Extended Standard Programming Language Rexx

Rexx Language ARB

12 Apr 2024

THE REXX LANGUAGE ASSOCIATION
RexxLA Symposium Proceedings Series
ISSN 1534-8954

Publication Data

A publication of **RexxLA Press**

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

The RexxLA Symposium Series is registered under ISSN 1534-8954
The 2023 edition is registered under ISBN 978-90-819090-1-3

ISBN 978-90-819090-1-3

2023-03-31 First printing

Contents

1

Foreword

1.1 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.

1.2 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.

In November, 1990, X3 announced the formation of a new technical committee, X3J18, to develop an American National Standard for Rexx. This standard was published as ANSI X3.274-1996.

The popularity of "Object Oriented" programming, and the need for Rexx to work with objects created in various ways, led to Rexx extensions and to a second X3J18 project which produced this standard. *(Ed - hopefully)*

Committee lists *(Here)*

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

There are annexes in this standard; they 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, NCITS. 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 NCITS Committee had the following members:

To be inserted

The people who contributed to Technical Committee J18 on Rexx, which developed this standard, include:

2

Introduction

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

3

Scope, purpose, and application

3.1 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.

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;
- the syntax of the configuration interfaces.

3.2 Purpose

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

3.3 Application

This standard is applicable to Rexx language processors.

3.4 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 the annexes should be read in conjunction with this standard.

4

Normative references

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

5

Definitions and document notation

Lots more for NetRexx

5.1 Definitions

application programming interface A set of functions which allow access to some Rexx facilities from non-Rexx programs.

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.

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

character string A sequence of zero or more characters.

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.

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`.

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.

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

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

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

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

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

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

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

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

directive Clauses which begin with two colons are directives. Directives are not executable, they indicate the structure of the program. Directives may also be written with the two colons implied.

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'.

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

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.

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.

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.

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.

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.

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

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.

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.

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

function Some processing which can be invoked by name and will produce a result. This term is used for both Rexx functions (See nnn) and functions provided by the configuration (see n).

identifier The name of a construct.

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'.

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

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

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.

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

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

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.

null clause A clause which has no tokens.

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

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

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.

routine Some processing which can be invoked by name.

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.

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.

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.

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.

subcode The decimal part of an error number.

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.

symbol A sequence of characters used as a name, see nnn. Symbols are used to name variables, functions, etc.

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.

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

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

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

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

5.2 Document notation

5.2.1 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.

5.2.2 Italics

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

6

Conformance

6.1 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 X3.274-1996) with which conformity is claimed.

6.2 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.

7

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 clause describes what they shall together do to provide the configuration for the Rexx language processing which is described in this standard.

We don't want to add undue "magic" to this section. It seems we will need the concept of a "reference" (equivalent to a machine address) so that this section can at least have composite objects as arguments. (As it already does but these are not Rexx objects)

Possibly we could unify "reference" with "variable pool number" since object one-to-one with its variable pool is a fair model. That way we don't need a new primitive for comparison of two references.

JAVA is only a "reference" for NetRexx so some generalized JAVA-like support is needed for that. It would provide the answers to what classes were in the context, what their method signatures were etc.

7.1 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 clause and nnn.

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.

7.1.1 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 nnn.
- 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.

7.2 Processing initiation

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

We could do REQUIRES in a macro-expansion way by adding an argument to Config_SourceChar to specify the source file. However, I'm assuming we will prefer to recursively "run" each required file. One of the results of that will be the classes and methods made public by that REQUIRES subject.

7.2.1 API Start

Syntax:

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

where:

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

What does OOI say for How when running REQUIREd files?

- `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 streams to be used.
- `Traps` is the list of traps to be used in processing (see nnn). 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.

If How was `'REQUIRED'` and the completion response was not `'E'`, the `Provides` argument is set to reference classes made available. See nnn for the semantics of these classes.

7.3 Source programs and character sets

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

7.3.1 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 bracket, left square bracket;
- (left parenthesis;
- % percent sign;
- . period, decimal point, full stop, dot;
- • plus sign;
- ? question mark;
- ” quotation mark, double quote;
- reverse slant, reverse solidus, backslash;
-] right bracket, right square bracket;
-) right parenthesis;
- ; semicolon;
- / slant, solidus, slash;
- a-z small letters a through z;
- ~ tilde, twiddle;
- _ underline, low line, underscore;
- vertical line, bar, vertical bar.

7.3.2 Extra_letters

A configuration may have a category of characters in source programs called *extra_-letters*. *Extra_letters* are determined by the configuration.

7.3.3 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.

7.3.4 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.

7.3.5 Other_characters

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

7.4 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`.

7.4.1 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.

7.4.2 Config_OtherBlankCharacters

Syntax:

```
call Config OtherBlankCharacters
#A11Blanks<Index "#Al1Blanks" # "" > = ' '#Outcome /* "Real" blank
   concatenated with
others */
#Bif Digits. = 9
call Config Constants
.true = '1'
.false = '0'
```

Semantics:

Get *other_blank_characters* (see nnn).

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

7.4.3 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.

7.4.4 Config_Lower

Syntax:

Config_Lower(Character)

where:

Character is the character to be translated to lowercase.

Semantics:

Translate Character to lowercase. Set #Outcome to the translated character. Characters which have been subject to this translation are referred to as being in lowercase. Config_-Lower applied to a character in lowercase must not change the character. Config_Upper of the outcome of Config_Lower(Character) shall be the original character.

