point, exclamation mark; |
| – | > | greater-than sign; |
| – | - | hyphen, minus sign; |
| – | < | less-than sign; |
| – | ( | left parenthesis; |
| – | % | percent sign; |
| – | . | period, decimal point, full stop, dot; |
| – | + | plus sign; |
| – | ? | question mark; |
| – | " | quotation mark, double quote; |
| – | \ | reverse slant, reverse solidus, backslash; |
| – | ) | right parenthesis; |
| – | ; | semicolon; |
| – | / | slant, solidus, slash; |
| – | a-z | small letters a through z; |
| – | _ | underline, low line, underscore; |
| – | \| | vertical line, bar, vertical bar. |

### 5.3.2 Extra_letters

A configuration may have a category of characters in source programs called extra_letters. Extra_letters are determined by the configuration.

### 5.3.3 Other_blank_characters

A configuration may have a category of characters in source programs called other_blank_characters. Other_blank_characters are determined by the configuration. Only the following characters represent possible characters of this category:

– carriage return;

– form feed;

| | | |
|---|---|---|
| – | | horizontal tabulation; |
| – | | new line; |
| – | | vertical tabulation. |

### 5.3.4 Other_negators

A configuration may have a category of characters in source programs called other_negators. Other_negators are determined by the configuration. Only the following characters represent possible characters of this category. The glyphs used to represent them in this document are also shown:

| | | |
|---|---|---|
| – | ^ | circumflex accent, caret; |
| – | ¬ | not sign. |

### 5.3.5 Other_characters

A configuration may have a category of characters in source programs called other_characters. Other_characters are determined by the configuration.

## 5.4 Configuration characters and encoding

The configuration characters and encoding interface consists of functions which the configuration shall provide which are concerned with the encoding of characters.

The following functions shall be provided:

- Config_SourceChar;
- Config_OtherBlankCharacters;
- Config_Upper;
- Config_Compare;
- Config_B2C;
- Config_C2B;
- Config_Substr;
- Config_Length;
- Config_Xrange.

### 5.4.1 Config_SourceChar

Syntax:

```
Config_SourceChar()
```

Semantics:

Supply the characters of the source program in sequence, together with the EOL and EOS events. The EOL event represents the end of a line. The EOS event represents the end of the source program. The EOS event must only occur immediately after an EOL event. Either a character or an event is supplied on each invocation, by setting #Outcome.

If this function is unable to supply a character because the source program encoding is incorrect the indicator of the completion response is 'E' and the description of the completion information is the encoding which is incorrect, in hexadecimal notation.

### 5.4.2 Config_OtherBlankCharacters

Syntax:

**`Config_OtherBlankCharacters()`**

Semantics:

Get other_blank_characters (see section 5.3.3).

Set #Outcome to a string of zero or more unique characters in arbitrary order. Each character is one that the configuration considers equivalent to the character Blank for the purposes of parsing.

### 5.4.3 Config_Upper

Syntax:

**`Config_Upper(Character)`**

where:

Character is the character to be translated to uppercase.

Semantics:

Translate Character to uppercase. Set #Outcome to the translated character. Characters which have been subject to this translation are referred to as being in uppercase. Config_Upper applied to a character in uppercase must not change the character.

### 5.4.4 Config_Compare

Syntax:

**`Config_Compare(Character1, Character2)`**

where:

Character1 is the character to be compared with Character2.

Character2 is the character to be compared with Character1.

Semantics:

Compare two characters. Set #Outcome to

– 'equal' if Character1 is equal to Character2;

– 'greater' if Character1 is greater than Character2;

– 'lesser' if Character1 is less than Character2.

The function shall exhibit the following characteristics. If Config_Compare(a,b) produces

– 'equal' then Config_Compare(b,a) produces 'equal';

– 'greater' then Config_Compare(b,a) produces 'lesser';

– 'lesser' then Config_Compare(b,a) produces 'greater';

– 'equal' and Config_Compare(b,c) produces 'equal' then Config_Compare(a,c) produces 'equal';

– 'greater' and Config_Compare(b,c) produces 'greater' then Config_Compare(a,c) produces 'greater';

– 'lesser' and Config_Compare(b,c) produces 'lesser' then Config_Compare(a,c) produces 'lesser'.

Syntactic characters which are different characters shall not compare equal by Config_Compare, see section 5.3.1.

### 5.4.5 Config_B2C

Syntax:

**`Config_B2C(Binary)`**

where:

Binary is a sequence of digits, each '0' or '1'. The number of digits shall be a multiple of eight.

Semantics:

Translate Binary to a coded string. Set #Outcome to the resulting string. The string may, or may not, correspond to a sequence of characters.

### 5.4.6 Config_C2B

Syntax:

**`Config_C2B(String)`**

where:

String is a string.

Semantics:

Translate String to a sequence of digits, each '0' or '1'. Set #Outcome to the result. This function is the inverse of Config_B2C.

### 5.4.7 Config_Substr

Syntax:

**`Config_Substr(String, n)`**

where:

String is a string.

n is an integer identifying a position within String.

Semantics:

Copy the n-th character from String. The leftmost character is the first character. Set #Outcome to the resulting character.

If this function is unable to supply a character because there is no n-th character in String the indicator of the completion response is 'M'.

If this function is unable to supply a character because the encoding of String is incorrect the indicator of the completion response is 'E' and the description of the completion information is the encoding which is incorrect, in hexadecimal notation.

### 5.4.8 Config_Length

Syntax:

**`Config_Length(String)`**

where:

String is a string.

Semantics:

Set #Outcome to the length of the string, that is, the number of characters in the string.

If this function is unable to determine a length because the encoding of String is incorrect the indicator of the completion response is 'E' and the description of the completion information is the encoding which is incorrect, in hexadecimal notation.

#### 5.4.9 Config_Xrange

Syntax:

**`Config_Xrange(Character1, Character2)`**

where:

Character1 is the null string, or a single character.

Character2 is the null string, or a single character.

Semantics:

If Character1 is the null string then let LowBound be the lowest ranked character in the character set according to the ranking order provided by Config_Compare; otherwise let LowBound be Character1.

If Character2 is the null string then let HighBound be the highest ranked character in the character set according to the ranking order provided by Config_Compare; otherwise let HighBound be Character2.

If #Outcome after Config_Compare(LowBound,HighBound) has a value of

– 'equal' then #Outcome is set to LowBound;

– 'lesser' then #Outcome is set to the sequence of characters between LowBound and HighBound inclusively, in ranking order;

– 'greater' then #Outcome is set to the sequence of characters HighBound and larger, in ranking order, followed by the sequence of characters LowBound and smaller, in ranking order.

### 5.5 Commands

The commands interface consists of a function which the configuration shall provide for strings to be passed as commands to an environment.

See sections 8.3.1 and 8.3.5 for a description of language features that use commands.

#### 5.5.1 Config_Command

Syntax:

**`Config_Command(Environment, Command)`**

where:

Environment is the environment to be addressed. It has components for:

— The name of the environment.

— The name of a stream from which the command will read its input. The null string indicates use of the default input stream.

— The name of a stream on to which the command will write its output. The null string indicates use of the default output stream. There is an indication of whether writing is to APPEND or REPLACE.

— The name of a stream on to which the command will write its output. The null string indicates use of the default error output stream. There is an indication of whether writing is to APPEND or REPLACE

Command is the command to be executed.

Semantics:

Perform a command.

– Set the indicator to 'E' or 'F' if the command ended with an ERROR condition, or a FAILURE condition, respectively.

– Set #RC to the return code string of the command.

## 5.6 External routines

The external routines interface consists of a function which the configuration shall provide to invoke external routines.

See sections 7.5 and 8.3.4 for a description of the language features that use external routines.

### 5.6.1 Config_ExternalRoutine

Syntax:

**`Config_ExternalRoutine(How, NameType, Name, Environment, Arguments, Streams, Traps)`**

where:

How is one of 'FUNCTION' or 'SUBROUTINE' and indicates how the external routine is to be invoked.

NameType is a specification of whether the name was provided as a symbol or as a string literal.

Name is the name of the routine to be invoked.

Environment is an environment value with the same components as on API_Start.

Arguments is a specification of the arguments to the routine, with the same components as on API_Start.

Streams is a specification of the default streams, with the same components as on API_Start.

Traps is the list of traps to be used in processing, with the same components as on API_Start.

Semantics:

Invoke an external routine. Set #Outcome to the result of the external routine, or set the indicator to 'D' if the external routine did not provide a result.

If this function is unable to locate the routine the indicator of the completion response is 'U'. As a result SYNTAX condition 43.1 is raised implicitly.

If How indicated that a result from the routine was required but the routine did not provide one the indicator of the completion response is 'H'. As a result SYNTAX condition 44.1 is raised implicitly.

If How indicated that a result from the routine was required but the routine provided one that was too long (see #Limit_String in section 5.10.1) the indicator of the completion response is 'L'. As a result SYNTAX condition 52 is raised implicitly.

If the routine failed in a way not indicated by some other indicator the indicator of the completion response is 'F'. As a result SYNTAX condition 40.1 is raised implicitly.

## 5.7 External data queue

The external data queue interface consists of functions which the configuration shall provide to manipulate an external data queue mechanism.

See sections 8.3.17, 8.3.19, 8.3.20, 8.3.21, and 9.8.2 for a description of language features that use the external data queue.

The configuration shall provide an external data queue mechanism. The following functions shall be provided:

– Config_Push;

– Config_Queue;

– Config_Pull;

– Config_Queued.

The configuration may permit the external data queue to be altered in other ways. In the absence of such alterations the external data queue shall be an ordered list. Config_Push adds the specified string to one end of the list, Config_Queue to the other. Config_Pull removes a string from the end that Config_Push adds to unless the list is empty.

### 5.7.1 Config_Push

Syntax:

**`Config_Push(String)`**

where:

String is the value to be retained in the external data queue.

Semantics:

Add String as item to the end of the external data queue from which Config_Pull will remove an item.

### 5.7.2 Config_Queue

Syntax:

**`Config_Queue(String)`**

where:

String is the value to be retained in the external data queue.

Semantics:

Add String as item to the opposite end of the external data queue from which Config_Pull will remove an item.

### 5.7.3 Config_Pull

Syntax:

**`Config_Pull()`**

Semantics:

Retrieve an item from the end of the external data queue to which Config_Push adds an element to the list. Set #Outcome to the value of the retrieved item.

If no item could be retrieved the indicator of the completion response is 'F'.

### 5.7.4 Config_Queued

Syntax:

**`Config_Queued()`**

Semantics:

Get the count of items in the external data queue. Set #Outcome to that number.

## 5.8 Streams

The streams interface consists of functions which the configuration shall provide to manipulate streams.

See sections 8.3.1, 8.3.17, and 9.7 for a description of language features which use streams.

Streams are identified by names and provide for the reading and writing of data. They shall support the concepts of characters, lines, positioning, default input stream and default output stream.

The concept of a persistent stream shall be supported and the concept of a transient stream may be supported. A persistent stream is one where the content is not expected to change except when the stream is explicitly acted on. A transient stream is one where the data available is expected to vary with time.

The concepts of binary and character streams shall be supported. The content of a character stream is expected to be characters.

The null string is used as a name for both the default input stream and the default output stream. The null string names the default output stream only when it is an argument to the Config_Stream_Charout operation.

The following functions shall be provided:

- Config_Stream_Charin;
- Config_Stream_Position;
- Config_Stream_Command;
- Config_Stream_State;
- Config_Stream_Charout;
- Config_Stream_Qualified;
- Config_Stream_Unique;
- Config_Stream_Query;
- Config_Stream_Close;
- Config_Stream_Count.

The results of these functions are described in terms of the following stems with tails which are stream names:

- #Charin_Position.Stream;
- #Charout_Position.Stream;
- #Linein_Position.Stream;
- #Lineout_Position.Stream.

### 5.8.1 Config_Stream_Charin

Syntax:

**`Config_Stream_Charin(Stream, OperationType)`**

where:

Stream is the name of the stream to be processed.

OperationType is one of 'CHARIN', 'LINEIN', or 'NULL'.

Semantics:

Read from a stream. Increase #Linein_Position.Stream by one when the end-of-line indication is encountered. Increase #Charin_Position.Stream when the indicator will be 'N'.

If OperationType is 'CHARIN' the state variables describing the stream will be affected as follows:

– When the configuration is able to provide data from a transient stream or the character at position #Charin_Position.Stream of a persistent stream then #Outcome shall be set to contain the data. The indicator of the response shall be 'N'.

– When the configuration is unable to return data because the read position is at the end of a persistent stream then the indicator of the response shall be 'O'.

- When the configuration is unable to return data from a transient stream because no data is available and no data is expected to become available then the indicator of the response shall be 'O'.

– Otherwise the configuration is unable to return data and does not expect to be able to return data by waiting; the indicator of the response shall be 'E'.

The data set in #Outcome will either be a single character or will be a sequence of eight characters, each '0' or '1'. The choice is decided by the configuration. The eight character sequence indicates a binary stream, see section 5.8.8.

If OperationType is 'LINEIN' then the action is the same as if Operation had been 'CHARIN' with the following additional possibility. If end-of-line is detected any character (or character sequence) which is an embedded indication of the end-of-line is skipped. The characters skipped contribute to the change of #Charin_Position.Stream. #Outcome is the null string.

If OperationType is 'NULL' then the stream is accessed but no data is read.

### 5.8.2 Config_Stream_Position

Syntax:

**`Config_Stream_Position(Stream, OperationType, Position)`**

where:

Stream is the name of the stream to be processed.

Operation is 'CHARIN', 'LINEIN', 'CHAROUT', or 'LINEOUT'.

Position indicates where to position the stream.

Semantics:

If the operation is 'CHARIN' or 'CHAROUT' then Position is a character position, otherwise Position is a line position.

If Operation is 'CHARIN' or 'LINEIN' and the Position is beyond the limit of the existing data then the indicator of the completion response shall be 'R'. Otherwise if Operation is 'CHARIN' or

'LINEIN' set #Charin_Position.Stream to the position from which the next Config_Stream_Charin on the stream shall read, as indicated by Position. Set #Linein_Position.Stream to correspond with this position.

If Operation is 'CHAROUT' or 'LINEOUT' and the Position is more than one beyond the limit of exisitng data then the indicator of the response shall be 'R'. Otherwise if Operation is 'CHAROUT' or 'LINEOUT' then #Charout_Position.Stream is set to the position at which the next Config_Stream_Charout on the stream shall write, as indicated by Position. Set #Lineout_Position.Stream to correspond with this position.

If this function is unable to position the stream because the stream is transient then the indicator of the completion response shall be 'T'.

### 5.8.3 Config_Stream_Command

Syntax:

**Config_Stream_Command(Stream, Command)**

where:

Stream is the name of the stream to be processed.

Command is a configuration specific command to be performed against the stream.

Semantics:

Issue a configuration specific command against a stream. This may affect all state variables describing Stream which hold position information. It may alter the effect of any subsequent operation on the specified stream. If the indicator is set to 'N', #Outcome shall be set to information from the command.

### 5.8.4 Config_Stream_State

Syntax:

**Config_Stream_State(Stream)**

where:

Stream is the name of the stream to be queried.

Semantics:

Set the indicator to reflect the state of the stream. Return an indicator equal to the indicator that an immediately subsequent Config_Stream_Charin(Stream, 'CHARIN') would return. Alternatively, return an indicator of 'U'.

The remainder of the response shall be a configuration dependent description of the state of the stream.

### 5.8.5 Config_Stream_Charout

Syntax:

**Config_Stream_Charout(Stream, Data)**

where:

Stream is the name of the stream to be processed.

Data is the data to be written, or 'EOL' to indicate that an end-of-line indication is to be written, or a null string. In the first case, if the stream is a binary stream then Data will be eight characters, each '0' or '1', otherwise Data will be a single character.

Semantics:

When Data is the null string, no data is written.

Otherwise write to the stream. The state variables describing the stream will be affected as follows:

- When the configuration is able to write Data to a transient stream or at position #Charout_Position.Stream of a persistent stream then the indicator in the response shall be 'N'. When Data is not 'EOL' then #Charout_Position.Stream is increased by one. When Data is 'EOL', then #Lineout_Position.Stream is increased by one and #Charout_Position.Stream is increased as necessary to account for any end-of-line indication embedded in the stream.
- When the configuration is unable to write Data the indicator is set to 'E'.

### 5.8.6 Config_Stream_Qualified

Syntax:

**`Config_Stream_Qualified(Stream)`**

where:

Stream is the name of the stream to be processed.

Semantics:

Set #Outcome to some name which identifies Stream.

Return a completion response with indicator 'B' if the argument is not acceptable to the configuration as identifying a stream.

### 5.8.7 Config_Stream_Unique

Syntax:

**`Config_Stream_Unique()`**

Semantics:

Set #Outcome to a name that the configuration recognizes as a stream name. The name shall not be a name that the configuration associates with any existing data.

### 5.8.8 Config_Stream_Query

Syntax:

**`Config_Stream_Query(Stream)`**

where:

Stream is the name of the stream to be queried.

Semantics:

Set #Outcome to 'B' if the stream is a binary stream, to 'C' if it is a character stream.

### 5.8.9 Config_Stream_Close

Syntax:

**`Config_Stream_Close(Stream)`**

where:

Stream is the name of the stream to be closed.

Semantics:

#Charout_Position.Stream and #Lineout_Position.Stream are set to 1 unless the stream has existing data, in which case they are set ready to write immediately after the existing data.

If this function is unable to position the stream because the stream is transient then the indicator of the completion response shall be 'T'.

### 5.8.10 Config_Stream_Count

Syntax:

**Config_Stream_Count(Stream, Operation, Option)**

where:

Stream is the name of the stream to be counted.

Operation is 'CHARS', or 'LINES'.

Option is 'N' or 'C'.

Semantics:

If the option is 'N', #Outcome is set to zero if:

- the file is transient and no more characters (or no more lines if the Operation is 'LINES') are expected to be available, even after waiting;

- the file is persistent and no more characters (or no more lines if the Operation is 'LINES') can be obtained from this stream by Config_Stream_Charin before use of some function which resets #Charin_Position.Stream and #Linein_Position.Stream.

If the option is 'N' and #Outcome is set non-zero, #Outcome shall be 1, or be the number of characters (or the number of lines if Operation is 'LINES') which could be read from the stream before resetting.

If the option is 'C', #Outcome is set to zero if:

- the file is transient and no characters (or no lines if the Operation is 'LINES') are available without waiting;

the file is persistent and no more characters (or no more lines if the Operation is 'LINES') can be obtained from this stream by Config_Stream_Charin before use of some function which resets #Charin_Position.Stream and #Linein_Position.Stream.

If the option is 'C' and #Outcome is set non-zero, #Outcome shall be the number of characters (or the number of lines if the Operation is 'LINES') which can be read from the stream without delay and before resetting .

## 5.9 External variable pools

The external variable pools interface consists of functions which the configuration shall provide to manipulate variables in external variable pools.

See section 9.8.6 for the VALUE built-in function which uses external variable pools.

The configuration shall provide an external variable pools mechanism. The following functions shall be provided:

– Config_Get;

– Config_Set.

The configuration may permit the external variable pools to be altered in other ways.

### 5.9.1 Config_Get

Syntax:

**`Config_Get(Poolid, Name)`**

where:

Poolid is a identification of the external variable pool.

Name is the name of a variable.

Semantics:

Get the value of a variable with name Name in the external variable pool Poolid. Set #Outcome to this value.

If Poolid does not identify an external pool provided by this configuration the indicator of the completion response is 'P'.

If Name is not a valid name of a variable in the external pool, the indicator of the completion response is 'F'.

### 5.9.2 Config_Set

Syntax:

**`Config_Set(Poolid, Name, Value)`**

where:

Poolid is a identification of the external variable pool.

Name is the name of a variable.

Value is the value to be assigned to the variable.

Semantics:

Set a variable with name Name in the external variable pool Poolid to Value.

If Poolid does not identify an external pool provided by this configuration the indicator of the completion response is 'P'.

If Name is not a valid name of a variable in the external pool, the indicator of the completion response is 'F'.

## 5.10 Configuration characteristics

The configuration characteristics interface consists of a function which the configuration shall provide which indicates choices decided by the configuration.

### 5.10.1 Config_Constants

Syntax:

**`Config_Constants()`**

Semantics:

Set the values of the following state variables:

– If there are any built-in functions which do not operate at NUMERIC DIGITS 9, then set variables #Bif_Digits. (with various tails which are the names of those built-in functions) to the values to be used.

– Set variables #Limit_Digits, #Limit_EnvironmentName, #Limit_ExponentDigits, #Limit_Literal, Limit_MessageInsert, #Limit_Name, #Limit_String, Limit_TraceData to the relevant limits. A configuration shall allow a #Limit_MessageInsert value of 50 to be specified. A configuration shall allow a #Limit_TraceData value of 250 to be specified.

- Set #Configuration to a string identifying the configuration.

- Set #Version to a string identifying the language processor. It shall have five words. Successive words shall be separated by a blank character. The first four letters of the first word shall be 'REXX'. The second word shall be the four characters '5.00'. The last three words shall be in the format which is the default for the DATE() built-in function.

## 5.11 Configuration routines

The configuration routines interface consists of functions which the configuration shall provide which provide functions for a language processor.

The following functions shall be provided:

– Config_Trace_Query;

– Config_Trace_Input;

– Config_Trace_Output;

– Config_Default_Input;

– Config_Default_Output;

– Config_Initialization;

– Config_Termination;

– Config_Halt_Query;

– Config_Halt_Reset;

– Config_NoSource;

– Config_Time;

– Config_Random_Seed;

– Config_Random_Next.

### 5.11.1 Config_Trace_Query

Syntax:

**`Config_Trace_Query()`**

Semantics:

Indicate whether external activity is requesting interactive tracing. Set #Outcome to 'Yes' if interactive tracing is currently requested. Otherwise set #Outcome to 'No'.

### 5.11.2 Config_Trace_Input

Syntax:

**`Config_Trace_Input()`**

Semantics:

Set #Outcome to a value from the source of trace input. The source of trace input is determined by the configuration.

### 5.11.3 Config_Trace_Output

Syntax:

**`Config_Trace_Output(Line)`**

where:

Line is a string.

Semantics:

Write String as a line to the destination of trace output. The destination of trace output is defined by the configuration.

### 5.11.4 Config_Default_Input

Syntax:

**`Config_Default_Input()`**

Semantics:

Set #Outcome to the value that LINEIN( ) would return.

### 5.11.5 Config_Default_Output

Syntax:

**`Config_Default_Output(Line)`**

where:

Line is a string.

Semantics:

Write the string as a line in the manner of LINEOUT( ,Line).

### 5.11.6 Config_Initialization

Syntax:

**`Config_Initialization()`**

Semantics:

This function is provided only as a counterpart to Trap_Initialization; in itself it does nothing except return the response. An indicator of 'F' gives rise to Msg3.1.

### 5.11.7 Config_Termination

Syntax:

**`Config_Termination()`**

Semantics:

This function is provided only as a counterpart to Trap_Termination; in itself it does nothing except return the response. An indicator of 'F' gives rise to Msg2.1.

### 5.11.8 Config_Halt_Query

Syntax:

**`Config_Halt_Query()`**

Semantics:

Indicate whether external activity has requested a HALT condition to be raised. Set #Outcome to 'Yes' if HALT is requested. Otherwise set #Outcome to 'No'.

### 5.11.9 Config_Halt_Reset

Syntax:

**`Config_Halt_Reset()`**

Semantics:

Reset the configuration so that further attempts to cause a HALT condition will be recognized.

### 5.11.10 Config_NoSource

Syntax:

**`Config_NoSource()`**

Semantics:

Indicate whether the source of the program may or may not be output by the language processor.

Set #NoSource to '1' to indicate that the source of the program may not be output by the language processor, at various points in processing where it would otherwise be output. Otherwise, set #NoSource to '0'.

A configuration shall allow any program to be processed in such a way that Config_NoSource() sets #NoSource to '0'. A configuration may allow any program to be processed in such a way that Config_NoSource() sets #NoSource to '1'.

### 5.11.11 Config_Time

Syntax:

**`Config_Time()`**

Semantics:

Get a time stamp. Set #Time to a string whose value is the integer number of microseconds that have elapsed between 00:00:00 on January first 0001 and the time that Config_Time is called, at longitude zero. Values sufficient to allow for any date in the year 9999 shall be supported. The value returned may be an approximation but shall not be smaller than the value returned by a previous use of the function.

Set #Adjust to an integer number of microseconds. #Adjust reflects the difference between the local date/time and the date/time corresponding to #Time. #Time + #Adjust is the local date/time.

### 5.11.12 Config_Random_Seed

Syntax:

**`Config_Random_Seed(Seed)`**

where:

Seed is a sequence of up to #Bif_Digits.RANDOM digits.

Semantics:

Set a seed, so that subsequent uses of Config_Random_Next will reproducibly produce quasi-random numbers.

### 5.11.13 Config_Random_Next

Syntax:

**`Config_Random_Next(Min, Max)`**

where:

Min is the lower bound, inclusive, on the number returned in #Outcome.

Max is the upper bound, inclusive, on the number returned in #Outcome.

Semantics:

Set #Outcome to a quasi-random non-negative integer in the range Min to Max.

#### 5.11.14 Config_Options

Syntax:

**`Config_Options(String)`**

where:

String is a string.

Semantics:

No effect beyond the effects common to all Config_ invocations. The value of the string will come from an OPTIONS instruction, see section 8.3.16.

### 5.12 Traps

The trapping interface consists of functions which may be provided by the caller of API_Start (see section 5.2.1) as a list of traps. Each trap may be specified or omitted. The language processor shall invoke a specified trap before, or instead of, using the corresponding feature of the language processor itself. This correspondence is implied by the choice of names, that is a name begining `Trap_` will correspond to a name begining `Config_` when the remainder of the name is the same. Corresponding functions are called with the same interface, with one exception. The exception is that a trap may return a null string. When a trap returns a null string, the corresponding `Config_` function is invoked; otherwise the invocation of the trap replaces the potential invocation of the `Config_` function.

In the rest of this standard, the trapping mechanism is not shown explicitly. It is implied by the use of a `Config_` function.

The names of the traps are

- Trap_Command;
- Trap_ExternalRoutine;
- Trap_Push;
- Trap_Queue;
- Trap_Pull;
- Trap_Queued;
- Trap_Trace_Query;
- Trap_Trace_Input;
- Trap_Trace_Output;
- Trap_Default_Input;
- Trap_Default_Output;

- Trap_Initialization;
- Trap_Termination;
- Trap_Halt_Query;
- Trap_Halt_Reset.

## 5.13 Variable pool

The variable pool interface consists of functions which the configuration shall provide to manipulate the variables and to obtain some characteristics of a Rexx program.

These functions can be called from programs not written in Rexx — commands and external routines invoked from a Rexx program, or traps invoked from the language processor.

All the functions comprising the variable pool interface shall return with an indication of whether an error occurred. They shall return indicating an error and have no other effect, if #API_Enabled has a value of '0' or if the arguments to them fail to meet the defined syntactic constraints.

These functions interact with the processing of clauses. To define this interaction, the functions are described here in terms of the processing of variables, see section 7.

Some of these functions have an argument which is a symbol. A symbol is a string. The content of the string shall meet the syntactic constraints of the left hand side of an assignment. Conversion to uppercase and substitution in compound symbols occurs as it does for the left hand side of an assignment. The symbol identifies the variable to be operated upon.

Some of the functions have an argument which is a direct symbol. A direct symbol is a string. The content of this string shall meet the syntactic constraints of a VAR_SYMBOL in uppercase with no periods or it shall be the concatenation of a part meeting the syntactic constraints of a stem in uppercase, and a part that is any string. In the former case the symbol identifies the variable to be operated upon. In the latter case the variable to be operated on is one with the specified stem and a tail which is the remainder of the direct symbol.

Functions that have an argument which is symbol or direct symbol shall return an indication of whether the identified variable existed before the function was executed.

Section 7.1 defines functions which manipulate Rexx variable pools. Where possible the functions comprising the variable pool interface are described in terms of the appropriate invocations of the functions defined in section 7.1. The first parameter on these calls is the state variable #Pool. If these `Var_` functions do not return an indicator 'N', 'R', or 'D' then the `API_` function shall return an error indication.

### 5.13.1 API_Set

Syntax:

```
API_Set(Symbol, Value)
```

where:

Symbol is a symbol.

Value is the string whose value is to be assigned to the variable.

Semantics:

Assign the value of Value to the variable identified by Symbol. If Symbol contains no periods or contains one period as its last character:

```
Var_Set(#Pool, Symbol, '0', Value)
```

Otherwise:

**`Var_Set(#Pool, #Symbol, '1', Value)`**

where:

#Symbol is Symbol after any replacements in the tail as described by section 7.3.1.

### 5.13.2 API_Value

Syntax:

**`API_Value(Symbol)`**

where:

Symbol is a symbol.

Semantics:

Return the value of the variable identified by Symbol. If Symbol contains no periods or contains one period as its last character this is the value of #Outcome after:

**`Var_Value(#Pool, Symbol, '0')`**

Otherwise the value of #Outcome after:

**`Var_Value(#Pool, #Symbol, '1')`**

where:

#Symbol is Symbol after any replacements in the tail as described by 7.3.1.

### 5.13.3 API_Drop

Syntax:

**`API_Drop(Symbol)`**

where:

Symbol is a symbol.

Semantics:

Drop the variable identified by Symbol. If Symbol contains no periods or contains one period as its last character:

**`Var_Drop(#Pool, Symbol, '0')`**

Otherwise:

**`Var_Drop(#Pool, #Symbol, '1')`**

where:

#Symbol is Symbol after any replacements in the tail as described by 7.3.1.

### 5.13.4 API_SetDirect

Syntax:

**`API_SetDirect(Symbol, Value)`**

where:

Symbol is a direct symbol.

Value is the string whose value is to be assigned to the variable.

Semantics:

Assign the value of Value to the variable identified by Symbol. If the Symbol contains no period:

**`Var_Set(#Pool, Symbol, '0', Value)`**

Otherwise:

**`Var_Set(#Pool, Symbol, '1', Value)`**

### 5.13.5 API_ValueDirect

Syntax:

**`API_ValueDirect(Symbol)`**

where:

Symbol is a direct symbol.

Semantics:

Return the value of the variable identified by Symbol. If the Symbol contains no period:

**`Var_Value(#Pool, Symbol, '0')`**

Otherwise:

**`Var_Value(#Pool, Symbol, '1')`**

### 5.13.6 API_DropDirect

Syntax:

**`API_DropDirect(Symbol)`**

where:

Symbol is a direct symbol.

Semantics:

Drop the variable identified by Symbol. If the Symbol contains no period:

**`Var_Drop(#Pool, Symbol, '0')`**

Otherwise:

**`Var_Drop(#Pool, Symbol, '1')`**

### 5.13.7 API_ValueOther

Syntax:

**`API_ValueOther(Qualifier)`**

where:

Qualifier is an indication distinguishing the result to be returned including any necessary further qualification.

Semantics:

Return characteristics of the program, depending on the value of Qualifier. The possibilities for the value to be returned are:

– the value of #Source;

– the value of #Version;

– the largest value of n such that #ArgExists.1.n is '1', see section 8.2.1;

- the value of #Arg.1.n where n is an integer value provided as input.

### 5.13.8 API_Next

Syntax:

```
API_Next()
```

Semantics:

Returns both the name and the value of some variable in the variable pool that does not have the attribute 'dropped' or the attribute 'implicit' and is not a stem; alternatively return an indication that there is no suitable name to return. When API_Next is called it will return a name that has not previously been returned; the order is undefined. This process of returning different names will restart whenever the Rexx processor executes Var_Reset.

### 5.13.9 API_NextVariable

Syntax:

```
API_NextVariable()
```

Semantics:

Returns both the name and the value of some variable in the variable pool that does not have the attribute 'dropped' or the attribute 'implicit'; alternatively return an indication that there is no suitable name to return. When API_NextVariable is called it will return data about a variable that has not previously been returned; the order is undefined. This process of returning different names will restart whenever the Rexx processor executes Var_Reset. In addition to the name and value, an indication of whether the variable was 'tailed' will be returned.

# 6 Syntax constructs

## 6.1 Notation

### 6.1.1 Backus Normal Form (BNF)

The syntax constructs in this standard are defined in Backus Normal Form (BNF). The syntax used in these BNF productions has

- a left-hand side (called identifier);
- the characters ':=';
- a right-hand side (called bnf_expression).

The left-hand side identifies syntactic constructs. The right-hand side describes valid ways of writing a specific syntactic construct.

The right-hand side consists of operands and operators, and may be grouped.

### 6.1.2 Operands

Operands may be terminals or non-terminals. If an operand appears as identifier in some other production it is called a non-terminal, otherwise it is called a terminal. Terminals are either literal or symbolic.

Literal terminals are enclosed in quotes and represent literally (apart from case) what must be present in the source being described.

Symbolic terminals formed with lower case characters represent something which the configuration may, or may not, allow in the source program, see sections 5.3.2, 5.3.3, 5.3.4, 5.3.5.

Symbolic terminals formed with uppercase characters represent events and tokens, see sections 6.2.1.1 and 6.2.1.2.

### 6.1.3 Operators

The following lists the valid operators, their meaning, and their precedence; the operator listed first has the highest precedence; apart from precedence recognition is from left to right:

- The postfix plus operator specifies one or more repetitions of the preceding construct.
- Abuttal specifies that the preceding and the following construct must appear in the given order.
- The operator '|' specifies alternatives between the preceding and the following constructs.

### 6.1.4 Grouping

Parentheses and square brackets are used to group constructs. Parentheses are used for the purpose of grouping only. Square brackets specify that the enclosed construct is optional.

### 6.1.5 BNF syntax definition

The BNF syntax, described in BNF, is:

```
production          :=  identifier ':=' bnf_expression         (6.1.5.1)
bnf_expression      :=  abuttal | bnf_expression '|' abuttal    (6.1.5.2)
abuttal             :=  [abuttal] bnf_primary                   (6.1.5.3)
bnf_primary         :=  '[' bnf_expression ']'                  (6.1.5.4)
                        | '(' bnf_expression ')' | literal
                        | identifier | message_identifier
                        | bnf_primary '+'
```

### 6.1.6 Syntactic errors

The syntax descriptions (see sections 6.2.2 and 6.3.2) make use of message_identifiers which are shown as Msgnn.nn or Msgnn, where nn is a number. These actions produce the correspondingly numbered error messages (see sections 6.4.6 and 8.2.1).

## 6.2 Lexical

The lexical level processes the source and provides tokens for further recognition by the top syntax level.

### 6.2.1 Lexical elements

#### 6.2.1.1 Events

The fully capitalized identifiers in the BNF syntax (see section 6.2.2) represent events. An event is either supplied by the configuration or occurs as result of a look-ahead in left-to-right parsing. The following events are defined:

– EOL occurs at the end of a line of the source. It is provided by Config_SourceChar, see section 5.4.1.

– EOS occurs at the end of the source. It is provided by Config_SourceChar.

– RADIX occurs when the character about to be scanned is 'X' or 'x' or 'B' or 'b' not followed by a *general_letter*, or a *digit,* or '.'.

– CONTINUE occurs when the character about to be scanned is ',', and the characters after the ',' up to EOL represent a repetition of *comment* or *blank,* and the EOL is not immediately followed by an EOS.

– EXPONENT_SIGN occurs when the character about to be scanned is '+' or '-', and the characters to the left of the sign, currently parsed as part of *Const_symbol,* represent a *plain_number* followed by 'E' or 'e', and the characters to the right of the sign represent a repetition of *digit* not followed by a *general_letter* or '.'.

#### 6.2.1.2 Actions and tokens

Mixed case identifiers with an initial capital letter cause an action when they appear as operands in a production. These actions perform further tests and create tokens for use by the top syntax level. The following actions are defined:

– *Special* supplies the source recognized as *special* to the top syntax level.

– *Eol* supplies a semicolon to the top syntax level.

– *Eos* supplies an end of source indication to the top syntax level.

– *Var_symbol* supplies the source recognized as *Var_symbol* to the top syntax level, as keywords or VAR_SYMBOL tokens, see section 6.2.3. The characters in a *Var_symbol* are converted by Config_Upper to uppercase. Msg30.1 shall be produced if *Var_symbol* contains more than #Limit_Name characters, see section 5.10.1.

– *Const_symbol* supplies the source recognized as *Const_symbol* to the top syntax level. If it is a *number* it is passed as a NUMBER token, otherwise it is passed as a CONST_SYMBOL token. The characters in a *Const_symbol* are converted by Config_Upper to become the characters that comprise that NUMBER or CONST_SYMBOL. Msg30.1 shall be produced if *Const_symbol* contains more than #Limit_Name characters.

– *Embedded_quotation_mark* records an occurrence of two consecutive quotation marks within a *string* delimited by quotation marks for further processing by the String action.

– *Embedded_apostrophe* records an occurrence of two consecutive apostrophes within a *string* delimited by apostrophes for further processing by the String action.

– *String* supplies the source recognized as *String* to the top syntax level as a STRING token. Any occurrence of *Embedded_quotation_mark* or *Embedded_apostrophe* is replaced by a single quotation mark or apostrophe, respectively. Msg30.2 shall be produced if the resulting string contains more than #Limit_Literal characters.

