CREXX
Language Reference

The CREXX team

June 30, 2024

THE REXX LANGUAGE ASSOCIATION
CREXX Programming Series
ISBN 978-90-819090-1-3

Publication Data

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.

This edition is registered under ISBN 978-90-819090-1-3

ISBN 978-90-819090-1-3

9 789081 909013 >

Contents

The CREXX Programming Series

This book is part of a library, the *CREXX Programming Series*, documenting the CREXX programming language and its use and applications. This section lists the other publications in this series, and their roles. These books can be ordered in convenient hardcopy and electronic formats from the Rexx Language Association.

User Guide	This guide is meant for an audience that has done some programming and wants to start quickly. It starts with a quick tour of the language, and a section on installing the CREXX translator and how to run it. It also contains help for troubleshooting if anything in the installation does not work as designed, and states current limits and restrictions of the open source reference implementation.
Application Programming Guide	The Application Programming Guide explains the working of the tools and has examples for building programs on the platforms it supports.
Language Reference	Referred to as the CRL, this is meant as the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementers.
The CREXX VM Specification	The CREXX VM Specification, documents the CREXX Assembly Language and its execution by the CREXX Virtual Machine. It also contains low level virtual machine and ABI specifications.

Typographical conventions

In general, the following conventions have been observed in the REXX publications:

- Body text is in this font
- Examples of language statements are in a `keyword` or **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are listed in this fashion:

```
-- salute the reader
say 'hello reader'
```

About This Book

The crexx language is a further development, and variant of the Rexx language[1].

This language reference provides a comprehensive and detailed description of the syntax, semantics, and features of the crexx programming language. It usually includes a complete listing of all the language constructs, such as statements, expressions, functions, classes, and modules, along with their syntax, parameters, and usage.

Audience

The crexx language reference is aimed at experienced programmers who are familiar with the Rexx language. It is an essential resource for developers who need to write code in the language, debug programs, or create tools, while the *crexx User's Guide* serves as an introduction to the language and is intended for new users.

History

mvp This version documents the Minimally Viable Product release, Q1 2022 and is intended for developers only. It documents crexx level B, which is a typed subset of Classic Rexx.

f29 First version, Git feature [F0029]

Document Structure

This document is in three parts:

1. The objectives of the crexx language and the concepts underlying its design, and acknowledgements.
2. An overview and introduction to the crexx language.
3. The definition of the language.

[1] Cowlishaw, 1979

Part I

Objectives

1

Introduction

crexx is a general-purpose programming language. **It is still designed for people, not computers**. In this respect it follows Rexx closely, with many of the concepts and most of the syntax taken directly from Rexx or its object-oriented variants, Object Rexx and Netrexx.

crexx will be delivered in different levels. Level B is the first and contained in this MVP release. Level C will be mostly compatible with ISO standard Classic Rexx. In addition to the mentioned Rexx variants, crexx level B contains static typing and binary arithmetic.

Later levels will contain an object model, exception handling and a library of classes. The resulting language not only provides the scripting capabilities and decimal arithmetic of Rexx, but also seamlessly extends to large application development with fast binary arithmetic.

The crexx team is trying to provide a new, up-to-date implementation of Rexx, and to incorporate many of the improvements ooRexx and Netrexx added to the language. In some cases, choices need to be made. These choices are made in dialogue with the RexxARB (Architecture Review Board). While, in this day and age of open source development every language implementation team is autonomous, and in fact everyone can fork a repository and strike out on their own, the crexx team seeks cooperation and efficient deployment of work.

The terms 'new' and 'up-to-date' refer to several different developments in our technical and social reality. The developments of CPU and GPU technology mean that, while we might have hit a wall with Moore's law, parallellism and pipelining have become more important, as are cache sizes, RISC and vectorising of instructions, like done in different SIMD implementations. As address ranges have become large and segmentation is something of the past, we are de-emphasising memory management and emphasising instruction pipelining in the crexx VM Kernel. Also, we are testing our hypothesis that Rexx can be a high-performance language, so apart from that VM Kernel, everything is written in that language.

We are convinced that the working of interpreters and compilers needs to be understandable to its users; with a more complete mental imagery of those workings, it is possible to write much better programs. We strive to have a completely transparent architecture for the language tools. For this reason we have tried to open up the bytecode compiler and interpreter layers to the user of the language, instead of hiding these. This also enables the team (or others) to start working on native, true compilers that generate executable

machine instructions for hardware instruction set architectures or future virtual machines.

Also from a software point of view the world has changed. Apart from fads and silver bullets, some developments have stuck and are sometimes missed in Rexx, which at this point in time is more than 40 years old. The introduction of Object Orientation in the 1990's was a long and arduous process, that might have taken too much time than was time-to-market and user expectations allowed. What did not exactly helped was putting this Orexx version into the hands of a different language architect, the product intentions published early and the product delivery taking a very long time - as a 'user switchable' implementation on a moribund operating system[2] Further announcements and retractions of its owner regarding its open sourcing status did not help it in any way. Another variant, for the Java Virtual Machine, has shown the divergence in ideas between Rexx's original architect, Mike Cowlishaw, and the Object Rexx implementation of the same ideas.

We would like to try to avoid this by opening up architecture, source and discussion in the user community. While Rexx variants may have diverged on small points, an effort could be set up to standardize as much as possible on new developments. What are these new developments in programming language technology?[3]

A discussion on various mailing fora brought about the following list of what different variants of Rexx need:

- Short circuit evaluation
- Functions as first class objects
- Multi-threading
- Dynamically typed, preferably with type hints
- Co-routintes
- Support for object oriented programming
- Regular expressions
- Libraries for web programming, jpegs, JSON/YAML parsing etc
- Language tools
- Logic programming

Note that this list is just a statement of intention and some things do exist in existing implementations while others might not be a good idea or would be done better by tools external to the language.

1.1 Level B Minimally Viable Product Snapshot

It is important to know that the Level B implementation is not complete yet. What still is missing from this snapshot with regard to the first crexx Level B release, is the following

[2] IBM's OS/2.
[3] some might be quite old

- Adress statement
- stream I/O
- structured variables

The team deems it important to release the working parts at this time, so people can play with it, be amazed by the speed of code implemented with Rexx, use Unicode and see how it works, and possibly, hopefully participate in its development. This means that crexx has entered the phase in which things are stable enough for working on specific tasks by a larger team. Everybody is welcome! Lots of things can be done in Rexx!

1.2 Language Objectives

1.2.1 Features of Rexx

The Rexx programming language[4] was designed with just one objective: to make programming easier than it was before. The design achieved this by emphasizing readability and usability, with a minimum of special notations and restrictions. It was consciously designed to make life easier for its users, rather than for its implementers.

The great strengths of Rexx are its human-oriented features, including

- simplicity
- coherent and uncluttered syntax
- comprehensive stringhandling
- case-insensitivity
- arbitrary precision decimal arithmetic.

Care has been taken to preserve these. Conversely, the interpretive nature of Classic Rexx has always entailed a lack of efficiency: excellent Rexx compilers do exist, from IBM and other companies, but cannot offer the full speed of statically-scoped languages such as C[5] or Java[6]. Where Netrexx already offers high performance on the Java Virtual Machine, crexx aims to offer this on native platforms.

1.3 Rexx in Rexx

An aim of the project is to implement as much of the crexx architecture as possible in Rexx itself. To achieve this however core Rexx level B facilities are needed. These will be used to implement classic Rexx and modernised Rexx variations.

[todo picture]

[4] Cowlishaw, M. F., **The REXX Language** (second edition), ISBN 0-13-780651-5, Prentice-Hall, 1990.

[5] Kernighan, B. W., and Ritchie, D. M., **The C Programming Language** (second edition), ISBN 0-13-110362-8, Prentice- Hall, 1988.

[6] Gosling, J. A., *et al.* **The Java Language Specification**, ISBN 0-201-63451-1, Addison-Wesley, 1996.