7.4.5 Config_Compare

Syntax:

Config Compare(Characterl, 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 Character' 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';
- 'equal' then Config_Compare(a,c) and Config_Compare(b,c) produce the same value.
- Syntactic characters which are different characters shall not compare equal by Config_Compare, see nnn.

7.4.6 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.

7.4.7 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.

7.4.8 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.

7.4.9 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.

7.4.10 Config_Xrange

Syntax:

Config Xrange(Characterl, 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 a 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 a highest ranked character in the character set according to the ranking order provided by Config_Compare; otherwise let HighBound be Character2lf #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.

7.5 Objects

The objects interface consists of functions which the configuration shall provide for creating objects.

7.5.1 Config_ObjectNew

Syntax:

Config ObjectNew

Semantics:

Set #Outcome to be a reference to an object. The object shall be suitable for use as a variable pool, see nnn. This function shall never return a value in #Outcome which compares equal with the value returned on another invokation of the function.

7.5.2 Config_Array_Size

Syntax:

Config Array Size(Object, size)

where:

Object is an object.

Size is an integer greater or equal to 0.

Semantics:

The configuration should prepare to deal efficiently with the object as an array with indexes having values up to the value of size.

7.5.3 Config_Array_Put

Syntax:

Config Array Put(Array, Item, Index)

where:

Array is an array.

Item is an object

Index is an integer greater or equal to 1.

Semantics:

The configuration shall record that the array has Item associated with Index.

7.5.4 Config_Array_At

Syntax:

Config Array At(Array, Index)

where:
Array is an array.
Index is an integer greater or equal to 1.

Semantics:

The configuration shall return the item that the array has associated with Index.

7.5.5 Config_Array_Hasindex

Syntax:

Config Array At(Array, Index)
where:
Array is an array.
Index is an integer greater or equal to 1.

Semantics:

Return '1' if there is an item in Array associated with Index, '0' otherwise.

7.5.6 Config_Array_Remove

Syntax:

Config Array At(Array, Index)
where:
Array is an array.
Index is an integer greater or equal to 1.

Semantics:

After this operation, no item is associated with the Index in the Array.

7.6 Commands

The commands interface consists of a function which the configuration shall provide for strings to be passed as commands to an environment.
See nnn and nnn for a description of language features that use commands.

7.6.1 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 onto 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 onto which the command will write its error 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.

7.7 External routines

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

See nnn and nnn for a description of the language features that use external routines.

7.7.1 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 of the completion response 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 nnn) 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.

7.7.2 Config_ExternalMethod

OOI has external classes explicitly via the ::CLASS abc EXTERNAL mechanism. Analogy with classic would also allow the subject of ::REQUIRES to be coded in non-Rexx. However ::REQUIRES subject is coded, we need to gather in knowledge of its method names because of the search algorithm that determines which method is called. Hence reasonable that the ultimate external call is to a method. Perhaps combine Config_-ExternalRoutine with Config_ExternalMethod.

There is a terminology clash on "environment". Perhaps easiest to change the classic to "address_environment". (And make it part of new "environment"?)

There are terminology decisions to make about "files", "programs", and "packages". Possibly "program" is the thing you run (and we don't say what it means physically), "file" is a unit of scope (ROUTINEs in current file before those in REQUIREd), and "package" we don't use (since a software package from a shop would probably have several files but not everything to run a program.) Using "file" this way may not be too bad since we used "stream" rather than "tile" in the classic definition.

The How parameter will need 'METHOD' as a value. Should API_Start also allow 'METHOD". If we pass the new Environment we don't have to pass Streams separately.

Text of Config_ExternalMethod waiting on such decisions.

Syntax:

Config ExternalMethod (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 of the completion response 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 nnn) 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.

7.8 External data queue

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

See nnn, nnn, nnn, nnn, and nnn 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.

7.8.1 Config Push

Syntax:

Config Push(String)

where:

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

Semantics:

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

7.8.2 Contig_Queue

Syntax:

Config Queue (String)

where:

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

Semantics:

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

7.8.3 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'.

7.8.4 Contig_Queued

Syntax:

Config Queued ()

Semantics:

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

7.9 Streams

The streams interface consists of functions which the configuration shall provide to manipulate streams. See nnn, nnn, and nnn 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.

7.9.1 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 nnn.

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.

7.9.2 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 'LINEOQUT' and the Position is more than one beyond the limit of existing 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'.

7.9.3 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.

7.9.4 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.

7.9.5 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'.

7.9.6 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.

7.9.7 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.

7.9.8 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, or to 'C' if it is a character stream.

7.9.9 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'.

7.9.10 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 nonzero, #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 nonzero, #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.

7.10 External variable pools

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

See nnn 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.

7.10.1 Config Get

Syntax:

Config Get (Poolid, Name)

where:

Poolid is an 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'.

7.10.2 Config Set

Syntax:

Config Set (Poolid, Name, Value)

where:

Poolid is an 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'.

7.11 Configuration characteristics

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

7.11.1 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 comprise a date. This shall be in the format which is the default for the DATE() built-in function.

- set .nil to a value which compares unequal with any other value that can occur in execution.
- set .local .kernel .system?

7.12 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.

7.12.1 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'.

7.12.2 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.

7.12.3 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.

7.12.4 Config _Default_Input

Syntax:

Config Default Input ()

Semantics:

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

7.12.5 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).

7.12.6 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.

7.12.7 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.

7.12.8 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'.