– *Binary_string* supplies the converted binary string to the top syntax level as a STRING token, after checking conformance to the *binary_string* syntax. If the *binary_string* does not contain any occurrence of a *binary_digit* a string of length 0 is passed to the top syntax level. The occurrences of *binary_digit* are concatenated to form a number in radix 2. Zero digits are added at the left if necessary to make the number of digits a multiple of 8. If the resulting number of digits exceeds 8 times #Limit_Literal then Msg30.2 shall be produced. The binary digits are converted to an encoding, see section 5.4.5. The encoding is supplied to the top syntax level as a STRING token.

– *Hex_string* supplies the converted hexadecimal string to the top syntax level as a STRING token, after checking conformance to the *hex_string* syntax. If the *hex_string* does not contain any occurrence of a *hex_digit* a string of length 0 is passed to the top syntax level. The occurrences of *hex_digit* are each converted to a number with four binary digits and concatenated. Zero digits are added at the left if necessary to make the number of digits a multiple of 8. If the resulting number of digits exceeds 8 times #Limit_Literal then Msg30.2 shall be produced. The binary digits are converted to an encoding. The encoding is supplied to the top syntax level as a STRING token.

– *Operator* supplies the source recognized as *Operator* (excluding characters that are not *operator_char* ) to the top syntax level. Any occurrence of an *other_negator* within *Operator* is supplied as '\'.

– *Blank* records the presence of a *blank* in the following token. This may subsequently be tested (see section 6.2.3).

Constructions of type *Number*, *Const_symbol*, *Var_symbol* or *String* are called operands.

#### 6.2.1.3 Source characters

The source is obtained from the configuration by the use of Config_SourceChar (see section 5.4.1). If no character is available because the source is not a correct encoding of characters, message Msg22.1 shall be produced.

The terms *extra_letter*, *other_blank_character*, *other_negator*, and *other_character* used in the productions of the lexical level refer to characters of the groups extra_letters (see section 5.3.2), other_blank_characters (see section 5.3.3), other_negators (see section 5.3.4) and other_characters (see section 5.3.5), respectively.

#### 6.2.1.4 Rules

In scanning, recognition that causes an action (see section 6.2.1.2) only occurs if no other recognition is possible.

### 6.2.2 Lexical level

```
digit              := '0' | '1' | '2' | '3' | '4' | '5'       (6.2.2.1)
                      | '6' | '7' | '8' | '9'
special            := ',' | ':' | ';' | ')' | '('             (6.2.2.2)
not                := '\' | other_negator                     (6.2.2.3)
operator_only      := '+' | '-' | '%' | '|' | '&' | '='       (6.2.2.4)
                      | not | '>' | '<'
```

```
operator_or_other       :=  '/' | '*'                                (6.2.2.5)
operator_char           :=  operator_only | operator_or_other        (6.2.2.6)
general_letter          :=  '_' | '!' | '?' | extra_letter | 'A'     (6.2.2.7)
                            | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
                            | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'
                            | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S'
                            | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y'
                            | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e'
                            | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
                            | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
                            | 'r' | 's' | 't' | 'u' | 'v' | 'w'
                            | 'x' | 'y' | 'z'
blank                   :=  ' ' | other_blank_character              (6.2.2.8)
bo                      :=  [blank+]                                 (6.2.2.9)
string_or_comment_char  :=  digit | '.' | special | operator_only   (6.2.2.10)
                            | general_letter | blank
                            | other_character

tokenize                :=  [between+] [tokenbetween+] EOS Eos      (6.2.2.11)
tokenbetween            :=  token [between+]                        (6.2.2.12)
token                   :=  operand | Operator | Special            (6.2.2.13)
operand                 :=  string_literal | Var_symbol             (6.2.2.14)
                            | Const_symbol
between                 :=  comment | blank_run Blank | EOL Eol     (6.2.2.15)
                            | Msg13.1
blank_run               :=  (blank | continuation)+                 (6.2.2.16)
continuation            :=  CONTINUE ',' [(comment | blank)+] EOL   (6.2.2.17)
comment                 :=  '/' '*' [commentpart+] ['*'+] ('*' '/'  (6.2.2.18)
                            | EOS Msg6.1)
commentpart             :=  comment | ['/'+] comment_char+          (6.2.2.19)
                            | '*'+ comment_char+
comment_char            :=  string_or_comment_char | '"' | "'"      (6.2.2.20)
                            | EOL
string_literal          :=  Hex_string | Binary_string | String     (6.2.2.21)
String                  :=  quoted_string                           (6.2.2.22)
Hex_string              :=  quoted_string RADIX ('x' | 'X')         (6.2.2.23)
Binary_string           :=  quoted_string RADIX ('b' | 'B')         (6.2.2.24)
quoted_string           :=  quotation_mark_string                   (6.2.2.25)
                            [(Embedded_quotation_mark
                            quotation_mark_string)+]
                            | apostrophe_string
                            [(Embedded_apostrophe
                            apostrophe_string)+]
quotation_mark_string   :=  '"' [(string_char | "'")+]              (6.2.2.26)
                            ('"' | EOL Msg6.3)
apostrophe_string       :=  "'" [(string_char | '"')+]              (6.2.2.27)
                            ("'" | EOL Msg6.2)
string_char             :=  string_or_comment_char | '*' | '/'      (6.2.2.28)
Var_symbol              :=  general_letter [var_symbol_char+]       (6.2.2.29)
var_symbol_char         :=  general_letter | digit | '.'            (6.2.2.30)
Const_symbol            :=  (digit | '.') [const_symbol_char+]      (6.2.2.31)
const_symbol_char       :=  var_symbol_char                         (6.2.2.32)
                            | EXPONENT_SIGN ('+' | '-')
```

```
Special                := special                                  (6.2.2.33)
Operator               := operator_char | '|' bo '|'               (6.2.2.34)
                          | '/' bo '/' | '*' bo '*' | not bo '='
                          | '>' bo '<' | '<' bo '>' | '>' bo '='
                          | not bo '<' | '<' bo '=' | not bo '>'
                          | '=' bo '=' | not bo '=' bo '='
                          | '>' bo '>' | '<' bo '<'
                          | '>' bo '>' bo '='
                          | not bo '<' bo '<'
                          | '<' bo '<' bo '='
                          | not bo '>' bo '>'
                          | '&' bo '&'

number                 := plain_number [exponent]                  (6.2.2.35)
plain_number           := ['.'] digit+ | digit+ '.' [digit+]       (6.2.2.36)
exponent               := ('e' | 'E') ['+' | '-'] digit+           (6.2.2.37)

hex_string             := (hex_digit [break_hex_digit_pair+]       (6.2.2.38)
                          | [hex_digit hex_digit
                          [break_hex_digit_pair+]]) | (Msg15.1 |
                          Msg15.3)
hex_digit              := digit | 'a' | 'A' | 'b' | 'B' | 'c'      (6.2.2.39)
                          | 'C' | 'd' | 'D' | 'e' | 'E' | 'f'
                          | 'F'
break_hex_digit_pair   := bo hex_digit hex_digit                   (6.2.2.40)

binary_string          := (binary_digit                            (6.2.2.41)
                          [break_binary_digit_quad+]
                          | binary_digit binary_digit
                          [break_binary_digit_quad+]
                          | binary_digit binary_digit
                          binary_digit
                          [break_binary_digit_quad+]
                          | [binary_digit binary_digit
                          binary_digit binary_digit
                          [break_binary_digit_quad+]])
                          | (Msg15.2 | Msg15.4)
binary_digit           := '0' | '1'                                (6.2.2.42)
break_binary_digit_quad := bo binary_digit binary_digit            (6.2.2.43)
                          binary_digit binary_digit
```

### 6.2.3 Interaction between levels of syntax

When the lexical process recognizes tokens to be supplied to the top level, there can be changes made or tokens added. Recognition is performed by the lexical process and the top level process in a synchronized way. The tokens produced by the lexical level can be affected by what the top level syntax has recognized. Those tokens will affect subsequent recognition by the top level. Both processes operate on the characters and the tokens in the order they are produced. The term "context" refers to the progress of the recognition at some point, without consideration of unprocessed characters and tokens.

If a token which is '+', '—', '\' or '(' appears in a lexical level context (other than after the keyword 'PARSE') where the keyword 'VALUE' could appear in the corresponding top level context, then 'VALUE' is passed to the top level before the token is passed.

If an '=' *operator_char* appears in a lexical level context where it could be the '=' of an *assignment* in the corresponding top level context then the *Var_symbol* preceding it is passed as a VAR_SYMBOL. If an operand is followed by a colon token in the lexical level context then the operand only is passed to the top level syntax as a LABEL, provided the context permits a LABEL.

Except where the rules above determine the token passed, a *Var_symbol* is passed as a terminal (a keyword) rather than as a VAR_SYMBOL under the following circumstances:

– If the symbol is spelled 'WHILE' or 'UNTIL' it is a keyword wherever a VAR_SYMBOL would be part of an *expression* within a *do_specification*.

– If the symbol is spelled 'TO' , 'BY', or 'FOR' it is a keyword wherever a VAR_SYMBOL would be part of an *expression* within a *do_rep*.

– If the symbol is spelled 'WITH' it is a keyword wherever a VAR_SYMBOL would be part of a *parsevalue*, or part of an *expression* or *taken_constant* within *address*.

– If the symbol is spelled 'THEN' it is keyword wherever a VAR_SYMBOL would be part of an *expression* immediately following the keyword 'IF' or 'WHEN'.

Except where the rules above determine the token passed, a *Var_symbol* is passed as a keyword if the spelling of it matches a keyword which the top level syntax recognizes in its current context, otherwise the *Var_symbol* is passed as a VAR_SYMBOL token.

When a *Var_symbol* is passed as a terminal because of the spelling match it is referred to as a keyword.

In a context where the top level syntax could accept a '||' token as the next token, a '||' operator or a ' ' operator may be inferred and passed to the top level provided that the next token from the lexical level is a left parenthesis or an operand that is not a keyword. If the blank action has recorded the presence of one or more blanks to the left of the next token then the ' ' operator is inferred. Otherwise, a '||' operator is inferred, except if the next token is a left parenthesis following an operand; in this case no operator is inferred.

When any of the keywords 'OTHERWISE', 'THEN', or 'ELSE' is recognized, a semicolon token is supplied as the following token. A semicolon token is supplied as the previous token when the 'THEN' keyword is recognized. A semicolon token is supplied as the token following a LABEL.

#### 6.2.3.1 Reserved symbols

A *Const_symbol* which starts with a period and is not a *Number* shall be spelled .MN, .RESULT, .RC, .RS, or .SIGL otherwise Msg50.1 is issued.

#### 6.2.3.2 Function name syntax

A *symbol* which is the leftmost component of a *function* shall not end with a period, otherwise Msg51.1 is issued.

## 6.3 Syntax

### 6.3.1 Syntax elements

The tokens generated by the actions described in section 6.2.1 form the basis for recognizing larger constructs.

### 6.3.2 Syntax level

```
/* The first part introduces the overall structure of a program */
program          := [ncl] [instruction_list] ['END' Msg10.1]       (6.3.2.1)
  ncl            := null_clause+ | Msg21.1                         (6.3.2.2)
    null_clause  := ';' [label_list]                               (6.3.2.3)
```

```
      label_list  := (LABEL ';')+                                (6.3.2.4)
  instruction_list:= instruction+                                (6.3.2.5)
    instruction   := group | single_instruction ncl              (6.3.2.6)
single_instruction:= assignment | keyword_instruction | command  (6.3.2.7)
  assignment      := VAR_SYMBOL '=' expression                   (6.3.2.8)
                  | NUMBER '=' Msg31.1
                  | CONST_SYMBOL '=' (Msg31.2 | Msg31.3)
  keyword_instruction:= address | arg | call | drop | exit       (6.3.2.9)
                  | interpret | iterate | leave
                  | nop | numeric | options
                  | parse | procedure | pull | push
                  | queue | return | say | signal | trace
                  | 'THEN' Msg8.1 | 'ELSE' Msg8.2
                  | 'WHEN' Msg9.1 | 'OTHERWISE' Msg9.2
  command         := expression                                 (6.3.2.10)
group             := do | if | select                           (6.3.2.11)
  do              := do_specification (ncl | Msg21.1 | Msg27.1) (6.3.2.12)
                  [instruction_list] do_ending
    do_ending     := 'END' [VAR_SYMBOL] ncl                     (6.3.2.13)
                  | EOS Msg14.1 | Msg35.1
  if              := 'IF' expression [ncl] (then | Msg18.1)     (6.3.2.14)
                  [else]
    then          := 'THEN' ncl                                 (6.3.2.15)
                  (instruction | EOS Msg14.3 | 'END' Msg10.5)
    else          := 'ELSE' ncl                                 (6.3.2.16)
                  (instruction | EOS Msg14.4 | 'END' Msg10.6)
  select          := 'SELECT' ncl select_body                   (6.3.2.17)
                  ('END' [VAR_SYMBOL Msg10.4] ncl
                  | EOS Msg14.2 | Msg7.2)
    select_body   := (when | Msg7.1) [when+] [otherwise]        (6.3.2.18)
      when        := 'WHEN' expression [ncl] (then | Msg18.2)   (6.3.2.19)
      otherwise   := 'OTHERWISE' ncl [instruction_list]         (6.3.2.20)

/*
Note:  The next part concentrates on the instructions.
It leaves unspecified the various forms of symbol, template
and expression.
*/

address           := 'ADDRESS' [(taken_constant [expression]    (6.3.2.21)
                  | Msg19.1 | valueexp)  [ 'WITH' connection]]
  taken_constant  := symbol | STRING                            (6.3.2.22)
  valueexp        := 'VALUE' expression                         (6.3.2.23)
  connection      := error [adio] | input [adeo]                (6.3.2.24)
                  | output [adei] | Msg25.5
    adio          := input [output] | output [input]            (6.3.2.25)
      input       := 'INPUT' (resourcei | Msg25.6)              (6.3.2.26)
        resourcei := resources | 'NORMAL'                       (6.3.2.27)
      output      := 'OUTPUT' (resourceo | Msg25.7)             (6.3.2.28)
        resourceo := 'APPEND' (resources | Msg25.8)             (6.3.2.29)
                  | 'REPLACE' (resources | Msg25.9)
                  | resources | 'NORMAL'
    adeo          := error [output] | output [error]            (6.3.2.30)
```

```
      error       := 'ERROR' (resourceo | Msg25.14)                (6.3.2.31)
    adei          := error [input] | input [error]                 (6.3.2.32)
resources         := 'STREAM' (VAR_SYMBOL | Msg53.1)               (6.3.2.33)
                  | 'STEM' (VAR_SYMBOL | Msg53.2)
  vref            := '(' var_symbol (')' | Msg46.1)                (6.3.2.34)
    var_symbol    := VAR_SYMBOL | Msg20.1                          (6.3.2.35)
arg               := 'ARG' [template_list]                         (6.3.2.36)
call              := 'CALL' (callon_spec|                          (6.3.2.37)
                  (taken_constant|Msg19.2)[expression_list])
  callon_spec     := 'ON' (callable_condition | Msg25.1)           (6.3.2.38)
                  ['NAME' (taken_constant | Msg19.3)]
                  | 'OFF' (callable_condition | Msg25.2)
    callable_condition:= 'ERROR' | 'FAILURE'                       (6.3.2.39)
                  | 'HALT' | 'NOTREADY'
  expression_list := expr | [expr] ',' [expression_list]           (6.3.2.40)
do_specification  := do_simple | do_repetitive                     (6.3.2.41)
  do_simple       := 'DO'                                          (6.3.2.42)
  do_repetitive   := 'DO' dorep | 'DO' docond                      (6.3.2.43)
                  | 'DO' dorep docond
                  | 'DO' 'FOREVER' [docond | Msg25.16]
  docond          := 'WHILE' whileexpr | 'UNTIL' untilexpr         (6.3.2.44)
    untilexpr     := expression                                    (6.3.2.45)
    whileexpr     := expression                                    (6.3.2.46)
  dorep           := assignment [docount] | repexpr                (6.3.2.47)
    repexpr       := expression                                    (6.3.2.48)
    docount       := dot [dobf] | dob [dotf] | dof [dotb]          (6.3.2.49)
      dobf        := dob [dof] | dof [dob]                         (6.3.2.50)
      dotf        := dot [dof] | dof [dot]                         (6.3.2.51)
      dotb        := dot [dob] | dob [dot]                         (6.3.2.52)
      dot         := 'TO' toexpr                                   (6.3.2.53)
        toexpr    := expression                                    (6.3.2.54)
      dob         := 'BY' byexpr                                   (6.3.2.55)
        byexpr    := expression                                    (6.3.2.56)
      dof         := 'FOR' forexpr                                 (6.3.2.57)
        forexpr   := expression                                    (6.3.2.58)
drop              := 'DROP' variable_list                          (6.3.2.59)
  variable_list   := (vref | var_symbol)+                          (6.3.2.60)
exit              := 'EXIT' [expression]                           (6.3.2.61)
interpret         := 'INTERPRET' expression                        (6.3.2.62)
iterate           := 'ITERATE' [VAR_SYMBOL | Msg20.2 ]             (6.3.2.63)
leave             := 'LEAVE' [VAR_SYMBOL | Msg20.2 ]               (6.3.2.64)
nop               := 'NOP'                                         (6.3.2.65)
numeric           := 'NUMERIC' (numeric_digits | numeric_form      (6.3.2.66)
                  | numeric_fuzz | Msg25.15)
  numeric_digits  := 'DIGITS' [expression]                         (6.3.2.67)
  numeric_form    := 'FORM' ('ENGINEERING' | 'SCIENTIFIC'          (6.3.2.68)
                  | valueexp | Msg25.11)
  numeric_fuzz    := 'FUZZ' [expression]                           (6.3.2.69)
options           := 'OPTIONS' expression                          (6.3.2.70)
parse             := 'PARSE'(parse_type | Msg25.12)[template_list]  (6.3.2.71)
                  | 'PARSE' 'UPPER' (parse_type | Msg25.13)
                  [template_list]
  parse_type      := parse_key | parse_value | parse_var           (6.3.2.72)
```

```
    parse_key      := 'ARG' | 'PULL' | 'SOURCE' | 'LINEIN'           (6.3.2.73)
                   | 'VERSION'
    parse_value    := 'VALUE' [expression] ('WITH' | Msg38.3)        (6.3.2.74)
    parse_var      := 'VAR' var_symbol                               (6.3.2.75)
procedure          := 'PROCEDURE'                                    (6.3.2.76)
                   ['EXPOSE' variable_list | Msg25.17]
pull               := 'PULL' [template_list]                         (6.3.2.77)
push               := 'PUSH' [expression]                            (6.3.2.78)
queue              := 'QUEUE' [expression]                           (6.3.2.79)
return             := 'RETURN' [expression]                          (6.3.2.80)
say                := 'SAY' [expression]                             (6.3.2.81)
signal             := 'SIGNAL' (signal_spec | valueexp               (6.3.2.82)
                   | taken_constant | Msg19.4)
  signal_spec      := 'ON' (condition | Msg25.3)                     (6.3.2.83)
                   ['NAME' (taken_constant | Msg19.3)]
                   | 'OFF' (condition | Msg25.4)
    condition      := callable_condition | 'NOVALUE' | 'SYNTAX'      (6.3.2.84)
                   | 'LOSTDIGITS'
trace              := 'TRACE'                                        (6.3.2.85)
                   [(taken_constant | Msg19.6) | valueexp]

/* Note:  The next section describes templates. */
template_list      := template | [template] ',' [template_list]      (6.3.2.86)
  template         := (trigger | target | Msg38.1)+                  (6.3.2.87)
   target          := VAR_SYMBOL | '.'                               (6.3.2.88)
   trigger         := pattern | positional                           (6.3.2.89)
     pattern       := STRING | vrefp                                 (6.3.2.90)
       vrefp       := '(' (VAR_SYMBOL  | Msg19.7) (')' | Msg46.1)    (6.3.2.91)
     positional    := absolute_positional | relative_positional      (6.3.2.92)
       absolute_positional:= NUMBER | '=' position                   (6.3.2.93)
         position := NUMBER | vrefp | Msg38.2                        (6.3.2.94)
     relative_positional:= ('+' | '-') position                      (6.3.2.95)

/* Note: The final part specifies the various forms of symbol, and
expression. */
symbol             := VAR_SYMBOL | CONST_SYMBOL | NUMBER             (6.3.2.96)
expression         := expr [(',' Msg37.1) | (')' Msg37.2 )]          (6.3.2.97)
  expr             := expr_alias                                     (6.3.2.98)
    expr_alias     := and_expression                                 (6.3.2.99)
                   | expr_alias or_operator and_expression
      or_operator := '|' | '&&'                                     (6.3.2.100)
      and_expression := comparison | and_expression '&' comparison  (6.3.2.101)
comparison         := concatenation                                 (6.3.2.102)
                   | comparison comparison_operator concatenation
  comparison_operator:= normal_compare | strict_compare             (6.3.2.103)
    normal_compare:= '=' | '\=' | '<>' | '><' | '>' | '<' | '>='    (6.3.2.104)
                   | '<=' | '\>' | '\<'
    strict_compare:= '==' | '\==' | '>>' | '<<' | '>>=' | '<<='     (6.3.2.105)
                   | '\>>' | '\<<'
concatenation      := addition                                      (6.3.2.106)
                   | concatenation (' ' | '||') addition
addition           := multiplication                                (6.3.2.107)
                   | addition additive_operator multiplication
  additive_operator:= '+' | '-'                                     (6.3.2.108)
multiplication    := power_expression                               (6.3.2.109)
                  | multiplication multiplicative_operator
                  power_expression
  multiplicative_operator:= '*' | '/' | '//' | '%'                  (6.3.2.110)
power_expression  := prefix_expression                              (6.3.2.111)
                  | power_expression '**' prefix_expression
  prefix_expression := ('+' | '-' | '\') prefix_expression          (6.3.2.112)
                  | term | Msg35.1
    term          := symbol | STRING | function                     (6.3.2.113)
                  | '(' expr_alias (',' Msg37.1 | ')' | Msg36)
      function    := taken_constant '(' [expression_list]           (6.3.2.114)
                  (')' | Msg36)
```

## 6.4 Syntactic information

### 6.4.1 VAR_SYMBOL matching

Any VAR_SYMBOL in a *do_ending* must be matched by the same VAR_SYMBOL occurring at the start of an *assignment* contained in the *do_specification* of the *do* that contains both the *do_specification* and the *do_ending*, as described in section 6.3.2.13.

If there is a VAR_SYMBOL in a *do_ending* for which there is no *assignment* in the corresponding *do_specification* then message Msg10.3 is produced and no further activity is defined.

If there is a VAR_SYMBOL in a *do_ending* for which does not match the one occurring in the *assignment* then message Msg10.2 is produced and no further activity is defined.

An *iterate* or *leave* must be contained in the *instruction_list* of some *do* with a *do_specification* which is *do_repetitive*, otherwise a message (Msg28.2 or Msg28.1 respectively) is produced and no further activity is defined.

If an *iterate* or *leave* contains a VAR_SYMBOL there must be a matching VAR_SYMBOL in a *do_specification*, otherwise a message (Msg28.1, Msg28.2, Msg28.3 or Msg28.4 appropriately) is produced and no further activity is defined. The matching VAR_SYMBOL will occur at the start of an *assignment* in the *do_specification*. The *do_specification* will be associated with a *do* by section 6.3.2.13. The *iterate* or *leave* will be a single *instruction* in an *instruction_list* associated with a *do* by section 6.3.2.13. These two *do*s shall be the same, or the latter nested one or more levels within the former. The number of levels is called the nesting_correction and influences the semantics of the *iterate* or *leave*. It is zero if the two *do*s are the same. The nesting_correction for *iterates* or *leaves* that do not contain VAR_SYMBOL is zero.

### 6.4.2 Trace-only labels

Instances of LABEL which occur within a *grouping_instruction* and are not in a *ncl* at the end of that *grouping_instruction* are instances of trace-only labels.

### 6.4.3 Clauses and line numbers

The activity of tracing execution is defined in terms of clauses. A program consists of clauses, each clause ended by a semicolon special token. The semicolon may be explicit in the program or inferred.

The line number of a clause is one more than the number of EOL events recognized before the first token of the clause was recognized.

### 6.4.4 Nested IF instructions

The syntax specification 6.3.2 allows *if*s to be nested and does not fully specify the association of an *else* with an *if*. An *else* associates with the closest prior *if* that it can associate with in conformance with the syntax.

### 6.4.5 Choice of messages

The specifications 6.2.2 and 6.3.2 permit two alternative messages in some circumstances. The following rules apply:

– Msg15.1 shall be preferred to Msg15.3 if the choice of Msg15.3 would result in the replacement for the insertion being a blank character.

– Msg15.2 shall be preferred to Msg15.4 if the choice of Msg15.4 would result in the replacement for the insertion being a blank character.

– Msg31.3 shall be preferred to Msg31.2 if the replacement for the insertion in the message starts with a period.

– Preference is given to the message that appears later in the list: Msg21.1, Msg27.1, Msg25.16, Msg36, Msg38.3, Msg35.1, other messages.

### 6.4.6 Creation of messages

The message_identifiers in section 6 correlate with the tails of stem #ErrorText., which is initialized in section 8.2.1 to identify particular messages. The action of producing an error message will replace any insertions in the message text and present the resulting text, together with information on the origin of the error, to the configuration by writing on the default error stream.

Further activity by the language processor is permitted, but not defined by this standard.

The effect of an error during the writing of an error message is not defined.

#### 6.4.6.1 Error message prefix

The error message selected by the message number is preceded by a prefix. The text of the prefix is #ErrorText.0.1 except when the error is in source that execution of an interpret instruction (see section 8.3.10) is processing, in which case the text is #ErrorText.0.2. The insert called <value> in these texts is the message number. The insert called <linenumber> is the line number of the error.

The line number of the error is one more than the number of EOL events encountered before the error was detectable, except for messages Msg6.1, Msg14, Msg14.1, Msg14.2, Msg14.3, and Msg14.4. For Msg6.1 it is one more than the number of EOL events encountered before the line containing the unmatched '/*'. For the others, it is the line number of the clause containing the keyword referenced in the message text.

The insert called <source> is the value provided on the API_Start function which started processing of the program, see section 5.2.1.

#### 6.4.6.2 Replacement of insertions

Within the text of error messages, an insertion consists of the characters '<', '>', and what is between those characters. There will be a word in the insertion that specifies the replacement text, with the following meaning:

– If the word is 'hex-encoding' and the message is not Msg23.1 then the replacement text is the value of the leftmost character which caused the source to be syntactically incorrect. The value is in hexadecimal notation.

– If the word is 'token' then the replacement text is the part of the source program which was recognized as the detection token, or in the case of Msg31.1 and Msg31.2, the token before the detection token, or in the case of Msg10.2 the VAR_SYMBOL of a *do_specification*.

The detection token is the leftmost token for which the program up to and including the token could not be parsed as the left part of a *program* without causing a message. If the detection token is a semicolon that was not present in the source but was supplied during recognition then the replacement is the text "end-of-line" without quotation marks.

– If the word is 'position' then the replacement text is a number identifying the detection character. The detection character is the leftmost character in the *hex_string* or *binary_string* which did not match the required syntax. The number is a count of the characters in the string which preceded the detection character, including the initial quote or apostrophe.

– If the word is 'char' then the replacement text is the detection character.

– If the word is 'linenumber' then the replacement text is the line number of a clause associated with the error. The wording of the message text specifies which clause that is.

– If the words is 'keywords' then the replacement text is a list of the keywords that the syntax would allow at the context where the error occurred. If there are two keywords they shall be separated by the four characters ' or '. If more, the last shall be preceded by the three characters 'or ' and the others shall be followed by the two characters ', '.

Replacement text is truncated to #Limit_MessageInsert characters if it would otherwise be longer than that, except for a keywords replacement.

When an insert is both truncated and appears within quotes in the message, the three characters '...' are inserted in the message after the trailing quote.

# 7 Evaluation

The syntax section describes how expressions and the components of expressions are written in a program. It also describes how operators can be associated with the strings, symbols and function results which are their operands.

This evaluation section describes what values these components have in execution, or how they have no value because a condition is raised.

This section refers to the DATATYPE built-in function when checking operands, see section 9.3.8. Except for considerations of limits on the values of exponents, the test:

```
datatype(Subject) == 'NUM'
```

is equivalent to testing whether the subject matches the syntax:

```
num  :=   [blank+]  ['+' | '-']  [blank+]  number  [blank+]
```

For the syntax of *number* see 6.2.2.35.

When the matching subject does not include a '–' the value is the value of the number in the match, otherwise the value is the value of the expression (0 – number).

The test:

```
datatype(Subject , 'W')
```

is a test that the Subject matches that syntax and also has a value that is "whole", that is has no non-zero fractional part.

When these two tests are made and the Subject matches the constraints but has an exponent that is not in the correct range of values then a condition is raised:

```
call #Raise 'SYNTAX', 41.7, Subject
```

This possibility is implied by the uses of DATATYPE and not shown explicitly in the rest of this section 7.

## 7.1 Variables

The values of variables are held in variable pools. The capabilities of variable pools are listed here, together with the way each function will be referenced in this definition.

The notation used here is the same as that defined in sections 5.1 and 5.1.1, including the fact that the `Var_` routines may return an indicator of 'N', 'S' or 'X'.

Each possible name in a variable pool is qualified as tailed or non-tailed name; names with different qualification and the same spelling are different items in the pool. For those `Var_` functions with a third argument this argument indicates the qualification; it is '1' when addressing tailed names or '0' when addressing non-tailed names.

Each item in a variable pool is associated with three attributes and a value. The attributes are 'dropped' or 'not-dropped', 'exposed' or 'not-exposed' and 'implicit' or 'not-implicit'.

A variable pool is associated with a pool number denoted by the first argument with name Pool. The value of Pool may alter during execution. The same name, in conjunction with different values of Pool, can correspond to different values.

### 7.1.1 Var_Empty

**Var_Empty(Pool)**

The function sets the variable pool associated with the given pool number to the state where every name is associated with attributes 'dropped', 'implicit' and 'not-exposed'.

### 7.1.2 Var_Set

#### Var_Set(Pool, Name, '0', Value)

The function operates on the variable pool with the specified pool number. The name is a non-tailed name. If the specified name has the 'exposed' attribute then Var_Set operates on the variable pool associated with (Pool-1) and this rule is applied to that pool. When the pool with attribute 'not-exposed' for this name is determined the specified value is associated with the specified name. It also associates the attributes 'not-dropped' and 'not-implicit'. If that attribute was previously 'not-dropped' then the indicator returned is 'R'. The name is a stem if it contains just one period, as its rightmost character. When the name is a stem Var_Set(Pool,TailedName, '1',Value) is executed for all possible valid tailed names which have Name as their stem, and then those tailed-names are given the attribute 'implicit'.

#### Var_Set(Pool, Name, '1', Value)

The function operates on the variable pool with the specified pool number. The name is a tailed name. The left part of the name, up to and including the first period, is the stem. The stem is a non-tailed name. If the specified stem has the 'exposed' attribute then Var_Set operates on the variable pool associated with (Pool-1) and this rule is applied to that pool. When the pool with attribute 'not-exposed' for the stem is determined the name is considered in that pool. If the name has the 'exposed' attribute then the variable pool associated with a pool number one less is considered and this rule applied to that pool. When the pool with attribute 'not-exposed' is determined the specified value is associated with the specified name. It also associates the attributes 'not-dropped' and 'not-implicit' . If that attribute was previously 'not-dropped' then the indicator returned is 'R'.

### 7.1.3 Var_Value

#### Var_Value(Pool, Name, '0')

The function operates on the variable pool with the specified pool number. The name is a non-tailed name. If the specified name has the 'exposed' attribute then Var_Value operates on the variable pool associated with (Pool-1) and this rule is applied to that pool. When the pool with attribute 'not-exposed' for this name is determined the indicator returned is 'D' if the name has 'dropped' associated, 'N' otherwise. In the former case #Outcome is set equal to Name, in the latter case #Outcome is set to the value most recently associated with the name by Var_Set.

#### Var_Value(Pool, Name, '1')

The function operates on the variable pool with the specified pool number. The name is a tailed name. The left part of the name, up to and including the first period, is the stem. The stem is a non-tailed name. If the specified stem has the 'exposed' attribute then Var_Value operates on the variable pool associated with (Pool-1) and this rule is applied to that pool. When the pool with attribute 'not-exposed' for the stem is determined the name is considered in that pool. If the name has the 'exposed' attribute then the variable pool associated with a pool number one less is considered and this rule applied to that pool. When the pool with attribute 'not-exposed' is determined the indicator returned is 'D' if the name has 'dropped' associated, 'N' otherwise. In the former case #Outcome is set equal to Name, in the latter case #Outcome is set to the value most recently associated with the name by Var_Set.

### 7.1.4 Var_Drop

#### Var_Drop(Pool, Name, '0')

The function operates on the variable pool with the specified pool number. The name is a non-tailed name. If the specified name has the 'exposed' attribute then Var_Drop operates on the variable pool associated with (Pool-1) and this rule is applied to that pool. When the pool with attribute 'not-exposed' for this name is determined the attribute 'dropped' is associated with the specified name. Also, when the name is a stem, Var_Drop(Pool,TailedName,'1') is executed for all possible valid tailed names which have Name as a stem.

**`Var_Drop(Pool, Name, '1')`**

The function operates on the variable pool with the specified pool number. The name is a tailed name. The left part of the name, up to and including the first period, is the stem. The stem is a non-tailed name. If the specified stem has the 'exposed' attribute then Var_Drop operates on the variable pool associated with (Pool-1) and this rule is applied to that pool. When the pool with attribute 'not-exposed' for the stem is determined the name is considered in that pool. If the name has the 'exposed' attribute then the variable pool associated with a pool number one less is considered and this rule applied to that pool. When the pool with attribute 'not-exposed' is determined the attribute 'dropped' is associated with the specified name.

#### 7.1.5 Var_Expose

**`Var_Expose(Pool, Name, '0')`**

The function operates on the variable pool with the specified pool number. The name is a non-tailed name. The attribute 'exposed' is associated with the specified name. Also, when the name is a stem, Var_Expose(Pool,TailedName,'1') is executed for all possible valid tailed names which have Name as a stem.

**`Var_Expose(Pool, Name, '1')`**

The function operates on the variable pool with the specified pool number. The name is a tailed name. The attribute 'exposed' is associated with the specified name.

#### 7.1.6 Var_Reset

**`Var_Reset(Pool)`**

The function operates on the variable pool with the specified pool number. It establishes the effect of subsequent API_Next and API_NextVariable functions (see sections 5.13.8 and 5.13.9). A Var_Reset is implied by any API_ operation other than API_Next and API_NextVariable.

## 7.2 Symbols

For the syntax of a symbol see section 6.3.2.95.

The value of a symbol which is a *NUMBER* or a *CONST_SYMBOL* which is not a reserved symbol is the content of the appropriate token.

The value of a *VAR_SYMBOL* which is "taken as a constant" is the *VAR_SYMBOL* itself, otherwise the *VAR_SYMBOL* identifies a variable and its value may vary during execution.

Accessing the value of a symbol which is not "taken as a constant" shall result in trace output, see section 8.3.26:

```
if #Tracing.#Level == 'I' then call #Trace Tag
```

where Tag is '>L>' unless the symbol is a VAR_SYMBOL which, when used as an argument to Var_Value, does not yield an indicator 'D'. In that case, the Tag is '>V>'.

## 7.3 Value of a variable

If *VAR_SYMBOL* does not contain a period, or contains only one period as its last character, the value of the variable is the value associated with *VAR_SYMBOL* in the variable pool, that is #Outcome after

**`Var_Value(Pool,VAR_SYMBOL,'0')`**

If the indicator is 'D', indicating the variable has the 'dropped' attribute, the NOVALUE condition is raised; see sections 7.3.1 and 9.8.6 for exceptions to this.

```
#Response = Var_Value(Pool, VAR_SYMBOL, '0')
if left(#Response,1) == 'D' then
  call #Raise 'NOVALUE', VAR_SYMBOL, ''
```

If *VAR_SYMBOL* contains a period which is not its last character, the value of the variable is the value associated with the derived name.

### 7.3.1 Derived names

A derived name is derived from a *VAR_SYMBOL* as follows:

```
VAR_SYMBOL          :=   Stem Tail
Stem                :=   PlainSymbol '.'
Tail                :=   (PlainSymbol | '.'
                        [PlainSymbol]) ['.'
                        [PlainSymbol]]+
PlainSymbol         :=   (general_letter | digit)+
```

The derived name is the concatenation of:

– the Stem, without further evaluation;

– the Tail, with the PlainSymbols replaced by the values of the symbols. The value of a PlainSymbol which does not start with a digit is #Outcome after

```
Var_Value(Pool,PlainSymbol,'0')
```

These values are obtained without raising the NOVALUE condition.

If the indicator from the Var_Value was not 'D' then:

```
if #Tracing.#Level == 'I' then call #Trace '>C>'
```

The value associated with a derived name is obtained from the variable pool, that is #Outcome after:

```
Var_Value(Pool,Derived_Name,'1')
```

If the indicator is 'D', indicating the variable has the 'dropped' attribute, the NOVALUE condition is raised; see section 9.8.6 for an exception.

### 7.3.2 Value of a reserved symbol

The value of a reserved symbol is the value of a variable with the corresponding name in the pool numbered zero.