- Stage 1 - Rexx built-in functions will be implemented in crexx level B
- Stage 2 - Rexx Languages (Interpreters / Compilers) will be re-implemented in crexx level B
- Stage 3 - native code compilers based on LLVM IR and other technologies will be made available

An important difference from earlier implementations of the language tools is the emphasis on a clear separation of architectural levels. Where more (and indeed the very first implementations on VM/SP) versions contain a virtual machine layer, this implementation opens up these layers with clear boundaries between the compiler, the assembler and different backends. In this MVP, the execution engine is a threaded[7] virtual machine, which optionally is delivered along with the application in a single native executable.

The interface between the compiler and the assembler is a conventional text file with crexx assembler instructions. It is possible to write crexx assembler without resorting to the compiler; also a conventient way is provided (in crexx level b) to use inline assembly. It is possible to implement other languages on top of the assembler and virtual machine, but for the moment that is out of the scope of this publication.

The binary produced by the assembler is portable between the implementations of crexx on all operating systems platforms and hardware instruction set combinations. The threaded executable will be OS/ISA specific, and (at this moment) is not yet guaranteed to be compatible over crexx releases. Native executables are, at this moment alrready, executable without any part of the language toolchain available, i.e. without any installation.

1.4 Unicode

One of the design goals of crexx is to enable the use of Unicode in the Rexx language on all platforms that support its use. Unicode is an important standard that helps ensure that different computer systems and software programs can handle and display text correctly, regardless of language, script, or writing system.

Here are some specific reasons why Unicode is important:

Universal character encoding Unicode provides a single, universal character encoding that can represent all characters used in modern writing systems, including those used in different languages, scripts, and symbols. This means that Unicode enables computers and software programs to exchange text in any language or script.

Multilingual support Unicode supports the display of text in many different languages, including those that use non-Latin scripts, such as Arabic, Chinese, Cyrillic, Devanagari, Hebrew, and many others.

[7]

Interoperability Unicode enables interoperability between different computer systems and software programs, allowing users to share and exchange text without worrying about compatibility issues.

Accessibility Unicode makes it easier to develop software and tools that can support multiple languages and scripts, which is important for ensuring accessibility and inclusivity in digital communication.

Standardization Unicode is a widely accepted standard that is used by many software developers, hardware manufacturers, and technology companies. This standardization helps ensure consistency and compatibility across different platforms and devices.

Overall, Unicode plays a critical role in facilitating communication and information sharing across different cultures and languages, and is essential for creating a more connected and inclusive digital world.

1.4.1 Unicode in cREXX

Consider the following program:

```
/* UTF Test 多变的 */
options levelb
/* Chinese! */多变的
 = "多变的 is 'variable' in Chinese (according to Google!)"
SAY 多变的

équipe = "René Vincent Jansen"
équipe = équipe ", Mike Großmann"
équipe = équipe " and all the rest (this is just a UTF-8 test! □)"

Say "cREXX Équipe:" ÉQUIPE

/* The ß experiment */
Großmann = 1
GROßMANN = 2□
GROMANN = 3 /* The new capital □ - TODO fix utp.h */
GROSSMANN = 4

say Großmann
say GROßMANN
say □GROMANN
say GROSSMANN

/* The lexer knows that □ is not a letter and so is not a valid
   symbol name */
/* □ = "Smile" */
```

```
多变的 is 'variable' in Chinese (according to Google!)
cREXX Équipe: René Vincent Jansen , Mike Großmann and all the rest (this is
just a UTF-8 test! □)
2
2
3
4
```

1.5 Performance

For precise technical documentation, we refer to the *crexx VM Specification* publication, which contains the last word on the architecture and implementation of the bytecode compiler, the assembler and the first two virtual machines. One of these is a conventional byte code interpreter, the other is a *threaded code interpreter*. These share over 99% of their source code.

The virtual machine executing the Rexxassembler instructions is written according to a strict set of rules that limits the possibility of pipeline stalls in modern processor architectures.

Part II

Overview

Part III

Reference

2

Structure and General Syntax

A CREXX program is built up out of a series of *clauses* that are composed of: zero or more blanks (which are ignored); a sequence of tokens (described in this section); zero or more blanks (again ignored); and the delimiter "**;**" (semicolon) which may be implied by line-ends or certain keywords. Conceptually, each clause is scanned from left to right before execution and the tokens composing it are resolved.

Identifiers (known as symbols) and numbers are recognized at this stage, comments (described below) are removed, and multiple blanks (except within literal strings) are reduced to single blanks. Blanks adjacent to operator characters and special characters are also removed.

2.1 Blanks and White Space

Blanks (spaces) may be freely used in a program to improve appearance and layout, and most are ignored. Blanks, however, are usually significant

- within literal strings (see below)
- between two tokens that are not special characters (for example, between two symbols or keywords)
- between the two characters forming a comment delimiter
- immediately outside parentheses ("(" and ")") or brackets ("[" and "]").

For implementations that support tabulation (tab) and form feed characters, these characters outside of literal strings are treated as if they were a single blank; similarly, if the last character in a CREXX program is the End-of-file character (EOF, encoded in ASCII as decimal 26), that character is ignored.

2.2 Comments

Commentary is included in a CREXX program by means of *comments*. Two forms of comment notation are provided: *line comments* are ended by the end of the line on which they start, and *block comments* are typically used for more extensive commentary.

Line comments The default line comment character in CREXX level B is the *hash* character #, also called *octothorpe*. This is associated with the options `options hashcomments (default) / nohashcomments`.

A line comment can also be started by a sequence of two adjacent slashes (//); all characters following that sequence up to the end of the line are then ignored by the CREXX language processor. This is associated with the options `Options slashcomments / noslashcomments (default).`
A line comment can also be started by a sequence of two adjacent hyphens ("– –"); all characters following that sequence up to the end of the line are then ignored by the CREXX language processor. This is associated with the options `Options dashcomments / nosdashcomments (default).`
Example:

```
i=j+7 # this line comment follows an assignment
```

Block comments A block comment is started by the sequence of characters "/*", and is ended by the same sequence reversed, "*/". Within these delimiters any characters are allowed (including quotes, which need not be paired). Block comments may be nested, which is to say that "/*" and "*/" must pair correctly. Block comments may be anywhere, and may be of any length. When a block comment is found, it is treated as though it were a blank (which may then be removed, if adjacent to a special character).
Example:

```
/* This is a valid block comment */
```

The two characters forming a comment delimiter ("/*" or "*/") must be adjacent (that is, they may not be separated by blanks or a line-end).

Note: It is recommended that CREXX programs start with a block comment that describes the program. Not only is this good programming practice, but some implementations may use this to distinguish CREXX programs from other languages. **Implementation minimum:** Implementations should support nested block comments to a depth of at least 10. The length of a comment should not be restricted, in that it should be possible to "comment out" an entire program.

2.3 Tokens

The essential components of clauses are called *tokens*. These may be of any length, unless limited by implementation restrictions, [8] and are separated by blanks, comments, ends of lines, or by the nature of the tokens themselves.

The tokens are:

Literal strings A sequence including any characters that can safely be represented in an implementation [9] and delimited by the single quote character (') or

[8] Wherever arbitrary implementation restrictions are applied, the size of the restriction should be a number that is readily memorable in the decimal system; that is, one of 1, 25, or 5 multiplied by a power of ten. 500 is preferred to 512, the number 250 is more "natural" than 256, and so on. Limits expressed in digits should be a multiple of three.

[9] Some implementations may not allow certain "control characters" in literal strings. These characters may be included in literal strings by using one of the escape sequences provided.

