The Unicode Tools Of REXX*

Josep Maria Blasco

Espacio Psicoanalítico de Barcelona
Balmes, 32, 2º 1ª – 08007 Barcelona
jose.maria.blasco@gmail.com
+34 93 454 89 78

March the 4th, 2024

Abstract

In this article, we present TUTOR, a software package implementing (parts of) the Unicode standard in REXX and OOREXX. TUTOR stands for *The Unicode Tools Of REXX*, and is a *prototype*, *experimental*, *partial*, *procedural-first*, *level-one*, *pure Open Object REXX*, implementation of the Unicode standard; the first part of the article is devoted to providing us with an insight into the most basic design decisions behind the software package.

After a short review of what can be done, today, with REXX and Unicode, a detailed discussion of the additions to Classic REXX that are needed for Unicode follows. The next section, much shorter, does the same for (Open) Object REXX. The following two chapters discuss necessary modifications to the existing built-in functions, and the new functions defined by TUTOR, respectively.

The article concludes, before the acknowledgements, with a review of the main utilities included with TUTOR, including RXU, the REXX Preprocessor for Unicode, and `rxutry`, a derivative of `rexxtry` distributed with the 0.5 release, which has been extended to support TUTOR-defined REXX.

The REXX tokenizer, which is distributed as part of TUTOR, can, however, be used independently of that software package. It is described in an accompanying document.

*URL of this document: `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx.pdf`. Presented to the *35th International Rexx Language Symposium*, held in Brisbane, Australia and online from the 3rd to the 6th of March, 2024.

Contents

1 Introduction

In this article, we will present and describe a set of programs collectively known as *The Unicode Tools Of Rexx (Tutor).* Tutor is a *prototype*, *partial*, *experimental*, *procedural-first*, *level one*, *pure Open Object Rexx* implementation of the Unicode standard for the Rexx language. The exact meaning of the highlighted terms will be made clear in the following paragraphs.

1.1 The Architecture Review Board

Tutor has been written by the author of this article, but its design and features have been greatly influenced by the debates and discussions held in the Architecture Review Board (arb) of the Rexx Language Association (RexxLA); some of the features of Tutor are the direct result of suggestions made by other members of the arb, or expression of a certain consensus reached in the board.

I am very thankful for all the inputs, commentaries, suggestions and general feedback received in the course of these conversations, to which I will make, in what follows, frequent reference.

1.2 A prototype

Tutor is a *prototype*, not a finished product. We strongly discourage using it in production; if you do so, you are doing it at your own risk.

In particular, the package may exhibit *incoherent behaviour*. For example, many of the procedural stream built-in functions (bifs) have been extended to support Unicode, but the stream classes have not. Operating on a Unicode stream using stream bifs and stream classes at the same time may produce unexpected results, result in data corruption, etcetera.

1.3 A partial implementation

Tutor is a *partial* implementation, because it does not implement the totality of the Unicode standard, and also because not all of the existing features of the Rexx language have been revised to add Unicode support.

1.4 An experimental implementation

Tutor is an *experimental* implementation, because its main purpose is to provide a collection of proof-of-concept Rexx implementations of several aspects of the Unicode standard, in such a manner that the author, and hopefully other Rexx users too, can play and *experiment*, to self-educate (“tutor”) themselves, by immersing in the Unicode standard and the intricacies of a possible future Unicode-enabled Rexx implementation.

1.5 History of Tutor

Before producing anything remotely related to Tutor, I spent several months learning Unicode by debating on the ARB list, and directly on GitHub, with Jean Louis Faucher and René Vincent Jansen.[1] I am very grateful for these conversations, from which I extracted ample benefits: Jean Louis’ work with his Executor is extraordinary, and his knowledge of the Unicode standard is far wider than mine. Revisiting these GitHub debates with the benefit of hindsight, one can see that many of the design choices taken by Tutor had already been considered, *in nuce* or explicitly, as possibilities, in the discussions that were maintained.

Tutor itself started on June 11, 2023, when I distributed “A toy ooRexx implementation of the `General_Category` Unicode property”[2] for comment in the ARB list.

Five days later, a package then called *The Unicode Toys For Rexx* was distributed, as a 0.1 release, with many improvements and additions. In particular, we were already isolating extended grapheme clusters, and we implemented three types of string: `BYTES`, composed of bytes; `RUNES`, composed of Unicode code points (`RUNES` has since been renamed to `CODEPOINTS`), and `TEXT`, composed of extended grapheme clusters.[3]

Development continued at a furious pace. On June 19, version 0.1d was distributed.[4] This version incorporated a tool that has since become essential: RXU, the Rexx Preprocessor for Unicode. The preprocessor was based on a version of our Rexx tokenizer that had been modified to include Unicode support. We had started work on the tokenizer, as a separate project, around

[1] The conversation was chaotic, but, at least for me, very instructive. The interested reader might want to browse `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/_draft_notes.md` and follow the links therein.

[2] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/pre-0.1-release-notes.md`.

[3] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.1-release-notes.md`.

[4] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.1d-release-notes.md`.

May 2023. The tokenizer, described in an accompanying document,[5] is since being distributed as part of the TUTOR package, but, if no Unicode support is needed, it can be used separately. This release also featured support for the new `U` strings.[6]

On June 26, version 0.2 was released, featuring Unicode implementations of `LOWER()` and `UPPER()`, an experimental **`Options DefaultString`** instruction to determine the semantics of unsuffixed strings, and another experimental instruction, **`Options`** `Conversions` (which was later renamed **`Options Coercions`**), to determine whether and how REXX should automatically convert between string types.[7]

I tend to be a magmatic programmer: I produce code at high speed, but then the documentation layer resents (it tends to be lacking), and sometimes I end up by implementing the same functionality in several different ways in different parts of the code. On June 30, I announced that a refactoring and documentation effort was undergoing. Documentation quickly improved, while the rhythm of development, unavoidably, dropped. I also decided, following general remarks in the ARB, that the term "Toys" was to be substituted by "Tools" in the package name.

On August 8, Chip Davis observed that the package could well be renamed to *The Unicode Tools Of* REXX (instead of "*...for* REXX"), "giving you a nice acronym". Given that the main purpose of the package is to facilitate playing, experimenting and *learning* about REXX and Unicode, this suggestion looked more than idoneous.

Version 0.3 was released on August 11.[8] It featured Unicode support for the stream BIFS, two new BIFS, `ENCODE()` and `DECODE()`, and new documentation in ooRexxDoc format (we have since migrated to GitHub Markdown). It also defined a general encoder/decoder interface, and added a number of sample programs.

Six days later, version 0.3b was released. Among other things, it contained four new encodings, kindly contributed by Rony G. Flatscher, and "a lot of new documentation".[9]

[5] See the Symposium presentation titled *A Tokenizer for* REXX *and* OOREXX, `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx.pdf`, and the corresponding slides, `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx-slides.pdf`.

[6] *Vid. infra.*, section 3.7, *Unicode strings*, on page 30.

[7] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.2-release-notes.md`.

[8] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.3-release-notes.md`.

[9] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.3b-release-notes.md`.

Version 0.4, released on September 1, contained "*(a)* a big upgrade to the Rexx tokenizer, and *(b)* a complete rewrite of `rxu.rex`, the Rexx Preprocessor for Unicode, to take advantage of the improvements in the tokenizer".[10] The "big upgrade" was the addition of what we now call "full tokenizing".[11]

Version 0.4a,[12] released on October 2, migrated the documentation to GitHub Markdown format, added some implementation notes, and provided a sophisticated `UTF8()` routine, supporting the `UTF-8`, `UTF-8Z`, `WTF-8`, `CESU-8` and `MUTF-8` formats. The routine is distributed as part of the TUTOR package, but it can also be used independently.