## 7.4 Expressions and operators

### 7.4.1 The value of a term

See section 6.3.2.113 for the syntax of a *term*.

The value of a STRING is the content of the token; see section 6.2.1.2.

The value of a *function* is the value it returns, see section 7.5.

If a *term* is a *symbol* or *STRING* then the value of the *term* is the value of that *symbol* or *STRING*.

If a *term* contains an *expr_alias* the value of the *term* is the value of the *expr_alias*, see section 7.4.9.

### 7.4.2 The value of a prefix_expression

If the *prefix_expression* is a *term* then the value of the *prefix_expression* is the value of the *term*, otherwise let rhs be the value of the *prefix_expression* within it — see section 6.3.2.112

If the *prefix_expression* has the form '+' *prefix_expression* then a check is made:

```
if datatype(rhs)\=='NUM' then
 call #Raise 'SYNTAX',41.3, rhs, '+'
```

and the value is the value of (0 + rhs).

If the prefix_expression has the form '–' *prefix_expression* then a check is made:

```
if datatype(rhs)\=='NUM' then
 call #Raise 'SYNTAX',41.3,rhs, '–'
```

and the value is the value of (0 – rhs).

If a *prefix_expression* has the form *not prefix_expression* then

```
if rhs \== '0' then if rhs \=='1' then call #Raise 'SYNTAX', 34.6, not, rhs
```

See section 6.2.2.3 for the value of the third argument to that #Raise.

If the value of rhs is '0' then the value of the *prefix_expression* value is '1', otherwise it is '0'.

If the *prefix_expression* is not a *term* then:

```
if #Tracing.#Level == 'I' then call #Trace '>P>'
```

### 7.4.3 The value of a power_expression

See section 6.3.2.111 for the syntax of a *power_expression*.

If the *power_expression* is a *prefix_expression* then the value of the *power_expression* is the value of the *prefix_expression*.

Otherwise, let lhs be the value of *power_expression* within it, and rhs be the value of *prefix_expression* within it.

```
if datatype(lhs)\=='NUM' then call #Raise 'SYNTAX',41.1,lhs,'**'
if \datatype(rhs,'W') then call #Raise 'SYNTAX',26.1,rhs,'**'
```

The value of the *power_expression* is

```
ArithOp(lhs,'**',rhs)
```

If the *power_expression* is not a *prefix_expression* then:

```
if #Tracing.#Level == 'I' then call #Trace '>O>'
```

### 7.4.4 The value of a multiplication

See section 6.3.2.109 for the syntax of a *multiplication*.

If the *multiplication* is a *power_expression* then the value of the *multiplication* is the value of the *power_expression*.

Otherwise, let lhs be the value of *multiplication* within it, and rhs be the value of *power_expression* within it.

```
if datatype(lhs)\=='NUM' then
 call #Raise 'SYNTAX',41.1,lhs,multiplicative_operation
if datatype(rhs)\=='NUM' then
 call #Raise 'SYNTAX',41.2,rhs,multiplicative_operation
```

The value of the *multiplication* is

```
ArithOp(lhs,multiplicative_operation,rhs)
```

If the *multiplication* is not a *power_expression* then:

```
if #Tracing.#Level == 'I' then call #Trace '>O>'
```

### 7.4.5 The value of an addition

See section 6.3.2.107 for the syntax of *addition*.

If the *addition* is a *multiplication* then the value of the *addition* is the value of the *multiplication*.

Otherwise, let lhs be the value of *addition* within it, and rhs be the value of the *multiplication* within it. Let operation be the *additive_operator*.

```
if datatype(lhs)\=='NUM' then
     call #Raise 'SYNTAX', 41.1, lhs, operation
if datatype(rhs)\=='NUM' then
     call #Raise 'SYNTAX', 41.2, rhs, operation
```

If either of rhs or lhs is not an integer then the value of the *addition* is

**`ArithOp(lhs,operation,rhs)`**

Otherwise if the operation is '+' and the length of the integer lhs+rhs is not greater than #Digits.#Level then the value of *addition* is
**`lhs+rhs`**

Otherwise if the operation is '–' and the length of the integer lhs-rhs is not greater than #Digits.#Level then the value of *addition* is
**`lhs-rhs`**

Otherwise the value of the *addition* is

**`ArithOp(lhs,operation,rhs)`**

If the *addition* is not a *multiplication* then:
**`if #Tracing.#Level == 'I' then call #Trace '>O>'`**

### 7.4.6 The value of a concatenation

See section 6.3.2.106 for the syntax of a *concatenation*.

If the *concatenation* is an *addition* then the value of the *concatenation* is the value of the *addition.*

Otherwise, let lhs be the value of *concatenation* within it, and rhs be the value of the *additive_expression* within it.

If the concatenation contains '||' then the value of the concatenation will have the following characteristics:

– Config_Length(Value) will be equal to Config_Length(lhs)+Config_Length(rhs).

– #Outcome will be 'equal' after each of:

  – Config_Compare(Config_Substr(lhs,n),Config_Substr(Value,n)) for values of n not less than 1 and not more than Config_Length(lhs);

  – Config_Compare(Config_Substr(rhs,n),Config_Substr(Value,Config_Length(lhs)+n)) for values of n not less than 1 and not more than Config_Length(rhs).

Otherwise the value of the concatenation will have the following characteristics:

– Config_Length(Value) will be equal to Config_Length(lhs)+1+Config_Length(rhs).

– #Outcome will be 'equal' after each of:

  – Config_Compare(Config_Substr(lhs,n),Config_Substr(Value,n)) for values of n not less than 1 and not more than Config_Length(lhs);

  – Config_Compare(' ',Config_Substr(Value,Config_Length(lhs)+1));

  – Config_Compare(Config_Substr(rhs,n),Config_Substr(Value,Config_Length(lhs)+1+n)) for values of n not less than 1 and not more than Config_Length(rhs).

If the *concatenation* is not an *addition* then:

```
if #Tracing.#Level == 'I' then call #Trace '>O>'
```

### 7.4.7 The value of a comparison

See section 6.3.2.102 for the syntax of a *comparison*.

If the *comparison* is a *concatenation* then the value of the *comparison* is the value of the *concatenation.*

Otherwise, let lhs be the value of the *comparison* within it, and rhs be the value of the *concatenation* within it.

If the comparison has a *comparison_operator* that is a *strict_compare* then the variable #Test is set as follows:

#Test is set to 'E'. Let Length be the smaller of Config_Length(lhs) and Config_Length(rhs). For values of n greater than 0 and not greater than Length, if any, in ascending order, #Test is set to the uppercased first character of #Outcome after:

Config_Compare(Config_Substr(lhs),Config_Substr(rhs)).

If at any stage this sets #Test to a value other than 'E' then the setting of #Test is complete. Otherwise, if Config_Length(lhs) is greater than Config_Length(rhs) then #Test is set to 'G' or if Config_Length(lhs) is less than Config_Length(rhs) then #Test is set to 'L'.

If the comparison has a comparison_operator that is a *normal_compare* then the variable #Test is set as follows:

```
if datatype(lhs)\== 'NUM' | datatype(rhs)\== 'NUM'  then do
  /* Non-numeric non-strict comparison */
  lhs=strip(lhs, 'B', ' ')   /* ExtraBlanks not stripped */
  rhs=strip(rhs, 'B', ' ')
  if length(lhs)>length(rhs) then rhs=left(rhs,length(lhs))
                             else lhs=left(lhs,length(rhs))
  if lhs>>rhs then #Test='G'
              else if lhs<<rhs then #Test='L'
                               else #Test='E'
  end
else do /* Numeric comparison */
  if left(-lhs,1) == '-' & left(+rhs,1) \== '-' then #Test='G'
  else if left(-rhs,1) == '-' & left(+lhs,1) \== '-'  then #Test='L'
       else do
         Difference=lhs - rhs  /* Will never raise an arithmetic condition. */
         if Difference > 0 then #Test='G'
         else if Difference < 0 then #Test='L'
                                else #Test='E'
         end
   end
```

The value of #Test, in conjunction with the *operator* in the *comparison*, determines the value of the *comparison.*

The value of the *comparison* is '1' if

- #Test is 'E' and the *operator* is one of '=', '==', '>=', '<=', '\>', '\<', '>>=', '<<=', '\>>', or '\<<';
- #Test is 'G' and the *operator* is one of '>', '>=', '\<', '\=', '<>', '><', '\==', '>>', '>>=', or '\<<';
- #Test is 'L' and the *operator* is one of '<', '<=', '\>', '\=', '<>', '><', '\==', '<<', '<<=', or '\>>'.

In all other cases the value of the *comparison* is '0'.

If the *comparison* is not a *concatenation* then:
**if #Tracing.#Level == 'I' then call #Trace '>O>'**

### 7.4.8 The value of an and_expression

See section 6.3.2.101 for the syntax of an *and_expression*.

If the *and_expression* is a *comparison* then the value of the *and_expression* is the value of the *comparison.*

Otherwise, let lhs be the value of the *and_expression* within it, and rhs be the value of the *comparison* within it.

```
if lhs \== '0' then if lhs \== '1' then call #Raise 'SYNTAX',34.5,lhs,'&'
if rhs \== '0' then if rhs \== '1' then call #Raise 'SYNTAX',34.6,rhs,'&'
Value='0'
if lhs == '1' then if rhs == '1' then Value='1'
```

If the *and_expression* is not a *comparison* then:
**if #Tracing.#Level == 'I' then call #Trace '>O>'**

### 7.4.9 The value of an expression

See section 6.3.2.97 for the syntax of an *expression*.

The value of an *expression*, or an *expr*, is the value of the *expr_alias* within it.

If the *expr_alias* is an *and_expression* then the value of the *expr_alias* is the value of the *and_expression.*

Otherwise, let lhs be the value of the *expr_alias* within it, and rhs be the value of the *and_expression* within it.

```
if lhs \== '0' then if lhs \== '1' then
  call #Raise 'SYNTAX',34.5,lhs,or_operator
if rhs \== '0' then if rhs \== '1' then
  call #Raise 'SYNTAX',34.6,rhs,or_operator
Value='1'
if lhs == '0' then if rhs == '0' then Value='0'
```

If the *or_operator* is '&&' then

```
if lhs == '1' then if rhs == '1' then Value='0'
```

If the *expr_alias* is not an *and_expression* then:
**if #Tracing.#Level == 'I' then call #Trace '>O>'**

The value of an *expression* or *expr* shall be traced when #Tracing.#Level is 'R'. The tag is '>=>' when the value is used by an assignment and '>>>' when it is not.
**if #Tracing.#Level == 'R' then call #Trace Tag**

### 7.4.10 Arithmetic operations

The user of this standard is assumed to know the results of the binary operators '+' and '-' applied to signed or unsigned integers.

The code of ArithOp itself is assumed to operate under a sufficiently high setting of numeric digits to avoid exponential notation.

```
   ArithOp:

 arg Number1, Operator, Number2
/* The Operator will be applied to Number1 and Number2 under the numeric
```

```
settings #Digits.#Level, #Form.#Level, #Fuzz.#Level */

/* The result is the result of the operation, or the raising of a 'SYNTAX' or
'LOSTDIGITS' condition.  */

/* Variables with digit 1 in their names refer to the first argument of the
operation. Variables with digit 2 refer to the second argument. Variables
with digit 3 refer to the result. */

/*  The quotations and page numbers are from the first reference in
Annex C of this standard.  */

/* The operands are prepared first. (Page 130)  Function Prepare does this,
 separating sign, mantissa and exponent. */

 v = Prepare(Number1,#Digits.#Level)
 parse var v Sign1 Mantissa1 Exponent1

 v = Prepare(Number2,#Digits.#Level)
 parse var v Sign2 Mantissa2 Exponent2

/* The calculation depends on the operator. The routines set Sign3
Mantissa3 and Exponent3. */

Comparator = ''
select
 when Operator == '*'  then call Multiply
 when Operator == '/'  then call DivType
 when Operator == '**' then call Power
 when Operator == '%'  then call DivType
 when Operator == '//' then call DivType
 otherwise call AddSubComp
 end

 call PostOp  /* Assembles Number3 */
 if Comparator \== '' then do

/* Comparison requires the result of subtraction made into a logical     */
/* value.                                                                */

  t = '0'
  select
   when left(Number3,1) == '-' then
     if wordpos(Comparator,'< <= <> >< \= \>') > 0 then t = '1'
   when Number3 \== '0' then
     if wordpos(Comparator,'> >= <> >< \= \<') > 0 then t = '1'
   otherwise
     if wordpos(Comparator,'>= = =< \< \>')    > 0 then t = '1'
   end
  Number3 = t
  end

  return Number3      /* From ArithOp */
```

```
/* Activity before every operation:                                     */

Prepare:  /* Returns Sign Mantissa and Exponent */
/* Preparation of operands, Page 130 */
/* "...terms being operated upon have leading zeros removed (noting the
position of any decimal point, and leaving just one zero if all the digits in
the number are zeros) and are then truncated to DIGITS+1 significant digits
(if necessary)..." */

  arg Number, Digits

  /* Blanks are not significant. */
  /* The exponent is separated */
  parse upper value space(Number,0) with Mantissa 'E' Exponent
  if Exponent == '' then Exponent = '0'

  /* The sign is separated and made explicit. */
  Sign = '+' /* By default */
  if left(Mantissa,1) == '-' then Sign = '-'
  if verify(left(Mantissa,1),'+-') = 0 then Mantissa = substr(Mantissa,2)

  /* Make the decimal point implicit; remove any actual Point from the
  mantissa. */
  p = pos('.',Mantissa)
  if p > 0 then Mantissa = delstr(Mantissa,p,1)
           else p = 1+length(Mantissa)

  /* Drop the leading zeros */
  do q = 1 to length(Mantissa) - 1
   if substr(Mantissa,q,1) \== '0' then leave
   p = p - 1
   end q
  Mantissa = substr(Mantissa,q)

  /* Detect if Mantissa suggests more significant digits than DIGITS
  caters for. */
  do j = Digits+1 to length(Mantissa)
    if substr(Mantissa,j,1) \== '0' then call #Raise 'LOSTDIGITS', Number
    end j

  /* Combine exponent with decimal point position, Page 127 */
  /* "Exponential notation means that the number includes a power of ten
  following an 'E' that indicates how the decimal point will be shifted. Thus
  4E9 is just a shorthand way of writing 4000000000 "   */
  /* Adjust the exponent so that decimal point would be at right of
  the Mantissa. */
  Exponent = Exponent - (length(Mantissa) - p + 1)

  /* Truncate if necessary */
  t = length(Mantissa) - (Digits+1)
  if t > 0 then do
```

```
    Exponent = Exponent + t
    Mantissa = left(Mantissa,Digits+1)
    end

  if Mantissa == '0' then Exponent = 0

return Sign Mantissa Exponent


/* Activity after every operation.                                       */
/* The parts of the value are composed into a single string, Number3.    */

PostOp:
 /* Page 130 */
 /* 'traditional' rounding */
 t = length(Mantissa3) - #Digits.#Level
 if t > 0 then do
    /* 'traditional' rounding */
    Mantissa3 = left(Mantissa3,#Digits.#Level+1) + 5
    if length(Mantissa3) > #Digits.#Level+1 then
       /* There was 'carry' */
       Exponent3 = Exponent3 + 1
    Mantissa3 = left(Mantissa3,#Digits.#Level)
    Exponent3 = Exponent3 + t
   end
 /* "A result of zero is always expressed as a single character '0' "*/
 if verify(Mantissa3,'0') = 0 then Number3 = '0'
 else do
   if Operator == '/' | Operator == '**' then do
     /* Page 130 "For division, insignificant trailing zeros are removed
     after rounding." */
     /* Page 133 "... insignificant trailing zeros are removed." */
     do q = length(Mantissa3) by -1 to 2
       if substr(Mantissa3,q,1) \== '0' then leave
       Exponent3 = Exponent3 + 1
       end q
     Mantissa3 = substr(Mantissa3,1,q)
     end

    if Floating() == 'E' then do  /* Exponential format */

      Exponent3 = Exponent3 + (length(Mantissa3)-1)

      /* Page 136 "Engineering notation causes powers of ten to expressed as a
      multiple of 3 - the integer part may therefore range from 1 through
      999." */
      g = 1
      if #Form.#Level == 'E' then do
      /* Adjustment to make exponent a multiple of 3 */
        g = Exponent3//3   /* Recursively using ArithOp as
                              an external routine. */
        if g < 0 then g = g + 3
        Exponent3 = Exponent3 - g
        g = g + 1
        if length(Mantissa3) < g then
           Mantissa3 = left(Mantissa3,g,'0')
        end  /* Engineering */

      /* Exact check on the exponent. */
      if Exponent3 > #Limit_ExponentDigits then
        call #Raise 'SYNTAX', 42.1, Number1, Operator, Number2
      if -#Limit_ExponentDigits > Exponent3 then
        call #Raise 'SYNTAX', 42.2, Number1, Operator, Number2

      /* Insert any decimal [point. */
      if length(Mantissa3) \= g then Mantissa3 = insert('.',Mantissa3,g)
      /* Insert the E */
      if Exponent3 >= 0 then Number3 = Mantissa3'E+'Exponent3
                        else Number3 = Mantissa3'E'Exponent3
      end /* Exponent format */
    else do /* 'pure number' notation */
      p = length(Mantissa3) + Exponent3 /* Position of the point within
                                           Mantissa  */
      /* Add extra zeros needed on the left of the point. */
      if p < 1 then do
        Mantissa3 = copies('0',1 - p)||Mantissa3
        p = 1
        end
      /* Add needed zeros on the right. */
      if p > length(Mantissa3) then
         Mantissa3 = Mantissa3||copies('0',p-length(Mantissa3))
      /* Format with decimal point. */
      Number3 = Mantissa3
      if p < length(Number3) then Number3 = insert('.',Mantissa3,p)
                             else Number3 = Mantissa3
      end /* pure */
    if Sign3 == '-' then Number3 = '-'Number3
    end /* Non-Zero */
 return


/* This tests whether exponential notation is needed.                    */

Floating:
 /* The rule in the reference has been improved upon. */
  t = ''
  if Exponent3+length(Mantissa3) > #Digits.#Level then t = 'E'

  if length(Mantissa3) + Exponent3 < -5 then t = 'E'
  return t


/* Add, Subtract and Compare.                                            */

AddSubComp:  /* Page 130 */
 /* This routine is used for comparisons since comparison is
 defined in terms of subtraction. Page 134 */
 /* "Numeric comparison is affected by subtracting the two numbers(calculating
 the difference) and then comparing the result with '0'." */
 NowDigits = #Digits.#Level
 if Operator \=='+' & Operator \== '-' then do
   Comparator = Operator
   /* Page 135 "The effect of NUMERIC FUZZ is to temporarily reduce the value
   of NUMERIC DIGITS by the NUMERIC FUZZ value for each numeric comparison" */
   NowDigits = NowDigits - #Fuzz.#Level
   end

 /* Page 130 "If either number is zero then the other number ... is used as
 the result (with sign adjustment as appropriate). */
 if Mantissa2 == '0' then do  /* Result is the 1st operand */
  Sign3=Sign1; Mantissa3 = Mantissa1; Exponent3 = Exponent1
  return ''
  end

 if Mantissa1 == '0' then do  /* Result is the 2nd operand */
  Sign3 = Sign2; Mantissa3 = Mantissa2; Exponent3 = Exponent2
  if Operator \== '+' then if Sign3 = '+' then Sign3 = '-'
                                          else Sign3 = '+'
  return ''
  end

/* The numbers may need to be shifted into alignment. */
/* Change to make the exponent to reflect a decimal point on the left,
so that right truncation/extension of mantissa doesn't alter exponent. */
  Exponent1 = Exponent1 + length(Mantissa1)
  Exponent2 = Exponent2 + length(Mantissa2)
/* Deduce the implied zeros on the left to provide alignment. */
  Align1 = 0
  Align2 = Exponent1 - Exponent2
  if Align2 > 0 then do /* Arg 1 provides a more significant digit */
    Align2 = min(Align2,NowDigits+1) /* No point in shifting further. */
    /* Shift to give Arg2 the same exponent as Arg1 */
    Mantissa2 = copies('0',Align2) || Mantissa2
    Exponent2 = Exponent1
    end
  if Align2 < 0 then do /* Arg 2 provides a more significant digit */
    /* Shift to give Arg1 the same exponent as Arg2 */
    Align1 = -Align2
    Align1 = min(Align1,NowDigits+1) /* No point in shifting further. */
    Align2 = 0
    Mantissa1 = copies('0',Align1) || Mantissa1
    Exponent1 = Exponent2
    end

/* Maximum working digits is NowDigits+1.  Footnote 41. */

 SigDigits = max(length(Mantissa1),length(Mantissa2))
 SigDigits = min(SigDigits,NowDigits+1)
```

```
/* Extend a mantissa with right zeros, if necessary. */
 Mantissa1 = left(Mantissa1,SigDigits,'0')
 Mantissa2 = left(Mantissa2,SigDigits,'0')

/* The exponents are adjusted so that
the working numbers are integers, ie decimal point on the right. */
 Exponent3 = Exponent1-SigDigits
 Exponent1 = Exponent3
 Exponent2 = Exponent3

 if Operator = '+' then
      Mantissa3 = (Sign1 || Mantissa1) + (Sign2 || Mantissa2)
 else Mantissa3 = (Sign1 || Mantissa1) - (Sign2 || Mantissa2)

 /* Separate the sign */
 if Mantissa3 < 0 then do
   Sign3 = '-'
   Mantissa3 = substr(Mantissa3,2)
   end
 else Sign3 = '+'

 /* "The result is then rounded to NUMERIC DIGITS digits if necessary,
 taking into account any extra (carry) digit on the left after addition,
 but otherwise counting from the position corresponding to the most
 significant digit of the terms being added or subtracted." */

 if length(Mantissa3) > SigDigits then SigDigits = SigDigits+1
 d = SigDigits - NowDigits   /* Digits to drop. */
 if d <= 0 then return
 t = length(Mantissa3) - d  /* Digits to keep. */
 /* Page 130. "values of 5 through 9 are rounded up, values of 0 through 4 are
 rounded down." */
 if t > 0 then do
    /* 'traditional' rounding */
    Mantissa3 = left(Mantissa3, t + 1) + 5
    if length(Mantissa3) > t+1 then
       /* There was 'carry' */
       /* Keep the extra digit unless it takes us over the limit. */
       if t < NowDigits then t = t+1
                         else Exponent3 = Exponent3+1
    Mantissa3 = left(Mantissa3,t)
    Exponent3 = Exponent3 + d
    end /* Rounding */
 else Mantissa3 = '0'
 return  /* From AddSubComp */


/* Multiply operation:  */

Multiply:      /* p 131 */
 /* Note the sign of the result */
 if Sign1 == Sign2 then Sign3 = '+'
                   else Sign3 = '-'
 /* Note the exponent */
 Exponent3 = Exponent1 + Exponent2
 if Mantissa1 == '0' then do
   Mantissa3 = '0'
   return
   end
 /* Multiply the Mantissas */
 Mantissa3 = ''
 do q=1 to length(Mantissa2)
  Mantissa3 = Mantissa3'0'
  do substr(Mantissa2,q,1)
    Mantissa3 = Mantissa3 + Mantissa1
    end
  end q
 return /* From Multiply */


/* Types of Division:  */

DivType:       /* p 131 */
 /* Check for divide-by-zero */
 if Mantissa2 == '0' then call #Raise 'SYNTAX', 42.3
 /* Note the exponent of the result  */
 Exponent3 = Exponent1 - Exponent2
 /* Compute (one less than) how many digits will be in the integer
part of the result. */
 IntDigits = length(Mantissa1) - Length(Mantissa2) + Exponent3
 /* In some cases, the result is known to be zero. */
 if Mantissa1 = 0 | (IntDigits < 0 & Operator = '%') then do
   Mantissa3 = 0
   Sign3 = '+'
   Exponent3 = 0
   return
   end
 /* In some cases, the result is known to be to be the first argument. */
 if IntDigits < 0 & Operator == '//' then do
   Mantissa3 = Mantissa1
   Sign3 = Sign1
   Exponent3 = Exponent1
   return
   end
 /* Note the sign of the result. */
 if Sign1 == Sign2 then Sign3 = '+'
                   else Sign3 = '-'
 /* Make Mantissa1 at least as large as Mantissa2 so Mantissa2 can be
  subtracted without causing leading zero to result. Page 131 */
 a = 0
 do while Mantissa2 > Mantissa1
   Mantissa1 = Mantissa1'0'
   Exponent3 = Exponent3 - 1
   a = a + 1
   end
 /* Traditional divide */
```

```
Mantissa3 = ''
/* Subtract from part of Mantissa1 that has length of Mantissa2 */
x = left(Mantissa1,length(Mantissa2))
y = substr(Mantissa1,length(Mantissa2)+1)
do forever
  /* Develop a single digit in z by repeated subtraction. */
  z = 0
  do forever
    x = x - Mantissa2
    if left(x,1) == '-' then leave
    z = z + 1
    end
  x = x + Mantissa2   /* Recover from over-subtraction */
  /* The digit becomes part of the result */
  Mantissa3 = Mantissa3 || z
  if Mantissa3 == '0' then Mantissa3 = '' /* A single leading
                                             zero can happen.  */
  /* x||y is the current residue */
  if y == '' then if x = 0 then leave /* Remainder is zero */
  if length(Mantissa3) > #Digits.#Level then leave /* Enough digits
                                                      in the result */
  /* Check type of division */
  if Operator \== '/' then do
    if IntDigits = 0 then leave
    IntDigits = IntDigits - 1
    end
  /* Prepare for next digit */
  /* Digits come from y, until that is exhausted. */
  /* When y is exhausted an extra zero is added to Mantissa1 */
  if y == '' then do
    y = '0'
    Exponent3 = Exponent3 - 1
    a = a + 1
    end
  x = x || left(y,1)
  y = substr(y,2)
  end /* Iterate for next digit. */
Remainder = x || y
Exponent3 = Exponent3 + length(y)  /* The loop may have been left early. */
/* Leading zeros are taken off the Remainder. */
do while length(Remainder) > 1 & Left(Remainder,1) == '0'
  Remainder = substr(Remainder,2)
  end
if Operator \== '/' then do
  /* Check whether % would fail, even if operation is // */
  /* Page 133.  % could fail by needing exponential notation */
  if Floating() == 'E' then do
    if Operator == '%' then MsgNum = 26.11
                       else MsgNum = 26.12
    call #Raise 'SYNTAX', MsgNum, Number1 , Number2, #Digits.#Level
    end
  end
if Operator == '//' then do
```

```
    /* We need the remainder */
    Sign3 = Sign1
    Mantissa3 = Remainder
    Exponent3 = Exponent1 - a
    end
 return  /* From DivType */


/* The Power operation:  */

Power:       /* page 132 */
/* The second argument should be an integer */
 if \WholeNumber2() then call #Raise 'SYNTAX', 26.8, Number2
/* Lhs to power zero is always 1 */
 if Mantissa2 == '0' then do
    Sign3 = '+'
    Mantissa3 = '1'
    Exponent3 = '0'
    return
    end

 /* Pages 132-133 The Power algorithm */
 Rhs = left(Mantissa2,length(Mantissa2)+Exponent2,'0')/* Explicit
                                                  integer form */
 L = length(Rhs)
 b = X2B(D2X(Rhs)) /* Makes Rhs in binary notation */
 /* Ignore initial zeros */
 do q = 1 by 1
   if substr(b,q,1) \== '0' then leave
   end q
 a = 1
 do forever
 /* Page 133 "Using a precision of DIGITS+L+1" */
  if substr(b,q,1) == '1' then do
    a = Recursion('*',Sign1 || Mantissa1'E'Exponent1)
    if left(a,2) == 'MN' then signal PowerFailed
    end
  /* Check for finished */
  if q = length(b) then leave
  /* Square a */
  a = Recursion('*',a)
  if left(a,2) == 'MN' then signal PowerFailed
  q = q + 1
 end
 /* Divide into one for negative power */
 if Sign2 == '-' then do
    Sign2 = '+'
    a = Recursion('/')
    if left(a,2) == 'MN' then signal PowerFailed
   end
 /* Split the value up so that PostOp can put it together with rounding */
  Parse value Prepare(a,#Digits.#Level+L+1) with Sign3 Mantissa3 Exponent3
  return
```

```
PowerFailed:
/* Distinquish overflow and underflow */
  RcWas = substr(a,4)
  if Sign2 = '-' then if RcWas == '42.1' then RcWas = '42.2'
                                          else RcWas = '42.1'
  call #Raise 'SYNTAX', RcWas, Number1, '**', Number2
  /* No return */

WholeNumber2:
   numeric digits Digits
   if #Form.#Level == 'S' then numeric form scientific
                          else numeric form engineering
   return datatype(Number2,'W')

Recursion: /* Called only from '**' */
  numeric digits #Digits.#Level + L + 1
  signal on syntax name Overflowed
/* Uses ArithOp again under new numeric settings. */
  if arg(1) == '/' then  return 1 / a
                   else  return a * arg(2)
Overflowed:
  return 'MN '.MN
```

## 7.5 Functions

### 7.5.1 Invocation

Invocation occurs when a *function* or a *call* is evaluated. Invocation of a function may result in a value, in which case:
**if #Tracing.#Level == 'I' then call #Trace '>F>'**

### 7.5.2 Evaluation of arguments

The argument positions are the positions in the *expression_list* where syntactically an *expression* occurs or could have occurred. Let ArgNumber be the number of an argument position, counting from 1 at the left; the range of ArgNumber is all whole numbers greater than zero.

For each value of ArgNumber, #ArgExists.#NewLevel.ArgNumber is set '1' if there is an *expression* present, '0' if not.

From the left, if #ArgExists.#NewLevel.ArgNumber is '1' then #Arg.#NewLevel.ArgNumber is set to the value of the corresponding *expression*. If #ArgExists.#NewLevel.ArgNumber is '0' then #Arg.#NewLevel.ArgNumber is set to the null string.

#ArgExists.#NewLevel.0 is set to the largest ArgNumber for which #ArgExists.#NewLevel.ArgNumber is '1', or to zero if there is no such value of ArgNumber.

### 7.5.3 The value of a label

The value of a LABEL, or of the *taken_constant* in the *function* or *call_instruction*, is taken as a constant, see section 7.2. If the *taken_constant* is not a *string_literal* it is a reference to the first *LABEL* in the program which has the same value. The comparison is made with the '==' operator.

If there is such a matching label and the label is trace-only (see section 6.4.2) then a condition is raised:
**call #Raise 'SYNTAX', 16.3, *taken_constant***

If there is such a matching label, and the label is not trace-only, execution continues at the label with routine initialization (see section 8.2.2). This is execution of an internal routine.

If there is no such matching label, or if the *taken_constant* is a *string_literal*, further comparisons are made.

If the value of the *taken_constant* matches the name of some built-in function then that built-in function is invoked. The names of the built-in functions are defined in section 9 and are in uppercase.

If the value does not match any built-in function name, Config_ExternalRoutine is used to invoke an external routine.

Whenever a matching label is found, the variables SIGL and .SIGL are assigned the value of the line number of the clause which caused the search for the label. In the case of an invocation resulting from a condition occurring that shall be the clause in which the condition occurred.

```
Var_Set(#Pool, 'SIGL', '0', #LineNumber)
Var_Set(0 , '.SIGL', '0', #LineNumber)
```

### 7.5.4 The value of a function

A built-in function completes when it returns from the activity defined in section 9. The value of a built-in function is defined in section 9.

An internal routine completes when #Level returns to the value it had when the routine was invoked. The value of the internal function is the value of the *expression* on the *return* which completed the routine.

The value of an external function is determined by Config_ExternalRoutine.

### 7.5.5 Use of Config_ExternalRoutine

The values of the arguments to the use of Config_ExternalRoutine, in order, are:

The argument How is 'SUBROUTINE' if the invocation is from a *call*, 'FUNCTION' if the invocation is from a *function*.

The argument NameType is '1' if the *taken_constant* is a *string_literal*, '0' otherwise.

The argument Name is the value of the *taken_constant*.

The argument Environment is the value of this argument on the API_Start which started this execution.

The argument Arguments is the #Arg. and #ArgExists. data.

The argument Streams is the value of this argument on the API_Start which started this execution.

The argument Traps is the value of this argument on the API_Start which started this execution.

Var_Reset is invoked and #API_Enabled set to '1' before use of Config_ExternalRoutine. #API_Enabled is set to '0' after.

The response from Config_ExternalRoutine is processed. If no conditions are (implicitly) raised, #Outcome is the value of the function.

# 8 Execution

This section describes the execution of instructions, and how the sequence of execution can vary from the normal execution in order of appearance in the program.

## 8.1 Notation

Notation functions are functions which are not directly accessible as functions in a program but are used in this standard as a notation for defining semantics.

Some notation functions allow reference to syntax constructs defined in section 6.3.2. Which instance of the syntax construct in the program is being referred to is implied; it is the one for which the semantics are being specified.

The BNF_primary referenced may be directly in the *production* or in some component referenced in the *production*, recursively. The components are considered in left to right order.

**#Contains(Identifier, BNF_primary)**

where:

Identifier is an *identifier* in a *production* (see section 6.1.5.1) defined in section 6.3.2.

BNF_primary is a *bnf_primary* (see section 6.1.5.4) in a *production* defined in section 6.3.2.

Return '1' if the *production* identified by Identifier contained a *bnf_primary* identified by BNF_primary, otherwise return '0'.

**#Instance(Identifier, BNF_primary)**

where:

Identifier is an *identifier* in a *production* defined in section 6.3.2.

BNF_primary is a *bnf_primary* in a *production* defined in section 6.3.2.

Returns the content of the particular instance of the BNF_primary. If the BNF_primary is a VAR_SYMBOL this is referred to as the symbol "taken as a constant."

**#Evaluate(Identifier, BNF_primary)**

where:

Identifier is an *identifier* in a *production* defined in section 6.3.2.

BNF_primary is a *bnf_primary* in a *production* defined in section 6.3.2.

Return the value of the BNF_primary in the production identified by Identifier.

**#Execute(Identifier, BNF_primary)**

where:

Identifier is an *identifier* in a *production* defined in section 6.3.2.

BNF_primary is a *bnf_primary* in a *production* defined in section 6.3.2.

Perform the instructions identified by the BNF_primary in the production identified by Identifier.

**#Parses(Value, BNF_primary)**

where:

Value is a string

BNF_primary is a *bnf_primary* in a *production* defined in section 6.3.2.

Return '1' if the Value matches the definition of the BNF_primary, by the rules of section 6, '0' otherwise.

**#Clause(Label)**

where:

Label is a label in code used by this standard to describe processing.

Return an identification of that label. The value of this identification is used only by the #Goto notation function.

**#Goto(Value)**

where:

Value identifies a label in code used by this standard to describe processing.

The description of processing continues at the identified label.

**#Retry()**

This notation is used in the description of interactive tracing to specify re-execution of the clause just previously executed. It has the effect of transferring execution to the beginning of that clause, with state variable #Loop set to the value it had when that clause was previously executed.

## 8.2 Initializations, terminations

### 8.2.1 Program initialization and message texts

Processing of a program begins when API_Start is executed. Some of the values which affect processing of the program are parameters of API_Start:

#HowInvoked is set to 'COMMAND', 'FUNCTION' or 'SUBROUTINE' according to the first parameter of API_Start.

#Source is set to the value of the second parameter of API_Start.

The third parameter of API_Start is used to determine the initial active environment.

The fourth parameter of API_Start is used to determine the arguments. For each argument position #ArgExists.1.ArgNumber is set '1' if there is an argument present, '0' if not. ArgNumber is the number of the argument position, counting from 1. If #ArgExists.1.ArgNumber is '1' then #Arg.1.ArgNumber is set to the value of the corresponding argument. If #ArgExists.1.ArgNumber is '0' then #Arg.1.Arg is set to the null string. #ArgExists.1.0 is set to the largest n for which #ArgExists.1.n is '1', or to zero if there is no such value of n.