TABLE 1: Escape sequences

\t	the escape sequence represents a tabulation (tab) character
\n	the escape sequence represents a new-line (line feed) character
\r	the escape sequence represents a return (carriage return) character
\f	the escape sequence represents a form-feed character
\"	the escape sequence represents a double-quote character
\'	the escape sequence represents a single-quote character
****	the escape sequence represents a backslash character
\-	the escape sequence represents a "null" character (the character whose encoding equals zero), used to indicate continuation in a `say` instruction
\0(zero)	the escape sequence represents a "null" character (the character whose encoding equals zero); an alternative to **\-**
\xhh	the escape sequence represents a character whose encoding is given by the two hexadecimal digits ("**hh**") following the "**x**". If the character encoding for the implementation requires more than two hexadecimal digits, they are padded with zero digits on the left.
\uhhhh	the escape sequence represents a character whose encoding is given by the four hexadecimal digits ("**hhhh**") following the "**u**". It is an error to use this escape if the character encoding for the implementation requires fewer than four hexadecimal digits.

the double-quote (**"**). Use **""** to include a **"** in a literal string delimited by **"**, and similarly use two single quotes to include a single quote in a literal string delimited by single quotes. A literal string is a constant and its contents will never be modified by CREXX. Literal strings must be complete on a single line (this means that unmatched quotes may be detected on the line that they occur). Any string with no characters (*i.e.*, a string of length 0) is called a *null string*.

Examples:

```
'Fred'
'Aÿ'
"Don't Panic!"
":x"
'You shouldn''t'     /* Same as "You shouldn't" */
''                   /* A null string */
```

Within literal strings, characters that cannot safely or easily be represented (for example "control characters") may be introduced using an *escape sequence*. An escape sequence starts with a *backslash* ("\"), which must then be followed immediately by one of the following (letters may be in either uppercase or lowercase) - see table 1.

[todo back to Rexx notation] Hexadecimal digits for use in the escape sequences above may be any decimal digit (0-9) or any of the first six alphabetic characters (a-f), in either lowercase or uppercase. **Examples:**

```
'You shouldnt'  /* Same as "You shouldn't" */
```

```
'\x6d\u0066\x63'  /* In Unicode: 'mfc' */
'\\\u005C'        /* In Unicode, two backslashes */
```

Implementation minimum: Implementations should support literal strings of at least 100 characters. (But note that the length of string expression results, *etc.*, should have a much larger minimum, normally only limited by the amount of storage available.)

Symbols Symbols are groups of characters selected from the Roman alphabet in uppercase or lowercase (A-Z, a-z), the Arabic numerals (0-9), or the characters underscore, dollar, and euro[10] ("_$ €") Implementations may also allow other alphabetic and numeric characters in symbols to improve the readability of programs in languages other than English. These additional characters are known as *extra letters* and *extra digits*. [11]

Examples:

```
DanYrOgof
minx
Élan
$Virtual3D
```

A symbol may include other characters only when the first character of the symbol is a digit (0-9 or an extra digit). In this case, it is a *numeric symbol* - it may include a period ("**.**") and it must have the syntax of a number. This may be *simple number*, which is a sequence of digits with at most one period (which may not be the final character of the sequence), or it may be a *hexadecimal or binary numeric symbol*, or it may be a number expressed in *exponential notation*.

A number in exponential notation is a simple number followed immediately by the sequence "**E**" (or "**e**"), followed immediately by a sign ("**+**" or "**-**"), [12] followed immediately by one or more digits (which may not be followed by any other symbol characters).

Examples:

```
1
1.3
12.007
17.3E-12
3e+12
0.03E+9
```

When *extra digits* are used in numeric symbols, they must represent values in the range zero through nine. When numeric symbols are used as numbers, any extra digits are first converted to the corresponding character in the range 0-9, and then the symbol follows the usual rules for numbers in CREXX (that is, the most significant digit is on the left, *etc.*).

3.03 *In the reference implementation, the extra letters are those characters (excluding A-Z, a-z, and underscore) that result in **1** when tested with **java.lang.Character.isJavaIdentifierPart**. Similarly, the extra digits are those*

[10] Note that only UTF8-encoded source files can currently use the euro character.

[11] It is expected that implementations of CREXX will be based on Unicode or a similarly rich character set. However, portability to implementations with smaller character sets may be compromised when extra letters or extra digits are used in a program.

[12] The sign in this context is part of the symbol; it is not an operator.

characters (excluding 0-9) that result in ***1*** *when tested with* ***java.lang.Character.isDigit****.*

The meaning of a symbol depends on the context in which it is used. For example, a symbol may be a constant (if a number), a keyword, the name of a variable, or identify some algorithm.
It is recommended that the dollar and euro only be used in symbols in mechanically generated programs or where otherwise essential. **Implementation minimum:** Implementations should support symbols of at least 50 characters. (But note that the length of its value, if it is a string variable, should have a much larger limit.)

Operator characters The characters `+ - * % |/ & \= < >` are used (sometimes in combination) to indicate operations in expressions. A few of these are also used in parsing templates, and the equals sign is also used to indicate assignment. Blanks adjacent to operator characters are removed, so, for example, the sequences:

```
345>=123
345 >=123
345 >=  123
345 > =  123
```

are identical in meaning. Some of these characters may not be available in all character sets, and if this is the case appropriate translations may be used. **Note:** The sequences "**–**", "**/***", and "***/**" are comment delimiters, as described earlier. The sequences "**++**" and "****" are not valid in CREXX programs.

Special characters The characters **.,;**) ( ] [together with the operator characters have special significance when found outside of literal strings, and constitute the set of *special characters*. They all act as token delimiters, and blanks adjacent to any of these (except the period) are removed, except that a blank adjacent to the outside of a parenthesis or bracket is only deleted if it is also adjacent to another special character (unless this is a parenthesis or bracket and the blank is outside it, too). Some of these characters may not be available in all character sets, and if this is the case appropriate translations may be used.

To illustrate how a clause is composed out of tokens, consider this example:

```
'REPEAT'   B + 3;
```

This is composed of six tokens: a literal string, a blank operator (described later), a symbol (which is probably the name of a variable), an operator, a second symbol (a number), and a semicolon. The blanks between the "**B**" and the "**+**" and between the "**+**" and the "**3**" are removed. However one of the blanks between the **'REPEAT'** and the "**B**" remains as an operator. Thus the clause is treated as though written:

```
'REPEAT' B+3;
```

2.4 Implied semicolons and continuations

A semicolon (clause end) is implied at the end of each line, except if:

1. The line ends in the middle of a block comment, in which case the clause continues at the end of the block comment.
2. The last token was a hyphen. In this case the hyphen is functionally replaced by a blank, and hence acts as a *continuation character*.

This means that semicolons need only be included to separate multiple clauses on a single line.

Notes:

1. A comment is not a token, so therefore a comment may follow the continuation character on a line.
2. Semicolons are added automatically by CREXX after certain instruction keywords when in the correct context. The keywords that may have this effect are `else`, `finally`, `otherwise`, `then`; they become complete clauses in their own right when this occurs. These special cases reduce program entry errors significantly.

2.5 The case of names and symbols

In general, CREXX is a *case-insensitive* language. That is, the names of keywords, variables, and so on, will be recognized independently of the case used for each letter in a name; the name "**Swildon**" would match the name "**swilDon**".

CREXX, however, uses names that may be visible outside the CREXX program, and these may well be referenced by case-sensitive languages. Therefore, any name that has an external use (such as the name of a property, method, constructor, or class) has a defined spelling, in which each letter of the name has the case used for that letter when the name was first defined or used.

Similarly, the lookup of external names is both case-preserving and case-insensitive. If a class, method, or property is referenced by the name "**Foo**", for example, an exact-case match will first be tried at each point that a search is made. If this succeeds, the search for a matching name is complete. If it does not succeed, a case-insensitive search in the same context is carried out, and if one item is found, then the search is complete. If more than one item matches then the reference is ambiguous, and an error is reported.

Implementations are encouraged to offer an option that requires that all name matches are exact (case-sensitive), for programmers or house-styles that prefer that approach to name matching.