7.12.9 Config _Halt_Reset

Syntax:

Config Halt Reset ()

Semantics:

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

7.12.10 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'.

7.12.11 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<Index "#Adjust" #" "> to an integer number of microseconds. #Adjust<Index "#Adjust" # ™ > reflects the difference between the local date/time and the date/time corresponding to #Time. #Time + #Adjust<Index "#Adjust" # " "> is the local date/time.

7.12.12 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 return quasi-random numbers.

7.12.13 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 nonnegative integer in the range Min to Max.

7.12.14 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 have come from an OPTIONS instruction, see nnn.

7.13 Traps

The trapping interface consists of functions which may be provided by the caller of API_-Start (see nnn) 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 beginning Trap_ will correspond to a name beginning 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.

7.14 Variable pool

How does this fit with variables as properties?

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 nnn.

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. Clause nnn 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 nnn. 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.

7.14.1 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 nnn.

7.14.2 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 nnn.

7.14.3 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 nnn.

7.14.4 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)

7.14.5 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')

7.14.6 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')

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

7.14.8 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.

7.14.9 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.

8

Syntax constructs

8.1 Notation

8.1.1 Backus-Naur Form (BNF)

The syntax constructs in this standard are defined in Backus-Naur 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.

8.1.2 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 nnn, nnn, nnn, nnn.

Symbolic terminals formed with uppercase characters represent events and tokens, see nnn and nnn.

8.1.3 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.

8.1.4 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.

8.1.5 BNF syntax definition

The BNF syntax, described in BNF, is:

```
production  :=    identifier ':=' bnf_expression
bnf_expression    := abuttal | bnf_expression '|' abuttal
abuttal     :=    [abuttal] bnf primary
bnf_primary :=    '[' bnf expression ']' | '(' bnf expression ')' |
   literal |
identifier | message identifier | bnf primary '+'
```

8.1.6 Syntactic errors

The syntax descriptions (see nnn and nnn) 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 nnn and nnn).

8.2 Lexical

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

8.2.1 Lexical elements

Events

The fully-capitalized identifiers in the BNF syntax (see nnn) 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 nnn;
- *EOS* occurs at the end of the source program. 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 `'.'`.

- I would put ASSIGN *here for the leftmost `'='` in a clause that is not within parentheses or brackets. But Simon not happy with message term being an assignment?*

Actions and tokens

Mixed case identifiers with an initial capital letter cause an action when they appear as operands ina 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 nnn. The characters in a Var_symbol are converted by Config_Upper to uppercase. Msg30.1 shall be produced if Var_-symbo/ contains more than #Limit_Name characters, see nnn;
- Const_symbol supplies the source recognized as Const_symbo! to the top syntax level. If itis 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_symbo! 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 or 4 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 nnn. 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. 0 to 7 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 ofher_negator within Operator is supplied as '';
- Blank records the presence of a blank. This may subsequently be tested (see nnn). 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 nnn). 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 nnn),

other_blank_characters (see nnn), other_negators (see nnn) and other_characters (see nnn), respectively.

Rules

In scanning, recognition that causes an action (see nnn) only occurs if no other recognition is possible, except that Embedded_apostrophe and Embedded_quotation_mark actions occur wherever possible.

8.2.2 Lexical level

8.2.3 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 ina 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 or message_instruction in the corresponding top level context then it is recognized as the '=“' of that instruction. (It will be outside of brackets and parentheses, and any Var_symbo/ immediately 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. In a context where the top level syntax could accept a '||' token as the next token, a'||' operator ora' ' 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 (see nnn); 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.

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.