Some of the values which affect processing of the program are provided by the configuration:

```
call Config_OtherBlankCharacters
```

```
#AllBlanks = ' '#Outcome /* "Real" blank concatenated with others */
#Bif_Digits. = 9
call Config_Constants
```

Some of the state variables set by this call are limits, and appear in the text of error messages. The relation between message numbers and message text is defined by the following list, where the message number appears immediately before an '=' and the message text follows in quotes.

```
   #ErrorText.    = ''

   #ErrorText.0.1 = 'Error <value> running <source>, line <linenumber>:'
   #ErrorText.0.2 = 'Error <value> in interactive trace:'
   #ErrorText.0.3 = 'Interactive trace.  "Trace Off" to end debug. ',
                    'ENTER to continue.'
   #ErrorText.2   = 'Failure during finalization'
   #ErrorText.2.1 = 'Failure during finalization: <description>'

   #ErrorText.3   = 'Failure during initialization'
   #ErrorText.3.1 = 'Failure during initialization: <description>'

   #ErrorText.4   = 'Program interrupted'
   #ErrorText.4.1 = 'Program interrupted with HALT condition: <description>'

   #ErrorText.5   = 'System resources exhausted'
   #ErrorText.5.1 = 'System resources exhausted: <description>'

   #ErrorText.6   = 'Unmatched "/*" or quote'
   #ErrorText.6.1 = 'Unmatched comment delimiter ("/*")'
   #ErrorText.6.2 = "Unmatched single quote (')"
   #ErrorText.6.3 = 'Unmatched double quote (")'

   #ErrorText.7   = 'WHEN or OTHERWISE expected'
   #ErrorText.7.1 = 'SELECT on line <linenumber> requires WHEN;',
                    'found "<token>"'
   #ErrorText.7.2 = 'SELECT on line <linenumber> requires WHEN, OTHERWISE,',
                    'or END; found "<token>"'
   #ErrorText.7.3 = 'All WHEN expressions of SELECT on line <linenumber> are',
                    'false; OTHERWISE expected'

   #ErrorText.8   = 'Unexpected THEN or ELSE'
   #ErrorText.8.1 = 'THEN has no corresponding IF or WHEN clause'
   #ErrorText.8.2 = 'ELSE has no corresponding THEN clause'

   #ErrorText.9   = 'Unexpected WHEN or OTHERWISE'
   #ErrorText.9.1 = 'WHEN has no corresponding SELECT'
   #ErrorText.9.2 = 'OTHERWISE has no corresponding SELECT'

   #ErrorText.10  = 'Unexpected or unmatched END'
   #ErrorText.10.1= 'END has no corresponding DO or SELECT'
   #ErrorText.10.2= 'END corresponding to DO on line <linenumber>',
                    'must have a symbol following that matches',
                    'the control variable (or no symbol);',
                    'found "<token>"'
   #ErrorText.10.3= 'END corresponding to DO on line <linenumber>',
                    'must not have a symbol following it because',
                    'there is no control variable;',
                    'found "<token>"'
   #ErrorText.10.4= 'END corresponding to SELECT on line <linenumber>',
                    'must not have a symbol following;',
                    'found "<token>"'
   #ErrorText.10.5= 'END must not immediately follow THEN'
   #ErrorText.10.6= 'END must not immediately follow ELSE'

   #ErrorText.13  = 'Invalid character in program'
   #ErrorText.13.1= 'Invalid character in program "<char>"',
                    "('<hex-encoding>'X)"

   #ErrorText.14  = 'Incomplete DO/SELECT/IF'
   #ErrorText.14.1= 'DO instruction requires a matching END'
   #ErrorText.14.2= 'SELECT instruction requires a matching END'
   #ErrorText.14.3= 'THEN requires a following instruction'
   #ErrorText.14.4= 'ELSE requires a following instruction'

   #ErrorText.15  = 'Invalid hexadecimal or binary string'
   #ErrorText.15.1= 'Invalid location of blank in position',
                    '<position> in hexadecimal string'
   #ErrorText.15.2= 'Invalid location of blank in position',
                    '<position> in binary string'
   #ErrorText.15.3= 'Only 0-9, a-f, A-F, and blank are valid in a',
                    'hexadecimal string; found "<char>"'
   #ErrorText.15.4= 'Only 0, 1, and blank are valid in a',
                    'binary string; found "<char>"'

   #ErrorText.16  = 'Label not found'
   #ErrorText.16.1= 'Label "<name>" not found'
   #ErrorText.16.2= 'Cannot SIGNAL to label "<name>" because it is',
                    'inside an IF, SELECT or DO group'
   #ErrorText.16.3= 'Cannot invoke label "<name>" because it is',
                    'inside an IF, SELECT or DO group'

   #ErrorText.17  = 'Unexpected PROCEDURE'
   #ErrorText.17.1= 'PROCEDURE is valid only when it is the first',
                    'instruction executed after an internal CALL',
                    'or function invocation'

   #ErrorText.18  = 'THEN expected'
   #ErrorText.18.1= 'IF keyword on line <linenumber> requires',
                    'matching THEN clause; found "<token>"'
   #ErrorText.18.2= 'WHEN keyword on line <linenumber> requires',
                    'matching THEN clause; found "<token>"'

   #ErrorText.19  = 'String or symbol expected'
   #ErrorText.19.1= 'String or symbol expected after ADDRESS keyword;',
                    'found "<token>"'
   #ErrorText.19.2= 'String or symbol expected after CALL keyword;',
                    'found "<token>"'
   #ErrorText.19.3= 'String or symbol expected after NAME keyword;',
                    'found "<token>"'
   #ErrorText.19.4= 'String or symbol expected after SIGNAL keyword;',
                    'found "<token>"'
   #ErrorText.19.6= 'String or symbol expected after TRACE keyword;',
                    'found "<token>"'
   #ErrorText.19.7= 'Symbol expected in parsing pattern;',
                    'found "<token>"'

   #ErrorText.20  = 'Name expected'
   #ErrorText.20.1= 'Name required; found "<token>"'
   #ErrorText.20.2= 'Found "<token>" where only a name is valid'

   #ErrorText.21  = 'Invalid data on end of clause'
   #ErrorText.21.1= 'The clause ended at an unexpected token;',
                    'found "<token>"'

   #ErrorText.22  = 'Invalid character string'
   #ErrorText.22.1= "Invalid character string '<hex-encoding>'X"

   #ErrorText.23  = 'Invalid data string'
   #ErrorText.23.1= "Invalid data string '<hex-encoding>'X"

   #ErrorText.24  = 'Invalid TRACE request'
   #ErrorText.24.1= 'TRACE request letter must be one of',
                    '"ACEFILNOR"; found "<value>"'

   #ErrorText.25  = 'Invalid sub-keyword found'
   #ErrorText.25.1= 'CALL ON must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.2= 'CALL OFF must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.3= 'SIGNAL ON must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.4= 'SIGNAL OFF must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.5= 'ADDRESS WITH must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.6= 'INPUT must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.7= 'OUTPUT must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.8= 'APPEND must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.9= 'REPLACE must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.11='NUMERIC FORM must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.12='PARSE must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.13='UPPER must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
   #ErrorText.25.14='ERROR must be followed by one of the',
                    'keywords <keywords>; found "<token>"'
```

```
#ErrorText.25.15='NUMERIC must be followed by one of the',
                 'keywords <keywords>; found "<token>"'
#ErrorText.25.16='FOREVER must be followed by one of the',
                 'keywords <keywords> or nothing; found "<token>"'
#ErrorText.25.17='PROCEDURE must be followed by the keyword',
                 'EXPOSE or nothing; found "<token>"'

#ErrorText.26  = 'Invalid whole number'
#ErrorText.26.1= 'Whole numbers must fit within current DIGITS',
                 'setting(<value>); found "<value>"'
#ErrorText.26.2= 'Value of repetition count expression in DO instruction',
                 'must be zero or a positive whole number;',
                 'found "<value>"'
#ErrorText.26.3= 'Value of FOR expression in DO instruction',
                 'must be zero or a positive whole number;',
                 'found "<value>"'
#ErrorText.26.4= 'Positional pattern of parsing template',
                 'must be a whole number; found "<value>"'
#ErrorText.26.5= 'NUMERIC DIGITS value',
                 'must be a positive whole number; found "<value>"'
#ErrorText.26.6= 'NUMERIC FUZZ value',
                 'must be zero or a positive whole number;',
                 'found "<value>"'
#ErrorText.26.7= 'Number used in TRACE setting',
                 'must be a whole number; found "<value>"'
#ErrorText.26.8= 'Operand to right of the power operator ("**")',
                 'must be a whole number; found "<value>"'
#ErrorText.26.11='Result of <value> % <value> operation would need',
                 'exponential notation at current NUMERIC DIGITS <value>'
#ErrorText.26.12='Result of % operation used for <value> // <value>',
                 'operation would need',
                 'exponential notation at current NUMERIC DIGITS <value>'

#ErrorText.27  = 'Invalid DO syntax'
#ErrorText.27.1= 'Invalid use of keyword "<token>" in DO clause'

#ErrorText.28  = 'Invalid LEAVE or ITERATE'
#ErrorText.28.1= 'LEAVE is valid only within a repetitive DO loop'
#ErrorText.28.2= 'ITERATE is valid only within a repetitive DO loop'
#ErrorText.28.3= 'Symbol following LEAVE ("<token>") must',
                 'either match control variable of a current',
                 'DO loop or be omitted'
#ErrorText.28.4= 'Symbol following ITERATE ("<token>") must',
                 'either match control variable of a current',
                 'DO loop or be omitted'

#ErrorText.29  = 'Environment name too long'
#ErrorText.29.1= 'Environment name exceeds',
                 #Limit_EnvironmentName 'characters; found "<name>"'

#ErrorText.30  = 'Name or string too long'
#ErrorText.30.1= 'Name exceeds' #Limit_Name 'characters'
#ErrorText.30.2= 'Literal string exceeds' #Limit_Literal 'characters'
```

```
#ErrorText.31  = 'Name starts with number or "."'
#ErrorText.31.1= 'A value cannot be assigned to a number;',
                 'found "<token>"'
#ErrorText.31.2= 'Variable symbol must not start with a number;',
                 'found "<token>"'
#ErrorText.31.3= 'Variable symbol must not start with a ".";',
                 'found "<token>"'

#ErrorText.33  = 'Invalid expression result'
#ErrorText.33.1= 'Value of NUMERIC DIGITS ("<value>")',
                 'must exceed value of NUMERIC FUZZ "(<value>)"'
#ErrorText.33.2= 'Value of NUMERIC DIGITS ("<value>")',
                 'must not exceed' #Limit_Digits
#ErrorText.33.3= 'Result of expression following NUMERIC FORM',
                 'must start with "E" or "S"; found "<value>"'

#ErrorText.34  = 'Logical value not "0" or "1"'
#ErrorText.34.1= 'Value of expression following IF keyword',
                 'must be exactly "0" or "1"; found "<value>"'
#ErrorText.34.2= 'Value of expression following WHEN keyword',
                 'must be exactly "0" or "1"; found "<value>"'
#ErrorText.34.3= 'Value of expression following WHILE keyword',
                 'must be exactly "0" or "1"; found "<value>"'
#ErrorText.34.4= 'Value of expression following UNTIL keyword',
                 'must be exactly "0" or "1"; found "<value>"'
#ErrorText.34.5= 'Value of expression to left',
                 'of logical operator "<operator>"',
                 'must be exactly "0" or "1"; found "<value>"'
#ErrorText.34.6= 'Value of expression to right',
                 'of logical operator "<operator>"',
                 'must be exactly "0" or "1"; found "<value>"'

#ErrorText.35  = 'Invalid expression'
#ErrorText.35.1= 'Invalid expression detected at "<token>"'

#ErrorText.36  = 'Unmatched "(" in expression'

#ErrorText.37  = 'Unexpected "," or ")"'
#ErrorText.37.1= 'Unexpected ","'
#ErrorText.37.2= 'Unmatched ")" in expression'

#ErrorText.38  = 'Invalid template or pattern'
#ErrorText.38.1= 'Invalid parsing template detected at "<token>"'
#ErrorText.38.2= 'Invalid parsing position detected at "<token>"'
#ErrorText.38.3= 'PARSE VALUE instruction requires WITH keyword'

#ErrorText.40  = 'Incorrect call to routine'
#ErrorText.40.1= 'External routine "<name>" failed'
#ErrorText.40.3= 'Not enough arguments in invocation of <bif>;',
                 'minimum expected is <argnumber>'
#ErrorText.40.4= 'Too many arguments in invocation of <bif>;',
                 'maximum expected is <argnumber>'
```

```
#ErrorText.40.5= 'Missing argument in invocation of <bif>;',
                 'argument <argnumber> is required'
#ErrorText.40.9= '<bif> argument <argnumber>',
                 'exponent exceeds' #Limit_ExponentDigits 'digits;',
                 'found "<value>"'
#ErrorText.40.11='<bif> argument <argnumber>',
                 'must be a number; found "<value>"'
#ErrorText.40.12='<bif> argument <argnumber>',
                 'must be a whole number; found "<value>"'
#ErrorText.40.13='<bif> argument <argnumber>',
                 'must be zero or positive; found "<value>"'
#ErrorText.40.14='<bif> argument <argnumber>',
                 'must be positive; found "<value>"'
#ErrorText.40.15='<bif> argument <argnumber>',
                 'must fit in <value> digits; found "<value>"'
#ErrorText.40.16='<bif> argument 1',
                 'requires a whole number fitting within',
                 'DIGITS(<value>); found "<value>"'
#ErrorText.40.17='<bif> argument 1',
                 'must have an integer part in the range 0:90 and a',
                 'decimal part no larger than .9; found "<value>"'
#ErrorText.40.18='<bif> conversion must',
                 'have a year in the range 0001 to 9999'
#ErrorText.40.19='<bif> argument 2, "<value>", is not in the format',
                  'described by argument 3, "<value>"'
#ErrorText.40.21='<bif> argument <argnumber> must not be null'
#ErrorText.40.23='<bif> argument <argnumber>',
                 'must be a single character; found "<value>"'
#ErrorText.40.24='<bif> argument 1',
                 'must be a binary string; found "<value>"'
#ErrorText.40.25='<bif> argument 1',
                 'must be a hexadecimal string; found "<value>"'
#ErrorText.40.26='<bif> argument 1',
                 'must be a valid symbol; found "<value>"'
#ErrorText.40.27='<bif> argument 1',
                 'must be a valid stream name; found "<value>"'
#ErrorText.40.28='<bif> argument <argnumber>,',
                 'option must start with one of "<optionslist>";',
                 'found "<value>"'
#ErrorText.40.29='<bif> conversion to format "<value>" is not allowed'
#ErrorText.40.31='<bif> argument 1 ("<value>") must not exceed 100000'
#ErrorText.40.32='<bif> the difference between argument 1 ("<value>") and',
                 'argument 2 ("<value>") must not exceed 100000'
#ErrorText.40.33='<bif> argument 1 ("<value>") must be less than',
                 'or equal to argument 2 ("<value>")'
#ErrorText.40.34='<bif> argument 1 ("<value>") must be less than',
                 'or equal to the number of lines',
                 'in the program (<sourceline()>)'
#ErrorText.40.35='<bif> argument 1 cannot be expressed as a whole number;',
                 'found "<value>"'
#ErrorText.40.36='<bif> argument 1',
                 'must be the name of a variable in the pool;',
                 'found "<value>"'
```

```
#ErrorText.40.37='<bif> argument 3',
                 'must be the name of a pool; found "<value>"'
#ErrorText.40.38='<bif> argument <argnumber>',
                 'is not large enough to format "<value>"'
#ErrorText.40.39='<bif> argument 3 is not zero or one; found "<value>"'
#ErrorText.40.41='<bif> argument <argnumber>',
                 'must be within the bounds of the stream;',
                 'found "<value>"'
#ErrorText.40.42='<bif> argument 1; cannot position on this stream;',
                  'found "<value>"'

#ErrorText.41  = 'Bad arithmetic conversion'
#ErrorText.41.1= 'Non-numeric value ("<value>")',
                 'to left of arithmetic operation "<operator>"'
#ErrorText.41.2= 'Non-numeric value ("<value>")',
                 'to right of arithmetic operation "<operator>"'
#ErrorText.41.3= 'Non-numeric value ("<value>")',
                 'used with prefix operator "<operator>"'
#ErrorText.41.4= 'Value of TO expression in DO instruction',
                 'must be numeric; found "<value>"'
#ErrorText.41.5= 'Value of BY expression in DO instruction',
                 'must be numeric; found "<value>"'
#ErrorText.41.6= 'Value of control variable expression of DO instruction',
                 'must be numeric; found "<value>"'
#ErrorText.41.7= 'Exponent exceeds' #Limit_ExponentDigits 'digits;',
                 'found "<value>"'

#ErrorText.42  = 'Arithmetic overflow/underflow'
#ErrorText.42.1= 'Arithmetic overflow detected at',
                 '"<value> <operation> <value>";',
                 'exponent of result requires more than',
                 #Limit_ExponentDigits 'digits'
#ErrorText.42.2= 'Arithmetic underflow detected at',
                 '"<value> <operation> <value>";',
                 'exponent of result requires more than',
                 #Limit_ExponentDigits 'digits'
#ErrorText.42.3= 'Arithmetic overflow; divisor must not be zero'

#ErrorText.43  = 'Routine not found'
#ErrorText.43.1= 'Could not find routine "<name>"'

#ErrorText.44  = 'Function did not return data'
#ErrorText.44.1= 'No data returned from function "<name>"'

#ErrorText.45  = 'No data specified on function RETURN'
#ErrorText.45.1= 'Data expected on RETURN instruction because',
                 'routine "<name>" was called as a function'

#ErrorText.46  = 'Invalid variable reference'
#ErrorText.46.1= 'Extra token ("<token>") found in variable',
                 'reference; ")" expected'

#ErrorText.47  = 'Unexpected label'
```

```
   #ErrorText.47.1= 'INTERPRET data must not contain labels;',
                    'found "<name>"'

   #ErrorText.48  = 'Failure in system service'
   #ErrorText.48.1= 'Failure in system service: <description>'

   #ErrorText.49  = 'Interpretation Error'
   #ErrorText.49.1= 'Interpretation Error: <description>'

   #ErrorText.50  = 'Unrecognized reserved symbol'
   #ErrorText.50.1= 'Unrecognized reserved symbol "<token>"'

   #ErrorText.51  = 'Invalid function name'
   #ErrorText.51.1= 'Unquoted function names must not end with a period;',
                    'found "<token>"'

   #ErrorText.52  = 'Result returned by "<name>" is longer than',
                    #Limit_String 'characters'

   #ErrorText.53  = 'Invalid option'
   #ErrorText.53.1= 'Variable reference expected',
                    'after STREAM keyword; found "<token>"'
   #ErrorText.53.2= 'Variable reference expected',
                    'after STEM keyword; found "<token>"'
   #ErrorText.53.3= 'Argument to STEM must have one period,',
                    'as its last character; found "<name>"'
   #ErrorText.54  = 'Invalid STEM value'
   #ErrorText.54.1= 'For this STEM APPEND, the value of "<name>" must be a',
                    'count of lines; found: "<value>"'
```

If the activity defined by section 6 does not produce any error message, execution of the program continues.

```
 call Config_NoSource
```

If Config_NoSource has set #NoSource to '0' the lines of source processed by section 6 are copied to #SourceLine. , with #SourceLine.0 being a count of the lines and #SourceLine.n for n=1 to #SourceLine.0 being the source lines in order.

If Config_NoSource has set #NoSource to '1' then #SourceLine.0 is set to 0.

The following state variables affect tracing:

```
 #InhibitPauses = 0
 #InhibitTrace = 0
 #AtPause = 0 /* Off until interactive input being received. */
 #Trace_QueryPrior = 'No'
```

An initial variable pool is established:

```
 #Pool = 1
 call Var_Empty #Pool
 call Var_Reset #Pool

 #Level = 1   /* Level of invocation */
 #IsFunction.#Level = (#HowInvoked == 'FUNCTION')
```

For this first level, there is no previous level from which values are inherited.  The relevant fields are initialized.

```
 #Digits.#Level = 9    /* Numeric Digits */
 #Form.#Level = ‘SCIENTIFIC’  /* Numeric Form */
 #Fuzz.#Level = 0  /* Numeric Fuzz */

 #StartTime.#Level = ‘’  /* Elapsed time boundary */
 #LineNumber = ''
 #Tracing.#Level = 'N'
 #Interactive.#Level = '0'
```

An environment is provided by the API_Start to become the initial active environment to which commands will be addressed. The alternate environment is made the same:

```
 /* Call the environments ACTIVE, ALTERNATE, TRANSIENT where these are
 never-initialized state variables.
 Similarly call the redirections I O and E */
 call EnvAssign ALTERNATE, #Level, ACTIVE, #Level
```

Conditions are initially disabled:

```
 #Enabling.SYNTAX.#Level = 'OFF'
 #Enabling.HALT.#Level = 'OFF'
 #Enabling.ERROR.#Level = 'OFF'
 #Enabling.FAILURE.#Level = 'OFF'
 #Enabling.NOTREADY.#Level = 'OFF'
 #Enabling.NOVALUE.#Level = 'OFF'
 #Enabling.LOSTDIGITS.#Level = 'OFF'
 #PendingNow.HALT.#Level = 0
 #PendingNow.ERROR.#Level = 0
 #PendingNow.FAILURE.#Level = 0
 #PendingNow.NOTREADY.#Level = 0
/* The following field corresponds to the results from the CONDITION built-in
function. */
 #Condition.#Level = ''
```

The opportunity is provided for a trap to initialize the pool.

```
 #API_Enabled = '1'
 call Var_Reset #Pool
 call Config_Initialization
 #API_Enabled = '0'
```

Execution of the program begins with its first clause.

### 8.2.2 Routine initialization

If the routine is invoked as a function, #IsFunction.#NewLevel shall be set to '1', otherwise to '0'; this affects the processing of a subsequent RETURN instruction.

```
#AllowProcedure.#NewLevel = '1'
```

Many of the initial values for a new invocation are inherited from the caller's values.

```
#Digits.#NewLevel = #Digits.#Level
#Form.#NewLevel = #Form.#Level
#Fuzz.#NewLevel = #Fuzz.#Level

#StartTime.#NewLevel = #StartTime.#Level

#Tracing.#NewLevel = #Tracing.#Level
#Interactive.#NewLevel = #Interactive.#Level

call EnvAssign ACTIVE, #NewLevel, ACTIVE, #Level
call EnvAssign ALTERNATE, #NewLevel, ALTERNATE, #Level

do t=1 to 7
  Condition = word('SYNTAX HALT ERROR FAILURE NOTREADY NOVALUE LOSTDIGITS',t)
  #Enabling.Condition.#NewLevel = #Enabling.Condition.#Level
  #Instruction.Condition.#NewLevel = #Instruction.Condition.#Level
  #TrapName.Condition.#NewLevel = #TrapName.Condition.#Level
  #EventLevel.Condition.#NewLevel = #EventLevel.Condition.#Level
  end t
```

If this invocation is not caused by a condition occurring, see section 8.4.1, the state variables for the CONDITION built-in function are copied.

```
  #Condition.#NewLevel = #Condition.#Level
  #ConditionDescription.#NewLevel = #ConditionDescription.#Level
  #ConditionExtra.#NewLevel = #ConditionExtra.#Level
  #ConditionInstruction.#NewLevel = #ConditionInstruction.#Level
```

Execution of the initialized routine continues at the new level of invocation.

```
#Level = #NewLevel
#NewLevel = #Level + 1
```

### 8.2.3 Clause initialization

The clause is traced before execution:

```
if pos(#Tracing.#Level, 'AIR') > 0  then call #TraceSource
```

The time of the first use of DATE or TIME will be retained throughout the clause.

```
#ClauseTime.#Level = ''
```

The state variable #LineNumber is set to the line number of the clause, see section 6.4.3.

A clause other than a null clause or label sets:

```
#AllowProcedure.#Level = '0'   /* See message 17.1 */
```

### 8.2.4 Clause termination

Polling for a HALT condition occurs:

```
#Response = Config_Halt_Query()
if #Outcome == 'Yes' then do
  call Config_Halt_Reset
  call #Raise 'HALT', substr(#Response,2)  /* May return */
  end
```

At the end of each clause there is a check for conditions which occurred and were delayed. It is acted on if this is the clause in which the condition arose.

```
do t=1 to 4
  #Condition=WORD('HALT FAILURE ERROR NOTREADY',t)
  /* HALT can be established during HALT handling. */
  do while #PendingNow.#Condition.#Level
     #PendingNow.#Condition.#Level = '0'
     call #Raise
     end
  end
```

Interactive tracing may be turned on via the configuration. Only a change in the setting is significant.

```
call Config_Trace_Query
if #AtPause = 0 & #Outcome == 'Yes' & #Trace_QueryPrior == 'No' then do
  /* External request for Trace '?R' */
  #Interactive.#Level = '1'
  #Tracing.#Level = 'R'
  end
#TraceQueryPrior = #Outcome
```

When tracing interactively, pauses occur after the execution of each clause except for CALL, DO the second or subsequent time around the loop, END, ELSE, EXIT, ITERATE, LEAVE, OTHERWISE, RETURN, SIGNAL, THEN and null clauses.

If the character '=' is entered in response to a pause, the prior clause is re-executed.

Anything else entered will be treated as a string of one or more clauses and executed by the language processor. The same rules apply to the contents of the string executed by interactive trace as do for strings executed by the INTERPRET instruction. If the execution of the string generates a syntax error, the standard message is displayed but no condition is raised. All condition traps are disabled during execution of the string. During execution of the string, no tracing takes place other than error or failure return codes from commands. The special variable RC is not set by commands executed within the string, nor is .RC.

If a TRACE instruction is executed within the string, the language processor immediately alters the trace setting according to the TRACE instruction encountered and leaves this pause point. If no TRACE instruction is executed within the string, the language processor simply pauses again at the same point in the program.

At a pause point:

```
if #AtPause = 0 & #Interactive.#Level & #InhibitTrace = 0 then do
  if #InhibitPauses > 0 then #InhibitPauses = #InhibitPauses-1
  else do
    #TraceInstruction = ‘0’
    do forever
      call Config_Trace_Query
      if #Outcome == 'No' & #Trace_QueryPrior == 'Yes' then do
        /* External request to stop tracing. */
        #Trace_QueryPrior=#Outcome
        #Interactive.#Level = '0'
        #Tracing.#Level = 'N'
        leave
        end
      if #Outcome == 'Yes' & #Trace_QueryPrior == 'No' then do
        /* External request for Trace '?R' */
        #Trace_QueryPrior = #Outcome
        #Interactive.#Level = '1'
        #Tracing.#Level = 'R'
        leave
        end
      if \#Interactive.#Level | #TraceInstruction then leave

      /* Accept input for immediate execution. */
      call Config_Trace_Input
      if length(#Outcome) = 0  |  #Outcome == '=' then leave
      #AtPause = #Level
      interpret #Outcome
      #AtPause = 0
      end /* forever loop */
    if #Outcome == '=' then call #Retry   /* With no return */
    end
  end
```

## 8.3 Instructions

### 8.3.1 ADDRESS

For a definition of the syntax of this instruction see section 6.3.2.21.

An external environment to which commands can be submitted is identified by an environment name. Environment names are specified in the ADDRESS instruction to identify the environment to which a command should be sent.

The concept of I/O redirection applies when submitting commands to an external environment. The submitted command's input stream can be taken from an existing stream, or from a set of compound variables with a given stem. In the latter case (that is, when a stem is specified as the source for the submitted commands input stream) whole number tails are used to select items from the compound variable collection and order them for presentation to the submitted command. Stem.0 must contain a whole number indicating the number of compound variables that should be presented, and stem.1 through stem.n (where n=stem.0) are the compound variables that will be presented to the submitted command.

Similarly, the submitted command's output stream can be directed to a stream, or to a set of compound variables with a given stem. In the latter case (i.e., when a stem is specified as the destination) compound variables will be created to hold the standard output, using whole number tails as described above. Output redirection can specify a REPLACE or APPEND option, which controls positioning prior to the command's execution. REPLACE is the default.

I/O redirection can be persistently associated with an environment name. The term “environment” is used to refer to an  environment name together with the I/O redirections.

At any given time, there will be two environments, the active environment and the alternate environment.

When an ADDRESS instruction specifies a command to the environment, any specified I/O redirection applies to that command's execution only, providing a third environment for the duration of the instruction.  When an ADDRESS command does not contain a command, that ADDRESS command creates a new active environment, which includes the specified I/O redirection.

The redirections specified on the ADDRESS instruction are requests.  The configuration may be aware that the command processor named does not perform I/O in a manner compatible with the request.  In that case the value of #Env_Type. may be set to 'UNUSED' as an alternative to 'STEM' and 'STREAM' where those values are assigned in the following code.

```
AddrInstr:
 /* If ADDRESS keyword alone, environments are swapped. */
 if \#Contains(address,taken_constant),
  & \#Contains(address,valueexp),
  & \#Contains(address,'WITH') then do
    call EnvAssign TRANSIENT, #Level, ACTIVE, #Level
    call EnvAssign ACTIVE, #Level, ALTERNATE, #Level
    call EnvAssign ALTERNATE, #Level, TRANSIENT, #Level
    return
    end
 /* The environment name will be explicitly specified. */
 if #Contains(address,taken_constant) then
   Name = #Instance(address, taken_constant)
 else
   Name = #Evaluate(valueexp, expression)
 if length(Name) > #LimitEnvironmentName then
   call #Raise 'SYNTAX', 29.1, Name

 if #Contains(address,expression) then do
   /* The command is evaluated (but not issued) at this point. */
   Command = #Evaluate(address,expression)
   if #Tracing.#Level == 'C' | #Tracing.#Level == 'A' then do
      call #Trace '>>>'
      end
   end
 call AddressSetup  /* Note what is specified on the address instruction. */
 /* If there is no command, the persistent environment is being set. */
 if \#Contains(address,expression) then do
    call EnvAssign ACTIVE, #Level, TRANSIENT, #Level
    return
    end

 call CommandIssue Command  /* See section 8.3.5 */

 return /* From AddrInstr */

AddressSetup:
 /* Note what is specified on the ADDRESS instruction,
 into the TRANSIENT environment. */
 EnvTail = 'TRANSIENT.'#Level
 /* Initialize with defaults. */
 #Env_Name.EnvTail = ''
 #Env_Type.I.EnvTail = 'NORMAL'
 #Env_Type.O.EnvTail = 'NORMAL'
 #Env_Type.E.EnvTail = 'NORMAL'
 #Env_Resource.I.EnvTail = ''
 #Env_Resource.O.EnvTail = ''
 #Env_Resource.E.EnvTail = ''
 /* APPEND / REPLACE does not apply to input. */
 #Env_Position.I.EnvTail = 'INPUT'
 #Env_Position.O.EnvTail = 'REPLACE'
 #Env_Position.E.EnvTail = 'REPLACE'

 /* If anything follows ADDRESS, it will include the command processor name.*/
 #Env_Name.EnvTail = Name

/* Connections may be explicitly specified. */
if #Contains(address,connection) then do
  if #Contains(connection,input) then do        /* input redirection */
    if #Contains(resourcei, 'STREAM') then do
      #Env_Type.I.EnvTail = 'STREAM'
      #Env_Resource.I.EnvTail=#Evaluate(resourcei, VAR_SYMBOL)
      end
    if #Contains(resourcei,'STEM') then do
      #Env_Type.I.EnvTail = 'STEM'
      Temp=#Instance(resourcei,VAR_SYMBOL)
      if \#Parses(Temp, stem /* See section 7.3.1 */) then
        call #Raise 'SYNTAX', 53.3, Temp
      #Env_Resource.I.EnvTail=Temp
      end
    end /* Input */

  if #Contains(connection,output) then /* output redirection */
    call NoteTarget O

  if #Contains(connection,error) then  /* error redirection */
    /* The prose on the description of #Contains specifies that the
    relevant resourceo is used in NoteTarget. */
    call NoteTarget E
  end /* Connection */

return /* from AddressSetup */

NoteTarget:
  /* Note the characteristics of an output resource. */
  arg Which  /* O or E */
  if #Contains(resourceo,'STREAM') then do
    #Env_Type.Which.EnvTail='STREAM'
    #Env_Resource.Which.EnvTail=#Evaluate(resourceo, VAR_SYMBOL)
    end
  if  #Contains(resourceo,'STEM') then do
    #Env_Type.Which.EnvTail='STEM'
    Temp=#Instance(resourceo,VAR_SYMBOL)
    if \#Parses(Temp, stem /* See section 7.3.1 */) then
      call #Raise 'SYNTAX', 53.3, Temp
    #Env_Resource.Which.EnvTail=Temp
    end
  if #Contains(resourceo,append) then
    #Env_Position.Which.EnvTail='APPEND'
 return /* From NoteTarget */

EnvAssign:
/* Copy the values that name an environment and describe its
redirections. */
 arg Lhs, LhsLevel, Rhs, RhsLevel
 #Env_Name.Lhs.LhsLevel = #Env_Name.Rhs.RhsLevel
 #Env_Type.I.Lhs.LhsLevel = #Env_Type.I.Rhs.RhsLevel
 #Env_Resource.I.Lhs.LhsLevel = #Env_Resource.I.Rhs.RhsLevel
 #Env_Position.I.Lhs.LhsLevel = #Env_Position.I.Rhs.RhsLevel
 #Env_Type.O.Lhs.LhsLevel = #Env_Type.O.Rhs.RhsLevel
 #Env_Resource.O.Lhs.LhsLevel = #Env_Resource.O.Rhs.RhsLevel
 #Env_Position.O.Lhs.LhsLevel = #Env_Position.O.Rhs.RhsLevel
 #Env_Type.E.Lhs.LhsLevel = #Env_Type.E.Rhs.RhsLevel
 #Env_Resource.E.Lhs.LhsLevel = #Env_Resource.E.Rhs.RhsLevel
 #Env_Position.E.Lhs.LhsLevel = #Env_Position.E.Rhs.RhsLevel
 return
```

### 8.3.2 ARG

For a definition of the syntax of this instruction see section 6.3.2.36.

The ARG instruction is a shorter form of the equivalent instruction:

**PARSE UPPER ARG *template_list***

### 8.3.3 Assignment

Assignment can occur as the result of executing a clause containing an *assignment* (see section 6.3.2.8 and section 6.3.2.47), or as a result of executing the VALUE built in function, or as part of the execution of a PARSE instruction.

Assignment involves an *expression* and a *VAR_SYMBOL*. The value of the expression is determined, see section 7.4.9.

If the *VAR_SYMBOL* does not contain a period, or contains only one period as its last character, the value is associated with the *VAR_SYMBOL*:

```
call Var_Set #Pool,VAR_SYMBOL,'0',Value
```

Otherwise, a name is derived, see section 7.3.1. The value is associated with the derived name:

```
call Var_Set #Pool,Derived_Name,'1',Value
```

### 8.3.4 CALL

For a definition of the syntax of this instruction see section 6.3.2.37.

The CALL instruction is used to invoke a routine, or is used to control the trapping of conditions.

If a *taken_constant* is specified, that name is used to invoke a routine, see section 7.5.1. If that routine does not return a result the RESULT and .RESULT variables become uninitialized. If the routine does return a result that value is assigned to RESULT and .RESULT. See section 8.4.1 for an exception to assigning results.

If the routine returns a result and the trace setting is 'R' then a trace with that result and a tag '>>>' shall be produced, associated with the call instruction.

If a *callon_spec* is specified:

```
If #Contains(call,callon_spec) then do
  Condition = #Instance(callon_spec,callable_condition)
  #Instruction.Condition.#Level = 'CALL'
  If #Contains(callon_spec,'OFF') then
    #Enabling.Condition.#Level = 'OFF'
  else
    #Enabling.Condition.#Level = 'ON'
  /* Note whether NAME supplied. */
  If Contains(callon_spec,taken_constant) then
    Name = #Instance(callable_condition,taken_constant)
  else
    Name = Condition
  #TrapName.Condition.#Level = Name
  end
```

### 8.3.5 Command to the configuration

For a definition of the syntax of a command see section 6.3.2.10.