Version 0.4b,[13] released on October 12, implemented the NFD normalization form. Non-strict equality was redefined to use NFD-equivalence. A new `UNICODE()` BIF was introduced and documented.

Around October 20, development was put on hold, because the author had to attend to other duties.

January, February and the first days of March 2024 have mainly been used to finalize and stabilize version 0.5, and to write this report and another one about the REXX tokenizer.

Version 0.5 adds support for the NFC normalization form, stipulates that `TEXT` strings are to be automatically normalized to NFC at creation time, and adds a new, intermediate type, called `GRAPHEMES`, composed of extended grapheme clusters that are *not* automatically normalized, and a utility called `rxutry.rex`, which mimics the `rexxtry.rex` utility distributed with REXX, but with all the Unicode extensions defined by TUTOR.

1.6 A pure ooRexx implementation

TUTOR does not depend on any external Unicode library, like ICU, the International Components for Unicode, as it is written in *pure Open Object REXX*. Obviously, such an approach has some drawbacks, but it also exhibits certain advantages.

The main —and obvious— drawback is that every feature, every aspect of the standard, has to be manually coded, written from scratch, which is very arduous. Since we do not rely on external libraries, we have to study

[10] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.4-release-notes.md`.

[11] Please refer to the accompanying documentation for the REXX tokenizer for additional details.

[12] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.4a-release-notes.md`.

[13] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.4b-release-notes.md`.

the standard thoroughly, and devise the means to implement every fragment of Unicode by hand, to the last strenuous detail. Clearly, this is much more laborious than simply adapting an existing framework; moving from one release of Unicode to another becomes also slower, since it may represent a considerable amount of work and all the programs have to be revised; and so on.

At the same time, the very same fact that everything has to be manually coded can also be pondered as an advantage. Coding everything is, no doubt, an exacting task; but, at the same time, it offers an excellent opportunity to learn, and to understand, in great depth and detail, the more subtle nuances of the standard. It is our hope that this effort may end by contributing, all things considered, to the creation of implementations of Unicode-enabled Rexx of much better quality.

Another, patent, advantage of this code-everything-by-yourself strategy is that a manually written implementation can be of great help to show the way for certain, severely memory-constrained, language implementations (for example, those operating under VM/370).

1.7 A level one implementation

The (admittedly somewhat arbitrary) denomination "*level one*" appeared in the ARB discussions, and has been imported here. The main idea behind that name is the following: the implementation of Unicode-enabled Rexx should not be done in a single step, but in a series of steps; every step should implement a *level* of Unicode-enabled Rexx; each level should be contained in the following level, so that, for example, all the features present in level one will appear in all the remaining levels, but some features will be exclusive to level two and to all the levels with numbers greater than two (if any), and so on.

There seems to be general consensus that all kind of *manipulations* of Unicode strings should pertain to level one, while more advanced features, like the *use* of Unicode as part of variable or constant symbol names, or non-ANSI numerals in numbers, should be left to levels higher than one.[14] Sticking to level one would also allow the implementation of Unicode-enabled Rexx with minimal modifications to the currently existing parsers.

Tutor adheres to this nomenclature, and is a level one implementation of Unicode-enabled Rexx.

[14] Besides, deciding how to implement Unicode identifiers is not a trivial task: see the discussion about the Julia language in `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/0650_Identifiers_security.md`.

1.8 Other implementations

Tutor is a particular implementation of Unicode for Rexx, and as such it is based on a set of design decisions which are outlined and justified, in the manner of a *rationale*, in this article. Other implementations, like Jean Louis Faucher's Executor, a ooRexx derivative that contains a trove of extensions to ooRexx, including a (also partial) Unicode implementation,[15] or Adrian Sutherland's CRexx project,[16] an experimental, Rexx-based low-level language, which is designed and built to use Unicode from the start, are both based in sets of design decisions which are different from the ones taken here; it is well-known that a single problem can usually be tackled in many different ways. In the rest of this article, I will sometimes make reference to these two implementations, in order to highlight the consequences of the different choices that each implementation has made.

1.9 The procedural-first approach

Tutor chooses a *procedural-first* approach to the implementation of Unicode-enabled Rexx. This means that its priority goal is to produce a (subset of) Unicode-enabled Classic Rexx — not Unicode-enabled ooRexx.

> Since Tutor is a pure ooRexx implementation, the *development* of this procedural Unicode-enabled version of Rexx will, unavoidably, require the creation of a series of object-oriented tools, like, for examples, new classes, or a rewrite of the current built-in methods. These new tools will serve as the basis for a possible Unicode-enabled version of ooRexx, and all care will be taken so that the object-oriented part of our development is coherent and harmonious with the Classic Rexx part. These extensions of ooRexx, however, are not an objective by themselves, but only a necessary consequence of the main goal of Tutor, that is, extending Classic Rexx so that it supports Unicode.

The implementation of Unicode versions of the built-in functions (bifs), to take an example, will have priority over the development of new versions of the built-in methods (bims): for instance, the current version implements a series of enhancements to many of the stream bifs, like `CharIn()`, `Chars()`, `LineIn()`, `Lines()` or `Stream()`, but no modification to the built-in stream classes.

The rationale behind this approach is the following: if we do not first think about the procedural dialects of Rexx, like Regina or BRexx, and concentrate instead only on producing an Unicode-enabled ooRexx derivative, it

[15] See `https://github.com/jlfaucher`.

[16] See `https://github.com/adesutherland/CREXX`.

is unclear how easy it will later be, or whether it will even be possible, to comfortably backport our new developments to Classic Rexx. Historically, Rexx was first procedural only, and only later it acquired object-oriented extensions; our impression is that, to properly support Unicode "in the Rexx way", we will have to repeat, in this sense, the history of Rexx.

Let us consider, as an example, Jean Louis Faucher's excellent Executor experiment. It follows a purely object-oriented approach, since it extends ooRexx by defining a new, Unicode-oriented class, called `.RexxText`, while maintaining the `.String` class as an ordered, byte-oriented, collection of characters. Executor does not tell us (it does not even attempt to, since, as far as I understand them, the goals of Executor are very different from ours) how a Classic Rexx interpreter like Regina could be modified to support Unicode. Adrian Sutherland's CRexx, on the other hand, is procedural, not object-oriented, but it is maybe too low-level, and too independent of Classic Rexx, to give us many ideas about how to modify Regina for Unicode; besides, being designed from scratch with Unicode support, it does not have to face the problems we are confronted with, since it avoids them by its very same design.

1.10 A single, universal, string interface

As we will see, we will need to define several new types of strings. All these string types will present a uniform and similar interface, as they will be usable in exactly the same way: all the Classic Rexx built-in functions (bifs) will work, completely unaltered (when this makes sense!), with all the new string types. Thus, the experienced Rexx programmer will not have to learn a new set of bifs specific to every new string type; to the contrary, she will be able to immediately leverage her experience with Classic Rexx bifs to create new, Unicode-enabled programs.

The main reason behind this design decision is to make the life of programmers more comfortable. But we will also see later that, at the same time, the task of building an experimental implementation of the whole hierarchy of string types is greatly simplified by following this unified approach.[17]

1.11 Experimenting with concepts

One of the main goals of Tutor is to create a toolbox that is as useful as possible *to experiment with language concepts* — in addition to experimenting with language *features*.