Function name syntax

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

8.3 Syntax

8.3.1 Syntax elements

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

8.3.2 Syntax level

```
starter:=x3j18
x3j18:=program Eos | Msg35.1
program   := [label_list] [ncl] [requires+] [prolog_instruction+]
                 (class definition [requires+])+
  requires          :=  'REQUIRES' ( taken constant | Msg19.8 ) ';'+
  prolog_instruction:= (package | import | options) ncl
    package         := 'PACKAGE'( NAME | Msgnn )
    import          := 'IMPORT' ( NAME | Msgnn ) ['.']
    options         := 'OPTIONS' ( symbol+ | Msgnn )
    ncl   := null_clause+ | Msg21.1
      null_clause   := ';' [label_list]
         label_list = (LABEL ';')+
class_definition    := class [property_info] [method_definition+]
  class             := 'CLASS' ( taken_constant | Msg19.12 ) [
     class_option+]
                        ['INHERIT' ( taken_constant | Msg19.13 )+] ncl
  class_option      := visibility | modifier | 'BINARY' | 'DEPRECATED
     '
                     | 'EXTENDS' ( NAME | Msgnn )
                     | 'USES' ( NAMElist | Msgnn )
                     | 'IMPLEMENTS' ( NAMElist | Msgnn )
                     | external | metaclass | submix /* | 'PUBLIC' */
    external        := 'EXTERNAL' (STRING | Msg19.14)
    metaclass       := 'METACLASS' ( taken_constant | Msg19.15 )
    submix          := 'MIXINCLASS' ( taken_constant | Msg19.16 )
                       | 'SUBCLASS' ( taken_constant | Msg19.17 )
    visibility      := 'PUBLIC' | 'PRIVATE'
    modifier        := 'ABSTRACT' | 'FINAL' | 'INTERFACE' | 'ADAPTER'
       !
    NAMElist        := NAME [(',' ( NAME | Msgnn ) )+]
  property_info     := numeric | property_assignment | properties |
     trace
    numeric         := 'NUMERIC' (numeric_digits | numeric_form |
       Msg25.15)
      numeric_digits:= 'DIGITS' [expression]
      numeric_form  := 'FORM' ['ENGINEERING' | 'SCIENTIFIC']
  property_assignment := NAME | assignment
  properties        := 'PROPERTIES' ( properties_option+ | Msgnn)
    properties_option := properties_visibility | properties_modifier
      properties_visibility := 'INHERITABLE' | 'PRIVATE' | 'PUBLIC' |
'INDIRECT'
      properties_modifier := 'CONSTANT' | 'STATIC' | 'VOLATILE' | '
         TRANSIENT'
  trace             := 'TRACE' ['ALL' | 'METHODS' | 'OFF' | 'RESULTS'
     ]
method_definition   := (method [expose ncl]| routine)
                        balanced
  expose            := 'EXPOSE' variable_list
  method            := 'METHOD' (taken_constant | Msg19.9)
                       [ '(' assigncommalist | Msgnn ( ')' | Msgnn )]
                       [method_option+] nel
    assigncommalist := assignment [',' ( assignment | Msgnn ) )+]
    method_option   := method_visibility | method_modifier | 'PROTECT
       '
                     | 'RETURNS' ( term | Msgnn )
                     | 'SIGNAL' ( termcommalist | Msgnn )
```

```
                    | 'DEPRECATED'
                    | 'CLASS' | 'ATTRIBUTE' | /*'PRIVATE' | */
                      guarded
        guarded     := 'GUARDED' | 'UNGUARDED!
      method_visibility := 'INHERITABLE' | 'PRIVATE' | 'PUBLIC' | '
         SHARED'
      method_modifier := 'ABSTRACT' | 'CONSTANT' | 'FINAL' | 'NATIVE'
          |
'STATIC'
      termcommalist := term [(',' ( term | Msgnn ) )+]
  routine           := 'ROUTINE' ( taken constant | Msg19.11 ) ['
     PUBLIC'] ncl
  balanced:= instruction_list ['END' Msgl0.1]
    instruction_list:= instruction+

/* The second part is about groups */

instruction       := group | single_instruction ncl
group             := do ncl | if | loop ncl | select ncl
  do              := do_specification ncl [instruction+] [
     group_handler]
                    ('END' [NAME] | Eos Msg14.1 | Msg35.1)
    group_option  := 'LABEL' ( NAME | Msgnn ) | 'PROTECT' ( term |
       Msgnn )
    group_handler := catch | finally | catch finally
      catch       := 'CATCH' [ NAME '=' ] ( NAME | Msgnn) ncl [
         instruction+]
/* FINALLY implies a semicolon. */
      finally     := 'FINALLY' ncl ( instruction+ | Msgnn )
  if              := 'IF' expression [ncl] (then | Msg18.1)
                  [else]
    then          := 'THEN' ncl
                  (instruction | EOS Msg14.3 | 'END' Msg10.5)
    else          := 'ELSE' nel
                  (instruction | EOS Msg14.4 | 'END' Msg10.6)
  loop            := 'LOOP' [group_option+] [repetitor] [conditional]
      ncl
                     instruction+ [group_handler]
                     loop_ending
    loop_ending   := 'END' [VAR SYMBOL] | EOS Msg14.n | Msg35.1
  conditional     := 'WHILE' whileexpr | 'UNTIL' untilexpr
    untilexpr     := expression
    whileexpr     := expression
  repetitor       := assignment [count option+] | expression | over |
'FOREVER'
    count_option  := loopt | loopb | loopf
      loopt       := 'TO' expression
      loopb       := 'BY' expression
      loopf       := 'FOR' expression
    over          := VAR_SYMBOL 'OVER' expression
                  | NUMBER 'OVER' Msg31.1
                  | CONST_SYMBOL 'OVER' (Msg31.2 | Msg31.3)
select            := 'SELECT' [group_option+] ncl select_body [
   group_handler]
                  ('END' [NAME Msg10.4] | EOS Msgl14.2 | Msg7.2)
    select_body   := (when | Msg7.1) [when+] [otherwise]
```

```
      when        := 'WHEN' expresgion [ncl] (then | Msg18.2)
      otherwise   := 'OTHERWISE' ncl [instruction+]

/* Third part is for single instructions. */
single_instruction:= assignment | message_instruction |
   keyword_instruction
                  |command
  assignment      := VAR SYMBOL '#' expression
                  | NUMBER '#' Msg31.1
                  | CONST_SYMBOL '#' (Msg31.2 | Mgg31.3)
  message_instruction := message_term | message:term '#' expression
  keyword_instruction:= address | arg | call | drop | exit
                  | interpret | iterate | leave
                  | nop | numeric | options
                  | parse | procedure | pull | push | queue
                  | raise | reply | return | say | signal | trace |
                    use
                  | 'THEN' Msg8.1 | 'ELSE' Msg8.2
                  | 'WHEN' Msg9.1 | 'OTHERWISE' Msg9.2
  command         := expression
address           := 'ADDRESS' [(taken_constant [expression]
                  | Msg19.1 | valueexp)  [ 'WITH' connection] ]
  taken_constant  := symbol | STRING
  valueexp        := 'VALUE' expression
  connection      := ad_option+
    ad_option     := error | input | output | Msg25.5
      error       := 'ERROR' (resourceo | Msg25.14)
      input       := 'INPUT' (resourcei | Msg25.6)
        resourcei := resources | 'NORMAL'
      output      := 'OUTPUT' (resourceo | Mgg25.7)
        resourceo := 'APPEND' (resources | Msg25.8)
                  | 'REPLACE' (resources | Msg25.9)
                  | resources | 'NORMAL'
resources         := 'STREAM' (VAR_SYMBOL | Msg53.1)
                  | 'STEM' (VAR_SYMBOL | Msg53.2)
  vref            := '(' var_symbol (')' | Msg46.1)