A command that is not part of an ADDRESS instruction is processed in the ACTIVE environment.

```
Command =  #Evaluate(command, expression)
if #Tracing.#Level == 'C' | #Tracing.#Level == 'A' then do
   call #Trace '>>>'
   end
call EnvAssign TRANSIENT, #Level, ACTIVE, #Level
call CommandIssue Command
```

CommandIssue is also used to describe the ADDRESS instruction:

```
CommandIssue:
  parse arg Cmd
  /* Issues the command, requested environment is TRANSIENT */
  /* This description does not require the command processor to understand
  stems, so it uses an altered environment. */
  call EnvAssign PASSED, #Level, TRANSIENT, #Level
  EnvTail = 'TRANSIENT.'#Level

  /* Note the command input. */
  if #Env_Type.I.EnvTail = 'STEM' then do
      /* Check reasonableness of the stem. */
      Stem = #Env_Resource.I.EnvTail
      Lines = value(Stem'0')
      if \datatype(Lines,'W') then
        call #Raise 'SYNTAX',54.1,Stem'0', Lines
      if Lines<0 then
        call #Raise 'SYNTAX',54.1,Stem'0', Lines
      /* Use a stream for the stem */
      #Env_Type.I.PASSED.#Level = 'STREAM'
      call Config_Stream_Unique
      InputStream = #Outcome
      #Env_Resource.I.PASSED.#Level = InputStream
      call charout InputStream ,  ,1
      do j = 1 to Lines
        call lineout InputStream, Value(Stem || j)
        end j
      call lineout InputStream
      end

  /* Note the command output. */
  if #Env_Type.O.EnvTail = 'STEM' then do
     Stem = #Env_Resource.O.EnvTail
     if #Env_Position.O.EnvTail == 'APPEND' then do
       /* Check that Stem.0 will accept incrementing. */
       Lines=value(Stem'0');
       if \datatype(Lines,'W') then
         call #Raise 'SYNTAX',54.1,Stem'0', Lines
       if Lines<0 then
         call #Raise 'SYNTAX',54.1,Stem'0', Lines
       end
     else call value Stem'0',0
     /* Use a stream for the stem */
     #Env_Type.O.PASSED.#Level = 'STREAM'
     call Config_Stream_Unique
     #Env_Resource.O.PASSED.#Level = #Outcome
     end

    /* Note the command error stream. */
    if #Env_Type.E.EnvTail = 'STEM' then do
       Stem = #Env_Resource.E.EnvTail
       if #Env_Position.E.EnvTail == 'APPEND' then do
         /* Check that Stem.0 will accept incrementing. */
         Lines=value(Stem'0');
         if \datatype(Lines,'W') then
           call #Raise 'SYNTAX',54.1,Stem'0', Lines
         if Lines<0 then
           call #Raise 'SYNTAX',54.1,Stem'0', Lines
         end
      else call value Stem'0',0
      /* Use a stream for the stem */
      #Env_Type.E.PASSED.#Level = 'STREAM'
      call Config_Stream_Unique
      #Env_Resource.E.PASSED.#Level = #Outcome
      end

  #API_Enabled  =  '1'
   call Var_Reset #Pool
   /* Specifying PASSED here implies all the
      components of that environment. */
  #Response = Config_Command(PASSED, Cmd)
  #Indicator = left(#Response,1)
  Description = substr(#Response,2)
  #API_Enabled = '0'
  /* Recognize success and failure. */
  if #AtPause = 0 then do
    call value 'RC', #RC
    call Var_Set 0,  '.RC', 0, #RC
    end
  select
    when #Indicator=='N' then Temp=0
    when #Indicator=='F' then Temp=-1  /* Failure */
    when #Indicator=='E' then Temp=1   /* Error */
    end
  call Var_Set 0, '.RS', 0, Temp
  /* Process the output */
  if #Env_Type.O.EnvTail='STEM' then do   /* get output into stem. */
    Stem = #Env_Resource.O.EnvTail
    OutputStream = #Env_Resource.O.PASSED.#Level
    do while lines(OutputStream)
      call value Stem'0',value(Stem'0')+1
```

```
      call value Stem||value(Stem'0'),linein(OutputStream)
      end
    end /* Stemmed Output */
  if #Env_Type.E.EnvTail='STEM' then do   /* get error output into stem. */
    Stem = #Env_Resource.E.EnvTail
    OutputStream = #Env_Resource.E.PASSED.#Level
    do while lines(OutputStream)
      call value Stem'0',value(Stem'0')+1
      call value Stem||value(Stem'0'),linein(OutputStream)
      end
    end /* Stemmed Error output */
  if #Indicator \== 'N' & #Tracing.#Level=='C' then
     call #Trace '+++'
  if (#Indicator \== 'N' & #Tracing.#Level=='E'),
   | (#Indicator=='F' & (#Tracing.#Level=='F' | #Tracing.#Level=='N')) then do
     call #Trace '>>>'
     call #Trace '+++'
     end
 if #Indicator='F' then call #Raise 'FAILURE' , Cmd
 if #Indicator='E' then call #Raise 'ERROR', Cmd
 return /* From CommandIssue */
```

The configuration may choose to perform the test for message 54.1 before or after issuing the command.

### 8.3.6 DO

For a definition of the syntax of this instruction see section 6.3.2.12.

The DO instructions is used to group instructions together and optionally to execute them repeatedly.

Executing a *do_simple* has the same effect as executing a *nop*, except in its trace output. Executing the *do_ending* associated with a *do_simple* has the same effect as executing a *nop*, except in its trace output.

A *do_instruction* that does not contain a *do_simple* is equivalent, except for trace output, to a sequence of instructions in the following order.

```
#Loop = #Loop+1
#Iterate.#Loop = #Clause(IterateLabel)
#Once.#Loop = #Clause(OnceLabel)
#Leave.#Loop = #Clause(LeaveLabel)
if #Contains(do_specification,assignment) then
    #Identity.#Loop = #Instance(assignment, VAR_SYMBOL)
if #Contains(do_specification, repexpr) then
   if \datatype(repexpr,'W') then
       call #Raise ‘SYNTAX’, 26.2,repexpr
   else do
       #Repeat.#Loop = repexpr+0
       if #Repeat.#Loop<0 then
          #Raise ‘SYNTAX’,26.2,#Repeat.#Loop
       end
if #Contains(do_specification,assignment) then do
   #StartValue.#Loop = #Evaluate(assignment,expression)
   if datatype(#StartValue.#Loop) \== 'NUM' then
```

```
        #Raise 'SYNTAX', 41.6, #StartValue.#Loop
   if \#Contains(do_specification,byexpr) then
        #By.#Loop = 1
   end
```

The following three assignments are made in the order in which ‘to’, ‘by’ and ‘for’ appear in docount.

```
if #Contains(do_specification, toexpr) then do
   if datatype(toexpr) \== 'NUM' then
      call #Raise 'SYNTAX', 41.4, toexpr
   #To.#Loop = toexpr+0
if #Contains(do_specification, byexpr) then do
   if datatype(byexpr)\==’NUM’ then
      call #Raise 'SYNTAX', 41.5, byexpr
   #By.#Loop = byexpr+0
if #Contains(do_specification, forexpr) then do
   if \datatype(forexpr, 'W') then
      call #Raise 'SYNTAX', 26.3, forexpr
   #For.#Loop = forexpr+0
   if #For.#Loop <0 then
      call #Raise 'SYNTAX', 26.3, #For.#Loop
   end
if #Contains(do_specification,assignment) then do
   call value #Identity.#Loop, #StartValue.#Loop
   end
   call #Goto  #Once.#Loop  /* to OnceLabel */
IterateLabel:
if #Contains(do_specification, untilexpr) then do
  Value = #Evaluate(untilexp, expression)
  if Value == '1'  then leave
  if Value \== '0' then call #Raise 'SYNTAX', 34.4, Value
  end
if #Contains(do_specification, byexpr) then do
   t = value(#Identity.#Loop)
   if #Indicator == 'D' then #Raise 'NOVALUE', #Identity.#Loop
   call value #Identity.#Loop, t + #By.#Loop
   end
OnceLabel:
if #Contains(do_specification, toexpr) then do
   if #By.#Loop>=0 then do
     if value(#Identity.#Loop) > #To.#Loop then leave
     end
     else nop
   else if value(#Identity.#Loop) < #To.#Loop then
               leave
  end
if #Contains(dorep, repexpr) then
   if #Repeat.#Loop = 0 then leave
#Repeat.#Loop = #Repeat.#Loop-1
if #Contains(do_specification, forexpr) then
   if #For.#Loop = 0 then leave
#For.#Loop = #For.#Loop - 1
if #Contains(do_specification, whileexpr) then do
  Value = #Evaluate(whileexp, expression)
  if Value == '0'  then leave
  if Value \== '1' then call #Raise 'SYNTAX', 34.3, Value
  end
  #Execute(do_instruction, instruction_list)
TraceOfEnd:
call #Goto #Iterate.#Loop  /* to IterateLabel */
LeaveLabel:
#Loop = #Loop - 1
```

#### 8.3.6.1 DO loop tracing

When clauses are being traced by #TraceSource, because **pos(#Tracing.#Level, 'AIR') > 0**, the DO instruction shall be traced when it is encountered and again each time the IterateLabel (see section 8.3.6) is encountered. The END instruction shall be traced when the TraceOfEnd label is encountered.

When expressions or intermediates are being traced they shall be traced in the order specified by section 8.3.6. Hence, in the absence of conditions arising, those executed prior to the first execution of OnceLabel shall be shown once per execution of the DO instruction, others shall be shown depending on the outcome of the tests.

The code in the DO description:

```
   t = value(#Identity.#Loop)
   if #Indicator == 'D' then #Raise 'NOVALUE', #Identity.#Loop
   call value #Identity.#Loop, t + #By.#Loop
```

represents updating the control variable of the loop. That assignment is subject to tracing, and other expressions involving state variables are not. When tracing intermediates, the BY value will have a tag of '>+>'.

### 8.3.7 DROP

For a definition of the syntax of this instruction see section 6.3.2.59.

The DROP instruction restores variables to an uninitialized state.

The words of the *variable_list* are processed from left to right.

A word which is a VAR_SYMBOL, not contained in parentheses, specifies a variable to be dropped. If VAR_SYMBOL does not contain a period, or has only a single period as its last character, the variable associated with VAR_SYMBOL by the variable pool is dropped:

```
#Response = Var_Drop(Level,VAR_SYMBOL,'0')
```

If VAR_SYMBOL has a period other than as last character, the variable associated with VAR_SYMBOL by the variable pool is dropped by:

```
#Response = Var_Drop (Level,VAR_SYMBOL,'1')
```

If the word of the *variable_list* is a VAR_SYMBOL enclosed in parentheses then the value of the VAR_SYMBOL is processed. The value is considered in uppercase:

```
#Value = Config_Upper(VAR_SYMBOL)
```

Each word in that value found by the WORD built-in function, from left to right, is subjected to this process:

If the word does not have the syntax of VAR_SYMBOL a condition is raised:

```
call #Raise 'SYNTAX', 20.1, word
```

Otherwise the VAR_SYMBOL indicated by the word is dropped, as if that VAR_SYMBOL was a word of the *variable_list*.

### 8.3.8 EXIT

For a definition of the syntax of this instruction see section 6.3.2.61.

The EXIT instruction is used to unconditionally complete execution of a program.

Any expression is evaluated:

```
if #Contains(exit, expression) then Value = #Evaluate(exit, expression)
#Level = 1
#Pool = 1
```

The opportunity is provided for a final trap.

```
 #API_Enabled = '1'
 call Var_Reset #Pool
 call  Config_Termination
 #API_Enabled = '0'
```

The processing of the program is complete. See section 5.2.1 for what API_Start returns as the result.

If the normal sequence of execution "falls through" the end of the program, that is, would execute a further statement if one were appended to the program, then the program is terminated in the same manner as an EXIT instruction with no argument.

### 8.3.9 IF

For a definition of the syntax of this instruction see section 6.3.2.14.

The IF instruction is used to conditionally execute an instruction, or to select between two alternatives.

The *expression* is evaluated. If the value is neither '0' nor '1' error 34.1 occurs. If the value is '1', the *instruction i* in the *then* is executed. If the value is '0' and *'else'* is specified, the *instruction* in the *else* is executed.

In the former case, if tracing clauses, the clause consisting of the THEN keyword shall be traced in addition to the instructions.

In the latter case, if tracing clauses, the clause consisting of the ELSE keyword shall be traced in addition the instructions.

### 8.3.10 INTERPRET

For a definition of the syntax of this instruction see section 6.3.2.62.

The INTERPRET instruction is used to execute instructions that have been built dynamically by evaluating an expression.

The *expression* is evaluated.

The HALT condition is tested for, and may be raised, in the same way it is tested at clause termination, see section 8.2.4.

The process of syntactic recognition described in section 6 is applied, with Config_SourceChar obtaining its results from the characters of the value, in left to right order, without producing any EOL or EOS events. When the characters are exhausted the event EOL occurs, followed by the event EOS.

If that recognition would produce any message then the *interpret* raises the corresponding 'SYNTAX' condition.

If the *program* recognized contains any LABELs then the *interprett*raises a condition:

```
call #Raise 'SYNTAX',47.1,Label
```

where Label is the first LABEL in the *program*.

Otherwise the *instruction_list* in the *program* is executed.

### 8.3.11 ITERATE

For a definition of the syntax of this instruction see section 6.3.2.63.

The ITERATE instruction is used to alter the flow of control within a repetitive DO.

For a definition of the nesting correction see section 6.4.1.

```
#Loop = #Loop - NestingCorrection
call #Goto #Iterate.#Loop
```

### 8.3.12 Execution of labels

The execution of a label has no effect, other than clause termination activity and any tracing.

```
if #Tracing.#Level=='L' then call #TraceSource
```

### 8.3.13 LEAVE

For a definition of the syntax of this instruction see section 6.3.2.64.

The LEAVE instruction is used to immediately exit one or more repetitive DOs.

For a definition of the nesting correction see section 6.4.1.

```
#Loop = #Loop - NestingCorrection
call #Goto #Leave.#Loop
```

### 8.3.14 NOP

For a definition of the syntax of this instruction see section 6.3.2.65.

The NOP instruction has no effect other than the effects associated with all instructions.

### 8.3.15 NUMERIC

For a definition of the syntax of this instruction see section 6.3.2.66.

The NUMERIC instruction is used to change the way in which arithmetic operations are carried out.

#### 8.3.15.1 NUMERIC DIGITS

For a definition of the syntax of this instruction see section 6.3.2.67.

NUMERIC DIGITS controls the precision under which arithmetic operations and arithmetic built-in functions will be evaluated.

```
if #Contains(numericdigits, expression) then
   Value = #Evaluate(numericdigits, expression)
else Value = 9
if \datatype(Value,'W') then
    #Raise 'SYNTAX',26.5,Value
Value = Value + 0
if Value<=#Fuzz.#Level then
    #Raise 'SYNTAX',33.1,Value
if Value>#Limit_Digits then
```

```
    #Raise 'SYNTAX',33.2,Value
#Digits.#Level = Value
```

#### 8.3.15.2 NUMERIC FORM

For a definition of the syntax of this instruction see section 6.3.2.68.

NUMERIC FORM controls which form of exponential notation is to be used for the results of operations and arithmetic built-in functions.

The value of form is either taken directly from the SCIENTIFIC or ENGINEERING keywords, or by evaluating *valueexp* .

```
if \#Contains(numeric,numericsuffix) then
  Value = 'SCIENTIFIC'
else if #Contains(numericformsuffix,'SCIENTIFIC') then
           Value = 'SCIENTIFIC'
        else
           if #Contains(numericformsuffix,'ENGINEERING') then
             Value = 'ENGINEERING'
           else do
             Value = #Evaluate(numericformsuffix,valueexp)
             Value = translate(left(Value,1))
             select
                when Value == 'S' then Value = 'SCIENTIFIC'
                when Value == 'E' then Value = 'ENGINEERING'
                otherwise call #Raise 'SYNTAX',33.3,Value
                end
            end
#Form.#Level  =  Value
```

#### 8.3.15.3 NUMERIC FUZZ

For a definition of the syntax of this instruction see section 6.3.2.69.

NUMERIC FUZZ controls how many digits, at full precision, will be ignored during a numeric comparison.

```
If #Contains(numericfuzz,expression) then
  Value = #Evaluate(numericfuzz,expression)
else
  Value = 0
If \datatype(Value,'W') then
  call #Raise 'SYNTAX',26.6,Value
Value  =  Value+0
If Value < 0 then
  call #Raise 'SYNTAX',26.6,Value
If Value >= #Digits.#Level then
  call #Raise 'SYNTAX',33.1,#Digits.#Level,Value
#Fuzz.#Level = Value
```

### 8.3.16 OPTIONS

For a definition of the syntax of this instruction see section 6.3.2.70.

The OPTIONS instruction is used to pass special requests to the language processor.

The *expression* is evaluated and the value is passed to the language processor. The language processor treats the value as a series of blank delimited words. Any words in the value that are not recognized by the language processor are ignored without producing an error.

## 8.3.17 PARSE

For a definition of the syntax of this instruction see section 6.3.2.71.

The PARSE instruction is used to assign data from various sources to variables.

The purpose of the PARSE instruction is to select substrings of the *parse_type* under control of the *template_list*. If the *template_list* is omitted, or a *template* in the list is omitted, then a template which is the null string is implied.

Processing for the PARSE instruction begins by constructing a value, the source to be parsed.

```
 ArgNum = 0
 select
   when #Contains(parse_type, 'ARG') then do
         ArgNum = 1
         ToParse = #Arg.#Level.ArgNum
         end
   when #Contains(parse_type, 'LINEIN') then ToParse = linein('')
   when #Contains(parse_type, 'PULL') then do
      /* Acquire from external queue or default input. */
      #Response = Config_Pull()
      if left(#Response, 1) == 'F' then
        call Config_Default_Input
        ToParse = #Outcome
        end
   when #Contains(parse_type, 'SOURCE') then
      ToParse = #Configuration #HowInvoked #Source
    when #Contains(parse_type, 'VALUE') then
      if \#Contains(parse_value, expression) then ToParse = ''
      else ToParse = #Evaluate(parse_value, expression)
   when #Contains(parse_type, 'VAR') then
     ToParse = #Evaluate(parse_var,VAR_SYMBOL)
   when #Contains(parse_type, 'VERSION') then ToParse = #Version
   end
 Uppering =  #Contains(parse, 'UPPER')
```

The first template is associated with this source. If there are further templates, they are matched against null strings unless 'ARG' is specified, when they are matched against further arguments.

The parsing process is defined by the following routine, ParseData. The template_list is accessed by ParseData as a stemmed variable. This variable Template. has elements which are null strings except for any elements with tails 1,2,3,... corresponding to the tokens of the template_list from left to right.

```
ParseData:
  /* Targets will be flagged as the template is examined. */
  Target.='0'
  /* Token is a cursor on the components of the template,
  moved by FindNextBreak. */
  Token = 1
  /* Tok is a cursor on the components of the template
  moved through the target variables by routine WordParse. */
  Tok = 1
do forever  /* Until commas dealt with. */
  /* BreakStart and BreakEnd indicate the position in the source
  string where there is a break that divides the source.  When the break
  is a pattern they are the start of the pattern and the position just
  beyond it. */
  BreakStart = 1
  BreakEnd = 1
  SourceEnd = length(ToParse) + 1
  If Uppering then ToParse = translate(ToParse)

  do while Template.Tok \== '' & Template.Tok \== ','

    /* Isolate the data to be processed on this iteration. */
    call FindNextBreak  /* Also marks targets. */

    /* Results have been set in DataStart which indicates the start
    of the isolated data and BreakStart and BreakEnd which are ready
    for the next iteration. Tok has not changed. */

    /* If a positional takes the break leftwards from the end of the
    previous selection, the source selected is the rest of the string, */

    if BreakEnd <= DataStart then
      DataEnd = SourceEnd
    else
      DataEnd = BreakStart

    /* Isolated data, to be assigned from: */
    Data=substr(ToParse,DataStart,DataEnd-DataStart)
    call WordParse  /* Does the assignments. */

    end /* while */
  if Template.Tok \== ',' then leave
  /* Continue with next source. */
  Token=Token+1
  Tok=Token
  if ArgNum <> 0 then do
     ArgNum = ArgNum+1
     ToParse = #Arg.ArgNum
     end
  else ToParse=''
  end

return  /* from ParseData */

FindNextBreak:
  do while Template.Token \== '' & Template.Token \== ','
    Type=left(Template.Token,1)
    /* The source data to be processed next will normally start at the end of
```

```
the break that ended the previous piece. (However, the relative
positionals alter  this.) */
DataStart = BreakEnd
select

  when Type='"' | Type="'" | Type='(' then do
    if Type='(' then do
      /* A parenthesis introduces a pattern which is not a constant. */
      Token = Token+1
      Pattern = value(Template.Token)
      if #Indicator == 'D' then call #Raise 'NOVALUE', Template.Token
      Token = Token+1
      end
    else
      /* The following removes the outer quotes from the
      literal pattern */
      interpret "Pattern="Template.Token
    Token = Token+1
    /* Is that pattern in the remaining source? */
    PatternPos=pos(Pattern,ToParse,DataStart)
    if PatternPos>0 then do
      /* Selected source runs up to the pattern. */
      BreakStart=PatternPos
      BreakEnd=PatternPos+length(Pattern)
      return
      end
    leave /* The rest of the source is selected. */
    end

  when datatype(Template.Token,'W') | pos(Type,'+-=') > 0 then do
    /* A positional specifies where the relevant piece of the subject
    ends. */
    if pos(Type,'+-=') = 0 then do
      /* Whole number positional */
      BreakStart = Template.Token
      Token = Token+1
      end
    else do
      /* Other forms of positional. */
      Direction=Template.Token
      Token = Token + 1
      /* For a relative positional, the position is relative to the start
      of the previous trigger, and the source segment starts there. */
      if Direction \== '=' then
         DataStart = BreakStart
      /* The adjustment can be given as a number or a variable in
      parentheses. */
      if Template.Token ='(' then do
         Token=Token + 1
         BreakStart = value(Template.Token)
         if #Indicator == 'D' then call #Raise 'NOVALUE', Template.Token
         Token=Token + 1
         end
```

```
          else BreakStart = Template.Token
          if \datatype(BreakStart,'W')
               then call #Raise 'SYNTAX', 26.4,BreakStart
          Token = Token+1
          if Direction='+'
            then BreakStart=DataStart+BreakStart
          else if Direction='-'
            then BreakStart=DataStart-BreakStart
          end
        /* Adjustment should remain within the ToParse */
        BreakStart = max(1, BreakStart)
        BreakStart = min(SourceEnd, BreakStart)
        BreakEnd = BreakStart  /* No actual literal marks the boundary. */
        return
        end

      when Template.Token \== '.' & pos(Type,'0123456789.')>0 then
        /* A number that isn't a whole number. */
        call #Raise 'SYNTAX', 26.4, Template.Token
        /* Raise will not return */

      otherwise do /* It is a target, not a pattern */
        Target.Token='1'
        Token = Token+1
        end

      end /* select */
    end /* while */
    /* When no more explicit breaks, break is at the end of the source. */
    DataStart=BreakEnd
    BreakStart=SourceEnd
    BreakEnd=SourceEnd
    return  /* From FindNextBreak */

WordParse:
/* The names in the template are assigned blank-delimited values from the
source string. */

  do while Target.Tok /* Until no more targets for this data. */
    /* Last target gets all the residue of the Data. */
    NextTok = Tok + 1
    if \Target.NextTok then do
      call Assign(Data)
      leave
      end
    /* Not last target; assign a word. */
    Data = strip(Data,'L')
    if Data == '' then call Assign('')
    else do
      Word=word(Data,1)
      call Assign Word
      Data = substr(Data,length(Word) + 1)
      /* The word terminator is not part of the residual data: */
      if Data \== '' then Data=substr(Data,2)
      end
   Tok = Tok + 1
   end
   Tok=Token /* Next time start on new part of template. */
   return

Assign:
   if Template.Tok=='.' then Tag='>.>'
   else do
     Tag='>=>'
     call value Template.Tok,arg(1)
     end
   /* Arg(1) is an implied argument of the tracing. */
   if #Tracing.#Level = 'R' then call #Trace Tag
   return
```

### 8.3.18 PROCEDURE

For a definition of the syntax of this instruction see section 6.3.2.76.

The PROCEDURE instruction is used within an internal routine to protect all the existing variables by making them unknown to following instructions. Selected variables may be exposed.

It is used at the start of a routine, after routine initialization:

```
if \#AllowProcedure then call #Raise 'SYNTAX', 17.1
/* It introduces a new variable pool: */
#Pool=#Pool+1
IsProcedure.#Level='1'
call Var_Empty #Pool
```

If there is a *variable_list*, it provides access to a previous variable pool.

The words of the *variable_list* are processed from left to right.

A word which is a VAR_SYMBOL, not contained in parentheses, specifies a variable to be made accessible. If VAR_SYMBOL does not contain a period, or has only a single period as its last character, the variable associated with VAR_SYMBOL by the variable pool (as a non-tailed name) is given the attribute 'exposed'.

```
call Var_Expose #Pool, VAR_SYMBOL, '0'
```

If VAR_SYMBOL has a period other than as last character, the variable associated with VAR_SYMBOL in the variable pool ( by the name derived from VAR_SYMBOL, see section 7.3.1) is given the attribute 'exposed'.

```
call Var_Expose #Pool, Derived_Name, '1'
```

If the word of the *variable_list* is a VAR_SYMBOL enclosed in parentheses then the VAR_SYMBOL is exposed, as if that VAR_SYMBOL was a word of the *variable_list*. The value of the VAR_SYMBOL is processed. The value is considered in uppercase:

```
Value = Config_Upper(VAR_SYMBOL)
```

Each word in that value found by the WORD built-in function, from left to right, is subjected to this process:

If the word does not have the syntax of VAR_SYMBOL a condition is raised:

```
call #Raise 'SYNTAX', 20.1, word
```

Otherwise the VAR_SYMBOL indicated by the word is exposed, as if that VAR_SYMBOL was a word of the *variable_list*.

### 8.3.19 PULL

For a definition of the syntax of this instruction see section 6.3.2.77.

A PULL instruction is a shorter form of the equivalent instruction:

**PARSE UPPER PULL *template_list***

### 8.3.20 PUSH

For a definition of the syntax of this instruction see section 6.3.2.78.

The PUSH instruction is used to place a value on top of the stack.

```
If #Contains(push,expression) then
  Value = #Evaluate(push,expression)
else
  Value = ''
call Config_Push Value
```

### 8.3.21 QUEUE

For a definition of the syntax of this instruction see section 6.3.2.79.

The QUEUE instruction is used to place a value on the bottom of the stack.

```
If #Contains(queue,expression) then
  Value = #Evaluate(queue,expression)
else
  Value = ''
call Config_Queue Value
```

### 8.3.22 RETURN

For a definition of the syntax of this instruction see section 6.3.2.80.

The RETURN instruction is used to return control and possibly a result from a program or internal routine to the point of its invocation.

The RETURN keyword may be followed by an optional expression, which will be evaluated and returned as a result to the caller of the routine.

Any expression is evaluated:

```
if #Contains(return,expression) then
   #Outcome = #Evaluate(return, expression)
else #IsFunction.#Level  then
   call #Raise 'SYNTAX', 45.1, #Name.#Level
```

If the routine started with a PROCEDURE instruction then the associated pool is taken out of use:

```
if #IsProcedure.#Level then #Pool = #Pool-1
```

A RETURN instruction which is interactively entered at a pause point leaves the pause point.

```
if #Level = #AtPause then #AtPause = 0
```

The activity at this level is complete:

```
    #Level = #Level-1
    #NewLevel = #Level+1
```

If #Level is not zero the processing of the RETURN instruction and the invocation is complete. Otherwise processing of the program is completed:

The opportunity is provided for a final trap.

```
 #API_Enabled = '1'
 call Var_Reset #Pool
 call Config_Termination
 #API_Enabled = '0'
```

The processing of the program is complete. See section 5.2.1 for what API_Start returns as the result.

### 8.3.23 SAY

For a definition of the syntax of this instruction see section 6.3.2.81.

The SAY instruction is used to write a line to the default output stream.

```
If #Contains(say,expression) then
  Value = Evaluate(say,expression)
else
  Value = ''
call Config_Default_Output Value
```

### 8.3.24 SELECT

For a definition of the syntax of this instruction see section 6.3.2.17.

The SELECT instruction is used to conditionally execute one of several alternative instructions.

When tracing, the clause containing the keyword SELECT is traced at this point.

The #Contains(*selectbody*, *when*) test in the following description refers to the items of the *whenlist* in order:

```
LineNum = #LineNumber
Ending = #Clause(EndLabel)
Value=#Evaluate(requiredwhen,expression)
if Value \== '1' & Value \== '0' then
  call #Raise 'SYNTAX',34.2,Value
If Value=='1' then
  call #Execute requiredwhen,instructionx14
else do
  do while #Contains(selectbody, when)
     Value = #Evaluate(when,expression)
     If Value=='1' then do
        call #Execute when,instructionx14
        call #Goto Ending
        end
     if Value \== '0' then
        call #Raise 'SYNTAX',34.2,Value
     end  /* Of each when */
  If \#Contains(selectbody,'OTHERWISE') then
     call #Raise 'SYNTAX',7.3,LineNum
  If #Contains(selectbody,instruction_list) then
     call #Execute selectbody,instruction_list
 end
EndLabel:
```

When tracing, the clause containing the END keyword is traced at this point.

### 8.3.25 SIGNAL

For a definition of the syntax of this instruction see section 6.3.2.82.

The SIGNAL instruction is used to cause a change in the flow of control or is used with the ON and OFF keywords to control the trapping of conditions.

```
If #Contains(signal,signal_spec) then do
  Condition = #Instance(signal_spec,condition)
  #Instruction.Condition.#Level = 'SIGNAL'
  If #Contains(signal_spec,'OFF') then
    #Enabling.Condition.#Level = 'OFF'
  else
    #Enabling.Condition.#Level = 'ON'
  If Contains(signal_spec,taken_constant) then
    Name = #Instance(condition,taken_constant)
  else
    Name = Condition
  #TrapName.Condition.#Level = Name
  end
```

If there was a *signal_spec* this complete the processing of the signal instruction. Otherwise:

```
 if #Contains(signal,valueexp)
          then Name = #Evaluate(valueexp, expression)
          else Name = #Instance(signal,taken_constant)
```

The Name matches the first LABEL in the program which has that value. The comparison is made with the '==' operator.

If no label matches then a condition is raised:

```
call #Raise 'SYNTAX',16.1, Name
```

If the name is not a label of the program that can be signalled then a condition is raised:

```
call #Raise 'SYNTAX', 16.2, Name
```

If the name matches a label, execution continues at that label after these settings:

```
#Loop.#Level = 0
/* A SIGNAL interactively entered leaves the pause point. */
if #Level = #AtPause then #AtPause = 0
```

### 8.3.26 TRACE

For a definition of the syntax of this instruction see section 6.3.2.85.

The TRACE instruction is used to control the trace setting which in turn controls the tracing of execution of the program.

The TRACE instruction is ignored if it occurs within the program (as opposed to source obtained by Config_Trace_Input) and interactive trace is requested (#Interactive.#Level = '1'). Otherwise:

```
#TraceInstruction = '1'
```

```
value = ''
if #Contains(trace, valueexp) then Value = #Evaluate(valueexp, expression)
if #Contains(trace, taken_constant) then Value =
#Instance(trace,taken_constant)
if datatype(Value) == 'NUM' & \datatype(Value,'W') then
   call #Raise 'SYNTAX', 26.7, Value
if datatype(Value,'W') then do
   /* Numbers are used for skipping. */
   if Value>=0 then #InhibitPauses = Value
               else #InhibitTrace = -Value
   end
else do
    if length(Value) = 0 then #Interacting.#Level = '0'
    /* Each question mark toggles the interacting. */
    else do while left(Value,1)=='?'
               #Interacting.#Level = \#Interacting.#Level
               Value = substr(Value,2)
               end
    /* The default setting is Normal */
    if length(Value) = 0 then Value = 'N'
    else do
      Value = translate( left(Value,1) )
      if verify(Value, 'ACEFILNOR') > 0 then
        call #Raise 'SYNTAX', 24.1, Value
      if Value=='O' then #Interacting.#Level='0'
      end
    #Tracing.#Level = Value
    end
```

### 8.3.26.1 Trace output

The routines #TraceSource and #Trace specify the output that results from the trace settings. That output is presented to the configuration by Config_Trace_Output as lines. Each line has a clause identifier at the left, followed by a blank, followed by a three character tag, followed by a blank, followed by the trace data.

The width of the clause identifier shall be large enough to hold the line number of the last line in the program, and no larger. The clause identifier is the source program line number, or all blank if the line number is the same as the previous line number indicated and no execution with trace Off has occurred since. The line number is right-aligned with leading zeros replaced by blank characters.

When input at a pause is being executed (#AtPause \= 0 ), #Trace does nothing when the tag is not '+++'.

When input at a pause is being executed, #TraceSource does nothing.

If #InhibitTrace is greater than zero, #TraceSource does nothing except decrement #InhibitTrace. Otherwise #TraceSource outputs all lines of the source program which contain any part of the current clause, with any characters in those lines which are not part of the current clause and not other_blank_characters replaced by blank characters. The possible replacement of other_blank_characters is defined by the configuration. The tag is '*-*', or if the line is not the first line of the clause. '*,*'.

#Trace output also has a clause identifier and has a tag which is the argument to the #Trace invocation. The data is truncated, if necessary, to #Limit_TraceData characters.The data is enclosed by quotation marks and the quoted data preceded by two blanks. If the data is truncated, the trailing quote has the three characters '...' appended. The data is dependent on the #Tracing.#Level:

— When this is 'C' or 'E' or 'F' and the tag is '>>>' then the data is the value of the command passed to the environment.

— When this is 'C' or 'E' or 'F' and the tag is '+++' then the data is the four characters 'RC "' concatenated with #RC concatenated with the character '"'.

— When this is 'I' or 'R' the data is the most recent evaluated value.

Trace output can also appear as the result of a 'SYNTAX' condition occurring, irrespective of the trace setting. If a 'SYNTAX' condition occurs and it is not trapped by SIGNAL ON SYNTAX, then the clause in error shall be traced, along with a traceback. A traceback is a display of each active CALL and INTERPRET instruction, and function invocation, displayed in reverse order of execution.

## 8.4 Conditions and Messages

When an error occurs during execution of a program, an error number and message are associated with it. The error number has two parts, the error code and the error subcode. These are the integer and decimal parts of the error number. Subcodes beginning or ending in zero are not used.

Error codes in the range 1 to 90 and error subcodes up to .9 are reserved for errors described here and for future extensions of this standard.

Error number 3 is available to report error conditions occuring during the initialization phase, error number 2 is available to report error conditions during the termination phase. These are error conditions recognized by the language processor, but the circumstances of their detection is outside of the scope of this standard.

The ERRORTEXT built-in function returns the text as initialized in section 8.2.1 when called with the 'Standard' option. When the 'Standard' option is omitted, implementation-dependent text may be returned.

When messages are issued any message inserts are replaced by actual values.

The notation for detection of a condition is:

```
call #Raise Condition, Arg2, Arg3, Arg4, Arg5, Arg6
```

Some of the arguments may be omitted. In the case of condition 'SYNTAX' the arguments are the message number and the inserts for the message. In other cases the argument is a further description of the condition.

The action of the program as a result of a condition is dependent on any *signal_spec* and *callon_spec* in the program.

### 8.4.1 Raising of conditions

The routine #Raise corresponds to raising a condition. In the following definition, the instructions containing SIGNAL VALUE and INTERPRET denote transfers of control in the program being processed. The instruction EXIT denotes termination. If not at an interactive pause, this will be termination of the program, see section 8.3.8, and there will be output by Config_Trace_Output of the message (with prefix — see section 6.4.6.1) and tracing (see section 8.3.26.1). If at an interactive pause (#AtPause \= 0), this will be termination of the interpretation of the interactive input; there will be output by Config_Trace_Output of the message (without traceback) before continuing. The description of the continuation is in section 8.2.4 after the "interpret #Outcome" instruction.

The instruction "interpret 'CALL' #TrapName.#Condition.#Level" below does not set the variables RESULT and .RESULT; any result returned is discarded.

```
#Raise:
/* If there is no argument, this is an action which has been delayed from the
time the condition occurred until an appropriate clause boundary. */
if \arg(1,'E') then do
     Description = #PendingDescription.#Condition.#Level
     Extra = #PendingExtra.#Condition.#Level
     end
else do
  #Condition = arg(1)
  if #Condition \== 'SYNTAX' then do
    Description = arg(2)
    Extra = arg(3)
    end
  else do
    Description = #Message(arg(2),arg(3),arg(4),arg(5))
    call Var_Set 0, '.MN', 0, arg(2)
    Extra = ''
    end
  end

/* The events for disabled conditions are ignored or cause termination. */
if Enabling.#Condition.#Level == 'OFF'  |  #AtPause \= 0 then do
  if #Condition \== 'SYNTAX' & #Condition \== 'HALT' then
     return  /* To after use of #Raise. */
  if #Condition == 'HALT' then Description = #Message(4.1, Description)
    exit  /* Terminate with Description as the message. */
  end

/* SIGNAL actions occur as soon as the condition is raised. */
if #Instruction.#Condition.#Level == 'SIGNAL' then do
  #ConditionDescription.#Level = Description
  #ConditionExtra.#Level = Extra
  #ConditionInstruction.#Level = ‘SIGNAL’
  #Enabling.#Condition.#Level = 'OFF'
  signal value #TrapName.#Condition.#Level
  end

/* All CALL actions are initially delayed until a clause boundary. */
if arg(1,'E') then do
   /* Events within the handler are not stacked up, except for one
   extra HALT while a first is being handled. */
   EventLevel = #Level
   if  #Enabling.#Condition.#Level == 'DELAYED' then do
      if #Condition \== 'HALT' then return
      EventLevel = #EventLevel.#Condition.#Level
      if #PendingNow.#Condition.EventLevel  then return
      /* Setup a HALT to come after the one being handled. */
      end
   /* Record a delayed event. */
   #PendingNow.#Condition.EventLevel = '1'
   #PendingDescription.#Condition.EventLevel = Description
   #PendingExtra.#Condition.EventLevel = Extra
   #Enabling.#Condition.EventLevel = 'DELAYED'
   return
   end
 /* Here for CALL action after delay. */
 /* Values for the CONDITION built-in function. */
 #Condition.#NewLevel = #Condition
 #ConditionDescription.#NewLevel = #PendingDescription.#Condition.#Level
 #ConditionExtra.#NewLevel = #PendingExtra.#Condition.#Level
 #ConditionInstruction.#NewLevel = 'CALL'
 interpret 'CALL'  #TrapName.#Condition.#Level
 #Enabling.#Condition.#Level = 'ON'
 return /* To clause termination */
```

### 8.4.2 Messages during execution

The state function #Message corresponds to constructing a message.

This definition is for the message text in section 8.2.1. Translations in which the message inserts are in a different order are permitted.

In addition to the result defined below, the values of MsgNumber and #LineNumber shall be shown when a message is output. Also there shall be an indication of whether the error occurred in code executed at an interactive pause, see section 6.4.6.1.

Messages are shown by writing them to the default error stream.

```
#Message:
    MsgNumber = arg(1)
    if #NoSource then t = t % 1  /* And hence no inserts */
    Text = #ErrorText.MsgNumber
    Expanded = ''
    Index = 2
    do forever
       parse var Text Begin '<' Insert '>' +1 Text
       if Insert = '' then leave
       Insert = arg(Index)
       Index=Index+1
       if length(Insert) > #Limit_MessageInsert then
         t=left(Insert,#Limit_MessageInsert)
       else t=Insert
       Expanded = Expanded || Begin || t
    end

    Expanded = Expanded || Begin
  say Expanded
return
```

# 9 Built-in functions

## 9.1 Notation

The built-in functions are defined mainly through code. The code refers to state variables. This is solely a notation used in this standard.