[17] Most probably, a production implementation should be fine-tuned for efficiency, and then some of these simplifications would lose their effectiveness.

Let us try to clarify, with a very simple example, the meaning we are trying to convey. As we will see below in detail,[18] TUTOR defines four types of strings: strings made of bytes (equivalent to the current, Classic REXX, strings); strings made of Unicode code points; and two types of strings made of extended grapheme clusters, one of them automatically normalized to the NFC Unicode normalization form. We will refer to these four string types by using the names `BYTES`, `CODEPOINTS`, `GRAPHEMES` and `TEXT`, respectively. Similarly, we will define four conversion functions, with the same denominations, to transform strings of one type into another.

There has been some discussion about whether defining such new names and functions is a good design decision or not. A number of people (most notably, Rony Flatscher) argued that it might be better to create a single new function, let us call it `STRING` for the sake of the argument, and then implement the conversion functions as options of that function. For example, to convert a variable `var` to the `BYTES` type, one would write

```
STRING(var, "BYTES")
```

instead of

```
BYTES(var).
```

This would have the advantage of helping to keep the language small, and additionally it would minimize the pollution of the function name space — two valuable and desirable objectives.

Our way of looking at this question, though, is a different one: we think that TUTOR should, *at the present moment*, include all these names, but, at the same time, we are also currently agnostic about whether a final implementation of Unicode-enabled REXX should include these names or not.

To justify our standpoint, we will argue as follows. One of the goals of TUTOR is to serve as a playground for new concepts,[19] so that these concepts are *socialized* in the diverse REXX groups, and also to provide a *common vocabulary*, so that discussions about these concepts can be held, and, eventually, pertinent decisions can be taken regarding the entities designated by these same concepts.

If one looks at TUTOR from this point of view, `BYTES(var)` is much clearer than `STRING(var, "BYTES")`. The relevant concept, `BYTES`, appears in a prominent way, almost alone; functional notation intuitively indicates

[18] See the section titled *What is a character, anyway?*, on page 24.

[19] *Playground*: that is one of the reasons why the first releases of TUTOR were defined as a set of *toys*.

that `var` is to be of type `BYTES`, or will be converted to `BYTES`, or something similar. The construction `STRING(var, "BYTES")`, on the other hand, has `STRING`, not `BYTES`, in the most visible first position, so that the fact that `var` is, or will be, transformed into a `BYTES` string is somewhat obscured. `STRING(var, "BYTES")` effectively *buries and hides the main concept* —`BYTES`— inside a string parameter, which is in turn an option, and this makes the formulation of certain statements, certain assertions about the concept, more complicated, when not directly impossible: it effectively ends up by *reducing our expressive power.*

Once the conceptual debate is over, however; once there has been an agreement on the final, desirable, features of Unicode-enabled Rexx, it can and it will have to be decided which of the new names have to be kept, and which ones should better be displaced — to be an option of a more general BIF, for example. But while the conversation is ongoing, it seems more practical that the concepts at stake can be named using simple, straightforward, first-class, labels and denominations. The proliferation of new names, functions, methods, etcetera, should, therefore, be construed as a methodic means to facilitate conceptual debate —as a temporary epistemic and sociological (group) aid, if you want—, and not as a set of formalized, concrete proposals to extend the language.

1.12 Structure of this article

Section 1, *Introduction*, on page 5, presents the main design decisions that are behind Tutor, details its origins, evolution and history, and gives justifications for some of its philosophical stances.

Section 2, *Using Unicode with Rexx, today*, on page 15, describes what can be done, today, with Unicode, when using any of the two most used interpreters, under Windows and Linux. As we will see, Unicode can be used, with some precautions and up to a certain point, but lack of explicit support may provide incorrect or unexpected results for many of the most usual operations.

Section 3, *Unicode for Classic Rexx*, on page 20, studies the most basic aspects of a possible extension of Classic Rexx to support Unicode. For compatibility reasons, supporting at least two types of strings appears as an absolute necessity, and the creation of new metaphors to accommodate the presence of different string types in what has traditionally been presented as an essentially typeless language (“everything is a string”) is explored. Tutor ends up by introducing not two, but *four* string types; this is amply justified. We also introduce a new, low-level, kind of string, the *Unicode string*, comparable to hexadecimal and binary strings. An experimental mechanism is defined to select the default string type (that is, the type of an unsuffixed string), and another mechanism to determine whether and how automatic type conversions are performed is presented.

Section 4, *Unicode for (Open) Object Rexx*, on page 33, presents the modifications to the object-oriented part of REXX defined by TUTOR. In accordance with the procedural-first approach taken by TUTOR, this section is relatively small, compared to the previous one. It limits itself to introducing the four string classes, namely `Bytes`, `Codepoint`, `Graphemes` and `Text`, and to show some examples of their use.

Section 5, *Modifications to existing built-in functions*, on page 35, is devoted to studying the modifications to existing (Classic) REXX built-in functions that are necessary to implement Unicode. We divide our work, in turn, between the string manipulation functions, the stream functions, and some low-level functions.

Section 6, *New built-in functions*, on page 45, examines the new set of built-in functions defined and implemented by TUTOR.

Section 7, *Utilities*, on page 51, presents a small set of utilities that are essential to the use of TUTOR: `setenv`, to prepare the environment to run TUTOR; `rxu`, the REXX preprocessor for Unicode, which allows to run TUTOR-extended programs as if they were standard REXX programs; the REXX tokenizer, which is described in detail in an accompanying document, an can be used independently of TUTOR; and `rxutry.rex`, a novelty that comes with release 0.5 and has a role comparable to the classical `rexxtry` program, while allowing all the language extensions defined by TUTOR.

Section 8, *Further work*, on page 56, details the different avenues that can be followed to extend and complete our work.

Section 9 on page 57 contain the *Acknowledgements*.

The appendices, starting on page 58, provide some more or less tedious listings which the reader may, nevertheless, find necessary for a thorough comprehension of the article as a whole. A list of links to PDFs for the articles, presentation slides, and other resources is also provided.

2 Using Unicode with Rexx, today

Neither ooRexx nor Regina implement Unicode;[20] however, we can still *use and partially manipulate* Unicode strings, getting in effect a very limited form of Unicode *usage*. In this section, we will review many of the possible uses of Unicode in actual Rexx programs.

> We will restrict ourselves to two implementations of Rexx, namely ooRexx and Regina Rexx ("Regina"), and to two operating systems, Windows and Linux.

2.1 Character encoding

Both ooRexx and Regina use a subset of ASCII as the alphabet of Rexx. To be able to write Unicode literals in our source programs, we will need an editor that supports the UTF-8 encoding, which is a strict superset of ASCII— and therefore of the alphabet of Rexx.

2.2 Unicode literal strings

Once we are using the UTF-8 encoding, we will be able to effortlessly create Unicode literal strings:

```
croissant = "🥐"
```

Additionally, if, for whatever reason, we need to use a Unicode code point not directly supported by our editor, but we know its UTF-8 encoding instead, we can resort to that encoding to use an hexadecimal string for that particular code point:

```
Say croissant == "F0 9F A5 90"X        /* 1                 */
```

> Indeed, in Tutor we will also be able to use a new, specialized string type, the *Unicode string*, similar to hexadecimal and binary strings, to denote Unicode characters by code point, name, label or alias. This is a much more convenient way to denote strings with Unicode content, since it is independent of the source file encoding, while hexadecimal notation is not.[21] See the section titled *Unicode strings* on page 30 for more details.
>
> ```
> Say croissant == "1F950"U          /* 1                 */
> Say croissant == "(Croissant)"U    /* 1                 */
> ```

[20] Beyond some anecdotal RexxUtil functions, `SysFromUnicode` and `SysToUnicode`, which, additionally, under ooRexx, are Windows-only (and, therefore, non portable).