2.6 Hexadecimal and binary numeric symbols

A *hexadecimal numeric symbol* describes a whole number, and is of the form *n***X***string*. Here, *n* is a simple number with no decimal part (and optional leading insignificant zeros) which describes the effective length of the hexadecimal string, the **X** (which may be in lowercase) indicates that the notation is hexadecimal, and *string* is a string of one or more hexadecimal characters (characters from the ranges "a-f", "A-F", and the digits "0-9").

The *string* is taken as a signed number expressed in *n* hexadecimal characters. If necessary, *string* is padded on the left with "**0**" characters (note, not "sign-extended") to length *n* characters.

If the most significant (left-most) bit of the resulting string is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both cases it is converted to a CREXX number which may, therefore, be negative. The result of the conversion is a number comprised of the Arabic digits 0-9, with no insignificant leading zeros but possibly with a leading "**-**".

The value *n* may not be less than the number of characters in *string*, with the single exception that it may be zero, which indicates that the number is always positive (as though *n* were greater than the the length of *string*).

Examples:

```
1x8    == -8
2x8    == 8
2x08   == 8
0x08   == 8
0x10   == 16
0x81   == 129
2x81   == -127
3x81   == 129
4x81   == 129
04x81  == 129
16x81  == 129
4xF081 == -3967
8xF081 == 61569
0Xf081 == 61569
```

A *binary numeric symbol* describes a whole number using the same rules, except that the identifying character is **B** or **b**, and the digits of *string* must be either **0** or **1**, each representing a single bit.

Examples:

```
1b0    == 0
1b1    == -1
0b10   == 2
0b100  == 4
4b1000 == -8
8B1000 == 8
```

Note: Hexadecimal and binary numeric symbols are a purely syntactic device for representing decimal whole numbers. That is, they are recognized only within the source of a CREXX program, and are not equivalent to a literal string

with the same characters within quotes.

3

Built-in functions for CREXX strings

This section describes the set of functions defined for CREXX strings These are called *built-in functions*, and include character manipulation, word manipulation, conversion, and arithmetic functions.

Implementations will also provide other functions for the `Rexx` class (for example, to implement the CREXX operators or to provide constructors with primitive arguments), but these are not part of the CREXX language.

Functions of the `RexxStream`, `RexxTime` and `RexxDate` classes are, while not strictly part of the `Rexx` string class, listed here, followed by a suitable referral to the proper class documentation. [13]

General notes on the built-in functions:

1. All functions work on a CREXX string of type `Rexx`; this is referred to by the name *string* in the descriptions of the functions. For example, if the **word** method were invoked using the term:

   ```
   "Three word phrase".word(2)
   ```

   then in the description of **word** the name *string* refers to the string "**Three word phrase**", and the name *n* refers to the string "**2**".
2. All method arguments are of type `Rexx` and all functions return a string of type `Rexx`; if a number is returned, it will be formatted as though 0 had been added with no rounding.
3. The first parenthesis in a method call must immediately follow the name of the method, with no space in between.
4. The parentheses in a method call can be omitted if no arguments are required and the method call is part of a [14]
5. A position in a string is the number of a character in the string, where the first character is at position 1, *etc.*
6. Where arguments are optional, commas may only be included between arguments that are present (that is, trailing commas in argument lists are not permitted).
7. A *pad* argument, if specified, must be exactly one character long.
8. If a method has a sub-option selected by the first character of a string, that character may be in upper or lowercase.
9. Conversion between character encodings and decimal or hexadecimal is dependent on the machine representation (encoding) of characters

[13] *Details of the functions provided in the reference implementation are included in Appendix C*) .
[14] Unless an implementation-provided option to disallow parenthesis omission is in force.

and hence will return appropriately different results for Unicode, ASCII, EBCDIC, and other implementations.

3.1 abbrev(info [,length])

returns 1 if *info* is equal to the leading characters of *string* and *info* is not less than the minimum length, *length*; 0 is returned if either of these conditions is not met. *length* must be a non-negative whole number; the default is the length of *info*. **Examples:**

```
'Print'.abbrev('Pri')   == 1
'PRINT'.abbrev('Pri')   == 0
'PRINT'.abbrev('PRI',4) == 0
'PRINT'.abbrev('PRY')   == 0
'PRINT'.abbrev('')      == 1
'PRINT'.abbrev('',1)    == 0
```

Note: A null string will always match if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired. **Example:**

```
say 'Enter option:';  option=ask
select  /* keyword1 is to be the default */
  when 'keyword1'.abbrev(option) then ...
  when 'keyword2'.abbrev(option) then ...
     ...
  otherwise ...
  end
```

3.2 abs()

returns the absolute value of *string*, which must be a number. Any sign is removed from the number, and it is then formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

Examples:

```
'12.3'.abs              == 12.3
' -0.307'.abs           == 0.307
'123.45E+16'.abs        == 1.2345E+18
'- 1234567.7654321'.abs == 1234567.7654321
```

3.3 b2d([n])

3.02 Binary to decimal. Converts *string*, a string of at least one binary (**0** and/or **1**) digits, to an equivalent string of decimal characters (a number), without rounding. The returned string will use digits, and will not include any blanks.

If the number of binary digits in the string is not a multiple of four, then up to three **'0'** digits will be added on the left before conversion to make a total that is a multiple of four. If *string* is the null string, 0 is returned. If n is not specified, *string* is taken to be an unsigned number.

Examples:

```
'01110'.b2d == 14
'10000001'.b2d == 129
'111110000001'.b2d == 3969
'1111111110000001'.b2d == 65409
'1100011011110000'.b2d == 50928
```

If n is specified, string is taken as a signed number expressed in n binary characters. If the most significant (left-most) bit is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both cases it is converted to a NetRexx number which may, therefore, be negative. If n is 0, 0 is always returned.

If necessary, string is padded on the left with '0' characters (note, not "signextended"), or truncated on the left, to length n characters; (that is, as though string.right(n, '0') had been executed.)

Examples:

```
'10000001'.b2d(8) == -127
'10000001'.b2d(16) == 129
'1111000010000001'.b2d(16) == -3967
'1111000010000001'.b2d(12) == 129
'1111000010000001'.b2d(8) == -127
'1111000010000001'.b2d(4) == 1
'0000000000110001'.b2d(0) == 0
```

3.4 b2x()

Binary to hexadecimal. Converts *string*, a string of at least one binary (**0** and/or **1**) digits, to an equivalent string of hexadecimal characters. The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks. If the number of binary digits in the string is not a multiple of four, then up to three **'0'** digits will be added on the left before conversion to make a total that is a multiple of four.

Examples:

```
'11000011'.b2x  == 'C3'
'10111'.b2x     == '17'
'0101'.b2x      == '5'
'101'.b2x       == '5'
'111110000'.b2x == '1F0'
```

3.5 center(length [,pad])

or

3.6 centre(length [,pad])

returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up the required length. *length* must be a non-negative whole number. The default *pad* character is blank. If the string is longer than *length*, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right hand end loses or gains one more character than the left hand end.

Examples:

```
'ABC'.centre(7)          == '  ABC  '
'ABC'.center(8,'-')      == '--ABC---'
'The blue sky'.centre(8) == 'e blue s'
'The blue sky'.center(7) == 'e blue '
```

Note: This method may be called either **centre** or **center**, which avoids difficulties due to the difference between the British and American spellings.

3.7 changestr(needle, new)

returns a copy of *string* in which each occurrence of the *needle* string is replaced by the *new* string. Each unique (non-overlapping) occurrence of the *needle* string is changed, searching from left to right and starting from the first (leftmost) position in *string*. Only the original *string* is searched for the *needle*, and each character in *string* can only be included in one match of the *needle*.

If the *needle* is the null string, the result is a copy of *string*, unchanged.