    var:symbol    := VAR_SYMBOL | Msg20.1
arg               := 'ARG' [template list]
call              := 'CALL' (callon_spec |
                  (taken_constant | vref | Msg19.2) [expression_list]
                      )
  callon_spec     := 'ON' (callable_condition | Msg25.1)
                  ['NAME' (symbol_constant_term | Msg19.3)]
                  | 'OFF' (callable_condition | Msg25.2)
    symbol_constant_term := term
    callable_condition:= 'ANY' | 'ERROR' | 'FAILURE' | 'HALT' | '
       NOTREADY'
                  | 'USER' ( symbol_constant_term | Msg19.18 )
    condition     := callable_condition | 'LOSTDIGITS'
                  | 'NOMETHOD' | 'NOSTRING' | 'NOVALUE' | 'SYNTAX'
  expression_list := expr | [expr] ',' [expression list]
do_specification  := do simple | do repetitive
  do_simple       := 'DO' [group_option+]
  do_repetitive   := do_simple (dorep | conditional | dorep
     conditional)
    dorep         := 'FOREVER' | repetitor
```

```
drop              := 'DROP' variable_list
  variable_list   := (vref | var_symbol)+
exit              := 'EXIT' [expression]
forward           := 'FORWARD' [forward_option+ | Msg25.18]
  forward_option  := 'CONTINUE' | ArrayArgOption |
                     MessageOption | ClassOption | ToOption
    ArrayArgOption:='ARRAY' arguments | 'ARGUMENTS' term
    MessageOption :='MESSAGE' term
    ClassOption   :='CLASS' term
    ToOption      :='TO' term
guard             := 'GUARD' ('ON' | Msg25.22) [('WHEN' | Msg25.21)
expression]
                        | ( 'OFF' | Msg25.19) [('WHEN' | Msg25.21)
expression]
interpret         := 'INTERPRET' expression
iterate           := 'ITERATE' [VAR SYMBOL | Msg20.2]
leave             := 'LEAVE' [VAR SYMBOL | Msg20.2]
nop               := 'NOP'
numeric           := 'NUMERIC' (numeric_digits | numeric_form
                  | numeric fuzz | Msg25.15)
  numeric_digits  := 'DIGITS' [expression]
  numeric_form    := 'FORM' [numeric_form_suffix]
    numeric_form_suffix:=('ENGINEERING' |'SCIENTIFIC'|valueexp |
       Msg25.11)
  numeric_fuzz    := 'FUZZ' [expression]
options           := 'OPTIONS' expression
parse             := 'PARSE' [translations] (parse_type
|Msg25.12) [template_list]
  translations    := 'CASELESS' ['UPPER' | 'LOWER']
                  | ('UPPER' | 'LOWER') ['CASELESS']
  parse_type      := parse_key | parse_value | parse_var | term
    parse_key     := 'ARG' | 'PULL' | 'SOURCE' | 'LINEIN'
                  | 'VERSION'
    parse_value   := 'VALUE' [expression] ('WITH' | Msg38.3)
    parse_var     := 'VAR' var_symbol
  template := NAME [( [pattern] NAME) +]
    pattern:= STRING | [indicator] NUMBER | [indicator] '(' symbol ')
        '
      indicator := '+' | '-' | '='
procedure         := 'PROCEDURE' [expose | Msg25.17]
pull              := 'PULL' [template_list]
push              := 'PUSH' [expression]
queue             := 'QUEUE' [expression]
raise             := 'RAISE' conditions (raise_option | Msg25.24)
  conditions      := 'ANY' | 'ERROR' term | 'FAILURE' term
                  | 'HALT'| 'LOSTDIGITS' | 'NOMETHOD' | 'NOSTRING' |
"NOTREADY'
                  | 'NOVALUE' | 'PROPAGATE' | 'SYNTAX' term
                  | 'USER' ( symbol_constant_term | Msg19.18) | Msg25
                     .23
  raise_option    := ExitRetOption | Description | ArrayOption
    ExitRetOption := 'EXIT' [term] | 'RETURN' [term]
    Description   :='DESCRIPTION' term
    ArrayOption   := 'ADDITIONAL' term | 'ARRAY' arguments
reply             := 'REPLY' [ expression]
return            := 'RETURN' [expression]
```

```
say                 := 'SAY' [expression]
signal              := 'SIGNAL' (signal_spec | valueexp
                    | symbol_constant_term | Msg19.4)
  signal_spec       := 'ON' (condition | Msg25.3)
                    ['NAME' (symbol_constant_term | Msg19.3)]
                    | 'OFF' (condition | Msg25.4)
trace               := 'TRACE' [(taken_constant | Msg19.6) | valueexp]
use                 := 'USE' ('ARG' | Msg25.26) [use_list]
  use_list          := VAR_SYMBOL | [VAR_SYMBOL] ',' [use_list]

/* Note:  The next part describes templates. */
template_list       := template | [template] ',' [template_list]
  template          := (trigger | target | Msg38.1)+
   target           := VAR_SYMBOL | '.'
   trigger          := pattern | positional
     pattern        := STRING | vrefp
       vrefp        := '(' (VAR_SYMBOL | Msg19.7) (')' | Msg46.1)
     positional     := absolute positional | relative positional
       absolute_positional:= NUMBER | '=' position
         position := NUMBER | vrefp | Msg38.2
     relative_positional:= ('+' | '-') position

/* Note: The final part specifies the various forms of symbol, and
expression. */
symbol              := VAR_SYMBOL | CONST_SYMBOL | NUMBER
expression          := expr [(',' Msg37.1) | (')' Msg37.2 )]
  expr              := expr_alias
    expr_alias      := and_expression
                    | expr_alias or_operator and_expression
      or_operator := '|' | '&&'
      and_expression := comparison | and_expression '&' comparison
comparison          := concatenation
                    | comparison comparison_operator concatenation
  comparison_operator:= normal_compare | strict_compare
    normal_compare:= '=' | '\=' | '<>' | '><' | '>' | '<' | '>='
                    | '<=' | '\>' | '\<'
    strict_compare:= '==' | '\==' | '>>' | '<<' | '>>=' | '<<='
                    | '\>>' | '\<<'
concatenation       := addition
                    | concatenation (' ' | '||') addition
addition            := multiplication
                    | addition additive_operator multiplication
  additive operator:= '+' | '-'
multiplication      := power_expression
                    | multiplication multiplicative_operator
                   power_expression
  multiplicative_operator:= '*' | '/' | '//' | '%'
power_expression    := prefix_expression
                    | power_expression '**' prefix_expression
  prefix_expression := ('+' | '-' | '\') prefix_expression
                    | term | Msg35.1
/* "Stub" has to be identified semantically? */
    term            := simple_term [ '.' ( term | Msgnn )]
      simple_term := symbol | STRING | invoke | indexed
                    | '(' expression ( ')' | Msg36 )
                    | initializer
```

```
                | message_term '##'
      message_term:= term ('~' | '~~') method_name [arguments]
                | term '['[ expression_list ] (']' | Msg36.2)

        method_name:=(taken constant | Msg19.19)
                          [':' ( VAR_SYMBOL | Msg19.21 )]
/* Method-call without arguments is syntactically like symbol. */
/* Editor - not sure of my notes about here. */
invoke       := (symbol | STRING) arguments
  arguments       := '#(' [expression list] (')' | Msg36)
    expression_list := expression | [expression] ',' [expression_list
      ]
indexed           := (symbol | STRING) indices
  indices         := '#[' [expression list] (']' | Msg36.n)
initializer       := '['expression list (']' | Msg36.n)
```