[21] For example, if the source program file was encoded using UTF-16, the hexadecimal representation of the croissant emoji would be `"D83E DD50"X` instead of `"F0 9F A5 90"X`.

2.3 Operating with Unicode strings

UTF-8 encoded strings are normal string values, which can be manipulated using the usual REXX BIFS and operators:

```
croissants  = Copies(croissant,2)
coffee      = "☕"
breakfast   = coffee || croissants
```

Some of these operations will get the desired results,

```
Say Copies("🦩", 2)          /* 🦩🦩                          */
```

while others will not:

```
Say Length("🦆🦆🦆")         /* 12 (instead of 3)            */
```

2.4 Unicode labels, and external programs

We can use Unicode strings as *labels*, for internal calls, as targets of **Signal** instructions, or for any other purpose, like tracing, with absolute normality:

```
   /* Internal calls to string labels need parentheses    */
   Call ("🐾🐾🐾🐾")           /* Follow the trail           */
   ...
"🐾🐾🐾🐾":
   ...
"🎨":                           /* Do something               */
   ...
   Signal "🎨"
```

You can also use Unicode strings as class or method names, etc.

Additionally, since file names can also be Unicode strings, we can call *external programs* written in REXX, and other *commands*, with Unicode file names:

```
/* Call "🦜🦜🦜🦜🦜.rex"                                   */
Call "🦜🦜🦜🦜🦜"              /* Fire the AI assistant     */

/* Invoke a command called "🐄.exe" (Windows)             */
Address Command "🐄"
```

2.5 String identity in Rexx, and its effects on labels

In Rexx, character, hexadecimal and binary strings are all different notations *for the same type of strings*. The ANSI standard,[22] for example, clearly states that "*String* supplies the source recognized as String to the top syntax level as a STRING token" (6.2.1.2), and then adds "*Binary_string* supplies the converted binary string to the top syntax level as a STRING token" (*ibid*), and, similarly, "*Hex_string* supplies the converted hexadecimal string to the top syntax level as a STRING token" (*ibid*).

This means that, according to the standard, and assuming an UTF-8 encoding,[23] the "top syntax level" sees *the same string* regardless of whether the source program contains "a", "61"X or "0110 0001"B.

```
Say "a" == "61"X          /* 1                          */
Say "a" == "0110 0001"B   /* 1                          */
```

Labels, as it is well known, can be symbols (which includes variable symbols, constant symbols — and numbers, with or without exponents[24])... or strings. When a label is a string, the above criteria for string equivalence also applies. Assume, for example, that you have a label "a":

```
"a": ...                  /* Do something               */
```

You can then use that label (with a function call, a Call or Signal instruction, etc.) by referring to it as "61"X, or as "0110 0001"B:

```
Call ("61"X)              /* Identical to: Call ("a") */
Signal "0110 0001"B       /* Identical to: Signal "a" */
```

> Regina does not allow the Call (<expression>) instruction format, and therefore all string calls are automatically calls to an external function. On the other hand, ooRexx does not bypass internal labels when the above instruction format is used (Rick McGuire brought this subtle detail to my attention).

What is valid for the label "a" is also valid for all other Unicode characters. Fix again a UTF-8 encoding, and assume that we have a label "🎨":

```
"🎨":                     /* Do something               */
```

[22] See https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/historic/j18pub.pdf.

[23] Indeed, ANSI is sufficient for the following example.

[24] This is not so well known. Open Objet Rexx honours this rare variation, but not Regina.

Then, since the UTF-8 encoding of `"🎨"` is `"D8 3C DF A8"X`, we will be able to `Call` or `Signal` this label using

```
Call ("D8 3C DF A8"X)      /* Identical to: Call ("🎨")*/
/* The following is identical to: Signal "🎨":        */
Signal "1101 1000 0011 1100 1101 1111 1010 1000"B
```

Some readers may find this behaviour of the interpreters strange, or surprising. Let us stress that these are not proposals made by TUTOR, but mere descriptions of the behaviour of some of today's REXX implementations. REGINA and OOREXX actually work as described in this section, including with Unicode —e.g., emoji— strings.[25]

2.6 Stream I/O

Since UTF-8 is a strict superset of ASCII, UTF-8 encoded Unicode strings can be read and written without problems.

```
Call LineOut dinner, "🦞🦑🦪"
```

We can even use UTF-8 strings as *file names*: the following code fragment

```
Call LineOut "🦢🦢🦢🦢🦢", "Hume's swans"
```

will write a line containing `"Hume's swans"` to a file called `"🦢🦢🦢🦢🦢"`.

> Be careful with file names containing emojis: Dropbox will not synchronize them, since it only allows Unicode characters from the Basic Multilingual Plane in file names.

2.7 Console I/O

The Linux terminal will happily work with Unicode, if the right locale is selected; it is also possible, if a little more tricky, to activate Unicode by default in the Windows terminal.

> This activation is somewhat convoluted. Under Windows 11 23H2, for example, one has to follow the path `Settings` → `Time and language` → `Language and region` → `Administrative language settings` → `Change system locale`, making sure that the `Beta: Use Unicode UTF-8 for worldwide language support` checkbox is selected, and then reboot. Additionally, a TrueType font that supports the Unicode subset being used should be selected for the Terminal application.[26]

[25] With the caveat about calculated `Call` instructions under REGINA mentioned above.

[26] Lucida Console works for us. See `https://stackoverflow.com/questions/57131654/using-utf-8-encoding-chcp-65001-in-command-prompt-windows-powershell-window` for additional details.

2.8 Other environments

Rexx cgi scripts written using the utf-8 encoding can easily create html5 web pages (html5 uses the utf-8 character encoding by default), or be used to process the contents of html5 forms.[27]

2.9 In summary

To summarize: while we can certainly *use* Unicode in programs written in the Rexx language today, given that the current implementations do not offer explicit support for Unicode, our experience will per force be somewhat limited (when not *very* limited). Since many of the built-in functions will not work as intended, programmers will have to implement by themselves many of the most basic operations (for example, upper- and lowercasing, or accent removal), in case they need them. This level of (implicit) Unicode support, though, however lacking it is, should be, at the same time, the conceptual core around which Rexx support for Unicode is built. What works today has to continue to work tomorrow — for compatibility. And this signals a certain path that implementations of Unicode for Rexx will have to follow, if they do not want to break compatibility with existing programs.

[27] See the web of my institution, `https://www.epbcn.com/`, for a working example: it is a quite complex web, entirely built using Rexx cgi programs.

3 Unicode for Classic Rexx

In this section, we will describe the main design decisions and features that affect the Classic Rexx aspect of our implementation.