Examples:

```
'elephant'.changestr('e','X')    == 'XlXphant'
'elephant'.changestr('ph','X')   == 'eleXant'
'elephant'.changestr('ph','hph') == 'elehphant'
'elephant'.changestr('e','')     == 'lphant'
'elephant'.changestr('','!!')    == 'elephant'
```

3.8 compare(target [,pad])

returns 0 if *string* and *target* are the same. If they are not, the returned number is positive and is the position of the first character that is not the same in both strings. If one string is shorter than the other, one or more *pad* characters are added on the right to make it the same length for the comparison. The default *pad* character is a blank.

Examples:

```
'abc'.compare('abc')      == 0
'abc'.compare('ak')       == 2
'ab '.compare('ab')       == 0
'ab '.compare('ab',' ')   == 0
'ab '.compare('ab','x')   == 3
'ab-- '.compare('ab','-') == 5
```

3.9 copies(n)

returns *n* directly concatenated copies of *string*. *n* must be positive or 0; if 0, the null string is returned.

Examples:

```
'abc'.copies(3) == 'abcabcabc'
'abc'.copies(0) == ''
''.copies(2)    == ''
```

3.10 countstr(needle)

returns the count of non-overlapping occurrences of the *needle* string in *string*, searching from left to right and starting from the first (leftmost) position in *string*.

If the *needle* is the null string, **0** is returned.

Examples:

```
'elephant'.countstr('e')  == '2'
'elephant'.countstr('ph') == '1'
'elephant'.countstr('')   == '0'
```

The **changestr** method can be used to change occurrences of *needle* to some other string.

3.11 c2d()

Coded character to decimal. Converts the encoding of the character in *string* (which must be exactly one character) to its decimal representation. The returned string will be a non-negative number that represents the encoding of the character and will not include any sign, blanks, insignificant leading zeros, or decimal part.

Examples:

```
'M'.c2d  == '77'  -- ASCII or Unicode
'7'.c2d  == '247' -- EBCDIC
'\textbackslash{}r'.c2d == '13'  -- ASCII or Unicode
'\textbackslash{}0'.c2d == '0'
```

The **c2x** method can be used to convert the encoding of a character to a hexadecimal representation.

3.12 c2x()

Coded character to hexadecimal. Converts the encoding of the character in *string* (which must be exactly one character) to its hexadecimal representation (unpacks). The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks. Insignificant leading zeros are removed.

Examples:

```
'M'.c2x  == '4D' -- ASCII or Unicode
'7'.c2x  == 'F7' -- EBCDIC
'\textbackslash{}r'.c2x == 'D'  -- ASCII or Unicode
'\textbackslash{}0'.c2x == '0'
```

The **c2d** method can be used to convert the encoding of a character to a decimal number.

3.13 datatype(option)

returns 1 if *string* matches the description requested with the *option*, or 0 otherwise. If *string* is the null string, 0 is always returned.

Only the first character of *option* is significant, and it may be in either uppercase or lowercase. The following *option* characters are recognized:

A (Alphanumeric); returns 1 if *string* only contains characters from the ranges "a-z", "A-Z", and "0-9".

B (Binary); returns 1 if *string* only contains the characters "0" and/or "1".

D (Digits); returns 1 if *string* only contains characters from the range "0-9".

L (Lowercase); returns 1 if *string* only contains characters from the range "a-z".

M (Mixed case); returns 1 if *string* only contains characters from the ranges "a-z" and "A-Z".

N (Number); returns 1 if *string* is a syntactically valid CREXX number that could be added to **'0'** without error,

S (Symbol); returns 1 if *string* only contains characters that are valid in non-numeric symbols (the alphanumeric characters and underscore), and does not start with a digit. Note that both uppercase and lowercase letters are permitted.

U (Uppercase); returns 1 if *string* only contains characters from the range "A-Z".

W (Whole Number); returns 1 if *string* is a syntactically valid CREXX number that can be added to **'0'** without error, and whose decimal part after that addition, with no rounding, is zero.

X (heXadecimal); returns 1 if *string* only contains characters from the ranges "a-f", "A-F", and "0-9".

Examples:

```
'101'.datatype('B')    == 1
'12.3'.datatype('D')   == 0
'12.3'.datatype('N')   == 1
'12.3'.datatype('W')   == 0
'LaArca'.datatype('M') == 1
''.datatype('M')       == 0
'Llanes'.datatype('L') == 0
'3 d'.datatype('s')    == 0
'BCd3'.datatype('X')   == 1
'BCgd3'.datatype('X')  == 0
```

Note: The **datatype** method tests the meaning of the characters in a string, independent of the encoding of those characters. Extra letters and Extra digits cause **datatype** to return 0 except for the number tests ("**N**" and "**W**"), which treat extra digits whose value is in the range 0-9 as though they were the corresponding Arabic numeral.

3.14 delstr(n [,length])

returns a copy of *string* with the sub-string of *string* that begins at the *nth* character, and is of length *length* characters, deleted. If *length* is not specified, or is greater than the number of characters from *n* to the end of the string, the rest of the string is deleted (including the *nth* character). *length* must be a non-negative whole number, and *n* must be a positive whole number. If *n* is greater than the length of *string*, the string is returned unchanged.

Examples:

```
'abcd'.delstr(3)    == 'ab'
'abcde'.delstr(3,2) == 'abe'
'abcde'.delstr(6)   == 'abcde'
```

3.15 delword(n [,length])

returns a copy of *string* with the sub-string of *string* that starts at the *nth* word, and is of length *length* blank-delimited words, deleted. If *length* is not specified, or is greater than number of remaining words in the string, it defaults to be the remaining words in the string (including the *nth* word). *length* must be a non-negative whole number, and *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the string is returned unchanged. The string deleted includes any blanks following the final word involved, but none of the blanks preceding the first word involved.

Examples:

```
'Now is the  time'.delword(2,2) == 'Now time'
'Now is the time '.delword(3)   == 'Now is '
'Now  time'.delword(5)          == 'Now  time'
```

3.16 d2b([n])

3.02 Decimal to binary. Returns a string of binary characters of length as needed or of length n, which is the binary representation of the decimal number. The returned string will use 0 and 1 characters for binary values. string must be a whole number, and must be non-negative unless n is specified, or an error will result. If n is not specified, the length of the result returned is such that there are no leading 0 characters, unless string was equal to 0 (in which case '0' is returned).

If n is specified it is the length of the final result in characters; that is, after conversion the input string will be sign-extended to the required length (negative numbers are converted assuming twos-complement form). If the number is too big to fit into n characters, it will be truncated on the left. n must be a nonnegative whole number.

Examples:

```
'0'.d2b == 0
'9'.d2b == 1001
'19'.d2b == 10011
'129'.d2b == 10000001
'129'.d2b(1) == 1
'129'.d2b(8) == 10000001
'127'.d2b(12) == 000001111111
'129'.d2b(16) == 0000000010000001
'257'.d2b(8) == 00000001
'-127'.d2b(8) == 10000001
'-127'.d2b(16) == 1111111110000001
'12'.d2b(0) ==
```

3.17 d2c()

Decimal to coded character. Converts the *string* (a CREXX *number*) to a single character, where the number is used as the encoding of the character.

string must be a non-negative whole number. An error results if the encoding described does not produce a valid character for the implementation (for example, if it has more significant bits than the implementation's encoding for characters).

Examples:

```
'77'.d2c  == 'M' -- ASCII or Unicode
'+77'.d2c == 'M' -- ASCII or Unicode
'247'.d2c == '7' -- EBCDIC
'0'.d2c   == '\textbackslash 0'
```

3.18 d2x([n])

Decimal to hexadecimal. Returns a string of hexadecimal characters of length as needed or of length *n*, which is the hexadecimal (unpacked) representation of

the decimal number. The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks. *string* must be a whole number, and must be non-negative unless *n* is specified, or an error will result. If *n* is not specified, the length of the result returned is such that there are no leading 0 characters, unless *string* was equal to 0 (in which case **'0'** is returned).