8.4 Syntactic information

8.4.1 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 nnn.

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* 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*. Tne *do_specification* will be associated with a *do* by nnn. The *iterate* or *leave* will be a single *instruction* in an *instruction_list* associated with a *do* by nnn. These two dos 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 dos are the same. The *nesting_correction* for *iterates* or *leaves* that do not contain *VAR_SYMBOL* is zero.

8.4.2 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.

8.4.3 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.

8.4.4 Nested IF instructions

The syntax specification nnn allows `'IF'` instructions to be nested and does not fully specify the association of an `'ELSE'` keyword with an `'IF'` keyword. An `'ELSE'` associates with the closest prior `'IF'` that it can associate with in conformance with the syntax.

8.4.5 Choice of messages

The specifications nnn and nnn 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.

8.4.6 Creation of messages

The *message_identifiers* in clause 6 correlate with the tails of stem #ErrorText., which is initialized in nnn 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.

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 interactive trace *interpret* instruction (see nnn) 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 nnn.

8.5 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.

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 previous token;

- 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. In deciding the leftmost blank in a quoted string of radix `'X'` or `'B'` that is erroneous note that:
  - A blank as the first character of the quoted string is an error.
  - The leftmost embedded sequence of blanks can validly follow any number of non-blank characters.
  - Otherwise a blank run that follows an odd numbered sequence of non-blanks (or a number not a multiple of four in the case of radix `'B'`) is not valid.
  - If the string is invalid for a reason not described above, the leftmost blank of the rightmost sequence of blanks is the invalid blank to be referenced in the message;
- 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 word 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 `','`. The keywords will be uppercased and in alphabetical order.

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.

8.6 Syntactic equivalence

If a message_term contains a `'['` it is regarded as an equivalent message_term without a `'['`, for execution. The equivalent is `term~'[]'(expression_list)`. See nnn. If a *message_instruction* has the construction `message_term '=' expression` it is regarded as equivalent to a *message_term* with the same components as the *message_term* left of the `'='`, except that the *taken_constant* has an `'='` character appended and *arguments* has the expression from the right of the `'='` as an extra first argument. See nnn.

9

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 nnn. 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 nnn.