We will center on the most nuclear features of the language: the compatibility conflict, from which stems the need to define at least two types of strings (see section 3.1, *The compatibility conflict*, on page 20), the possible difficulties of implementing different string types in a language that is supposed to be typeless (see section 3.2, *Implementing types in a "typeless" language*, on page 22), the "glasses" or "view" metaphor (see section 3.3, *Changing glasses: The view metaphor*, on page 22), the new string types introduced by Tutor (see section 3.4, *What is a character, anyway?*, on page 24), how to specify the default semantics of an unsuffixed string (see section 3.5, *Defining the default string type*, on page 28), the different possibilities for automatic type coercions (see section 3.6, *Coercions*, on page 29), and the definition of the specialized, low-level, *Unicode strings* (see section 3.7, *Unicode strings*, on page 30).

More specialized topics, like stream I/O, will be studied later, in their own sections, below.

3.1 The compatibility conflict

3.1.1 The need for two types of strings

Unicode support may be implemented in several, very different, ways.

A possible approach is to leave the language practically untouched, add some mechanism by which the fact that a string is a Unicode string can be indicated, and then provide a set of BIFs that work with these Unicode strings. The main drawback with such an approach is that Unicode support would be *second class*. If the language has to be left untouched, a literal string has to be, by default, a Classic Rexx string, i.e., not a Unicode string, and then Unicode strings will have to be denoted by a special mechanism, be it, for example, by a new string suffix,

```
var = "string"U  /* Assume that "U" strings are Unicode */
```

or by a call to a special BIF:

```
var = Unicode("string")
```

Both possibilities imply that

```
var = "string"
```

denotes a non-Unicode, Classic Rexx, string, and this is what makes Unicode second-class.

To avoid Unicode being second class, we could stipulate that

```
var = "string"
```

should refer to a Unicode string, but then we would break existing programs, creating a huge compatibility problem. And we would still a way to indicate that a string is a “Classic” string, i.e., one composed of bytes, maybe by using a new string prefix:

```
var = "string"Y          /* "Y" strings are bYte strings */
```

In both cases, we will need to extend Rexx to accept *two kinds of strings*, one composed of bytes and another one composed of Unicode characters (the exact meaning of “Unicode character” will be discussed below).

> Rick McGuire suggested the names `.Bytes` and `.Text` for two hypothetical ooRexx classes implementing these two kinds of strings, a temporary denomination that has since stuck and we have adopted in Tutor.

3.1.2 Selectable default string types

It would appear that we will be forced to either have Unicode strings as second-class strings, or to create a serious compatibility problem. A way out of the conflict is to invent some mechanism to indicate, for every program, what should be the semantics of a default, unsuffixed string. When we indicate that an unsuffixed string has to be a “classic” string, i.e., a string composed of bytes, we are writing a *compatibility* program, and Unicode strings will have to be specified using a special notation (like a string suffix); when we indicate that an unsuffixed string has to be a Unicode string, we are writing a *Unicode-enabled* program, and byte strings will have to be specified using a special notation. The actual method used to indicate the semantics of unsuffixed strings is irrelevant at this point of the discussion: it could be an `Options` instruction, a directive, or some other mechanism.

> The current release of Tutor implements an experimental form of the `Options` instruction to address this need:
>
> ```
> Options DefaultString Bytes
> ```
>
> indicates that default, unsuffixed strings are classic strings, composed of bytes, while
>
> ```
> Options DefaultString Text
> ```
>
> indicates that default, unsuffixed strings are Unicode strings, composed of Unicode extended grapheme clusters. We will discuss these and other forms of the `Options DefaultString` instruction in more detail below.

3.2 Implementing types in a "typeless" language

In the new universe of Unicode-enabled Rexx, we will have "pure" compatibility programs (e.g., all the old, non-Unicode programs), "pure" Unicode programs (i.e., programs where all the strings are Unicode strings), and, independently of the specified semantics of an unsuffixed string in a particular program, *mixed* programs, in which some of the strings will be byte strings and others will be Unicode strings. This represents a challenge for a language which, in its classic form, presents itself as being somehow typeless ("everything is a string").

There has been some discussion, in the ARB, about this topic. Some would argue that Rexx is, indeed, a *typed* language, only that its typing is *dynamic* (i.e., the type of a variable can change with time); the existence of the `DataType` BIF seems to support this idea. Others would retort that these different types, in case they existed, should indeed refer to different basic ways of storing data, and not to a single type (namely `String`, in ooRexx parlance).[28]

A Unicode enabled version of Rexx should support the coexistence of bytes strings and Unicode strings. Tutor provides a new `StringType` BIF that returns the type of a string

```
Say StringType(var1)        /* "BYTES" (maybe)         */
Say StringType(var2)        /* "TEXT"  (maybe)         */
```

and several new BIFS and string notations that allow to create strings of the different types; these will be described in detail below.

3.3 Changing glasses: The view metaphor

Assume for the moment being that the meaning of the expression "Unicode character" has been determined,[29] and that we have strings consisting of bytes and strings consisting of Unicode characters. Let us assume, further, that we have a BIF, called `TEXT()`, that transforms a (bytes or Unicode) string into a Unicode string, and another one, `BYTES()`, that transforms a (bytes or Unicode) string into a string composed of bytes.

[28] Regardless of the underlying implementation details: some implementations of Rexx actually use, in some cases, internal representations that are not strings. For example, ooRexx has two hidden classes, `Integer` and `NumberString`, that masquerade as `String` objects. I owe the details of this information to Rick McGuire.

[29] It has not, as it can mean either a Unicode code point or an extended grapheme cluster, but we will take care of these differences later. See section, 3.4, *What is a character, anyway?*, below, on page 24.

```
/* Assume that hexadecimal strings are BYTES strings   */
string = "F0 9F 91 A9"X     /* "Woman" emoji (UTF-8)    */
Say StringType(string)      /* "BYTES"                  */
Say Length(string)          /* 4 (4 bytes)              */
string2 = Text(string)      /* Conversion               */
Say StringType(string2)     /* "TEXT"                   */
Say Length(string2)         /* 1 (1 Unicode code point) */
```

What has changed, really, when we execute `string2 = Text(string)`, what is the difference between `string` and `string2`, if any? Indeed, we do not need to believe that the string itself has changed; what has necessarily to change is *the way we look at it*, our *view* of the string, i.e., the semantics of the various BIFS, when applied to that string.

> The mere fact that we can write “the string itself” refers to an outdated belief, which states that values have canonical internal representations. This might well be true of the native data types, like bytes, integers or reals, but it is more than doubtful when applied to objects like strings. Java, for example, stores strings composed exclusively of Latin-1 characters (i.e., of code points not greater than `'FF'X`) as an array of bytes, while other strings are stored as an array of 2-byte UTF-16 chars. Contrary to native data types, which are dependent on the architecture and therefore invariable, there is no single, unique, fixed or canonical representation of a string: the optimization for Latin-1 strings, for example, appeared in a certain release of Java. The *interface* (or, if we want to be more philosophical, the *presentation*) of a string has to remain constant, if one wants to avoid compatibility problems; to the contrary, its *internal structure*, its *representation*, is opaque to the programmer, and it may vary between releases, as it is implementation-dependent.

When the program starts, `string` is a `Bytes` string; this means that *we are looking at it as if it was a sequence of bytes*, and therefore its length has to be 4 bytes. When we promote the string, by using the `TEXT()` BIF, we obtain a new variable, called `string2`. We just have *put on new glasses*, which allows us to change our *view* of the string. The string itself might well contain the same binary data as before, or maybe not; what should now matter to us is that now we are looking at it in a new way: it *appears* as composed of Unicode characters, and, consequently, it has to have a length of 1 (because `"F0 9F 91 A9"X` is the UTF-8 representation of the “Woman” emoji, `"👩"`.