If *n* is specified it is the length of the final result in characters; that is, after conversion the input string will be sign-extended to the required length (negative numbers are converted assuming twos-complement form). If the number is too big to fit into *n* characters, it will be truncated on the left. *n* must be a non-negative whole number.

Examples:

```
'9'.d2x       == '9'
'129'.d2x     == '81'
'129'.d2x(1)  == '1'
'129'.d2x(2)  == '81'
'127'.d2x(3)  == '07F'
'129'.d2x(4)  == '0081'
'257'.d2x(2)  == '01'
'-127'.d2x(2) == '81'
'-127'.d2x(4) == 'FF81'
'12'.d2x(0)   == ''
```

3.19 exists(index)

returns 1 if *index* names a sub-value of *string* that has explicitly been assigned a value, or 0 otherwise.

Example: Following the instructions:

```
vowel=0
vowel['a']=1
vowel['b']=1
vowel['b']=null -- drops previous assignment
```

then:

```
vowel.exists('a') == '1'
vowel.exists('b') == '0'
vowel.exists('c') == '0'
```

3.20 format([before [,after]])

formats (lays out) *string*, which must be a number.

The number, *string*, is first formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). If no arguments are given, the result is precisely that of this operation.

The arguments *before* and *after* may be specified to control the number of characters to be used for the integer part and decimal part of the result respectively. If either of these is omitted (with no arguments specified to its right), or is **null**, the number of characters used will be as many as are needed for that part.

before must be a positive number; if it is larger than is needed to contain the integer part, that part is padded on the left with blanks to the requested length. If *before* is not large enough to contain the integer part of the number (including the sign, for negative numbers), an error results.

after must be a non-negative number; if it is not the same size as the decimal part of the number, the number will be rounded (or extended with zeros) to fit. Specifying 0 for *after* will cause the number to be rounded to an integer (that is, it will have no decimal part or decimal point).

Examples:

```
' - 12.73'.format         == '-12.73'
'0.000'.format            == '0'
'3'.format(4)             == '   3'
'1.73'.format(4,0)        == '   2'
'1.73'.format(4,3)        == '   1.730'
'-.76'.format(4,1)        == '  -0.8'
'3.03'.format(4)          == '   3.03'
' - 12.73'.format(null,4) == '-12.7300'
```

Further arguments may be passed to the `format` method to control the use of exponential notation. The full syntax of the method is then:

`format([before[,after[,explaces[,exdigits[,exform]]]]])` The first two arguments are as already described. The other three (*explaces*, *exdigits*, and *exform*) control the exponent part of the result. The default for any of the arguments may be selected by omitting them (if there are no arguments to be specified to their right) or by using the value **null**.

explaces must be a positive number; it sets the number of places (digits after the sign of the exponent) to be used for any exponent part, the default being to use as many as are needed. If *explaces* is specified and is not large enough to contain the exponent, an error results. If *explaces* is specified and the exponent will be 0, then *explaces*+2 blanks are supplied for the exponent part of the result.

exdigits sets the trigger point for use of exponential notation. If, after the first formatting, the number of places needed before the decimal point exceeds *exdigits*, or if the absolute value of the result is less than **0.000001**, then exponential form will be used, provided that *exdigits* was specified. When *exdigits* is not specified, exponential notation will never be used. The current setting of `numeric digits` may be used for *exdigits* by specifying the special word **digits** . If 0 is specified for *exdigits*, exponential notation is always used unless the exponent would be 0.

exform sets the form for exponential notation (if needed). *exform* may be either **'Scientific'** (the default) or **'Engineering'**. Only the first character of *exform* is significant and it may be in uppercase or in lowercase. The current setting of `numeric form` may be used by specifying the special word **form** . If engineering

form is in effect, up to three digits (plus sign) may be needed for the integer part of the result (*before*).

Examples:

```
'12345.73'.format(null,null,2,2) == '1.234573E+04'
'12345.73'.format(null,3,null,0) == '1.235E+4'
'1.234573'.format(null,3,null,0) == '1.235'
'123.45'.format(null,3,2,0)      == '1.235E+02'
'1234.5'.format(null,3,2,0,'e')  == '1.235E+03'
'1.2345'.format(null,3,2,0)      == '1.235    '
'12345.73'.format(null,null,3,6) == '12345.73     '
'12345e+5'.format(null,3)        == '1234500000.000'
```

Implementation minimum: If exponents are supported in an implementation, then they must be supported for exponents whose absolute value is at least as large as the largest number that can be expressed as an exact integer in default precision, *i.e.*, 999999999. Therefore, values for *explaces* of up to 9 should also be supported.

3.21 insert(new [,n [,length [,pad]]])

inserts the string *new*, padded or truncated to length *length*, into a copy of the target *string* after the *nth* character; the string with any inserts is returned. *length* and *n* must be a non-negative whole numbers. If *n* is greater than the length of the target string, padding is added before the *new* string also. The default value for *n* is 0, which means insert before the beginning of the string. The default value for *length* is the length of *new*. The default *pad* character is a blank.

Examples:

```
'abc'.insert('123')         == '123abc'
'abcdef'.insert(' ',3)      == 'abc def'
'abc'.insert('123',5,6)     == 'abc  123   '
'abc'.insert('123',5,6,'+') == 'abc++123+++'
'abc'.insert('123',0,5,'-') == '123--abc'
```

3.22 lastpos(needle [,start])

returns the position of the last occurrence of the string *needle* in *string* (the "haystack"), searching from right to left. If the string *needle* is not found, or is the null string, 0 is returned. By default the search starts at the last character of *string* and scans backwards. This may be overridden by specifying *start*, the point at which to start the backwards scan. *start* must be a positive whole number, and defaults to the value *string*.**length** if larger than that value or if not specified (with a minimum default value of one).

Examples:

```
'abc def ghi'.lastpos(' ')   == 8
'abc def ghi'.lastpos(' ',7) == 4
```

```
'abcdefghi'.lastpos(' ')     == 0
'abcdefghi'.lastpos('cd')    == 3
''.lastpos('?')              == 0
```

3.23 left(length [,pad])

returns a string of length *length* containing the left-most *length* characters of *string*. The string is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be a non-negative whole number. This method is exactly equivalent to *string*.**substr**(**1**, *length* [, *pad*]).

Examples:

```
'abc d'.left(8)     == 'abc d   '
'abc d'.left(8,'.') == 'abc d...'
'abc defg'.left(6)  == 'abc de'
```

3.24 length()

returns the number of characters in *string*.

Examples:

```
'abcdefgh'.length == 8
''.length         == 0
```

3.25 linein(name,string)

reads a line from the stream named by the first argument, unless the third argument is zero, see page for Class `RexxStream` information.

3.26 lineout(name,string)

returns '1' or '0', indicating whether the second argument has been successfully written to the stream named by the first argument. A result of '1' means an unsuccessful write, see page for Class `RexxStream` information.

3.27 lower([n [,length]])

returns a copy of *string* with any uppercase characters in the sub-string of *string* that begins at the *nth* character, and is of length *length* characters, replaced by their lowercase equivalent.

n must be a positive whole number, and defaults to 1 (the first character in *string*). If *n* is greater than the length of *string*, the string is returned unchanged.

length must be a non-negative whole number. If *length* is not specified, or is greater than the number of characters from *n* to the end of the string, the rest of the string (including the *nth* character) is assumed.

Examples:

```
'SumA'.lower      == 'suma'
'SumA'.lower(2)   == 'Suma'
'SuMB'.lower(1,1) == 'suMB'
'SUMB'.lower(2,2) == 'SumB'
''.lower          == ''
```

3.28 max(number)

returns the larger of *string* and *number*, which must both be numbers. If they compare equal (that is, when subtracted, the result is 0), then *string* is selected for the result.

The comparison is effected using a numerical comparison with a digits setting that is either nine or, if greater, the larger of the number of digits in the mantissas of the two numbers (excluding leading insignificant zeros).