The code refers to functions with names that start with 'Config_'; these are the functions described in section 5.

The code is specified as an external routine that produces a result from the values #Bif (which is the name of the built-in function), #Bif_Arg.0 (the number of arguments), #Bif_Arg.i and #Bif_ArgExists.i (which are the argument data.)

The value of #Level is the value for the clause which invoked the built-in function.

The code either returns the result of the built-in or exits with an indication of a condition that the invocation of the built-in raises.

The code below uses built-in functions. Such a use invokes another use of this code with a new value of #Level. On these invocations, the CheckArgs function is not relevant.

Numeric settings as follows are used in the code. When an argument is being checked as a number by 'NUM' or 'WHOLENUM' the settings are those current in the caller. When an argument is being checked as an integer by an item containing 'WHOLE' the settings are those for the particular built-in function. Elsewhere the settings have sufficient numeric digits to avoid values which would require exponential notation.

## 9.2 Routines used by built-in functions

The routine CheckArgs is concerned with checking the arguments to the built-in. The routines Time2Date and Leap are for date calculations. ReRadix is used for radix conversion. The routine Raise raises a condition and does not return.

### 9.2.1 Argument checking

```
 /* Check arguments. Some further checks will be made in particular built-
ins.*/
 /* The argument to CheckArgs is a checklist for the allowable arguments. */
 /* NUM, WHOLENUM and WHOLE have a side-effect, 'normalizing' the number. */
 /* Calls to raise syntax conditions will not return. */

 CheckArgs:
    CheckList = arg(1)  /* This refers to the argument of CheckArgs. */

    /* Move the checklist information from a string to individual variables */
    ArgType. = ''
    ArgPos = 0  /* To count arguments */
    MinArgs  = 0
    do j = 1 to length(CheckList)
       ArgPos = ArgPos+1
       /* Count the required arguments. */
       if substr(CheckList,j,1) == 'r' then MinArgs = MinArgs + 1
       /* Collect type information. */
       do while j < length(CheckList)
         j = j + 1
         t = substr(CheckList,j,1)
         if t==' ' then leave
```

```
        ArgType.ArgPos = ArgType.ArgPos || t
        end
      /* A single space delimits parts. */
      end j
   MaxArgs = ArgPos

   /* Check the number of arguments to the built-in, in this instance. */
   NumArgs  = #Bif_Arg.0
   if NumArgs < MinArgs then call Raise 40.3, MinArgs
   if NumArgs > MaxArgs then call Raise 40.4, MaxArgs

   /* Check the type(s) of the arguments to the built-in. */
   do ArgPos = 1 to NumArgs
      if #Bif_ArgExists.ArgPos then
         call CheckType
      else
         if ArgPos <= MinArgs then call Raise 40.5, ArgPos
      end ArgPos

   /* No errors found by CheckArgs. */
   return

CheckType:

   Value = #Bif_Arg.ArgPos
   Type  = ArgType.ArgPos

   select
      when Type == 'ANY' then nop                         /* Any string */

      when Type == 'NUM' then do                          /* Any number */
         /* This check is made with the caller's digits setting. */
         if \Cdatatype(Value, 'N') then
            if #DatatypeResult=='E' then call Raise 40.9, ArgPos, Value
                                    else call Raise 40.11, ArgPos, Value
         #Bif_Arg.ArgPos=#DatatypeResult /* Update argument copy. */
         end

      when Type == 'WHOLE' then do                      /* Whole number */
         /* This check is made with digits setting for the built-in. */
         if \Edatatype(Value,'W') then
            call Raise 40.12, ArgPos, Value
         #Bif_Arg.ArgPos=#DatatypeResult
         end

      when Type == 'WHOLE>=0' then do      /* Non-negative whole number */
         if \Edatatype(Value,'W') then
            call Raise 40.12, ArgPos, Value
         if #DatatypeResult < 0 then
            call Raise 40.13, ArgPos, Value
         #Bif_Arg.ArgPos=#DatatypeResult
         end
```

```
when Type == 'WHOLE>0' then do           /* Positive whole number */
   if \Edatatype(Value,'W') then
      call Raise 40.12, ArgPos, Value
   if #DatatypeResult <= 0 then
      call Raise 40.14, ArgPos, Value
   #Bif_Arg.ArgPos=#DatatypeResult
   end

when Type == 'WHOLENUM' then do  /* D2X type whole number */
   /* This check is made with digits setting of the caller. */
   if \Cdatatype(Value,'W') then
      call Raise 40.12, ArgPos, Value
   #Bif_Arg.ArgPos=#DatatypeResult
   end

when Type == 'WHOLENUM>=0' then do /* D2X Non-negative whole number */
   if \Cdatatype(Value,'W') then
      call Raise 40.12, ArgPos, Value
   if #DatatypeResult < 0 then
      call Raise 40.13, ArgPos, Value
   #Bif_Arg.ArgPos=#DatatypeResult
   end

when Type == '0_90' then do                         /* Errortext */
   if \Edatatype(Value,'N') then
      call Raise 40.11, ArgPos, Value
   Value=#DatatypeResult
   #Bif_Arg.ArgPos=Value
   Major=Value % 1
   Minor=Value - Major
   if Major < 0 | Major > 90 | Minor > .9 | pos('E',Value)>0 then
      call Raise 40.16, Value  /* ArgPos will be 1 */
   end

when Type == 'PAD' then do    /* Single character, usually a pad. */
   if length(Value) \= 1 then
      call Raise 40.23, ArgPos, Value
   end

when Type == 'HEX' then                     /* Hexadecimal string */
   if \datatype(Value, 'X') then
      call Raise 40.25, Value    /* ArgPos will be 1 */

when Type == 'BIN' then                          /* Binary string */
   if \datatype(Value,'B') then
      call Raise 40.24, Value    /* ArgPos will be 1 */

when Type == 'SYM' then                                 /* Symbol */
   if \datatype(Value, 'S') then
      call Raise 40.26, Value   /* ArgPos will be 1 */

when Type == 'STREAM' then do
  call Config_Stream_Qualify Value
```

```
         if left(#Response, 1) == 'B' then
           call Raise 40.27, Value  /* ArgPos will be 1 */
         end

       when Type = 'ACEFILNOR' then do                         /* Trace */
          /* Allow leading '?'s */
          Val = strip(Value,'Left','?')
          if pos(translate(left(Val, 1)), 'ACEFILNOR') = 0 then
             call Raise 40.28, ArgPos, Type, Val
          end

       otherwise do                                            /* Options */
          /* The checklist item is a list of allowed characters */
          if Value == '' then
             call Raise 40.21, ArgPos
          #Bif_Arg.ArgPos = translate(left(Value, 1))
          if pos(#Bif_Arg.ArgPos, Type) = 0 then
             call Raise 40.28, ArgPos, Type, Value
          end

    end   /* Select */

    return

Cdatatype:
 /* This check is made with the digits setting of the caller. */
 /* #DatatypeResult will be set by use of datatype() */
          numeric digits #Digits.#Level
          numeric form value #Form.#Level
          return datatype(arg(1), arg(2))

Edatatype:
 /* This check is made with digits setting for the particular built-in. */
 /* #DatatypeResult will be set by use of datatype() */
          numeric digits #Bif_Digits.#Bif
          numeric form scientific
          return datatype(arg(1),arg(2))
```

### 9.2.2 Date calculations

```
Time2Date:
   if arg(1) < 0 then
     call Raise 40.18
   if arg(1) >= 315537897600000000 then
     call Raise 40.18
   return Time2Date2(arg(1))

Time2Date2:Procedure
   /*  Convert a timestamp to a date.
   Argument is a timestamp (the number of microseconds relative to
   0001 01 01 00:00:00.000000)
   Returns a date in the form
     year month day hour minute second microsecond base days     */
   numeric digits 18
   /* Adjust to the virtual date 0001 01 01 00:00:00.000000 */
   Time=arg(1)+59926608000000000

   Second = Time   % 1000000    ; Microsecond = Time   // 1000000
   Minute = Second %      60    ; Second      = Second //      60
   Hour   = Minute %      60    ; Minute      = Minute //      60
   Day    = Hour   %      24    ; Hour        = Hour   //      24
   /* At this point, the days are the days since the 0001 base date. */
   BaseDays = Day
   Day = Day + 1
   /* Compute either the fitting year, or some year not too far earlier.
   Compute the number of days left on the first of January of this
   year. */
   Year   = Day % 366
   Day    = Day - (Year*365 + Year%4 - Year%100 + Year%400)
   Year   = Year + 1
   /* Now if the number of days left is larger than the number of days
   in the year we computed, increment the year, and decrement the
   number of days accordingly. */
   do while Day > (365 + Leap(Year))
     Day = Day - (365 + Leap(Year))
     Year = Year + 1
   end
   /* At this point, the days left pertain to this year. */
   YearDays = Day
   /* Now step through the months, increment the number of the month,
   and decrement the number of days accordingly (taking into
   consideration that in a leap year February has 29 days), until
   further reducing the number of days and incrementing the month
   would lead to a negative number of days */
   Days = '31 28 31 30 31 30 31 31 30 31 30 31'
   do Month = 1 to words(Days)
     ThisMonth = Word(Days, Month) + (Month = 2) * Leap(Year)
     if Day <= ThisMonth then leave
     Day = Day - ThisMonth
   end

Return Year Month Day Hour Minute Second Microsecond BaseDays YearDays

Leap: Procedure
   /* Return 1 if the year given as argument is a leap year, or 0
   otherwise. */
   Return (arg(1)//4 = 0) & ((arg(1)//100 \= 0) | (arg(1)//400 = 0))
```

### 9.2.3 Radix conversion

```
ReRadix: /* Converts Arg(1) from radix Arg(2) to radix Arg(3) */
  procedure
  Subject=arg(1)
  FromRadix=arg(2)
  ToRadix=arg(3)
  /* Radix range is 2-16.  Conversion is via decimal */
  Integer=0
  do j=1 to length(Subject)
    /* Individual digits have already been checked for range. */
    Integer=Integer*FromRadix+pos(substr(Subject,j,1),'0123456789ABCDEF')-1
    end
  r = ''
  do while Integer>0
    r = substr('0123456789ABCDEF',1 + Integer // ToRadix, 1) || r
    Integer = Integer % ToRadix
    end
  /* When between 2 and 16, there is no zero suppression. */
    if FromRadix = 2 & ToRadix = 16 then
      r=right(r, (length(Subject)+3) % 4, '0')
    else if FromRadix = 16 & ToRadix = 2 then
      r=right(r, length(Subject) * 4, '0')
  return r
```

### 9.2.4 Raising the SYNTAX condition

```
 Raise:
/* These 40.nn messages always include the built-in name as an insert.*/
    call #Raise 'SYNTAX', arg(1), #Bif, arg(2), arg(3), arg(4)
    /* #Raise does not return. */
```

## 9.3 Character built-in functions

These functions process characters or words in strings. Character positions are numbered from one at the left. Words are delimited by blanks and their equivalents, word positions are counted from one at the left.

### 9.3.1 ABBREV

ABBREV returns '1' if the second argument is equal to the leading characters of the first and the length of the second argument is not less than the third argument.

```
    call CheckArgs 'rANY rANY oWHOLE>=0'

    Subject = #Bif_Arg.1
    Subj    = #Bif_Arg.2
    if #Bif_ArgExists.3 then Length = #Bif_Arg.3
                        else Length = length(Subj)
    Cond1 = length(Subject) >= length(Subj)
    Cond2 = length(Subj) >= Length
    Cond3 = substr(Subject, 1, length(Subj)) == Subj
    return Cond1 & Cond2 & Cond3
```

### 9.3.2 CENTER

CENTER returns a string with the first argument centered in it. The length of the result is the second argument and the third argument specifies the character to be used for padding.

```
    call CheckArgs   'rANY rWHOLE>=0 oPAD'

    String = #Bif_Arg.1
    Length = #Bif_Arg.2
    if #Bif_ArgExists.3 then Pad = #Bif_Arg.3
                        else Pad = ' '

    Trim = length(String) - Length

    if Trim > 0 then
       return substr(String, Trim % 2 + 1, Length)

    return overlay(String, copies(Pad, Length), -Trim % 2 + 1)
```

### 9.3.3 CENTRE

This is an alternative spelling for the CENTER built-in function.

### 9.3.4 CHANGESTR

CHANGESTR replaces all occurrences of the first argument within the second argument, replacing them with the third argument.

```
    call CheckArgs   'rANY rANY rANY'

    Output = ''
    Position = 1
    do forever
      FoundPos = pos(#Bif_Arg.1, #Bif_Arg.2, Position)
      if FoundPos = 0 then leave
      Output = Output || substr(#Bif_Arg.2, Position, FoundPos - Position),
               || #Bif_Arg.3
      Position = FoundPos + length(#Bif_Arg.1)
      end
    return Output || substr(#Bif_Arg.2, Position)
```

### 9.3.5 COMPARE

COMPARE returns '0' if the first and second arguments have the same value. Otherwise, the result is the position of the first character that is not the same in both strings.

```
    call CheckArgs  'rANY rANY oPAD'

    Str1 = #Bif_Arg.1
    Str2 = #Bif_Arg.2
    if #Bif_ArgExists.3 then Pad = #Bif_Arg.3
                        else Pad = ' '

    /* Compare the strings from left to right one character at a time */
    if length(Str1) > length(Str2) then do
      Length = length(Str1)
      Str2=left(Str2,Length,Pad)
      end
    else do
      Length = length(Str2)
      Str1=left(Str1,Length,Pad)
      end
    do i = 1 to Length
      if substr(Str1, i, 1) \== substr(Str2, i, 1) then return i
      end
    return 0
```

### 9.3.6 COPIES

COPIES returns concatenated copies of the first argument. The second argument is the number of copies.

```
    call CheckArgs   'rANY rWHOLE>=0'

    Output = ''
    do #Bif_Arg.2
      Output = Output || #Bif_Arg.1
      end
    return Output
```

### 9.3.7 COUNTSTR

COUNTSTR counts the appearances of the first argument in the second argument.

```
    call CheckArgs   'rANY rANY'

    Output = 0
    Position = pos(#Bif_Arg.1,#Bif_Arg.2)
    do while Position > 0
      Output = Output + 1
      Position = pos(#Bif_Arg.1, #Bif_Arg.2, Position + length(#Bif_Arg.1))
      end
    return Output
```

### 9.3.8 DATATYPE

DATATYPE tests for characteristics of the first argument. The second argument specifies the particular test.

```
  call CheckArgs  'rANY oABLMNSUWX'

  /* As well as returning the type, the value for a 'NUM' is set in
  #DatatypeResult.  This is a convenience when DATATYPE is used
  by CHECKARGS. */

  String = #Bif_Arg.1

  /* If no second argument, DATATYPE checks whether the first is a number. */
  if \#Bif_ArgExists.2 then return DtypeOne()

  Type = #Bif_Arg.2
  /* Null strings are a special case. */
  if String == '' then do
    if Type == "X" then return 1
    if Type == "B" then return 1
    return 0
    end

  /* Several of the options are shorthands for VERIFY */
  azl="abcdefghijklmnopqrstuvwxyz"
  AZU="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  D09="0123456789"
  if Type == "A" then return verify(String,azl||AZU||D09)=0
  if Type == "B" then do
      /* Check blanks in allowed places. */
    if pos(left(String,1),#AllBlanks)>0 then return 0
    if pos(right(String,1),#AllBlanks)>0 then return 0
    BinaryDigits=0
    do j = length(String) by -1 to 1
      c = substr(String,j,1)
      if pos(c,#AllBlanks)>0 then do
        /* Blanks need four BinaryDigits to the right of them. */
        if BinaryDigits//4 \= 0 then return 0
        end
      else do
        if verify(c,"01") \= 0  then return 0
        BinaryDigits = BinaryDigits + 1
        end
      end j
    return 1
    end /* B */
  if Type == "L" then return(verify(String,azl)=0)
  if Type == "M" then return(verify(String,azl||AZU)=0)
  if Type == "N" then return(datatype(String)=="NUM")
  if Type == "S" then return(symbol(String)\=='BAD')
  if Type == "U" then return(verify(String,AZU)=0)
  if Type == "W" then do
    /* It may not be a number. */
    if DtypeOne(String) == 'CHAR' then return '0'
    /* It can be "Whole" even if originally in exponential notation,
       provided it can be written as non-exponential. */
    if pos('E',#DatatypeResult)>0 then return '0'
    /* It won't be "Whole" if there is a non-zero after the decimal point. */
    InFraction='0'
    do j = 1 to length(String)
      c = substr(String,j,1)
      if pos(c,'Ee')>0 then leave j
      if InFraction & pos(c,'+-')>0 then leave j
      if c == '.' then InFraction='1'
                  else if InFraction & c\=='0' then return 0
      end j
    /* All tests for Whole passed. */
    #DatatypeResult = #DatatypeResult % 1
    return 1
    end /* W */
  /* Type will be "X" */
  if pos(left(String,1),#AllBlanks)>0 then return 0
```

```
  if pos(right(String,1),#AllBlanks)>0 then return 0
  HexDigits=0
  do j=length(String) by -1 to 1
    c=substr(String,j,1)
    if pos(c,#AllBlanks)>0 then do
      /* Blanks need a pair of HexDigits to the right of them. */
      if HexDigits//2 \= 0 then return 0
      end
    else do
      if verify(c,"abcdefABCDEF"D09) \= 0  then return 0
      HexDigits=HexDigits+1
      end
    end
  return 1
  /* end X */

DtypeOne:
  /* See section 7 for the syntax of a number. */
  #DatatypeResult = 'S' /* If not syntactically a number */
  Residue = strip(String) /* Blanks are allowed at both ends. */
  if Residue == '' then return "CHAR"
  Sign = ''
  if left(Residue,1) == '+' | left(Residue,1) == '-' then do
    Sign = left(Residue, 1)
    Residue = strip(substr(Residue,2),'L') /* Blanks after sign */
    end
  if Residue == '' then return "CHAR"
  /* Now testing Number, section 6.2.2.35 */
  if left(Residue,1) == '.' then do
    Residue = substr(Residue, 2)
    Before = ''
    After = DigitRun()
    if After == '' then return "CHAR"
    end
  else do
    Before = DigitRun()
    if Before == '' then return "CHAR"
    if left(Residue,1) == '.' then do
      Residue = substr(Residue, 2)
      After = DigitRun()
      end
    end
  Exponent = 0
  if Residue \== '' then do
    if left(Residue, 1) \== 'e' & left(Residue, 1) \== 'E' then
       return "CHAR"
    Residue = substr(Residue, 2)
    if Residue == '' then return "CHAR"
    Esign = ''
    if left(Residue, 1) == '+' | left(Residue, 1) == '-' then do
      Esign = left(Residue, 1)
      Residue = substr(Residue, 2)
      if Residue == '' then return "CHAR"
```

```
        end
      Exponent = DigitRun()
      if Exponent == '' then return "CHAR"
      Exponent = Esign || Exponent
      end
    if Residue \== '' then return "CHAR"

    /*DATATYPE tests for exponent out of range. */
    #DatatypeResult = 'E' /* If exponent out of range */
    Before = strip(Before,'L','0')
    if Before == '' then Before = '0'
    Exponent = Exponent + length(Before) -1   /* For SCIENTIFIC */

    /* "Engineering notation causes powers of ten to expressed as a
    multiple of 3 - the integer part may therefore range from 1 through 999." */
    g = 1
    if #Form.#Level == 'E' then do
      /* Adjustment to make exponent a multiple of 3 */
      g = Exponent//3
      if g < 0 then g = g + 3
      Exponent = Exponent - g
      end

    /* Check on the exponent. */
    if Exponent > #Limit_ExponentDigits then return "CHAR"
    if -#Limit_ExponentDigits > Exponent then return "CHAR"

    /* Format to the numeric setting of the caller of DATATYPE */
    numeric digits #Digits.#Level
    numeric form value #Form.#Level
    #DatatypeResult = 0 + #Bif_Arg.1
    return "NUM"

   DigitRun:
     Outcome = ''
     do while Residue \== ''
       if pos(left(Residue, 1), '0123456789') = 0 then leave
       Outcome = Outcome || left(Residue, 1)
       Residue = substr(Residue, 2)
       end
     return Outcome
```

### 9.3.9 DELSTR

DELSTR deletes the sub-string of the first argument which begins at the position given by the second argument. The third argument is the length of the deletion.

```
    call CheckArgs  'rANY rWHOLE>0 oWHOLE>=0'

    String = #Bif_Arg.1
    Num    = #Bif_Arg.2
    if #Bif_ArgExists.3 then Len = #Bif_Arg.3
```

```
if Num > length(String) then return String

Output = substr(String, 1, Num - 1)
if #Bif_ArgExists.3 then
  if Num + Len <= length(String) then
    Output = Output || substr(String, Num + Len)
return Output
```

### 9.3.10 DELWORD

DELWORD deletes words from the first argument. The second argument specifies position of the first word to be deleted and the third argument specifies the number of words.

```
call CheckArgs 'rANY rWHOLE>0 oWHOLE>=0'

String = #Bif_Arg.1
Num    = #Bif_Arg.2
if #Bif_ArgExists.3 then Len = #Bif_Arg.3

if Num > words(String) then return String

EndLeft = wordindex(String, Num) - 1
Output = left(String, EndLeft)
if #Bif_ArgExists.3 then do
   BeginRight = wordindex(String, Num + Len)
   if BeginRight>0 then
      Output = Output || substr(String, BeginRight)
   end
return Output
```

### 9.3.11 INSERT

INSERT insets the first argument into the second. The third argument gives the position of the character before the insert and the fourth gives the length of the insert. The fifth is the padding character.

```
call CheckArgs 'rANY rANY oWHOLE>=0 oWHOLE>=0 oPAD'

New    = #Bif_Arg.1
Target = #Bif_Arg.2
if #Bif_ArgExists.3 then Num = #Bif_Arg.3
                    else Num = 0
if #Bif_ArgExists.4 then Length = #Bif_Arg.4
                    else Length = length(New)
if #Bif_ArgExists.5 then Pad = #Bif_Arg.5
                    else Pad = ' '

return left(Target, Num, Pad),        /* To left of insert   */
   || left(New, Length, Pad),         /* New string inserted */
   || substr(Target, Num + 1)         /* To right of insert  */
```

### 9.3.12 LASTPOS

LASTPOS returns the position of the last occurrence of the first argument within the second. The third argument is a starting position for the search.

```
call CheckArgs 'rANY rANY oWHOLE>0'

Needle   = #Bif_Arg.1
Haystack = #Bif_Arg.2
if #Bif_ArgExists.3 then Start = #Bif_Arg.3
                    else Start = length(Haystack)

NeedleLength = length(Needle)
if NeedleLength = 0 then return 0
Start = Start - NeedleLength + 1
do i = Start by -1 while i > 0
  if substr(Haystack, i, NeedleLength) == Needle then return i
  end i
return 0
```

### 9.3.13 LEFT

LEFT returns characters that are on the left of the first argument. The second argument specifies the length of the result and the third is the padding character.

```
call CheckArgs 'rANY rWHOLE>=0 oPAD'

if #Bif_ArgExists.3 then Pad = #Bif_Arg.3
                    else Pad = ' '

return substr(#Bif_Arg.1, 1, #Bif_Arg.2, Pad)
```

### 9.3.14 LENGTH

Length returns a count of the number of characters in the argument.

```
call CheckArgs 'rANY'

String = #Bif_Arg.1

#Response = Config_Length(String)
Length = #Outcome
call Config_Substr #Response, 1
if #Outcome \== 'E' then return Length
/* Here if argument was not a character string. */
call Config_C2B String
call #Raise 'SYNTAX', 23.1, b2x(#Outcome)
/* No return to here */
```

### 9.3.15 OVERLAY

OVERLAY overlays the first argument onto the second. The third argument is the starting position of the overlay. The fourth argument is the length of the overlay and the fifth is the padding character.

```
call CheckArgs 'rANY rANY oWHOLE>0 oWHOLE>=0 oPAD'

New    = #Bif_Arg.1
Target = #Bif_Arg.2
if #Bif_ArgExists.3 then Num = #Bif_Arg.3
                    else Num = 1
if #Bif_ArgExists.4 then Length = #Bif_Arg.4
                    else Length = length(New)
if #Bif_ArgExists.5 then Pad = #Bif_Arg.5
                    else Pad = ' '

return left(Target, Num - 1, Pad),    /* To left of overlay  */
   || left(New, Length, Pad),         /* New string overlaid */
   || substr(Target, Num + Length)    /* To right of overlay */
```

### 9.3.16 POS

POS returns the position of the first argument within the second.

```
call CheckArgs 'rANY rANY oWHOLE>0'

Needle   = #Bif_Arg.1
Haystack = #Bif_Arg.2
if #Bif_ArgExists.3 then Start = #Bif_Arg.3
                    else Start = 1

if length(Needle) = 0 then return 0
do i = Start to length(Haystack)+1-length(Needle)
  if substr(Haystack, i, length(Needle)) == Needle then return i
  end i
return 0
```

### 9.3.17 REVERSE

REVERSE returns its argument, swapped end for end.

```
call CheckArgs 'rANY'

String = #Bif_Arg.1

Output = ''
do i = 1 to length(String)
  Output = substr(String,i,1) || Output
  end
return Output
```

### 9.3.18 RIGHT

RIGHT returns characters that are on the right of the first argument. The second argument specifies the length of the result and the third is the padding character.

```
    call CheckArgs 'rANY rWHOLE>=0 oPAD'

    String = #Bif_Arg.1
    Length = #Bif_Arg.2
    if #Bif_ArgExists.3 then Pad = #Bif_Arg.3
                        else Pad = ' '

    Trim = length(String) - Length
    if Trim >= 0 then return substr(String,Trim + 1)
    return copies(Pad, -Trim) || String    /* Pad string on the left */
```

### 9.3.19 SPACE

SPACE formats the blank-delimited words in the first argument with pad characters between each word. The second argument is the number of pad characters between each word and the third is the pad character.

```
    call CheckArgs 'rANY oWHOLE>=0 oPAD'

    String = #Bif_Arg.1
    if #Bif_ArgExists.2 then Num = #Bif_Arg.2
                        else Num = 1
    if #Bif_ArgExists.3 then Pad  = #Bif_Arg.3
                        else Pad = ' '

    Padding = copies(Pad, Num)
    Output = subword(String, 1, 1)
    do i = 2 to words(String)
       Output = Output || Padding || subword(String, i, 1)
       end
    return Output
```

### 9.3.20 STRIP

STRIP removes characters from its first argument. The second argument specifies whether the deletions are leading characters, trailing characters or both. Each character deleted is equal to the third argument, or equivalent to a blank if the third argument is omitted.

```
    call CheckArgs 'rANY oLTB oPAD'

    String = #Bif_Arg.1
    if #Bif_ArgExists.2 then Option = #Bif_Arg.2, 1
                        else Option = 'B'
    if #Bif_ArgExists.3 then Unwanted = #Bif_Arg.3
                        else Unwanted = #AllBlanks

    if Option == 'L' | Option == 'B' then do
       /* Strip leading characters */
       do while String \== '' & pos(left(String, 1), Unwanted) > 0
         String = substr(String, 2)
         end
       end
    if Option == 'T' | Option == 'B' then do
       /* Strip trailing characters */
       do while String \== '' & pos(right(String, 1), Unwanted) > 0
         String = left(String, length(String)-1)
         end   /* of while */
       end
    return String
```

### 9.3.21 SUBSTR

SUBSTR returns a sub-string of the first argument. The second argument specifies the position of the first character and the third specifies the length of the sub-string. The fourth argument is the padding character.

```
    call CheckArgs 'rANY rWHOLE>0 oWHOLE>=0 oPAD'

    String = #Bif_Arg.1
    Num    = #Bif_Arg.2
    if #Bif_ArgExists.3 then Length = #Bif_Arg.3
                        else Length = length(String)+1-Num
    if #Bif_ArgExists.4 then Pad = #Bif_Arg.4
                        else Pad = ' '

    Output = ''
    do Length
      #Response = Config_Substr(String,Num) /* Attempt to fetch character.*/
      Character = #Outcome
      Num = Num + 1
      call Config_Substr #Response,1 /* Was there such a character? */
      if #Outcome == 'E' then do
        /* Here if argument was not a character string. */

        call Config_C2B String
        call #Raise 'SYNTAX', 23.1, b2x(#Outcome)
        /* No return to here */
        end
      if #Outcome == 'M' then Character = Pad
      Output=Output||Character
      end
    return Output
```

### 9.3.22 SUBWORD

SUBWORD returns a sub-string of the first argument, comprised of words. The second argument is the position in the first argument of the first word of the sub-string. The third argument is the number of words in the sub-string.

```
    call CheckArgs 'rANY rWHOLE>0 oWHOLE>=0'

    String = #Bif_Arg.1
    Num    = #Bif_Arg.2
    if #Bif_ArgExists.3 then Length = #Bif_Arg.3
                        else Length = length(String) /* Avoids call  */
                                                     /*  to WORDS() */
    if Length = 0 then return ''
    /* Find position of first included word */
    Start = wordindex(String,Num)
    if Start = 0 then return ''                   /* Start is beyond end */

    /* Find position of first excluded word */
    End = wordindex(String,Num+Length)
    if End = 0 then End = length(String)+1

    Output=substr(String,Start,End-Start)

    /* Drop trailing blanks */
    do while Output \== ''
      if pos(right(Output,1),#AllBlanks) = 0 then leave
      Output = left(Output,length(Output)-1)
      end
    return Output
```

### 9.3.23 TRANSLATE

TRANSLATE returns the characters of its first argument with each character either unchanged or translated to another character.

```
    call CheckArgs 'rANY oANY oANY oPAD'
    String = #Bif_Arg.1
    /* If neither input nor output tables, uppercase. */
    if \#Bif_ArgExists.2 & \#Bif_ArgExists.3 then do
      Output = ''
      do j=1 to length(String)
        #Response = Config_Upper(substr(String,j,1))
        Output = Output || #Outcome
        end j
      return Output
      end
    /* The input table defaults to all characters. */
    if \#Bif_ArgExists.3 then do
      #Response = Config_Xrange()
      Tablei = #Outcome
      end
    else Tablei = #Bif_Arg.3
    /* The output table defaults to null */
    if #Bif_ArgExists.2 then Tableo = #Bif_Arg.2
                        else Tableo = ''
    /* The tables are made the same length */
    if #Bif_ArgExists.4 then Pad = #Bif_Arg.4
                        else Pad = ' '
    Tableo=left(Tableo,length(Tablei),Pad)

    Output=''
    do j=1 to length(String)
```

```
      c=substr(String,j,1)
      k=pos(c,Tablei)
      if k=0 then Output=Output||c
             else Output=Output||substr(Tableo,k,1)
      end j
    return Output
```

### 9.3.24 VERIFY

VERIFY checks that its first argument contains only characters that are in the second argument, or that it contains no characters from the second argument; the third argument specifies which check is made. The result is '0', or the position of the character that failed verification. The fourth argument is a starting position for the check.

```
    call CheckArgs 'rANY rANY oMN oWHOLE>0'

    String    = #Bif_Arg.1
    Reference = #Bif_Arg.2
    if #Bif_ArgExists.3 then Option = #Bif_Arg.3,1
                        else Option = 'N'
    if #Bif_ArgExists.4 then Start = #Bif_Arg.4
                        else Start = 1

    Last = length(String)
    if Start > Last then return 0
    if Reference == '' then
       if Option == 'N' then return Start
                        else return 0

    do i = Start to Last
       t = pos(substr(String, i, 1), Reference)
       if Option == 'N' then do
         if t = 0 then return i  /* Return position of NoMatch character. */
         end
       else
         if t > 0 then return i  /* Return position of Matched character. */
       end i
    return 0
```

### 9.3.25 WORD

WORD returns the word from the first argument at the position given by the second argument.

```
    call CheckArgs 'rANY rWHOLE>0'

    return subword(#Bif_Arg.1, #Bif_Arg.2, 1)
```

### 9.3.26 WORDINDEX

WORDINDEX returns the character position in the first argument of a word in the first argument. The second argument is the word position of that word.

```
    call CheckArgs 'rANY rWHOLE>0'
```

```
String = #Bif_Arg.1
Num    = #Bif_Arg.2

/* Find starting position */
Start = 1
Count = 0
do forever
   Start = verify(String, #AllBlanks, 'N', Start)  /* Find non-blank */
   if Start = 0 then return 0                    /* Start is beyond end */
   Count = Count + 1                                /* Words found */
   if Count = Num then leave
   Start = verify(String, #AllBlanks, 'M', Start + 1) /* Find blank  */
   if Start = 0 then return 0                 /* Start is beyond end */
   end
return Start
```

### 9.3.27 WORDLENGTH

WORDLENGTH returns the number of characters in a word from the first argument. The second argument is the word position of that word.

```
call CheckArgs 'rANY rWHOLE>0'

return length(subword(#Bif_Arg.1, #Bif_Arg.2, 1))
```

### 9.3.28 WORDPOS

WORDPOS finds the leftmost occurrence in the second argument of the sequence of words in the first argument. The result is '0' or the word position in the second argument of the first word of the matched sequence. Third argument is a word position for the start of the search.

```
call CheckArgs 'rANY rANY oWHOLE>0'

Phrase = #Bif_Arg.1
String = #Bif_Arg.2
if #Bif_ArgExists.3 then Start = #Bif_Arg.3
                    else Start = 1

Phrase = space(Phrase)
PhraseWords = words(Phrase)
if PhraseWords = 0 then return 0
String = space(String)
StringWords = words(String)
do WordNumber = Start to StringWords - PhraseWords + 1
  if Phrase == subword(String, WordNumber, PhraseWords) then
    return WordNumber
  end WordNumber
return 0
```

### 9.3.29 WORDS

WORDS counts the number of words in its argument.

```
call CheckArgs 'rANY'

do Count = 0 by 1
  if subword(#Bif_Arg.1, Count + 1) == '' then return Count
  end Count
```

### 9.3.30 XRANGE

XRANGE returns an ordered string of all valid character encodings in the specified range.

```
call CheckArgs 'oPAD oPAD'

if \#Bif_ArgExists.1 then #Bif_Arg.1 = ''
if \#Bif_ArgExists.2 then #Bif_Arg.2 = ''
#Response = Config_Xrange(#Bif_Arg.1, #Bif_Arg.2)
return #Outcome
```

## 9.4 Arithmetic built-in functions

These functions perform arithmetic at the numeric settings current at the invocation of the built-in function.   Note that CheckArgs formats any 'NUM'  (numeric) argument.