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', 20.1, word
 constant
```

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

9.1 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 nnn and nnn, 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 reference 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.

9.1.1 Var_Empty

```
Var_Empty(Pool)
```

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

9.1.2 Var Set

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

The function operates on the variable pool with the specified reference. The name is a non-tailed name. If the specified name has the 'exposed' attribute then Var_Set operates on the variable pool referenced by #Upper in this pool 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 reference. 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 referenced by #Upper in this pool 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 referenced by #Upper in the pool 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'.

9.1.3 Var_Value

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

The function operates on the variable pool with the specified reference. The name is a non-tailed name. If the specified name has the 'exposed' attribute then Var_Value

operates on the variable pool referenced by `#Upper` in this pool 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 reference. 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 referenced by `#Upper` in this pool 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 referenced by #Upper in the pool 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`.

9.1.4 Var_Drop

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

The function operates on the variable pool with the specified reference. The name is a non-tailed name. If the specified name has the 'exposed' attribute then `Var_Drop` operates on the variable pool referenced by `#Upper` in this pool 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 reference. 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 referenced by #Upper in this pool 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 referenced by `#Upper` in the pool 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.

9.1.5 Var_Expose

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

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

9.1.6 Var_Reset

```
Var_ Reset (Pool)
```

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

9.2 Symbols

For the syntax of a symbol see nnn.

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 nnn:

```
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>'`.

9.3 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 nnn and nnn 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.

9.3.1 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 nnn for an exception.

9.3.2 Value of a reserved symbol

The value of a reserved symbol is the value of a variable with the corresponding name in the reserved pool, see nnn.

9.4 Expressions and operators

Add a load of string coercions. Equality can operate on non-strings. What if one operand non-string?

9.4.1 The value of a term

See nnn for the syntax of a term.

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

The value of a function is the value it returns, see nnn.

If a termis 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 nnn.

9.4.2 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 ferm, otherwise let rhs be the value of the prefix_expression within it__ see nnn

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 nnn 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>'

9.4.3 The value of a power_expression

See nnn 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,'**'
```

power_expression is

ArithOp(lhs,'**',rhs)

If the power_expression is not a prefix_expression then: if #Tracing.#Level == 'I' then call #Trace '>0O>'

9.4.4 The value of a multiplication

See nnn 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 rns 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 '>0O>'

9.4.5 The value of an addition

See nnn 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 Ihs be the value of ad¢difion within it, and rhs be the value of the multiplication within it. Let

operation be the adaltive_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 Ihs 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 Ihs+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 Ihs-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 '>0O>'

9.4.6 The value of a concatenation

See nnn 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 Ihs 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(Ihs)+Config_Length(rhs).
- #Outcome will be 'equal' after each of:
- Config_Compare(Config_Subsir(Ihs,n)},Config_Subsitr(Value,n)) for values of n not less than 1 and not more than Config_Length(Ihs);
- Config_Compare(Config_Subsir(rhs,n),Config_Substr(Value,Config_Length(Ihs)+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(Ihs)+1+Config_Length(rhs).
- #Outcome will be 'equal' after each of:
- Config_Compare(Config_Subsir(Ihs,n)},Config_Subsitr(Value,n)) for values of n not less than 1 and not more than Config_Length(Ihs);
- Config_Compare(' ',Config_Substr(Value,Config_Length(Ihs)}+1));
- Config_Compare(Config_Subsitr(rhs,n),Config_Substr(Value,Config_Length(Ins)+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 '>0O>'

9.4.7 The value of a comparison

See nnn 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 Ihs be the value of the comparison within it, and rns 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(Ihs) and Config_Length(rhs). For values of n greater than O and not greater than Length, if any, in ascending order, #Test is set to the uppercased first character of #Outcome after:

Config_Compare(Config_Subsir(Ihs),Contfig_Subsir(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(Ihs) is greater than Config_Length(rhs) then #Test is set to 'G' or if Config_Length(Ihs) 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 '=" ";==" ", '>=', <=", '>', '<', 'p>=', '«=', », or «)

- #Test is 'G' and the operator is one of '>', '>=", '<', '=', '<>', '><', Nes",»! 'p>', 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 '>0O>'

9.4.8 The value of an and_expression

See nnn 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 Ihs 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 '>0O>'

9.4.9 The value of an expression

See nnn 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 Ihs 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 '>0O>'

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

9.4.10 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 `ArithOpp` itself is assumed to operate under a sufficiently high setting of numeric digits to avoid exponential notation.

```
   ArithoOp:

 arg Numberl, Operator, Number2
/* The Operator will be applied to Numberl 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 (Numberl, #Digits.#Level)
parse var v Signl Mantissal Exponentl

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
when Operator

' then call DivType
*' 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. */
Pp = 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,g,1) \== '0' then leave
p=p-l
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 be
   expressed as a

multiple of 3 - the integer part may therefore range from 1 through
999." */

g=l

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
geaqgitl
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, Numberl, Operator, Number2
if -#Limit ExponentDigits > Exponent3 then

call #Raise 'SYNTAX', 42.2, Numberl, Operator, Number2

/* Insert any decimal [point. */

if length(Mantissa3) \= g then Mantissa3 = insert('.',Mantissa3,g)
/* Insert the E */
if Exponent3 >= 0 then Number3
else Number3
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=il
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

Mantissa3'E+'Exponent3
Mantissa3'E'Exponent3

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. */
Ct = ter
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 lst operand */
Sign3=Signl; Mantissa3 = Mantissal; Exponent3 = Exponentl
return ''
end

if Mantissal
Sign3 = Sign

== '0' then do /* Result is the 2nd operand */
2
if Operator \

; Mantissa3 = Mantissa2; Exponent3 = Exponent2
== '+' then if Sign3 = '+' then Sign3 rat
else Sign3

bat
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
   . */
Exponentl = Exponentl + length (Mantissal1)
Exponent2 = Exponent2 + length (Mantissa2)

/* Deduce the implied zeros on the left to provide alignment. */

Aligni = 0

Align2 = Exponentl - 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 Argl */