The selected result is formatted by adding zero to the selected number with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

Examples:

```
0.max(1)          ==1
'-1'.max(1)       ==1
'+1'.max(-1)      ==1
'1.0'.max(1.00)   =='1.0'
'1.00'.max(1.0)   =='1.00'
'123456700000'.max(1234567E+5)   == '123456700000'
'1234567E+5'.max('123456700000') == '1.234567E+11'
```

3.29 min(number)

returns the smaller of *string* and *number*, which must both be numbers. If they compare equal (that is, when subtracted, the result is 0), then *string* is selected for the result.

The comparison is effected using a numerical comparison with a digits setting that is either nine or, if greater, the larger of the number of digits in the mantissas of the two numbers (excluding leading insignificant zeros).

The selected result is formatted by adding zero to the selected number with a digits setting that is either nine or, if greater, the number of digits in the mantissa

of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

Examples:

```
0.min(1)           ==0
'-1'.min(1)        =='-1'
'+1'.min(-1)       =='-1'
'1.0'.min(1.00)    =='1.0'
'1.00'.min(1.0)    =='1.00'
'123456700000'.min(1234567E+5)   == '123456700000'
'1234567E+5'.min('123456700000') == '1.234567E+11'
```

3.30 overlay(new [,n [,length [,pad]]])

overlays the string *new*, padded or truncated to length *length*, onto a copy of the target *string* starting at the *nth* character; the string with any overlays is returned. Overlays may extend beyond the end of the original *string*. If *length* is specified it must be a non-negative whole number. If *n* is greater than the length of the target string, padding is added before the *new* string also. The default *pad* character is a blank, and the default value for *n* is 1. *n* must be greater than 0. The default value for *length* is the length of *new*.

Examples:

```
'abcdef'.overlay(' ',3)      == 'ab def'
'abcdef'.overlay('.',3,2)    == 'ab. ef'
'abcd'.overlay('qq')         == 'qqcd'
'abcd'.overlay('qq',4)       == 'abcqq'
'abc'.overlay('123',5,6,'+') == 'abc+123+++'
```

3.31 pos(needle [,start])

returns the position of the string *needle*, in *string* (the "haystack"), searching from left to right. If the string *needle* is not found, or is the null string, 0 is returned. By default the search starts at the first character of *string* (that is, *start* has the value 1). This may be overridden by specifying *start* (which must be a positive whole number), the point at which to start the search; if *start* is greater than the length of *string* then 0 is returned. **Examples:**

```
'Saturday'.pos('day')    == 6
'abc def ghi'.pos('x')   == 0
'abc def ghi'.pos(' ')   == 4
'abc def ghi'.pos(' ',5) == 8
```

3.32 reverse()

returns a copy of *string*, swapped end for end.

Examples:

```
'ABc.'.reverse        == '.cBA'
'XYZ '.reverse        == ' ZYX'
'Tranquility'.reverse == 'ytiliuqnarT'
```

3.33 right(length [,pad])

returns a string of length *length* containing the right-most *length* characters of *string* - that is, padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. *length* must be a non-negative whole number.

Examples:

```
'abc  d'.right(8)  == '  abc  d'
'abc def'.right(5) == 'c def'
'12'.right(5,'0')  == '00012'
```

3.34 sequence(final)

returns a string of all characters, in ascending order of encoding, between and including the character in *string* and the character in *final*. *string* and *final* must be single characters; if *string* is greater than *final*, an error is reported.

Examples:

```
'a'.sequence('f')           == 'abcdef'
'\\0'.sequence('\\x03')       == '\\x00\\x01\\x02\\x03'
'\\ufffe'.sequence('\\uffff') == '\\ufffe\\uffff'
```

3.35 sign()

returns a number that indicates the sign of *string*, which must be a number. *string* is first formatted, just as though the operation "**string+0**" had been carried out with sufficient digits to avoid rounding. If the number then starts with **'-'** then **'-1'** is returned; if it is **'0'** then **'0'** is returned; and otherwise **'1'** is returned.

Examples:

```
'12.3'.sign    ==  1
'0.0'.sign     ==  0
' -0.307'.sign == -1
```

3.36 soundex()

returns the normalized soundex value of the string. This implementation is for the English language. 3.08

Examples:

```
'EULER'.soundex()  == 'E460'
```

3.37 space([n [,pad]])

returns a copy of *string* with the blank-delimited words in *string* formatted with *n* (and only *n*) *pad* characters between each word. *n* must be a non-negative whole number. If *n* is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is 1, and the default *pad* character is a blank.

Examples:

```
'abc  def  '.space        == 'abc def'
'  abc def '.space(3)     == 'abc   def'
'abc  def  '.space(1)     == 'abc def'
'abc  def  '.space(0)     == 'abcdef'
'abc  def  '.space(2,'+') == 'abc++def'
```

3.38 stream(name, [operation], [stream_command])

q (Operations) returns a description of the state of, or the result of an operation upon, the stream named by the first argument, for Class `RexxStream` information.

3.39 strip([option [,char]])

returns a copy of *string* with Leading, Trailing, or Both leading and trailing characters removed, when the first character of *option* is L, T, or B respectively (these may be given in either uppercase or lowercase). The default is B. The second argument, *char*, specifies the character to be removed, with the default being a blank. If given, *char* must be exactly one character long.

Examples:

```
'  ab c  '.strip        == 'ab c'
'  ab c  '.strip('L')   == 'ab c  '
'  ab c  '.strip('t')   == '  ab c'
'12.70000'.strip('t',0) == '12.7'
'0012.700'.strip('b',0) == '12.7'
```

3.40 substr(n [,length [,pad]])

returns the sub-string of *string* that begins at the *nth* character, and is of length *length*, padded with *pad* characters if necessary. *n* must be a positive whole number, and *length* must be a non-negative whole number. If *n* is greater than *string*.**length**, then only pad characters can be returned. If *length* is omitted it

defaults to be the rest of the string (or 0 if *n* is greater than the length of the string). The default *pad* character is a blank.

Examples:

```
'abc'.substr(2)       == 'bc'
'abc'.substr(2,4)     == 'bc  '
'abc'.substr(5,4)     == '    '
'abc'.substr(2,6,'.') == 'bc....'
'abc'.substr(5,6,'.') == '......'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting sub-strings, especially if more than one sub-string is to be extracted from a string.

3.41 subword(n [,length])

returns the sub-string of *string* that starts at the *nth* word, and is up to *length* blank-delimited words long. *n* must be a positive whole number; if greater than the number of words in the string then the null string is returned. *length* must be a non-negative whole number. If *length* is omitted it defaults to be the remaining words in the string. The returned string will never have leading or trailing blanks, but will include all blanks between the selected words.

Examples:

```
'Now is the  time'.subword(2,2) == 'is the'
'Now is the  time'.subword(3)   == 'the  time'
'Now is the  time'.subword(5)   == ''
```

3.42 translate(tableo, tablei [,pad])

returns a copy of *string* with each character in *string* either unchanged or translated to another character.

The **translate** method acts by searching the input translate table, *tablei*, for each character in *string*. If the character is found in *tablei* (the first, leftmost, occurrence being used if there are duplicates) then the corresponding character in the same position in the output translate table, *tableo*, is used in the result string; otherwise the original character found in *string* is used. The result string is always the same length as *string*.

The translate tables may be of any length, including the null string. The output table, *tableo*, is padded with *pad* or truncated on the right as necessary to be the same length as *tablei*. The default *pad* is a blank.

Examples:

```
'abbc'.translate('&','b')           == 'a&&c'
'abcdef'.translate('12','ec')       == 'ab2d1f'
'abcdef'.translate('12','abcd','.') == '12..ef'
'4123'.translate('abcd','1234')     == 'dabc'
'4123'.translate('hods','1234')     == 'shod'
```

Note: The last two examples show how the **translate** method may be used to move around the characters in a string. In these examples, any 4-character string could be specified as the first argument and its last character would be moved to the beginning of the string. Similarly, the term:

```
'gh.ef.abcd'.translate(19970827,'abcdefgh')
```

(which returns "**27.08.1997**") shows how a string (in this case perhaps a date) might be re-formatted and merged with other characters using the **translate** method.

3.43 trunc([n])

returns the integer part of *string*, which must be a number, with *n* decimal places (digits after the decimal point). *n* must be a non-negative whole number, and defaults to zero.

The number *string* is formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). It is then truncated to *n* decimal places (or trailing zeros are added if needed to make up the specified length). If *n* is 0 (the default) then an integer with no decimal point is returned. The result will never be in exponential form.

Examples:

```
'12.3'.trunc          == 12
'127.09782'.trunc(3) == 127.097
'127.1'.trunc(3)      == 127.100
'127'.trunc(2)        == 127.00
'0'.trunc(2)          == 0.00
```

3.44 upper([n [,length]])

returns a copy of *string* with any lowercase characters in the sub-string of *string* that begins at the *nth* character, and is of length *length* characters, replaced by their uppercase equivalent.

n must be a positive whole number, and defaults to 1 (the first character in *string*). If *n* is greater than the length of *string*, the string is returned unchanged.

length must be a non-negative whole number. If *length* is not specified, or is greater than the number of characters from *n* to the end of the string, the rest of the string (including the *nth* character) is assumed.

Examples:

```
'Fou-Baa'.upper         == 'FOU-BAA'
'Mad Sheep'.upper       == 'MAD SHEEP'
'Mad sheep'.upper(5)    == 'Mad SHEEP'
'Mad sheep'.upper(5,1) == 'Mad Sheep'
'Mad sheep'.upper(5,4) == 'Mad SHEEp'
```

```
'tinganon'.upper(1,1)  == 'Tinganon'
''.upper               == ''
```

3.45 verify(reference [,option [,start]])

verifies that *string* is composed only of characters from *reference*, by returning the position of the first character in *string* that is not also in *reference*. If all the characters were found in *reference*, 0 is returned. The *option* may be either **'Nomatch'** (the default) or **'Match'**. Only the first character of *option* is significant and it may be in uppercase or in lowercase. If **'Match'** is specified, the position of the first character in *string* that **is** in *reference* is returned, or 0 is returned if none of the characters were found. The default for *start* is 1 (that is, the search starts at the first character of *string*). This can be overridden by giving a different *start* point, which must be positive. If *string* is the null string, the method returns 0, regardless of the value of the *option*. Similarly if *start* is greater than *string*.**length**, 0 is returned. If *reference* is the null string, then the returned value is the same as the value used for *start*, unless **'Match'** is specified as the *option*, in which case 0 is returned.

Examples:

```
'123'.verify('1234567890')          == 0
'1Z3'.verify('1234567890')          == 2
'AB4T'.verify('1234567890','M')     == 3
'1P3Q4'.verify('1234567890','N',3)  == 4
'ABCDE'.verify('','n',3)            == 3
'AB3CD5'.verify('1234567890','m',4) == 6
```

3.46 word(n)

returns the *nth* blank-delimited word in *string*. *n* must be positive. If there are fewer than *n* words in *string*, the null string is returned. This method is exactly equivalent to *string*.**subword**(*n*,**1**).

Examples:

```
'Now is the time'.word(3) == 'the'
'Now is the time'.word(5) == ''
```

3.47 wordindex(n)

returns the character position of the *nth* blank-delimited word in *string*. *n* must be positive. If there are fewer than *n* words in the string, 0 is returned.

Examples:

```
'Now is the time'.wordindex(3) == 8
'Now is the time'.wordindex(6) == 0
```

3.48 wordlength(n)

returns the length of the *nth* blank-delimited word in *string*. *n* must be positive. If there are fewer than *n* words in the string, 0 is returned.

Examples:

```
'Now is the time'.wordlength(2)    == 2
'Now comes the time'.wordlength(2) == 5
'Now is the time'.wordlength(6)    == 0
```

3.49 wordpos(phrase [,start])

searches *string* for the first occurrence of the sequence of blank-delimited words *phrase*, and returns the word number of the first word of *phrase* in *string*. Multiple blanks between words in either *phrase* or *string* are treated as a single blank for the comparison, but otherwise the words must match exactly. Similarly, leading or trailing blanks on either string are ignored. If *phrase* is not found, or contains no words, 0 is returned. By default the search starts at the first word in *string*. This may be overridden by specifying *start* (which must be positive), the word at which to start the search.

Examples:

```
'now is the time'.wordpos('the')       == 3
'now is the time'.wordpos('The')       == 0
'now is the time'.wordpos('is the')    == 2
'now is the time'.wordpos('is    the') == 2
'now is the time'.wordpos('is  time')  == 0
'To be or not to be'.wordpos('be')     == 2
'To be or not to be'.wordpos('be',3)   == 6
```

3.50 words()

returns the number of blank-delimited words in *string*.

Examples:

```
'Now is the time'.words == 4
' '.words              == 0
''.words               == 0
```

3.51 x2b()

Hexadecimal to binary. Converts *string* (a string of at least one hexadecimal characters) to an equivalent string of binary digits. Hexadecimal characters may be any decimal digit character (0-9) or any of the first six alphabetic characters (a-f), in either lowercase or uppercase. *string* may be of any length;

each hexadecimal character with be converted to a string of four binary digits. The returned string will have a length that is a multiple of four, and will not include any blanks.

Examples:

```
'C3'.x2b  == '11000011'
'7'.x2b   == '0111'
'1C1'.x2b == '000111000001'
```

3.52 x2c()

Hexadecimal to coded character. Converts the *string* (a string of hexadecimal characters) to a single character (packs). Hexadecimal characters may be any decimal digit character (0-9) or any of the first six alphabetic characters (a-f), in either lowercase or uppercase.

string must contain at least one hexadecimal character; insignificant leading zeros are removed, and the string is then padded with leading zeros if necessary to make a sufficient number of hexadecimal digits to describe a character encoding for the implementation.

An error results if the encoding described does not produce a valid character for the implementation (for example, if it has more significant bits than the implementation's encoding for characters). **Examples:**

```
'004D'.x2c == 'M' -- ASCII or Unicode
'4d'.x2c   == 'M' -- ASCII or Unicode
'A2'.x2c   == 's' -- EBCDIC
'0'.x2c    == '\textbackslash 0'
```

The **d2c** method can be used to convert a CREXX number to the encoding of a character.

3.53 x2d([n])

Hexadecimal to decimal. Converts the *string* (a string of hexadecimal characters) to a decimal number, without rounding. If *string* is the null string, 0 is returned.

If *n* is not specified, *string* is taken to be an unsigned number.

Examples:

```
'0E'.x2d    == 14
'81'.x2d    == 129
'F81'.x2d   == 3969
'FF81'.x2d  == 65409
'c6f0'.x2d  == 50928
```

If *n* is specified, *string* is taken as a signed number expressed in *n* hexadecimal characters. If the most significant (left-most) bit is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both

cases it is converted to a CREXX number which may, therefore, be negative. If *n* is 0, 0 is always returned.

If necessary, *string* is padded on the left with **'0'** characters (note, not "sign-extended"), or truncated on the left, to length *n* characters; (that is, as though *string*.**right**(*n*, **'0'**) had been executed.)

Examples:

```
'81'.x2d(2)   == -127
'81'.x2d(4)   == 129
'F081'.x2d(4) == -3967
'F081'.x2d(3) == 129
'F081'.x2d(2) == -127
'F081'.x2d(1) == 1
'0031'.x2d(0) == 0
```

The **c2d** method can be used to convert a character to a decimal representation of its encoding.

List of Tables

Index

ISBN 978-90-819090-1-3