### 9.4.1 ABS

ABS returns the absolute value of its argument.

```
call CheckArgs  'rNUM'

Number=#Bif_Arg.1
if left(Number,1) = '-' then Number = substr(Number,2)
return Number
```

### 9.4.2 FORMAT

FORMAT formats its first argument.   The second argument specifies the number of characters to be used for the integer part and the third specifies the number of characters for the decimal part. The fourth argument specifies the number of characters for the exponent and the fifth determines when exponential notation is used.

```
call CheckArgs,
  'rNUM oWHOLE>=0 oWHOLE>=0 oWHOLE>=0 oWHOLE>=0'

if #Bif_ArgExists.2 then Before = #Bif_Arg.2
if #Bif_ArgExists.3 then After  = #Bif_Arg.3
if #Bif_ArgExists.4 then Expp   = #Bif_Arg.4
if #Bif_ArgExists.5 then Expt   = #Bif_Arg.5
/* In the simplest case the first is the only argument. */
Number=#Bif_Arg.1
if #Bif_Arg.0 < 2 then return Number

/* Dissect the Number. It is in the normal Rexx format. */
parse var Number Mantissa 'E' Exponent
if Exponent == '' then Exponent = 0
```

```
Sign = 0
if left(Mantissa,1) == '-' then do
  Sign = 1
  Mantissa = substr(Mantissa,2)
  end
parse var Mantissa Befo '.' Afte
/* Count from the left for the decimal point. */
Point = length(Befo)
/* Sign Mantissa and Exponent now reflect the Number. Befo Afte and
Point reflect Mantissa. */

/* The fourth and fifth arguments allow for exponential notation. */
/* Decide whether exponential form to be used, setting ShowExp. */
ShowExp = 0
if #Bif_ArgExists.4 | #Bif_ArgExists.5  then do
  if \#Bif_ArgExists.5 then Expt = #Digits.#Level
  /* Decide whether exponential form to be used. */
  if (Point + Exponent) > Expt then ShowExp = 1 /* Digits before rule. */
  LeftOfPoint = 0
  if length(Befo) > 0 then LeftOfPoint = Befo /* Value left of
  the point */

  /* Digits after point rule for exponentiation: */

  /* Count zeros to right of point. */
  z = 0
  do while substr(Afte,z+1,1) == '0'
    z = z + 1
    end
  if LeftOfPoint = 0 & (z - Exponent) > 5 then ShowExp = 1

  /* An extra rule for exponential form: */
  if #Bif_ArgExists.4 then if Expp = 0 then ShowExp = 0

  /* Construct the exponential part of the result. */
  if ShowExp then do
    Exponent = Exponent + ( Point - 1 )
    Point = 1 /* As required for 'SCIENTIFIC' */
    if #Form.#Level == 'ENGINEERING' then
      do while Exponent//3  \=  0
        Point = Point+1
        Exponent = Exponent-1
        end
    end
  if \ShowExp then Point = Point + Exponent
  end /* Expp or Expt given */
else do
  /* Even if Expp and Expt are not given, exponential notation will
  be used if the original Number+0 done by CheckArgs led to it. */
  if Exponent \= 0 then do
    ShowExp = 1
    end
  end
```

```
/* ShowExp now indicates whether to show an exponent,
   Exponent is its value. */
/* Make this a Number without a point. */
Integer = Befo||Afte
/* Make sure Point position isn't disjoint from Integer. */
if Point<1 then do /* Extra zeros on the left. */
  Integer = copies('0',1 - Point) || Integer
  Point = 1
  end
if Point > length(Integer) then
  Integer = left(Integer,Point,'0') /* And maybe on the right. */

/* Deal with right of decimal point first since that can affect the
left. Ensure the requested number of digits there. */
Afters = length(Integer)-Point
if #Bif_ArgExists.3 = 0 then After = Afters  /* Note default. */
/* Make Afters match the requested After */
do while Afters < After
  Afters = Afters+1
  Integer = Integer'0'
  end
if Afters > After then do
  /* Round by adding 5 at the right place. */
  r=substr(Integer, Point + After + 1, 1)
  Integer = left(Integer, Point + After)
  if r >= '5' then Integer = Integer + 1
  /* This can leave the result zero. */
  If Integer = 0 then Sign = 0
  /* The case when rounding makes the integer longer is an awkward
  one. The exponent will have to be adjusted. */
  if length(Integer) > Point + After then do
     Point = Point+1
     end
  if ShowExp = 1 then do
     Exponent=Exponent + (Point - 1)
     Point = 1 /* As required for 'SCIENTIFIC' */
     if form() = 'ENGINEERING' then
       do while Exponent//3  \=  0
         Point = Point+1
         Exponent = Exponent-1
         end
     end
     t = Point-length(Integer)
     if t > 0 then Integer = Integer||copies('0',t)
    end /* Rounded */
/* Right part is final now. */
if After > 0 then Afte = '.'||substr(Integer,Point+1,After)
             else Afte = ''

/* Now deal with the integer part of the result. */
Integer = left(Integer,Point)
if #Bif_ArgExists.2  =  0 then Before  =  Point + Sign /* Note default. */
```

```
/* Make Point match Before */
if Point > Before - Sign then call Raise  40.38, 2, #Bif_Arg.1
do while Point<Before
   Point = Point+1
   Integer = '0'Integer
   end

/* Find the Sign position and blank leading zeroes. */
r = ''
Triggered = 0
do j = 1 to length(Integer)
  Digit = substr(Integer,j,1)
  /* Triggered is set when sign inserted or blanking finished. */
  if Triggered = 1 then do
    r = r||Digit
    iterate
    end
  /* If before sign insertion point then blank out zero. */
  if Digit = '0' then
     if substr(Integer,j+1,1) = '0' & j+1<length(Integer) then do
       r = r||' '
       iterate
       end
  /* j is the sign insertion point. */
  if Digit = '0' & j \= length(Integer) then Digit = ' '
  if Sign = 1 then Digit = '-'
  r = r||Digit
  Triggered = 1
  end j
Number = r||Afte

if ShowExp = 1 then do
  /* Format the exponent. */
  Expart = ''
  SignExp = 0
  if Exponent<0 then do
    SignExp = 1
    Exponent = -Exponent
    end
  /* Make the exponent to the requested width. */
  if #Bif_ArgExists.4 = 0 then Expp = length(Exponent)
  if length(Exponent) > Expp then
    call Raise 40.38, 4, #Bif_Arg.1
  Exponent=right(Exponent,Expp,'0')
  if Exponent = 0 then do
    if #Bif_ArgExists.4 then Expart = copies(' ',expp+2)
    end
  else if SignExp = 0 then Expart = 'E+'Exponent
                      else Expart = 'E-'Exponent
  Number = Number||Expart
  end
return Number
```

### 9.4.3 MAX

MAX returns the largest of its arguments.

```
if #Bif_Arg.0 <1 then
  call Raise 40.3, 1
call CheckArgs 'rNUM'||copies(' rNUM', #Bif_Arg.0 - 1)

Max = #Bif_Arg.1
do i = 2 to #Bif_Arg.0 by 1
  Next = #Bif_Arg.i
  if Max < Next then Max = Next
  end i
return Max
```

### 9.4.4 MIN

MIN returns the smallest of its arguments.

```
if #Bif_Arg.0 <1 then
  call Raise 40.3, 1
call CheckArgs 'rNUM'||copies(' rNUM', #Bif_Arg.0 - 1)

Min = #Bif_Arg.1
do i = 2 to #Bif_Arg.0 by 1
  Next = #Bif_Arg.i
  if Min > Next then Min = Next
  end i
return Min
```

### 9.4.5 SIGN

SIGN returns '1', '0' or '-1' according to whether its argument is greater than, equal to, or less than zero.

```
call CheckArgs 'rNUM'

Number = #Bif_Arg.1

select
   when Number < 0 then Output = -1
   when Number = 0 then Output =  0
   when Number > 0 then Output =  1
   end
return Output
```

### 9.4.6 TRUNC

TRUNC returns the integer part of its argument, or the integer part plus a number of digits after the decimal point, specified by the second argument.

```
    call CheckArgs 'rNUM oWHOLE>=0'

    Number = #Bif_Arg.1
    if #Bif_ArgExists.2 then Num = #Bif_Arg.2
                        else Num = 0

    Integer =(10**Num  * Number)%1
    if Num=0 then return Integer

    t=length(Integer)-Num
    if t<=0 then return '0.'right(Integer,Num,'0')
              else return insert('.',Integer,t)
```

## 9.5 State built-in functions

These functions return values from the state of the execution.

### 9.5.1 ADDRESS

ADDRESS returns the name of the environment to which commands are currently being submitted. Optionally, under control by the argument, it also returns information on the targets of command output and the source of command input.

```
    call CheckArgs  'oEINO'

    if #Bif_ArgExists.1 then Option1 = #Bif_Arg.1
                        else Option1='N'

    if Option1 == 'N' then return #Env_Name.ACTIVE.#Level

    Tail = Option1'.ACTIVE.'#Level
    return #Env_Position.Tail #Env_Type.Tail #Env_Resource.Tail
```

### 9.5.2 ARG

ARG returns information about the argument strings to a program or routine, or the value of one of those strings.

```
    ArgData = 'oWHOLE>0 oENO'
    if #Bif_ArgExists.2 then ArgData = 'rWHOLE>0 rENO'
    call CheckArgs ArgData

    if \#Bif_ArgExists.1 then return #Arg.#Level.0

    ArgNum=#Bif_Arg.1
    if \#Bif_ArgExists.2 then return #Arg.#Level.ArgNum
    if #Bif_Arg.2 =='O' then return \#ArgExists.#Level.ArgNum
                        else return #ArgExists.#Level.ArgNum
```

### 9.5.3 CONDITION

CONDITION returns information associated with the current condition.

```
    call CheckArgs 'oCDEIS'
```

```
/* Values are null if this is not following a condition. */
if #Condition.#Level == '' then do
   #ConditionDescription.#Level = ''
   #ConditionExtra.#Level = ''
   #ConditionInstruction = ''
   end

Option=#Bif_Arg.1
if Option=='C' then return #Condition.#Level
if Option=='D' then return #ConditionDescription.#Level
if Option=='E' then return #ConditionExtra.#Level
if Option=='I' then return #ConditionInstruction.#Level
/* State is the current state. */
if #Condition.#Level = '' then return ""
return #Enabling.#Condition.#Level
```

### 9.5.4 DIGITS

DIGITS returns the current setting of NUMERIC DIGITS.

```
call CheckArgs ''

return #Digits.#Level
```

### 9.5.5 ERRORTEXT

ERRORTEXT returns the unexpanded text of the message which is identified by the first argument. A second argument of 'S' selects the standard English text, otherwise the text may be translated to another national language. This translation is not shown in the code below.

```
call CheckArgs 'r0_90 oSN'

msgcode = #Bif_Arg.1
if #Bif_ArgExists.2 then Option = #Bif_Arg.2
                    else Option = 'N'
return #ErrorText.msgcode
```

### 9.5.6 FORM

FORM returns the current setting of NUMERIC FORM.

```
call CheckArgs ''

return #Form.#Level
```

### 9.5.7 FUZZ

FUZZ returns the current setting of NUMERIC FUZZ.

```
call CheckArgs ''

return #Fuzz.#Level
```

### 9.5.8 SOURCELINE

If there is no argument, SOURCELINE returns the number of lines in the program, or '0' if the source program is not being shown on this execution. If there is an argument it specifies the number of the line of the source program to be returned.

```
call CheckArgs 'oWHOLE>0'

if \#Bif_ArgExists.1 then return #SourceLine.0
Num = #Bif_Arg.1
if Num > #SourceLine.0 then
   call Raise 40.34, Num, #SourceLine.0
return #SourceLine.Num
```

### 9.5.9 TRACE

TRACE returns the trace setting currently in effect, and optionally alters the setting.

```
call CheckArgs 'oACEFILNOR'      /* Also checks for '?' */

/* With no argument, this a simple query. */
Output=#Tracing.#Level
if #Interactive.#Level then Output = '?'||Output
if \#Bif_ArgExists.1 then return Output

Value=#Bif_Arg.1
if Value=='' then #Interacting.#Level='0'

/* Each question mark toggles the interacting. */
do while left(Value,1)=='?'
  #Interacting.#Level = \#Interacting.#Level
  Value=substr(Value,2)
  end
/* The default setting is 'Normal' */
if Value=='' then Value='N'
Value=translate(left(Value,1))
if Value=='O' then #Interacting.#Level='0'
#Tracing.#Level = Value
return Output
```

## 9.6 Conversion built-in functions

Conversions between Binary form, Decimal form, and heXadecimal form do not depend on the encoding (see section 5.4) of the character data.

Conversion to Coded form gives a result which depends on the encoding. Depending on the encoding, the result may be a string that does not represent any sequence of characters.

### 9.6.1 B2X

B2X performs binary to hexadecimal conversion.

```
call CheckArgs  'rBIN'
```

```
    String = space(#Bif_Arg.1,0)
    return ReRadix(String,2,16)
```

### 9.6.2 BITAND

The functions BITAND, BITOR and BITXOR operate on encoded character data. Each binary digit from the encoding of the first argument is processed in conjunction with the corresponding bit from the second argument.

```
    call CheckArgs  'rANY oANY oPAD'

    String1 = #Bif_Arg.1
    if #Bif_ArgExists.2 then String2 = #Bif_Arg.2
                        else String2 = ''

    /* Presence of a pad implies character strings. */
    if #Bif_ArgExists.3 then
      if length(String1) > length(String2) then
        String2=left(String2,length(String1),#Bif_Arg.3)
      else
        String1=left(String1,length(String2),#Bif_Arg.3)

    /* Change to manifest bit representation. */
    #Response=Config_C2B(String1)
    String1=#Outcome
    #Response=Config_C2B(String2)
    String2=#Outcome
    /* Exchange if necessary to make shorter second. */
    if length(String1)<length(String2) then do
      t=String1
      String1=String2
      String2=t
      end

    /* Operate on common length of those bit strings. */
    r=''
    do j=1 to length(String2)
      b1=substr(String1,j,1)
      b2=substr(String2,j,1)
      select
        when #Bif='BITAND' then
          b1=b1&b2
        when #Bif='BITOR' then
          b1=b1|b2
        when #Bif='BITXOR' then
          b1=b1&&b2
        end
      r=r||b1
      end j
    r=r || right(String1,length(String1)-length(String2))

    /* Convert back to encoded characters. */
    return x2c(b2x(r))
```

### 9.6.3 BITOR

See section 9.6.2

### 9.6.4 BITXOR

See section 9.6.2

### 9.6.5 C2D

C2D performs coded to decimal conversion.

```
call CheckArgs 'rANY oWHOLE>=0'

if length(#Bif_Arg.1)=0 then return 0

if #Bif_ArgExists.2 then do
  /* Size specified */
  Size = #Bif_Arg.2
  if Size = 0 then return 0
  /* Pad will normally be zeros */
  t=right(#Bif_Arg.1,Size,left(xrange(),1))
  /* Convert to manifest bit */
  call Config_C2B t
  /* And then to signed decimal. */
  Sign = Left(#Outcome,1)
  #Outcome = substr(#Outcome,2)
  t=ReRadix(#Outcome,2,10)
  /* Sign indicates 2s-complement. */
  if Sign then t=t-2**length(#Outcome)
  if abs(t) > 10 ** #Digits.#Level - 1 then call Raise 40.35, t
  return t
  end
/* Size not specified. */
call Config_C2B #Bif_Arg.1
t = ReRadix(#Outcome,2,10)
if t > 10 ** #Digits.#Level - 1 then call Raise 40.35, t
return t
```

### 9.6.6 C2X

C2X performs coded  to hexadecimal conversion.

```
call CheckArgs 'rANY'

if length(#Bif_Arg.1) = 0 then return ''
call Config_C2B #Bif_Arg.1
return ReRadix(#Outcome,2,16)
```

### 9.6.7 D2C

D2C performs decimal to coded conversion.

```
if \#Bif_ArgExists.2 then ArgData = 'rWHOLENUM>=0'
```

```
                     else ArgData = 'rWHOLENUM rWHOLE>=0'
    call CheckArgs ArgData

    /* Convert to manifest binary */
    Subject = abs(#Bif_Arg.1)
    r = ReRadix(Subject,10,2)
    /* Make length a multiple of 8, as required for Config_B2C */
    Length = length(r)
    do while Length//8 \= 0
       Length = Length+1
       end
    r = right(r,Length,'0')
    /* 2s-complement for negatives. */
    if #Bif_Arg.1<0 then do
      Subject = 2**length(r)-Subject
      r = ReRadix(Subject,10,2)
      end
    /* Convert to characters */
    #Response = Config_B2C(r)
    Output = #Outcome
    if \#Bif_ArgExists.2 then return Output

    /* Adjust the length with appropriate characters. */
    if #Bif_Arg.1>=0 then return right(Output,#Bif_Arg.2,left(xrange(),1))
                 else return right(Output,#Bif_Arg.2,right(xrange(),1))
```

### 9.6.8 D2X

D2X performs decimal to hexadecimal conversion.

```
   if \#Bif_ArgExists.2 then ArgData = 'rWHOLENUM>=0'
                     else ArgData = 'rWHOLENUM rWHOLE>=0'
    call CheckArgs ArgData

    /* Convert to manifest hexadecimal */
    Subject = abs(#Bif_Arg.1 )
    r = ReRadix(Subject,10,16)
    /* Twos-complement for negatives */
    if #Bif_Arg.1<0 then do
      Subject = 16**length(r)-Subject
      r = ReRadix(Subject,10,16)
      end
    if \#Bif_ArgExists.2 then return r
    /* Adjust the length with appropriate characters. */
    if #Bif_Arg.1>=0 then return right(r,#Bif_Arg.2,'0')
                 else return right(r,#Bif_Arg.2,'F')
```

### 9.6.9 X2B

X2B performs hexadecimal to binary conversion.

```
   call CheckArgs 'rHEX'
   Subject = #Bif_Arg.1
   if Subject == '' then return ''
   /* Blanks were checked by CheckArgs, here they are ignored. */
   Subject = space(Subject,0)
   return ReRadix(translate(Subject),16,2)
```

### 9.6.10 X2C

X2C performs hexadecimal to coded character conversion.

```
   call CheckArgs 'rHEX'

   Subject = #Bif_Arg.1
   if Subject == '' then return ''
   Subject = space(Subject,0)
   /* Convert to manifest binary */
   r = ReRadix(translate(Subject),16,2)
   /* Convert to character */
   Length = 8*((length(Subject)+1)%2)
   #Response = Config_B2C(right(r,Length,'0'))
   return #Outcome
```

### 9.6.11 X2D

X2D performs hexadecimal to decimal conversion.

```
   call CheckArgs 'rHEX oWHOLE>=0'

   Subject = #Bif_Arg.1
   if Subject == '' then return '0'

   Subject = translate(space(Subject,0))
   if #Bif_ArgExists.2 then
     Subject = right(Subject,#Bif_Arg.2,'0')
   if Subject =='' then return '0'
   /* Note the sign */
   if #Bif_ArgExists.2 then SignBit = left(x2b(Subject),1)
                       else SignBit = '0'
   /* Convert to decimal */
   r = ReRadix(Subject,16,10)
   /* Twos-complement */
   if SignBit then r = 2**(4*#Bif_Arg.2) - r
   if abs(r)>10 ** #Digits.#Level - 1 then call Raise 40.35, t
   return r
```

## 9.7 Input/Output built-in functions

The configuration shall provide the ability to access streams. Streams are identified by character string identifiers and provide for the reading and writing of data. They shall support the concepts of characters, lines, and positioning. The input/output built-in functions interact with one another, and they make use of Config_ functions, see section 5.8. When the operations are successful the following characteristics shall be exhibited:

– The CHARIN/CHAROUT functions are insensitive to the lengths of the arguments. The data written to a stream by CHAROUT can be read by a different number of CHARINs.

– The CHARIN/CHAROUT functions are reflective, that is, the concatenation of the data read from a persistent stream by CHARIN (after positioning to 1, while CHARS(Stream)>0), will be the same as the concatenation of the data put by CHAROUT.

– All characters can be used as CHARIN/CHAROUT data.

– The CHARS(Stream, 'N') function will return zero only when a subsequent read (without positioning) is guaranteed to raise the NOTREADY condition.

– The LINEIN/LINEOUT functions are sensitive to the length of the arguments, that is, the length of a line written by LINEOUT is the same as the length of the string returned by successful LINEIN of the line.

– Some characters, call them line-banned characters, cannot reliably be used as data for LINEIN/LINEOUT. If these are not used, LINEIN/LINEOUT is reflective. If they are used, the result is not defined. The set of characters which are line-barred is a property of the configuration.

– The LINES(Stream, 'N') function will return zero only when a subsequent LINEIN (without positioning) is guaranteed to raise the NOTREADY condition.

– When a persistent stream is repositioned and written to with CHAROUT, the previously written data is not lost, except for the data overwritten by this latest CHAROUT.

– When a persistent stream is repositioned and written to with LINEOUT, the previously written data is not lost, except for the data overwritten by this latest LINEOUT, which may leave lines partially overwritten.

### 9.7.1 CHARIN

CHARIN returns a string read from the stream named by the first argument.

```
call CheckArgs 'oSTREAM oWHOLE>0 oWHOLE>=0'

if #Bif_ArgExists.1 then Stream  = #Bif_Arg.1
                    else Stream  = ''
#StreamState.Stream = ''
/* Argument 2 is positioning. */
if #Bif_ArgExists.2 then do
  #Response = Config_Stream_Position(Stream,'CHARIN',#Bif_Arg.2)
  if left(#Response, 1) == 'R' then call Raise 40.41, 2, #Bif_Arg.2
  if left(#Response, 1) == 'T' then call Raise 40.42,Stream
  end
/* Argument 3 is how many. */
if #Bif_ArgExists.3 then Count = #Bif_Arg.3
                    else Count = 1
if Count = 0 then do
  call Config_Stream_Charin Stream, 'NULL' /* "Touch" the stream */
  return ''
  end
/* The unit may be eight bits (as characters) or one character. */
call Config_Stream_Query Stream
Mode = #Outcome
r = ''
do until Count = 0
```

```
      #Response = Config_Stream_Charin(Stream, 'CHARIN')
      if left(#Response, 1) \== 'N' then do
        if left(#Response, 1) == 'E' then #StreamState.Stream = 'ERROR'
        /* This call will return. */
        call #Raise 'NOTREADY', Stream, substr(#Response, 2)
        leave
        end
      r = r||#Outcome
      Count = Count-1
      end
  if Mode == 'B' then do
    call Config_B2C r
    r = #Outcome
    end
  return r
```

### 9.7.2 CHAROUT

CHAROUT returns the count of characters remaining after attempting to write the second argument to the stream named by the first argument.

```
    call CheckArgs 'oSTREAM oANY oWHOLE>0'

    if #Bif_ArgExists.1 then Stream  = #Bif_Arg.1
                        else Stream  = ''

    #StreamState.Stream = ''
    if \#Bif_ArgExists.2  &  \#Bif_ArgExists.3 then do
      /* Position to end of stream. */
      #Response = Config_Stream_Close(Stream)
      if left(#Response,1) == 'T' then call Raise 40.42,Stream
      return 0
      end

    if #Bif_ArgExists.3 then do
      /* Explicit positioning. */
      #Response = Config_Stream_Position(Stream,'CHAROUT', #Bif_Arg.3)
      if left(#Response,1) == 'T' then call Raise 40.42,Stream
      if left(#Response, 1) == 'R' then  call Raise 40.41, 3, #Bif_Arg.3
      end

    if \#Bif_ArgExists.2  | #Bif_Arg.2 == '' then do
      call Config_Stream_Charout Stream, 'NULL' /* "Touch" the stream */
      return 0
      end

    String = #Bif_Arg.2
    Stride = 1
    Residue = length(String)
    call Config_Stream_Query Stream
    Mode = #Outcome
    if Mode == 'B' then do
      call Config_C2B String
      String = #Outcome
      Stride = 8
      Residue = length(String)/8
      end
    Cursor = 1
    do while Residue>0
      Piece = substr(String,Cursor,Stride)
      Cursor = Cursor+Stride
      call Config_Stream_Charout Stream,Piece
       if left(#Response, 1) \== 'N' then do
         if left(#Response, 1) == 'E' then #StreamState.Stream = 'ERROR'
       call #Raise 'NOTREADY', Stream, substr(#Response, 2)
       return Residue
       end
      Residue = Residue - 1
      end
    return 0
```

### 9.7.3 CHARS

CHARS indicates whether there are characters remaining in the named stream. Optionally, it returns a count of the characters remaining and immediately available.

```
    call CheckArgs 'oSTREAM oCN'

    if #Bif_ArgExists.1 then Stream = #Bif_Arg.1
                        else Stream = ''
    if #Bif_ArgExists.2 then Option = #Bif_Arg.2
                        else Option = 'N'

    call Config_Stream_Count Stream, 'CHARS', Option
    return #Outcome
```

### 9.7.4 LINEIN

LINEIN reads a line from the stream named by the first argument, unless the third argument is zero.

```
  call CheckArgs 'oSTREAM oWHOLE>0 oWHOLE>=0'

  if #Bif_ArgExists.1 then Stream = #Bif_Arg.1
                      else Stream = ''
  #StreamState.Stream = ''
  if #Bif_ArgExists.2 then do
    #Response = Config_Stream_Position(Stream, 'LINEIN', #Bif_Arg2)
    if left(#Response, 1) == 'T' then call Raise 40.42,Stream
    if left(#Response, 1) == 'R' then  call Raise 40.41, 2, #Bif_Arg.2
    end
  if #Bif_ArgExists.3 then Count = #Bif_Arg.3
                      else Count = 1
  if Count>1 then call Raise 40.39, Count
  if Count = 0 then do
      call Config_Stream_Charin Stream, 'NULL' /* "Touch" the stream */
      return ''
      end
  /* A configuration may recognise lines even in 'binary' mode. */
  call Config_Stream_Query Stream
  Mode = #Outcome
  r = ''
  t = #Linein_Position.Stream
  /* Config_Stream_Charin will alter #Linein.Position. */
  do until t \= #Linein_Position.Stream
      #Response = Config_Stream_Charin(Stream,'LINEIN')
      if left(#Response, 1) \== 'N' then do
        if left(#Response, 1) == 'E' then #StreamState.Stream = 'ERROR'
        call #Raise 'NOTREADY', Stream, substr(#Response, 2)
        leave
        end
     r = r||#Outcome
     end
  if Mode == 'B' then do
    call Config_B2C r
    r = #Outcome
    end
  return r
```

### 9.7.5 LINEOUT

LINEOUT returns '1' or '0', indicating whether the second argument has been successfully written to the stream named by the first argument. A result of '1' means an unsuccessful write.

```
call CheckArgs 'oSTREAM oANY oWHOLE>0'

if #Bif_ArgExists.1 then Stream  = #Bif_Arg.1
                    else Stream  = ''
#StreamState.Stream = ''
if \#Bif_ArgExists.2  &  \#Bif_ArgExists.3 then do
  /* Position to end of stream. */
  #Response = Config_Stream_Close(Stream)
  if left(#Response,1) == 'T' then call Raise 40.42,Stream
  return 0
  end

if #Bif_ArgExists.3 then do
  #Response = Config_Stream_Position(Stream,'LINEOUT', #Bif_Arg.3)
  if left(#Response, 1) == 'T' then call Raise 40.42,Stream
  if left(#Response, 1) == 'R' then call Raise 40.41, 3, #Bif_Arg.3
  end

if \#Bif_ArgExists.2  then do
  call Config_Stream_Charout Stream, '' /* "Touch" the stream */
  return 0
  end

String = #Bif_Arg.2
Stride = 1
Residue = length(String)
```

```
call Config_Stream_Query Stream
Mode = #Outcome
if Mode == 'B' then do
  call Config_C2B String
  String = #Outcome
  Stride = 8
  Residue = length(String)/8
  end
Cursor = 1
do while Residue > 0
  Piece = substr(String,Cursor,Stride)
  Cursor = Cursor+Stride
  call Config_Stream_Charout Stream, Piece
  if left(#Response, 1) \== 'N' then do
    if left(#Response, 1) == 'E' then #StreamState.Stream = 'ERROR'
    call #Raise 'NOTREADY', Stream, substr(#Response, 2)
    return 1
    end
  Residue = Residue-1
  end
call Config_Stream_Charout Stream, 'EOL'
return 0
```

### 9.7.6 LINES

LINES returns the number of lines remaining in the named stream.

```
call CheckArgs 'oSTREAM oCN'

if #Bif_ArgExists.1 then Stream = #Bif_Arg.1
                    else Stream = ''
if #Bif_ArgExists.2 then Option = #Bif_Arg.2
                    else Option = 'N'

Call Config_Stream_Count Stream, 'LINES', Option
return #Outcome
```

### 9.7.7 QUALIFY

QUALIFY returns a name for the stream named by the argument. The two names are currently associated with the same resource and the result of QUALIFY may be more persistently associated with that resource.

```
call CheckArgs 'oSTREAM'

if #Bif_ArgExists.1 then Stream  = #Bif_Arg.1
                    else Stream  = ''

#Response = Config_Stream_Qualified(Stream)
return #Outcome
```

### 9.7.8 STREAM

STREAM returns a description of the state of, or the result of an operation upon, the stream named by the first argument.

```
    /* Third argument is only correct with 'C' */
    if #Bif_ArgExists.2 & translate(left(#Bif_Arg.2, 1) == 'C' then
       ArgData = 'rSTREAM rCDS rANY'
    else
       ArgData = 'rSTREAM oCDS'
    call CheckArgs ArgData

    Stream = #Bif_Arg.1

    if #Bif_ArgExists.2 then Operation = #Bif_Arg.2
                        else Operation = 'S'

   Select
     when Operation == 'C' then do
       call Config_Stream_Command Stream,#Bif_Arg.3
       return #Outcome
       end
     when Operation == 'D' then do
       #Response =  Config_Stream_State(Stream)
       return substr(#Response, 2)
       end
     when Operation == 'S' then do
       if StreamState.Stream == 'ERROR' then return 'ERROR'
       #Response =  Config_Stream_State(Stream)
       if left(#Response, 1) == 'N' then return 'READY'
       if left(#Response, 1) == 'U' then return 'UNKNOWN'
       return 'NOTREADY'
       end
     end
```

## 9.8 Other built-in functions

### 9.8.1 DATE

DATE with fewer than two arguments returns the local date. Otherwise it converts the second argument (which has a format given by the third argument) to the format specified by the first argument.

```
    call CheckArgs 'oBDEMNOSUW oANY oBDENOSU'
    /* If the third argument is given then the second is mandatory. */
    if #Bif_ArgExists.3 & \#Bif_ArgExists.2 then
      call Raise 40.19, '', #Bif_Arg.3

    if #Bif_ArgExists.1 then Option = #Bif_Arg.1
                        else Option = 'N'

    /* The date/time is 'frozen' throughout a clause. */
    if #ClauseTime.#Level == '' then do
      #Response = Config_Time()
      #ClauseTime.#Level = #Time
      #ClauseLocal.#Level = #Time + #Adjust
      end
    /* English spellings are used, even if messages not in English are used.
*/
    Months = 'January February March April May June July',
             'August September October November December'
    WeekDays = 'Monday Tuesday Wednesday Thursday Friday Saturday Sunday'

    /* If there is no second argument, the current date is returned. */
    if \#Bif_ArgExists.2 then
      return DateFormat(#ClauseLocal.#Level, Option)

    /* If there is a second argument it provides the date to be
    converted. */
    Value = #Bif_Arg.2
    if #Bif_ArgExists.3 then InOption = #Bif_Arg.3
                        else InOption = 'N'
    /* First try for Year Month Day */
    Logic = 'NS'
    select
      when InOption == 'N' then do
        parse var Value Day MonthIs Year
        do Month = 1 to 12
          if left(word(Months, Month), 3) == MonthIs then leave
          end Month
        end
      when InOption == 'S' then parse var Value Year +4 Month +2 Day
      otherwise Logic = 'EOU' /* or BD */
      end
    /* Next try for year without century */
    if logic = 'EOU' then
    Select
      when InOption == 'E' then parse var Value Day '/' Month '/' YY
      when InOption == 'O' then parse var Value YY '/' Month '/' Day
      when InOption == 'U' then parse var Value Month '/' Day '/' YY
      otherwise Logic = 'BD'
      end
    if Logic = 'EOU' then do
      /* The century is assumed, on the basis of the current year. */
      if \datatype(YY,'W') then
        call Raise 40.19, Value, InOption
      parse value Time2Date(#ClauseLocal.#Level) with YearNow .
      Year = YY
      do while Year < YearNow-50
        Year = Year + 100
        end
      end /* Century assumption */
    if Logic \= 'BD' then do
      /* Convert Month & Day to Days of year. */
      if \datatype(Month,'W') | \datatype(Day,'W') | \datatype(Year,'W') then
        call Raise 40.19, Value, InOption
      Days = word('0 31 59 90 120 151 181 212 243 273 304 334',Month) +,
        (Month>2)*Leap(Year) + Day-1
      end
    else
      if \datatype(Value,'W') then
        call Raise 40.19, Value, InOption
    if InOption == 'D' then do
      parse value Time2Date(#ClauseLocal.#Level) with Year .
      Days = Value - 1 /* 'D' includes current day */
      end
    /* Convert to BaseDays */
    if InOption \== 'B' then
      BaseDays = (Year-1)*365 + (Year-1)%4 - (Year-1)%100 + (Year-1)%400,
                 + Days
    else Basedays = Value
    /* Convert to microseconds from 1900 */
    Micro = BaseDays * 86400 * 1000000 - 59926608000000000
    /* Reconvert to check the original. (eg for Month = 99) */
    if DateFormat(Micro,InOption) \== Value then
      call Raise 40.19, Value, InOption
    return DateFormat(Micro, Option)
    end /* Conversion */

DateFormat:
    /* Convert from microseconds to given format. */
    parse value Time2Date(arg(1)) with,
         Year Month Day Hour Minute Second Microsecond Base Days
    select
      when arg(2) == 'B' then
        return Base
      when arg(2) == 'D' then
        return Days
      when arg(2) == 'E' then
        return right(Day,2,'0')'/'right(Month,2,'0')'/'right(Year,2,'0')
      when arg(2) == 'M' then
        return word(Months,Month)
      when arg(2) == 'N' then
        return Day left(word(Months,Month),3) right(Year,4,'0')
      when arg(2) == 'O' then
        return right(Year,2,'0')'/'right(Month,2,'0')'/'right(Day,2,'0')
      when arg(2) == 'S' then
        return right(Year,4,'0')||right(Month,2,'0')||right(Day,2,'0')
      when arg(2) == 'U' then
        return right(Month,2,'0')'/'right(Day,2,'0')'/'right(Year,2,'0')
      otherwise /* arg(2) == 'W' */
        return word(Weekdays,1+Base//7)
      end
```

### 9.8.2 QUEUED

QUEUED returns the number of lines remaining in the external data queue.

```
     call CheckArgs ''
```

```
 #Response = Config_Queued()
 return #Outcome
```

### 9.8.3 RANDOM

RANDOM returns a quasi-random number.

```
call CheckArgs  'oWHOLE>=0 oWHOLE>=0 oWHOLE>=0'

if #Bif_Arg.0 = 1 then do
   Minimum = 0
   Maximum = #Bif_Arg.1
   if Maximum>100000 then
      call Raise 40.31, Maximum
   end
else do
   if #Bif_ArgExists.1 then Minimum = #Bif_Arg.1
                       else Minimum = 0
   if #Bif_ArgExists.2 then Maximum = #Bif_Arg.2
                       else Maximum = 999
   end

if Maximum-Minimum>100000 then
   call Raise 40.32, Minimum, Maximum

if Maximum-Minimum<0 then
   call Raise 40.33, Minimum, Maximum

if #Bif_ArgExists.3 then call Config_Random_Seed #Bif_Arg.3
call Config_Random_Next Minimum, Maximum
return #Outcome
```

### 9.8.4 SYMBOL

The function SYMBOL takes one argument, which is evaluated. Let String be the value of that argument.  If Config_Length(String) returns an indicator 'E' then the SYNTAX condition 23.1 shall be raised.

Otherwise, if the syntactic recognition described in section 6 would not recognize String as a *symbol* then the result of the function SYMBOL is 'BAD'.

If String would be recognized as a symbol the result of the function SYMBOL depends on the outcome of accessing the value of that symbol, see section 7.3. If the final use of Var_Value leaves the indicator with value 'D' then the result of the function SYMBOL is 'LIT', otherwise 'VAR'.

### 9.8.5 TIME

TIME with less than two arguments returns the local time within the day, or an elapsed time. Otherwise it converts the second argument (which has a format given by the third argument) to the format specified by the first argument.

```
call CheckArgs 'oCEHLMNORS oANY oCHLMNS'
/* If the third argument is given then the second is mandatory. */
if #Bif_ArgExists.3 & \#Bif_ArgExists.2 then
  call Raise 40.19, '', #Bif_Arg.3
```

```
    if #Bif_ArgExists.1 then Option = #Bif_Arg.1
                        else Option = 'N'

    /* The date/time is 'frozen' throughout a clause. */
    if #ClauseTime.#Level == '' then do
       #Response = Config_Time()
       #ClauseTime.#Level = #Time
       #ClauseLocal.#Level = #Time + #Adjust
       end

    /* If there is no second argument, the current time is returned. */
    if \#Bif_ArgExists.2 then
      return TimeFormat(#ClauseLocal.#Level, Option)

    /* If there is a second argument it provides the time to be
    converted. */
    if pos(Option, 'ERO') > 0 then
      call Raise 40.29, Option
    InValue = #Bif_Arg.2
    if #Bif_ArgExists.3 then InOption = #Bif_Arg.3
                        else InOption = 'N'
    HH = 0
    MM = 0
    SS = 0
    HourAdjust = 0
    select
      when InOption == 'C' then do
        parse var InValue HH ':' . +1 MM +2 XX
        if XX == 'pm' then HourAdjust = 12
        end
      when InOption == 'H' then HH = InValue
      when InOption == 'L' | InOption == 'N' then
        parse var InValue HH ':' MM ':' SS
      when InOption == 'M' then MM = InValue
      otherwise SS = InValue
      end
    if \datatype(HH,'W') | \datatype(MM,'W') | \datatype(SS,'N') then
      call Raise 40.19, InValue, InOption
    HH = HH + HourAdjust
    /* Convert to microseconds */
    Micro = trunc((((HH * 60) + MM) * 60 + SS) * 1000000)
    /* Reconvert to check the original. (eg for hour = 99) */
    if TimeFormat(Micro,InOption) \== InValue then
      call Raise 40.19, InValue, InOption
    return TimeFormat(Micro, Option)
    end /* Conversion */

TimeFormat:
    /* Convert from microseconds to given format. */
    parse value Time2Date(arg(1)) with,
         Year Month Day Hour Minute Second Microsecond Base Days
    select
```

```
      when arg(2) == 'C' then
         if Hour>12 then
            return Hour-12':'right(Minute,2,'0')'pm'
         else
            return Hour':'right(Minute,2,'0')'am'
      when arg(2) == 'E' | arg(2) == 'R' then do
         /* Special case first time */
         if #StartTime.#Level == '' then do
            #StartTime.#Level = #ClauseTime.#Level
            return '0'
            end
         Output = #ClauseTime.#Level-#StartTime.#Level
         if arg(2) == 'R' then
           #StartTime.#Level = #ClauseTime.#Level
         return Output * 1E-6
         end  /* E or R */
      when arg(2) == 'H' then return Hour
      when arg(2) == 'L' then
         return right(Hour,2,'0')':'right(Minute,2,'0')':'right(Second,2,'0'),
            || '.'right(Microsecond,6,'0')
      when arg(2) == 'M' then return 60*Hour+Minute
      when arg(2) == 'N' then
         return right(Hour,2,'0')':'right(Minute,2,'0')':'right(Second,2,'0')
      when arg(2) == 'O' then
         return trunc(#ClauseLocal.#Level - #ClauseTime.#Level)
      otherwise /* arg(2) == 'S' */
        return 3600*Hour+60*Minute+Second
      end
```

### 9.8.6 VALUE

VALUE returns the value of the symbol named by the first argument, and optionally assigns it a new value.

```
    if #Bif_ArgExists.3 then ArgData = 'rANY oANY oANY'
                        else ArgData = 'rSYM oANY oANY'
    call CheckArgs ArgData

    Subject = #Bif_Arg.1
    if #Bif_ArgExists.3 then do  /* An external pool. */
      /* Fetch the original value */
      Pool = #Bif_Arg.3
      #Response = Config_Get(Pool,Subject)
      #Indicator = left(#Response,1)
      if #Indicator == 'F' then
         call Raise 40.36, Subject
      if #Indicator == 'P' then
         call Raise 40.37, Pool
      Value = #Outcome
      if #Bif_ArgExists.2 then do
         /* Set the new value. */
         #Response = Config_Set(Pool,Subject,#Bif_Arg.2)
         if #Indicator == 'P' then
```

```
       call Raise 40.37, Pool
     if #Indicator == 'F' then
       call Raise 40.36, Subject
     end
  /* Return the original value. */
  return Value
  end
/* Not external */
Subject = translate(Subject)
/* See section 7.3 */
p = pos(Subject, '.')
if p = 0 | p = length(Subject) then do
  /* Not compound */
  #Response = Var_Value(#Pool, Subject, '0')
  /* The caller, in the code of the standard, may need
  to test whether the Subject was dropped. */
  #Indicator = left(#Response, 1)
  Value = #Outcome
  if #Bif_ArgExists.2 then
     #Response = Var_Set(#Pool, Subject, '0', #Bif_Arg.2)
  return Value
  end
/* Compound */
Expanded = left(Subject,p-1)  /* The stem */
do forever
  Start = p+1
  p = pos(Subject,'.',Start)
  if p = 0 then p = length(Subject)
  Item = substr(Subject,Start,p-Start) /* Tail component symbol */
  if Item\=='' then if pos(left(Item,1),'0123456789') = 0 then do
     #Response = Var_Value(#Pool, Item, '0')
     Item = #Outcome
     end
  /* Add tail component. */
  Expanded = Expanded'.'Item
  end
#Response = Var_Value(#Pool, Expanded, '1')
#Indicator = left(#Response, 1)
Value = #Outcome
if #Bif_ArgExists.2 then
   #Response = Var_Set(#Pool, Expanded, '1', #Bif_Arg.2)
return Value
```

# Annex A — Rationale (informative)

This Annex explains some of the decisions made by the committee that drafted this standard, and gives help in understanding this standard. Some of the statements made are opinions rather than facts. These should be interpreted as if prefixed by "In the opinion of the X3J18 committee...". The numbering in this appendix reflects the numbering of the normative part of this standard, for example section A.4.1 explains section 4.1.