Mantissa2 = copies('0',Align2) || Mantissa2
Exponent2 = Exponentl
end

if Align2 < 0 then do /* Arg 2 provides a more significant digit */
/* Shift to give Argl the same exponent as Arg2 */

Aligni = -Align2

Alignl = min(Align1l,NowDigits+1) /* No point in shifting further. */
Align2 = 0

Mantissal = copies('0',Alignl) || Mantissal

Exponentl = Exponent2

end

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

SigDigits
SigDigits

max (length (Mantissal) , length (Mantissaz2) )
min (SigDigits,NowDigits+1)

/* Extend a mantissa with right zeros, if necessary. */
Mantissal = left (Mantissal,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 = Exponentl-SigDigits
Exponentl = Exponent3
Exponent2 = Exponent3
if Operator = '+' then
Mantissa3 = (Signl || Mantissal) + (Sign2 || Mantissa2)

else Mantissa3 (Signl || Mantigsal) - (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+l
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+l
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 Signl == Sign2 then Sign3 = '+'
else Sign3 = '-'
/* Note the exponent */
Exponent3 = Exponentl + Exponent2
if Mantissal == '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 + Mantissal
end
```

```
end q
return /* From Multiply */

/* Types of Division: */

DivType: /* p 131 */
/* Check for divide-by-zero */

if Mantissa2 == '0' then call #Raise 'SYNTAX',

/* Note the exponent of the result */
Exponent3 = Exponentl - Exponent2

/* Compute (one less than) how many digits will be in the integer

part of the result. */

IntDigits = length(Mantissal) - Length(Mantissa2) + Exponent3
/* In some cases, the result is known to be zero.
if Mantigsal = 0 | (IntDigits < 0 & Operator

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 = Mantissal
Sign3 = Signl
Exponent3 = Exponentl
return
end
/* Note the sign of the result. */
if Signl == Sign2 then Sign3 = '+'
else Sign3 = '-'

/* Make Mantissal at least as large as Mantissa2 so Mantissa2 can be
subtracted without causing leading zero to result. Page 131 */

az 0
do while Mantissa2 > Mantissal

Mantissal = Mantissal'0'
Exponent3 = Exponent3 - 1
aztadtl
end
/* Traditional divide */
Mantissa3 = ''

/* Subtract from part of Mantissal that has length of Mantissa2 */

left (Mantissal,length(Mantissa2) )
substr (Mantissal, length (Mantissa2)+1)
o forever

x
Y
d
```

```
/* Develop a single digit in z by repeated subtraction.

ze=O0
do forever
xX = kK - Mantissa2
if left(x,1) == '-' then leave
Zeze¢t+tl
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


##

*f
'%') then do

*/
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 Mantissal */
if y == '' then do
y = ror
Exponent3 = Exponent3 - 1
aztadtl
end
xX = xX | | 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 // */
```

```
if Floating( ) 'E' then do
if Operator '%' then MsgNum
else MsgNum

/* Page 133. % could fail by needing exponential notation */

26.11
26.12

call #Raise 'SYNTAX', MsgNum, Numberl , Number2, #Digits.#Level

end
end
if Operator == '//' then do
/* We need the remainder */
Sign3 = Signl

Mantissa3 = Remainder
Exponent3 = Exponentl - 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
Exponent3
return
end

i
ror

/* 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=l1byl
if substr(b,q,1) \== '0' then leave
end q
ael
do forever
/* Page 133 "Using a precision of DIGITS+L+1" */
if substr(b,q,1) == '1' then do
a = Recursion('*',Signl || Mantissal'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=qrtil
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 */
ReWas = substr(a,4)
if Sign2 = '-' then if ReWas == '42.1' then RcWas
else RcWas
call #Raise 'SYNTAX', RcWas, Numberl, '**', Number2
/* No return */

"42.2!
"42.1!

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 1l/a
else return a * arg(2)
Over flowed:
return 'MN '.MN
```

9.5 Functions

9.5.1 Invocation

Invocation occurs when a *function* or a *message_term* 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>'
```

Invocation of a *message_term* may result in a value, in which case:

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

9.5.2 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`.

9.5.3 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 nnn. 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 nnn) then a condition is raised:

```
call #Raise 'SYNTAX', 20.1, word
 constant
```

If there is such a matching label, and the label is not trace-only, execution continues at the label with routine initialization (see nnn). 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 nnn 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)
```

The name used in the invocation is held in `#Name.#Level` for possible use in an error message from the `RETURN` clause, see nnn

9.5.4 The value of a function

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

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`.

9.5.5 The value of a method

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

An internal method completes when `#Level` returns to the value it had when the routine was invoked. The value of the internal method is the value of the *expression* on the *return* which completed the method. The value of an external method is determined by `Config_ExternalMethod`.

9.5.6 The value of a message term

See nnn for the syntax of a *message_term*. The value of the *term* within a *message_term* is called the receiver.

The receiver and any arguments of the term are evaluated, in left to

```
r = #evaluate(message_term, term)
```

If the message term contains `'~~'` the value of the message term is the receiver.

Any effect on .Result?

Otherwise the value of a *message_term* is the value of the method it invokes. The method invoked is determined by the receiver and the *taken_constant* and *symbol*.

```
t = #Instance(message term, taken constant)
```

If there is a *symbol*, it is subject to a constraints.

```
if #contains(message term, symbol) then do
   if r <> #Self then
      call #Raise 'SYNTAX', nn.n
```