A *view*, therefore, is nothing other than a metaphor for a collection of BIFS (i.e., an *interface*, a *presentation*). Wherever it makes sense, the BIFS will be always the same, irrespective of the view. Conceptually, they will effect the same operation; for example, `Length` will return the number of elements in the array, and `string[1]` will return the first element in the sequence that holds the string, if such an element exists, and the null string otherwise.

Conversion functions change our view of a string. Not all strings can be converted to Unicode, because sometimes a string does not contain valid UTF-8, in which case the operation would not make sense. When one attempts to promote a string that does not contain valid UTF-8, a syntax error is raised.[30] Furthermore, some Unicode types can impose additional transformations to the converted string; for example, a type could automatically normalize all strings to a certain Unicode normalization form, say NFC. In this case, something about the string *may* have been be altered by the conversion, and then the round-trip bytes-Unicode-bytes conversion cycle will not amount to the identity.

3.4 What is a character, anyway?

Rexx is well known for its extensive string manipulation BIFS; when we think of an Unicode-enabled version of Rexx, we are naturally led to think of a new set of BIFS, or, to be more precise, of a set of polymorphic extensions of the existing BIFS, so that we can manipulate Unicode string values as easily as we manipulate classic string values today. When we write

```
breakfast   = "🥐🥐🥐"
nCroissants = Length(breakfast)
```

we would expect that `nCroissants == 3`; similarly, we would expect that `SubStr("🦞🦑🦪",2) == "🦑🦪"` and `Left("🦞🦑🦪",1) == "🦞"`, and so on.

> For some of the BIFS, we will expect a behaviour which will *not* be a direct extension of the Classic Rexx behaviour, but a new one that is able to leverage the extended features of Unicode and makes Rexx more general. For example, one would expect that `Lower()` and `Upper()` would implement the full `toLowerCase` and `toUpperCase` Unicode functions, instead of only uppercasing characters in the ASCII range, as Classic Rexx does.

3.4.1 Code points and extended grapheme clusters

When we write "manipulating Unicode string values", we are obviously assuming that things like "Unicode string values" exist; as a consequence, if Unicode string values exist, we should be in a position to clearly define

[30] In the same way that an error is produced when you attempt to add strings which do not contain numbers.

what is a *component* of such Unicode string values, that is, *what is a Unicode character*. The fact is that Unicode provides *two* such definitions of character. The most basic one is the *code point* (or *codepoint*), an integer value between zero and "10FFFF"X;

> Not all code points represent valid Unicode characters, but this should not concern us now.

the other definition consists on saying that characters are, indeed, *user perceived characters*, called *extended grapheme clusters* ("graphemes", in short), which are themselves sequences of code points.

What definition should REXX use? This is a quite involved question. Most languages opt for the first, most basic, definition: characters are Unicode code points; a set of built-in methods and functions provide then access to the extended grapheme clusters. Some few languages, on the contrary (most notably, Apple's Swift) decide that Unicode characters are extended grapheme clusters, and then they define a set of built-in methods and functions to provide access to code points.

Managing strings composed of graphemes is generally more expensive (i.e., it is may be slower and it may occupy more storage) than managing strings composed of code points. On the other hand, string manipulations at the grapheme level feel more "natural" than string manipulations at the code point level, in the sense that they represent a smaller astonishment factor, they better match user expectations.

As an example, let us consider "CC 81"X, the UTF-8 value that encodes the `Combining acute accent` Unicode code point. When concatenated after a lower case "a", the result shows as an accented letter:

```
acute  = "CC 81"X
aacute = "a" || acute
Say aacute                    /* "á"                        */
```

`Aacute` is a string composed of *two* code points, even if it shows as a single character when printed; on the other hand, `aacute` contains only *one* extended grapheme cluster, a "user perceived character", namely `"á"`.

If we decide that REXX strings are code points, we have to accept that a "single" character, like `"á"`, has an internal substructure (since it is composed of *two* code points); on the other hand, if we decide that REXX strings are graphemes, then we have to accept a new, transmuting semantics for concatenation, which does not occur in Classic REXX, since the concatenation of *two* graphemes, namely `"a"` and `acute`, can have as result a *single* grapheme.

Neither of the two approaches is entirely satisfactory, but the graphemes approach seems easier to explain to new users: we only have to relax the expectations about concatenation. In fact, assuming that a letter concatenated to an accent is an accented letter should appear to us as something natural, if we were not so tainted by the usual semantics of programming languages.

3.4.2 Abstract and encoded characters

Further complexity stems from the fact that Unicode offers *several* ways to express "the same" character.[31] To continue with our example, and considering that `"a" == "61"X`,

```
a      = "61"X
acute  = "CC 81"X
aacute = "61 CC 81"X
Say aacute                       /* "á"                          */
```

but the *single* Unicode code point with a UTF-8 encoding of `"C3 A1"X`, `Latin small letter a with acute`, also prints as `"á"` — in fact, `"C3 A1"X` and `"61 CC 81"X` are *visually indistinguishable*. We say that they represent the same *abstract character*, even if the respective *encodings* are different.

3.4.3 Normalization forms and string equivalence

`"C3 A1"X` and `"61 CC 81"X` are visually indistinguishable, but are they really *the same* character? In Unicode parlance, they are not equal, but *equivalent* — according to a certain Unicode Normalization Form, namely Normalization Form C, NFC. We say that `"C3 A1"X` and `"61 CC 81"X` are *NFC-equivalent*. If we stipulated that strings composed of graphemes are to be automatically normalized to the NFC form, then `"a"` concatenated to `acute` would indeed be *identical* to `"C3 A1"X`.[32]

This will be our definition of a default Unicode string: a string composed of graphemes, i.e., of extended grapheme clusters, automatically normalized to the NFC form.

3.4.4 Defining the four string types

We will say that a classic Rexx string, i.e., a string composed of bytes, is a `BYTES` string, and that a string composed of graphemes with automatic

[31] See, for example, *Figure 2-8. Abstract and Encoded Characters*, on page 29 in *The Unicode Standard. Version 15.0 – Core Specification*, https://www.unicode.org/versions/Unicode15.0.0/UnicodeStandard-15.0.pdf.

[32] Because the NFC normalization of UTF-8 `"61 CC 81"X` is precisely UTF-8 `"C3 A1"X`.

NFC normalization is a TEXT string. TEXT and BYTES will be the basic string types of REXX; TEXT will be used by default in most of the cases, for instance, when we need to uppercase or lowercase strings, when we need to single out certain characters or count them, and so on. BYTES, on the other hand, will be a low-level type; we will use it when we are only interested in the bytes that compose a string. We will also use BYTES when we will be manipulating strings encoded in a one-byte-is-one-char encoding.

In some occasions, we will need to manipulate strings composed of graphemes that are not automatically normalized. We will say that these strings are GRAPHEMES strings.

In some other occasions, we will need to manipulate strings at the code point level. We will say that a string composed of code points is a CODEPOINTS string.

Classic REXX programs will use BYTES strings by default, and new, Unicode-enabled programs will use TEXT strings by default. CODEPOINTS and GRAPHEMES are specialized string types, and, as such, will only be used in certain, special, circumstances.

TUTOR built-in functions and operators are defined to work with the four string types.

> The semantics of most of the string manipulation BIFS can be defined in terms of LENGTH() and a choice of SUBSTR() or []; this is the approach followed by TUTOR.