All the implementations of Rexx known to the committee used the book "The Rexx Language" as a specification, directly or indirectly. This has not meant that all implementations are identical, but it indicates that this book should be the basis for this standard. Implementations have also offered interfaces to the environment, and some commonality has occurred. The commonality extends to a wider area of interface than the environment considerations covered in "The Rexx Language". The commonality occurred because of a desire to provide approximately the interfaces specified in the committee's second base document.

Because not all implementations are identical, and because this standard differs from the base documents, there is a possibility of 'breakage'; programs written before this standard existed which do not execute as originally intended when processed by a conforming processor. The committee has balanced the aims of portability, maintainability and coherence against the desire to avoid breakage.

Some language features in this standard are extensions, not found in existing implementations or the base documents. Extensions were only added when they were essential for the goals of portability or maintainability. It was not the intention to add extensions merely because they might improve the productivity of the programmers using Rexx.

### A.4.1 Conformance

Note that irrespective of how this standard is written, the obligation on a conforming processor is only to achieve the defined results, not to follow the algorithms in this standard.

It is expected that some implementations of this standard will be constructed as part of larger implementations that have extra features. It is expected that such an implementation will provide an option (external to any Rexx program) that allows processing to be either conforming (no extra features) or non-conforming (extra features used). Note that this is an alternative; there can be no mode which is "ANSI Rexx conforming, with extensions".

### A.4.2 Limits

There is no practical combination of characteristics in a program which could be specified to provide portability for a broad range of programs, without making implementation of Rexx on small systems unduly difficult. Therefore capacity requirements for program size, nested CALLs etc. have not been given. The limits for a particular implementation will show as a "System resources exhausted" message.

In a formal sense, an implementation which processed any program solely by raising the "System resources exhausted" message would be a conforming implementation. The good sense of implementers will ensure that in practice the spirit of this standard is upheld. These limits in section 4.2 should be regarded in that light; an indication of what might be expected but not a portability guarantee.

No value is suggested in this section for the limit on the length of an environment name since this is not an important portability issue.

## A.5 Configuration

In addition to it's use for self-contained programs, Rexx is used as a means of controlling other programs and system services, and is used to provide customization of other programs (that is, used as a 'macro' language). This standard defines interfaces to that organized structure, but only insofar as necessary to allow a Rexx language processor to be implemented.

This standard does not define how programs in the system interact by making joint use of components of the system. In practice, it is to be expected that the file system which supports streams will be shared, the External Data Queue will be shared, and the External Variable Pools will be shared.

This standard does not define how multiple language processors in the system interact with one another. In practice there may be a 'Rexx co-ordination' feature in the system to determine which language processor is to process a particular source file, on the basis of the content or attributes of the file.

#### A.5.1.1 Notation for completion response and conditions

It would be unreasonable to specify exactly when the "Resources Exhausted" condition should be raised, but since there may be a handler for the condition it is intended that implementors avoid raising it in circumstances that would be confusing, for example with an EXPOSE list partially processed.

#### A.5.2.1 API_Start

This mechanism provides a general way for non-Rexx programs to access Rexx features; a Rexx program can be constructed and run by the non-Rexx program.

When a Rexx program invokes an external routine which is written in Rexx, the configuration will use API_Start to execute the external routine. This standard does not specify the relation between the parameters on Config_ExternalRoutine and on API_Start, although is expected that in practice at least some of the parameters, for example How, will be propagated.

Note that some of the arguments have components, so they will not be representable as a single string.

### A.5.3 Source programs and character sets

This standard does not require particular binary encodings for the characters.

Certain characters are defined as being necessary in order to write Rexx programs. This makes it impossible to have a conforming implementation on hardware that does not support those characters. This situation could have been ameliorated by defining substitutes for some of the less widely available characters, in terms of more widely available characters. Since the characters required by Rexx are in the most commonly available character sets and since the trend is for character sets to become larger, the complication of a general substitution mechanism is not justified.

The ideal of a fully specified set of characters for Rexx programs is balanced against other objectives by the inclusion of the categories extra_letters, other_negators, and other_blank_characters. These allow an implementation to offer a choice between maximum portability and benefits such as reduced breakage and tailoring to national languages, without loss of conformance.

#### A.5.3.2 Extra_letters

The intent of this clause is to allow for existing programs that treat as letters characters such as the @ sign, and to allow local variation such as the use of accented characters in symbols.

#### A.5.3.3 Other_blank_characters

The intent of this clause is to allow some characters, such as tab characters, to be treated as the blank character in certain contexts. Note that if any of these characters are being used as the end-of-line indication they may be converted to EOL events and hence not seen as blank characters.

#### A.5.3.4 Other_negators

The intent of this clause is to allow for existing programs which use characters other than the backslash in the role of negation.

### A.5.4 Configuration characters and encoding

The base reference has some dependencies on each single character being encoded as an octet in the range '00'x to 'FF'x. The intent of this section is to retain the semantics for encodings which are like that, while allowing for other encodings. The mapping between strings and bits (and hence numbers) is defined by the configuration.

Note that while this allows varied encodings to be used, it does not provide portability over different encodings for all programs. In particular, the collating sequence for characters is defined by the configuration.

#### A.5.4.4 Config_Compare

This comparison could be done by comparing the numeric values of the encodings of the characters but this standard does not require that.

#### A.5.4.5 Config_B2C

Much of Rexx is in terms of characters, without concern for the binary values which the characters are encoded to. Two bytes per character, and a variable number of bytes per character have been used in implementations. The encoding is an implementation choice, and some characters may not appear on the associated keyboards. The built-in function X2C, which uses Config_B2C, allows such characters to be constructed from a knowledge of their numeric encodings.

It may happen that for some integers, with the particular encoding scheme, there is no corresponding character string. It might seem natural for X2C to fail in this circumstance. However, there are some built-in functions, for example BITAND, which operate on a string as a sequence of bytes, with all possible values of each byte allowed. This implies that there are strings which are not character strings. So that these strings can be created, Config_B2C does not detect whether the string it creates is a character string or not.

The function Config_Substr does checking. This means that functions which depend on Config_Substr, for example LENGTH, may give rise to SYNTAX condition 40.23. The configuration decides whether an encoding is correct or incorrect, and it may take all of the first argument to Config_Substr into account.

### A.5.5 Commands

What the commands do is not defined. An implementation that did nothing for every command would be conforming.

It is not defined when Config_Command should return 'E' or 'F'. The intent is that 'E' is indicated when the command has not executed normally and the program using the command would normally be prepared for what occurred. The intent is that 'F' is indicated when the command has not executed normally and the program using the command would not normally be prepared for what occurred.

The content of the return code string is not defined. The intent is that it should relate to the configuration, for example being an error number that can be looked up in information about the configuration.

The standard does not define what environments will be available. It is expected that there will be an environment that submits commands to the operating system and there may be other application "registered" with the language processor as environments.

No special message is provided for the case where the environment name is unacceptable to the configuration. A Config_Command result of 'S' followed by a description of the problem would be appropriate, leading to a message 48.

### A.5.6 External routines

The NameType parameter allows the search for the external routine to do, for example, lower-casing of the name when the name was not written in quotes.

### A.5.7 External data queue

The sentence "The configuration may permit the external data queue to be altered in other ways" is specifically intended to allow:

– Execution of commands to communicate with the execution by taking data from the external data queue or by adding to it. Note, however, that better methods of communication are described in section 8.3.1.

– Programs to communicate through an external data queue having a wider scope than the execution of a single program.

– Restrictions to be placed on the length of any string to be held in the external data queue.

### A.5.8 Streams

A balance has to be made between leaving too much of I/O undefined and defining I/O in a way that cannot be reasonably implemented on some file systems. This standard provides a 'safety valve' for implementation by specifying the required behavior of successful operations. An implementation that was unable to achieve the required behavior could raise an error when the relevant operations were attempted. Another 'safety valve' is provided by Config_Stream_Command, which allows for implementation defined I/O operations.

Calls to the configuration are shown even for operations that "do nothing", for example reading zero characters, to emphasize that the configuration may make a file system change, for example altering a last-used time stamp.

This standard does not define how end-of-line is detected. For systems which have the concept of 'records' it should corresponds to a junction between records or the end of the final record. For systems which have an end-of-line character (or sequence of characters) embedded in the data, those characters are not part of any line. The line position is incremented when they are encountered. Notionally all the characters of an end-of-line sequence are ignored (and the character position advanced) when the end-of-line is detected.

#### A.5.8.1 Config_Stream_Charin

If no characters are immediately available but further characters are expected the configuration should wait for the input before returning. The programmer can test for the possibility of delay using the CHARS built-in function.

#### A.5.8.6 Config_Stream_Qualified

This standard does not define the circumstances in which the configuration may make two different stream names refer to the same data, for example the stream 'C:\MYDIR\MYFILE' may refer to the same data file as stream 'myfile' at one time during execution, but not at another time. The intent of Config_Stream_Qualified is to obtain a stream name that is as robust as possible; one that is likely to address the same file in a persistent way.

#### A.5.8.9 Config_Stream_Close

It is not intended that this operation will necessarily have all the characteristics of an operating system "close" operation.

The positions for reading are not altered so a program may need to reset them explicitly in addition to "closing".

#### A.5.8.10 Config_Stream_Count

The intent of this function is  to support algorithms which require to count chars and lines in persistent files, to provide a test whether there is data available immediately on a transient file and to allow CHARS()=0 as a reliable test for end-of-file.

Since providing an exact count may be expensive on some implementations, the option allows the programmer to request this only when needed.

#### A.5.9.1 Config_Get

No special indicator is provided for the case where a variable exists but is not initialized. An implementation might use an 'F' or an 'S' indicator for this case, or might return a value for the variable.

#### A.5.10.1 Config_Constants

Existing practice is usually to implement the built-in functions using NUMERIC DIGITS 9 for the checks on the arguments. That may be inadequate for some purposes, such as the character position within a very large stream, so the configuration is allowed to indicate different values of NUMERIC DIGITS for different built-in functions.

It is intended that the right part of the first word of #Version should identify the language processor in use. It is intended that the date in the last three words should be the release date of the language processor.

The base document specifies that the first word of #Version should not contain a period, so as to simplify parsing #Version. Existing practice has not honored that rule, and the rule is not part of this standard.

#### A.5.11.2 Config_Trace_Input

It is common existing practice to take the trace input from the default input stream.

#### A.5.11.3 Config_Trace_Output

It is common existing practice to send the trace output (and the message output) to either the default error stream or the default output stream.

#### A.5.11.9 Config_Halt_Reset

The Config_Trace_Query and Config_Halt_Query mechanisms could have been defined as more similar to one another.  However, the committee preferred to follow the existing practice defined in the second base document.

#### A.5.11.10 Config_NoSource

This routine is used to ask the configuration about each program. A response of '1' is used to indicate that features which readily expose parts of the source program, such as tracing, message inserts and full use of the SOURCELINE built-in function, are to be disabled. This might occur, for example, when a configuration has this increased security as an option, or when the program was compiled in a way that discarded the source. An implementation is conforming if it always returns '0' but not if it always returns '1'.

#### A.5.11.11 Config_Time

There may be a noticeable lack of accuracy in the time returned by the configuration. In such circumstances the configuration should return a micro-second timing which is an integer with appropriate zeros on the right (when in decimal notation) so as to avoid a spurious impression of accuracy.

The date 1/1/0001 is a "date that never happened" because it preceded the introduction of the Gregorian calendar. It denotes a time extrapolated backwards from Gregorian times using Gregorian conventions and ignoring leap seconds.

#Adjust is intended to allow for time zone and "daylight saving" effects.

#### A.5.11.12 Config_Random_Seed

Notice that sections 5.11.12 and 5.11.13 apply to a program, so that if a program calls an external routine which is a program the two sequences of random numbers will be unrelated.

### A.5.13 Variable Pool

It is common existing practice for implementations to support a Variable Pool similar to that described in reference 2. Although debatable whether this is "Rexx language", it is valuable to have it standardized.

Direct symbols containing no periods are allowed although the same effects can be achieved using the functions that take ordinary symbols as arguments: this follows existing practice.

The intent of the API enabling mechanism (see #API_Enabled) is to prevent contexts where variables are altered in ways that are not apparent from a reading of the source program. Thus, for example, a trap of Config_Halt_Query cannot use API_Set.

#### A.5.13.9 API_NextVariable

It is not common existing practice to have a way to return the values of stems. However, if the interface is to be used for debugging and maintenance of programs it will be useful to be able to 'dump' all variables.

#### A.6.1.6 Syntactic errors

When a message is produced as a result of lexical or higher level syntax, any further activity is implementation defined. The implementation may for example:

– Produce or not produce a further message, possibly with a better indication of the position of the error.

– Produce or not produce messages relating to the rest of the program.

– Begin or not begin execution of the program.

While a program is being processed according to this standard, and has not finished, this standard defines the output through the Config_Say and Config_Trace interfaces; implementations should avoid directing extra output, for example implementation defined extra messages, to the same destinations.

#### A.6.2.2.34 Operator

For the benefit of maintainability, comments are not permitted in multi-character operators, for example */*...*/* is not an allowed form of the power operator.

#### A.6.2.3 Interaction between levels of syntax (Quoted labels)

Although it is not common existing practice to have labels which are quoted strings it is right to allow this, to be consistent with having function names that are quoted strings. Function names which are quoted strings may have special significance for the configuration, when the search for external functions is done. (See NameType in section 5.6.1)

#### A.6.2.3.1 Reserved symbols

Extending the set of special variables RC, RESULT, and SIGL is undesirable because of breakage. Something special is needed to improve the portability of handling errors when commands are issued. Possible future standards are expected to need other special symbols. The committee chose to make use of some symbols (such as .RS) and reserve other symbols beginning with period for these special purposes. This is a balance of benefit and loss, since some existing programs use these symbols and hence will need change if they are to be conforming.

#### A.6.2.3.2 Function name syntax

The purpose of this rule is to allow any future standard to specify that ABC.(K+1) is a reference to a compound variable with a tail which was the value of K+1, rather than an invocation of function ABC.. There is loss as well as benefit, since some existing programs may need to have altered function names, or names enclosed in quotes, if they are to become conforming.

#### A.6.4.1 VAR_SYMBOL matching

The control variable from the *do_specification* and the symbol from the *do_ending* may both be compound variables. The matching is done on the symbols, without consideration of the values that the tails might have at execution.

#### A.6.4.6 Creation of messages

This description also applies to messages resulting from #Raise activity.

Messages (and the output of tracing) are meant to be read by humans but they may contain arbitary data. The standard does not prevent the configuration altering the data from the program before presenting it to humans, for example changing all the 'unprintable' characters to question marks.

#### A.6.4.6.2 Replacement of insertions

Some insertions are not described here. The #Raise invocations leading to the relevant messages have these inserts as arguments.

### A.7 Evaluation

The need to qualify names in variable pools arises because, for example, the name derived from ABC.DEF when the value of DEF is a string with zero length might be confused with the stem name ABC..

#### A.7.4.3 The value of a power_expression

Note that for dyadic operations, the order of evaluation is:

- evaluation of the left hand side operand;
- evaluation the right hand side operand;
- checking of the left hand side operand;
- checking of the right hand side operand;
- evaluation of the operation.

#### A.7.4.7 The value of a comparison

Most REXX implementations have two forms of comparison, an 'exact' form which compares strings character by character and a form in which the comparison depends on whether the two strings are numbers. The latter comparisons are not transitive, which had led to proposals for a third form. In view of the fact that there is an idiom, a+0 < b+0 which can be written to force numeric comparison, extra comparison operators are not justified.

#### A.7.4.10 Arithmetic operations

Existing practice uses a rule that non-exponential notation is used when there is zero before the decimal point and the number of digits after the decimal point does not exceed twice the value of the NUMERIC DIGITS setting. This has proved unsatisfactory because numbers with a large number of consecutive zero digits are easily misread. This standard limits the leading fractional zeros to five.

Existing practice uses a rule, in some contexts, that an argument which is required to be an integer may be provided as a value with non-zero digits after the decimal point, provided rounding removed such digits. This standard requires the original argument to be without non-zero digits after the decimal point.

In general REXX arithmetic follows the everyday rules of arithmetic, and the availability of high precision is sometimes valuable. However, with any arithmetic scheme there can be anomalies when the numbers used are at the limits; the effects of rounding and guard digits are not always those that might be intuitively expected. This can happen in REXX when numbers are over-precise, that is when the character form of the number contains more digits than the current setting of NUMERIC DIGITS. Such numbers cannot occur as the result of arithmetic (at that setting of NUMERIC DIGITS) but can be the result of character operations.

The committee looked at the prospects for removing the anomalies by suitable choice of arithmetic rules — such change would not have affected many existing programs since the use of over-precise numbers is rare. However, no way of avoiding the anomalies was found. Programs with over-precise numbers are not (necessarily) in error, but the programmer may want to be advised when they occur. The committee chose to add the LOSTDIGITS condition; most arithmetic anomalies do not happen when it is zeros that are truncated.

The intent of NUMERIC DIGITS is to allow increased precision where it is required. It will rarely be of value to reduce the precision below the default setting, and could be unsatisfactory if, for example, the length of a string could not be represented without exponential notation.

#### A.8.2.1 Program Initialization

The default value of NUMERIC DIGITS for arithmetic (as opposed to the checking of built-in function arguments) is nine. It is nine irrespective of the configuration since it is chosen to make the layout of numbers appropriate for humans to read, rather than being matched to hardware characteristics.

Message numbers have an integer part which is referred to as the error code and a decimal part which is the error subcode. No trailing zeros are written on a subcode since, for example, 8.1 and 8.10 are the same message number. This standard is not entitled to specify what message numbers a non-conforming implementation should use but the intent is for this standard (and any future standards) to use subcodes less than or equal to .9. Thus a subcode with leading 9 and further digits in a non-conforming implementation would highlight a message not matching any message in this standard. Analogously, this standard will use error codes less than 91.

#### A.8.2.4 Clause termination

The polling of the configuration by Config_Trace_Query and Config_Halt_Query is defined as occurring at clause termination. However, this says nothing about responsiveness to any button pushing because that relation is a matter for the configuration to decide. The intent is that there should never be a long delay from the user's perspective.

In interactive trace, some clauses do not have an associated pause. Note in particular that when conditions are raised the condition handling may transfer control so that the current clause does not reach clause termination and hence does not pause. Note also that most expressions on a DO specification are not re-evaluated on each iteration of the loop so the corresponding clause is not paused after, except on the first iteration.

The standard, like the base reference, specifies that when interactive input is being intepreted all conditions are temporarily disabled. The benefit of avoiding the confusing handling of conditions outweights the fact that some routines invoked from interactive input will not behave as intended — for example those that use NOTREADY rather CHARS(Stream)=0 as a test for end-of-file.

### A.8.3.1 ADDRESS

There is diversity in common practice about how to communicate data between a REXX program and the commands it issues. It would be valuable to establish a more common practice for this communication, even where the commands themselves are not portable. The redirection features of ADDRESS have been added to provide a goal for this common practice. However, since the environment determines what data it regards as belonging to its data streams, there may be no support for indirection beyond the ADDRESS built-in and recognising the ADDRESS instruction in full.

The programmer can check whether the language processor and specific environment allow redirection by executing ADDRESS with the redirection specified and then testing for ADDRESS built-in functions results of 'UNUSED'.

In addition to STEM and STREAM, a STACK or QUEUE option could have been added. However, existing practice is that redirection to a queue is done by altering the actual command issued, rather than altering the environment in which it is issued.

The environment, not this standard, determines what data the environment regards as the input stream, output stream, and error output stream for a command.

The allowed syntax for specifying indirections specifies all indirections (INPUT, OUTPUT, ERROR) when it specifies any indirection. An incremental method of specifying indirection could have been added, using an instruction beginning ADDRESS WITH, but this would have been complicated.

The RC special variable is intended to provide feedback on the processing of a command in terms that are relevant to the configuration, and hence can be looked up in the documentation of the configuration. The RC cannot be relied on as an indicator of success or failure.

The .RS reserved symbol is a reflection of whether a command gave rise to ERROR and FAILURE conditions. Although ERROR and FAILURE are themselves conditions determined by the configuration, a program that tests .RS is showing the intention to test for unsuccessful command execution.

It is intended that an environment name should uniquely identify the destination for a command, so blanks and upper/lower case are significant in the environment name.

### A.8.3.3 Assignment

Note that tails in the left hand side of assignment are accessed after evaluation of the right hand side.

### A.8.3.10 INTERPRET

If there are multiple clauses in the string to be interpreted they must be separated by semicolons rather than whatever the configuration uses as an end-of-line indicator.

### A.8.3.16 OPTIONS

The intent of the OPTIONS instruction is to pass a series of blank-delimited words to the language processor. Since a conforming processor cannot use this to over-ride the specifications of this standard, this standard specifies that there is no effect. However, a conforming processor may use the words to alter factors that are outside the scope of this standard, for example performance.

### A.8.3.26 TRACE

The letters allowed as a trace setting are the initial letters of modes of tracing known as 'All', 'Command', 'Errors', 'Failures', 'Intermediates', 'Labels', 'Normal', 'Off', and 'Results'.

The single letters that appear in tags correspond to variables, literals, function invocations, compound variable substitutions, prefix operations, and other operations.

#### A.8.3.26.1 Trace output

In current practice, trace output has considerable differences in layout on different implementations, and some differences of information content. The committee made a number of decisions in tightening the specification of trace. The presence of line numbers on the trace data (as well as on the trace of source) avoids the possibility of successive outputs from TRACE 'Intermediates' without line numbers to distinguish the (differing) source instructions that they emanated from.

### A.8.4 Conditions and messages

The original implementation of REXX was on a configuration where minimization of the amount of message text was important. It is nowadays appropriate to have a wider variety of messages since some messages, particularly message 40, have proved not to be specific enough.

To add more messages in a way that changed the number produced for a particular error would produce 'breakage' since existing programs do computation with the number. The introduction of a 'subcode' as the fractional part of the error number allows extra discrimination while leaving the integer part in accord with reference 1.

Maintainability considerations are also the cause for extra inserts in messages. Some of these inserts tailor the messages to the configuration, others reflect the cause of a particular instance of the message.

The message number 49 is retained and can be the result of the ERRORTEXT built-in function. A processor which produced a SYNTAX condition 49 would not be conforming to this standard.

Error 2 and Error 3 cannot be trapped by a handler of the SYNTAX condition because no handler could be established at the time these errors occur.

#### A.8.4.1 Raising of conditions

The standard defines what happens when a condition is delayed and a function invokation subsequently takes place in the same clause. The standard specifies that the function can re-enable for another occurrence of the same condition; there is potential for any number of occurrences of the same condition to be delayed at one time. Note that delayed conditions are raised at the end of the clause in which they occurred.

#### A.8.4.2 Messages during execution

The standard does not describe any splitting of the text into shorter lines which might be desirable before presentation of the message to the user, since this is a matter for the configuration to decide.

## A.9 Built-in functions

Some features have been included in the built-in functions which are not in all current implementations - TIME and DATE conversions, CHANGESTR and COUNTSTR. Defining these makes it less likely that implementations will provide these features (for which there is some current practice) in differing ways.

#### A.9.3.8 The DATATYPE built-in function

The SYMBOL built-in is sensitive to the extra_letter characters. Except where it refers to SYMBOL, the DATATYPE built-in is not sensitive to the extra letters. For example DATATYPE(x,'U') refers to the 26 letters 'A' through 'Z' only.

### A.9.7 I/O built-in functions

If STREAM(Stream, 'S') returns 'ERROR' this is a reference to the latest read or write operation on the Stream. The results from a CONDITION built-in within a handler of the NOTREADY condition refer to an event that was not raised when it occurred but was delayed until a clause boundary. Hence these STREAM and CONDITION built-ins do not necessarily refer to the same event.

#### A.9.8.4 The SYMBOL built-in function

The SYMBOL built-in is sensitive to the extra_letter characters and also to any configuration limit on the length of symbols.

#### A.9.8.5 The TIME built-in function

Timestamps denoting dates prior to introduction of the Gregorian calendar are treated as if the Gregorian calendar had been introduced 0001 01 01 00:00:00.000000 and do not represent valid dates.

The Gregorian calendar is off by one day for every 3000 years. Although the computation is guaranteed up to the end of 9999, most likely there will be a calendar reform earlier (most likely the year 3200 will not be a leap year), which would require modification of the algorithm.

TIME('Offset') is provided so that a program can construct a "time stamp" which is monotonic even when the time offset changes.

# Annex B — Method of definition (informative)

This annex describes the methods chosen to describe REXX, for this standard.

The numbering in this appendix reflects the numbering of the normative part of this standard, for example section B.4.1 explains section 4.1.

## B.3 Definitions

Definitions are given for some terms which are both used in this standard and also may be used elsewhere. This does not include names of syntax constructions, for example *group* which are distinguished in this standard by the use of italic font.

## B.4.1 Conformance

Note that irrespective of how this standard is written, the obligation on a conforming processor is only to achieve the defined results, not to follow the algorithms in this standard.

## B.5.1 Notation

The notation used to describe functions provided by the configuration is like a REXX function call but it is not defined as a REXX function call since a REXX function call is described in terms of one of these configuration functions.

Note that the mechanism of a returned string with a distinguishing first character is part of the notation used in this standard to explain the functions; implementations may use a different mechanism.

### B.5.1.1 Notation for completion response and conditions

The testing of 'X' and 'S' indicators is made implicit, for brevity. Even when written as a subroutine call, each use of a configuration routine implies the testing. Thus:

```
call Config_Time
```

implies

```
#Response = Config_Time()
if left(#Response,1) == 'X' then call #Raise 'SYNTAX', 5.1, substr(#Response,2)
if left(#Response,1) == 'S' then call #Raise 'SYNTAX', 48.1, substr(#Response,2)
```

## B.5.3 Source programs and character sets

The characters required by REXX are identified by name, with a glyph associated so that they can be printed in this standard. Alternative names are shown as a convenience for the reader.

## B.5.8.1 Config_Stream_Charin

The purpose of the OperatorType operand on Config_Stream_Charin is primarily to identify the built-in function which is reading characters.

## B.6.1 Notation

Note that section 6.1 is not specifying the syntax of a program; it is specifying the notation used in this standard for describing syntax.

## B.6.2.2 Lexical level

Productions 6.2.2.17 and 6.2.2.18 contain a recursion of *comment*. Apart from this recursion, the lexical level is a finite state automaton.

## B.6.3.2 Syntax level

This syntax shows a null_clause list, which is minimally a semicolon, being required in places where programmers do not normally write semicolons, for example after 'THEN'. This is because the 'THEN' implies a semicolon. This approach to the syntax was taken to allow the rule 'semicolons separate clauses' to define 'clauses'.

The precedence rules for the operators are built into this grammar.

## B.7 Evaluation (Definitions written as code)

There is no single definitional mechanism for describing semantics that is predominantly used in standards describing programming languages, except for the use of prose. The committee has chosen to define some parts of this standard using algorithms written in Rexx. This has the advantages of being rigorous and familiar to many of the intended readers of this standard. It has the potential disadvantage of circularity - a definition based on an assumption that the reader already understands what is being defined.

Circularity has been avoided by:

– Specifying the language incrementally, so that the algorithms for more complex details are specified in code that uses only more simple Rexx. For example, the notion that an expression evaluates to a result can be understood by the reader even without a complete specification of all operators and built-in functions that might be used in the expression.

– Specifying the valid syntax of Rexx programs without using Rexx coding. The method used, Backus Normal Form, can adequately be introduced by prose.

Ultimately some understanding of programming languages is assumed in the reader (just as the ability to read prose is assumed) but any remaining circularity in this standard is harmless.

The comparison of two single characters is an example of such a circularity; Config_Compare can compare two characters but the outcome can only be tested by comparing characters. It has to be assumed that the reader understands such a comparison.

Some of the definition using code is repeating earlier definition in prose. This duplication is to make the document easier to understand when read from front to back.

Note that the layout of the code, in the choices of instructions-per-line, indentations etc., is not significant. (The layout style used follows the examples in the base reference and it is deliberate that the DO and END of a group are not at the same alignment.)

The code is not intended as an example of good programming practice or style.

The variables in this code cannot be directly referenced by any program, even if the spelling of some VAR_SYMBOL coincides. These variables, referred to as state variables, are referenced throughout this document; they are not affected by any execution activity involving scopes. Some of more significant variables and routines are written with # as their first character. The following list of them is intended as an aid to understanding the code. The index of this standard shows the main usage, but not all usage, of these names.

The following are constants set by the configuration, by Config_Constants:

**#Configuration** is used for PARSE SOURCE.

**#Version** is used for PARSE VERSION.

**#Bif_Digits.** represents numeric digits settings, tails are built-in function names.

**#Limit_Digits** is the maximum significant digits.

**#Limit_EnvironmentName** is a maximum length.

**#Limit_ExponentDigits** is the maximum digits in an exponent.

**#Limit_Literal** is a maximum length.

**#Limit_MessageInsert** is a maximum length.

**#Limit_Name** is a maximum length.

**#Limit_String** is a maximum length.

**#Limit_TraceData** is a maximum length.

These are named outputs of configuration routines:

**#Response** is used to hold the result from a configuration routine.

**#Indicator** is used to hold the leftmost character of Response.

**#Outcome** is the main outcome of a configuration routine.

**#RC** is set by Config_Command.

**#NoSource** is set by Config_NoSource.

**#Time** is set by Config_Time

**#Adjust** is set by Config_Time

These variables are set up with output from configuration routines:

**#HowInvoked** records from API_Start, for use by PARSE SOURCE.

**#Source** records from API_Start for use by PARSE SOURCE.

**#AllBlanks** is a string including Blank and equivalents.

**#ErrorText.MsgNumber** is the text as altered by limits.

**#SourceLine.** is a record of the source, retained unless NoSource is set. #SourceLine.0 is a count of lines.

These are variables not initialized from the configuration:

**#Level** is a count of invocation depth, starting at one.

**#NewLevel** equals #Level plus one.

**#Pool** is a count of PROCEDURE invocation depth, starting at one.

**#Loop** is a count of loop nesting.

**#LineNumber** is the line number of the current clause.

**#Symbol** is a symbol after tails replacement.

**#API_Enabled** determines when the application programming interface for variable pools is available.

**#Test** is the Greater/Lesser/Equal result.

**#InhibitPauses** is a numeric trace control.

**#InhibitTrace** is a numeric trace control.

**#AtPause** is on when executing interactive input.

**#AllowProcedure** provides a check for the label needed before a procedure.

**#DatatypeResult** is a by-product of DATATYPE().

**#Condition** is a condition, eg ‘SYNTAX’.

**#Trace_QueryPrior** detects an external request for tracing.

**#TraceInstruction** detects TRACE as interactive input.

These are variables that are per-Level, that is, have #Level as a tail component:

**#IsFunction.** indicates a function call.

**#IsProcedure.** indicates indicates the routine is a procedure.

**#Condition.** indicates whether the routine is handling a condition.

**#ArgExists.#Level.ArgNumber** indicates whether an argument exists. (Initialized from API_Start for Level=1)

**#Arg.#Level.ArgNumber** provides the value of an argument. (Initialized from API_Start for Level=1) When ArgNumber=0 this gives a count of the arguments.

**#Tracing.** is the trace setting letter.

**#Interactive.** indicates when tracing is interactive. ('?' trace setting)

**#ClauseLocal.** ensures that DATE/TIME are consistent across a clause.

**#ClauseTime.** is the TIME/DATE frozen for the clause.

**#StartTime.** is for ‘Elapsed’ time calculations.

**#Digits.** is the current numeric digits.

**#Form.** is the current numeric form.

**#Fuzz.** is the current numeric fuzz.

These are qualified by #Condition as well as #Level:

**#Enabling.** is 'ON', 'OFF' or 'DELAYED'.

**#Instruction.** is 'CALL' or 'SIGNAL'

**#TrapName.** is the label.

**#ConditionDescription.** is for CONDITION('D')

**#ConditionExtra.** is for CONDITION('E')

**#ConditionInstruction.** is for CONDITION('I')

**#PendingNow.** indicates a DELAYED condition.

**#PendingDescription.** is the description of a DELAYED condition.

**#PendingExtra.** is the extra description for a DELAYED condition.

**#EventLevel.** is the #Level at which an event was DELAYED.

These are qualified by ACTIVE, ALTERNATE, or TRANSIENT as well as #Level:

**#Env_Name.** is the environment name.

**#Env_Type.** is the type of a resource, and is additionally qualified by input/output/error distinction.

**#Env_Resource.** is the name of a resource, and is additionally qualified by input/output/error distinction.

**#Env_Position.** is INPUT or APPEND or REPLACE, and is additionally qualified by input/output/error distinction.

These are variables that are per-loop:

**#Identity.** is the control variable.

**#Repeat.** is the repetition count.

**#By.** is the increment.

**#To.** is the limit.

**#For.** is that count.

**#Iterate.** holds a position in code describing DO instruction semantics.

**#Once.** holds a position in code describing DO instruction semantics.

**#Leave.** holds a position in code describing DO instruction semantics.

These are variables that are per-stream:

**#Charin_Position.**

**#Charout_Position.**

**#Linein_Position.**

**#Lineout_Position.**

**#StreamState.** records ERROR state for return by STREAM built-in function.

These are commonly used  prefixes:

**Config_** is used for a function provided by the configuration.

**API_** is used for an application programming interface.

**Trap_** is used for a routine called from the processor, not provided by it.

**Var_** is used for the routines operating on the variable pools.

These are notation routines, only available to code in this standard:

**#Contains** checks whether some construct is in the source.
**#Instance** returns the content of some construct in the source.
**#Evaluate** returns the value of some construct in the source.
**#Execute** causes execution of some construct found in the source.
**#Parses** checks whether a string matches some construct.
**#Clause** notes some position in the code.
**#Goto** continues execution at some noted position.
**#Retry** causes execution to continue at a previous clause.

These are frequently used routines:

**#Raise** is a routine for condition raising.
**#Trace** is a routine for trace output.
**#TraceSource** is a routine to trace the trace the source program.
**#CheckArgs** processes the arguments to a built-in function.

# Annex C — Bibliography (informative)

Cowlishaw, Mike F, *The Rexx Language*, Prentice Hall, Englewood Cliffs, N.J. 07632; second edition 1990. ISBN 0-13-780651-5

*Systems Application Architecture Common Programming Interface Rexx Level 2 Reference,* IBM SC24-5549, second edition 1992

## Index