This helps to keep the language small, allows the transfer of knowledge, techniques and code between the different string types, and avoids forcing the programmer to learn special functions to manipulate unnormalized grapheme strings or code point strings.

3.4.5 String suffixes

A string with the "Y" suffix, "string"Y, is a BYTES string. A string with the "P" suffix, "string"P, is a CODEPOINTS string. A string with the "G" suffix, "string"G, is a GRAPHEMES string. A string with the "T" suffix, "string"T, is a TEXT string. A string with no suffix will be a BYTES, CODEPOINTS, GRAPHEMES or TEXT string, depending on the setting specified by the Options DefaultString instruction.

3.4.6 Conversion functions

TUTOR defines a set of new built-in functions to convert between these four types.

- BYTES(string) works with a string of any type, and converts it to a BYTES string, without altering the contents of the string argument.
- CODEPOINTS(string) takes as its argument a string containing valid UTF-8, and returns a CODEPOINTS string, without altering the contents of the string argument. When string does not contain valid Unicode under the current program file encoding,[33] a syntax error is raised; beyond that requirement, string can be of any type.
- GRAPHEMES(string) takes as its argument a string containing valid UTF-8, and returns a GRAPHEMES string, without altering the contents of the string argument. When string does not contain valid Unicode under the current program file encoding, a syntax error is raised; beyond that requirement, string can be of any type.
- TEXT(string) takes as its argument a string containing valid UTF-8, and returns a TEXT string, normalizing first the string argument to NFC if necessary. When string does not contain valid Unicode under the current program file encoding, a syntax error is raised; beyond that requirement, string can be of any type.

3.4.7 The STRINGTYPE built-in function

STRINGTYPE(string) will return "BYTES", "CODEPOINTS", "GRAPHEMES" or "TEXT", depending on the type of the string argument.

```
name = "Łukasiewicz"T
Say StringType(name)        /* "TEXT"                    */
```

3.5 Defining the default string type

TUTOR defines an experimental Options DefaultString instruction to determine the default string type, i.e., the type of an unsuffixed string:

```
Options DefaultString type
```

where type may be one of BYTES, CODEPOINTS, GRAPHEMES or TEXT, defines the type (and therefore the semantics) of an unsuffixed string, "string".

```
Options DefaultString CODEPOINTS
var = "Gödel"               /* a CODEPOINTS string       */
```

[33] In the current implementation, this amounts to UTF-8.

3.6 Coercions

Should binary operations be allowed, when the operands are of distinct types? And, if the reply to the previous question is affirmative, what should be the result of such an operation?

```
a = "Löb's"T                /* A TEXT string          */
b = "theorem"Y              /* A BYTES string         */
c = a b                     /* Should this be allowed? */
                            /* What is StringType(c)?  */
```

Tutor implements a special form of the `Options` instruction so that the programmer can experiment with all the possibilities. We will order the set of string types as follows: `BYTES < CODEPOINTS < GRAPHEMES < TEXT`, i.e., we will stipulate that `BYTES` is the smallest of all types, `TEXT` is the biggest one, and so on. With this definition in mind, we will define

```
Options Coercions option
```

where `option` can be:

- `None`. When `Options Coercions None` is in effect, binary operations between strings of different type are forbidden. A syntax error is raised if such an operation is attempted.
- `Promote`. When `Options Coercions Promote` is in effect, the result of a binary operation between two strings of different type `a`, `b` has a type of `Max(StringType(a), StringType(b))`, i.e., the operation works as if the string of smaller type was *promoted* to the type of the other string before attempting the operation.
- `Demote`. When `Options Coercions Demote` is in effect, the result of a binary operation between two strings of different type `a`, `b` has a type of `Min(StringType(a), StringType(b))`, i.e., the operation works as if the string of bigger type was *demoted* to the type of the other string before attempting the operation.
- `Left`. When `Options Coercions Left` is in effect, the result of a binary operation between two strings of different type if of the same type as the left operand. This may imply a promotion or a demotion of the right operand.
- `Right`. When `Options Coercions Right` is in effect, the result of a binary operation between two strings of different type if of the same type as the right operand. This may imply a promotion or a demotion of the left operand.

It should be taken into consideration that although demotions always succeed, promotions may fail (when the promoted string does not contain valid Unicode).

```
a = "Löb's"T                  /* A TEXT string           */
b = "theorem"Y                /* A BYTES string          */
Options Coercions None
c = a b                       /* Syntax error            */
Options Coercions Promote
c = a b                       /* "Löb's theorem"T        */
c = a "FF"X                   /* Syntax error            */
Options Coercions Demote
c = a b                       /* "Löb's theorem"Y        */
Options Coercions Left
c = a b                       /* "Löb's theorem"T        */
Options Coercions Right
c = a b                       /* "Löb's theorem"Y        */
```

Since our impression is that the “right” setting for `Options Coercions` is `Promote`, this is the default in TUTOR.

3.7 Unicode strings

There has been some discussion about whether REXX should implement escape sequences in strings, that is, special combinations of characters that are translated to other characters, like `"\r"` for the carriage return character, `"0D"X`, or `"\n"` for the line feed character, `"0A"X`, and so on. Many languages implement these escape sequences, including NetRexx, and it would probably be a good idea that REXX implemented them too. The problem here is, once more, compatibility with existing programs: Classic REXX, as it is well known, does not implement escape sequences; if you want special characters, you have to resort to hexadecimal (or binary) strings.

If we were to implement escape sequences in REXX strings, we would need either (a) having two sets of suffixes —as Python, for example, does— for escaped and unescaped strings, or (b) to introduce an asymmetry between unsuffixed strings in Classic REXX and the rest of strings (i.e., to preserve compatibility with old programs, unsuffixed strings could not contain escape sequences in the compatibility dialect, but these same escape sequences would be allowed in other types of string).

Since all this is quite controversial and there is no clear consensus about this problem, TUTOR has opted for a conservative approach. It does not allow the use of escape sequences, but it defines instead a new type of low-level string, the *Unicode string*, similar to hexadecimal and binary strings. Unicode strings are terminated by a "U" character. They can contain blank-separated Unicode code points (with or without the "U+" prefix that many languages use),

```
"41"U == "A"
/* Leading zeros are ignored                                    */
"0041"U == "A"
/* The "U+" prefix is optional                                  */
"U+0041"U == "A"
"1F3B7"U == "🎷"
"41 1F3B7"U == "A🎷"
```

and Unicode code point names, alias or labels, written between parentheses, as defined by the Unicode standard.

```
"(LATIN CAPITAL LETTER A)"U == "A"
/* Casing and blanks are irrelevant                             */
"(LatinCapitalLetterA)"U == "A"
/* An alias:                                                    */
"(End of line)"U == "0A"X
/* A label:                                                     */
"(<control-000A>)" == "0A"X
"(Saxophone)"U == "🎷"
/* Blank separator not needed here                              */
"(Saxophone)(Guitar)"U  == "🎷🎸"
```

U strings are low-level constructions, equivalent to X and B strings, and therefore they are BYTES strings. You can always promote them, if you so please, by using the CODEPOINTS(), GRAPHEMES() or the TEXT() built-in functions.

Please note that U strings are first-class strings: both "(Crab)"U and "0001F980"U are equivalent to "🦀", and "🦀", in turn, is equivalent, if the program encoding is UTF-8, to "F0 9F A6 80"X.

> Let us emphasize once more the fact that the equivalence between "🦀"and "F0 9F A6 80"X is a current feature of Rexx, defined in the ANSI standard and implemented, for example, by ooRexx and Regina, and not a proposal or a feature introduced by Tutor.

All of them can be used, interchangeably, as labels and as targets of the Call and Signal instructions. The following code, for example, is perfectly legitimate:

```
   /* Parentheses are necessary for internal calls       */
   Call ("F0 9F A6 80"X)       /* Calls "🦀"              */
   Call ("1F980"U)             /* Calls also "🦀"         */
   ...
"🦀":                          /* Do something            */
   ...
   Signal "(Crab)"U            /* Transfers control to "🦀"*/
```

4 Unicode for (Open) Object Rexx

4.1 The four string classes

The four string types are implemented by four string classes.

4.2 The BYTES class

The Bytes class subclasses the built-in String class. It overloads the operator methods to support coercion selection (using the Options Coercions instruction), and it reimplements many text manipulation BIMS in terms of the Length() and [] methods.

> Currently, only the BIMS that correspond to BIFS that have been Unicode-enabled are implemented.

```
  /* The following method works equally well              */
  /* with bytes, code points or graphemes                 */
::Method Reverse
  ret = .MutableBuffer~new(, self~length : .String)
  Do i = self~length To 1 By -1
    ret~append( self[i] )
  End
  Return self~class~new(ret~makeString)
```

Every subclass of Bytes will only need to redefine these two methods to get full access to all the usual BIMS, but now applied to code points or to extended grapheme clusters, or whatever the definition of ‘character” is for the new string type.

The Bytes class also extends the DataType() BIM to support Unicode, and defines some few new BIMS, like C2U() and U2C().

4.3 The CODEPOINTS class

The Codepoints class subclasses Bytes, and redefines the Length() and [] methods so that they operate on Unicode code points. It implements some normalization methods,[34] and redefines non-strict equality to be NFC equivalence.

```
  Options Coercions Promote
  a = "a"P                    /* A CODEPOINTS string     */
```

[34] Currently, NFC and NFD.

```
acute = "(Combining acute accent)"U  /* BYTES           */
aacute = "á"P              /* A CODEPOINTS string      */
Say aacute =  a || acute   /* 1 Equal, but not..       */
Say aacute == a || acute   /* 0 ..strictly equal       */
Say Length(aacute)         /* 1 (one code point)       */
Say Length(C2X(aacute))    /* 4 ("C3A1" [UTF8])        */
```

4.4 The GRAPHEMES class

The Graphemes class subclasses Codepoints, and redefines the Length() and [] methods so that they operate on Unicode extended grapheme clusters.

```
Options Coercions Promote
/* C2X output prettyprinted for readability            */
jose = "Jose"G             /* A GRAPHEMES string       */
/* "301"U is the combining acute accent                */
Say C2X("301"U)            /* CC 81                    */
Say jose"301"U             /* José                     */
Say C2X(jose"301"U)        /* 6A 6F 73 65CC81          */
                           /* j- o- s- e-´---          */
rev = Reverse(jose"301"U)
Say rev                    /* ésoJ                     */
Say C2X(rev)               /* 65CC81 73 6F 6A          */
                           /* e-´--- s- o- j-          */
```

4.5 The TEXT class

The Text class subclasses Graphemes and implements automatic NFC normalization on string creation, including operation results.

```
Options Coercions Promote
a = "a"T                   /* A TEXT string            */
acute = "(Combining acute accent)"U  /* BYTES           */
aacute = "á"T              /* A TEXT string            */
Say aacute =  a || acute   /* 1 Equal, and..           */
Say aacute == a || acute   /* 1 ..strictly equal       */
```

5 Modifications to existing built-in functions

This section is devoted to the study of the modifications to the existing (Classic) Rexx built-in functions that are necessary to implement Unicode.[35]

We divide out work between the string manipulation functions (p. 35), the stream I/O functions (p. 38), and some few low-level functions (p. 43).

5.1 String manipulation functions

5.1.1 Semantics of string manipulation built-in functions

Rexx is well-known for its extensive and powerful set of string manipulation functions. Classic Rexx functions operate on strings composed of bytes. Unicode-enabled string manipulation functions should operate on Classic Rexx strings, i.e., on `BYTES` strings, and also on strings of the new types, that is, `CODEPOINTS`, `GRAPHEMES` and `TEXT`, with the usual semantics, as defined in the following section.

5.1.2 Methods and functions definable in terms of `LENGTH` and `[]`

Many of the usual string manipulation built-in functions can be defined in terms of `LENGTH()` and `[]` (or the alternate pair `LENGTH()` and `SUBSTR()`. The same is true of the corresponding methods of the String class.

Consider this reimplementation of the `REVERSE()` built-in method:

```
::Method Reverse
  ret = .MutableBuffer~new( , self~length:.String )
  Do i = self~length To 1 By -1
    ret~append( self[i] )
  End
  Return self~class~new( ret~makeString )
```

The method first creates a `MutableBuffer` to hold the result, for efficiency reasons; its suggested size is the size, *in bytes*, of the receiving string. We then run a counter `i` from `1` to the length of `self`. But, what is `self~length` (as opposed to `self~length:.String`)? Well, `self~length` is the length of the receiving string *in the terms of the string type definition itself*, that is, the number of *bytes* in `self` when `self` is a `BYTES` string; the number of *code points* in `self` when `self` is a `CODEPOINS` string; and the number of

[35] Please remember that this is a *partial* implementation of Unicode-enabled Rexx, and, therefore, our assertions of universality and necessity have to be understood in a limited sense.

extended grapheme clusters in `self` when `self` is a `GRAPHEMES` or a `TEXT` string. The loop runs over all the elements of `self`, which are picked using the `[]` method.

As we can see in this example, a simple method consisting of six lines of code provides us with a general implementation of the `REVERSE()` method that will work equally well, and without modification, with `BYTES`, `CODEPOINTS`, `GRAPHEMES` and `TEXT` strings — provided that we have written correct implementations of `LENGTH()` and `[]` for each string type.

It is now trivial to define the a polymorphic `REVERSE()` BIF in terms of this enhanced `REVERSE()` method.

5.1.3 Methods and functions definable in terms of the corresponding String method

Some few other string manipulation built-in methods and functions can be defined in terms of the corresponding method of the built-in `String` class. For example, the `COPIES()` method can be redefined as follows:

```
::Method Copies
  Use Strict Arg n
  .Validate~nonNegativeWholeNumber( "n" , n )
  If \self~isA(.Codepoints) Then
    Return Bytes( self~copies:.String(n) )
  Return self~class~new( Copies( self~makeString, n ) )
```

The method checks that the number of copies argument, `n`, is present and is a non negative whole number. If the receiving object is not a `CODEPOINTS` string (which implies that it is not a `GRAPHEMES` or `TEXT` string either, since these classes are subclasses of `CODEPOINTS`), it has to be a `BYTES` string, and then we resort to the built-in method of the `String class`, after which we coerce the result to `BYTES`; in all other cases, we transform the receiving object into a `String`, we create `n` copies of that string, and then we coerce the result into the same string type as the receiving object.

It is now very easy to define a polymorphic `COPIES()` BIF in terms of this enhanced `COPIES()` method.

5.1.4 Examples

Let `var` be defined as follows:

```
var = "(Man)(ZWJ)(Woman)(ZWJ)(Girl)(ZWJ)(Boy)"U
```

The value of `var` is the compound emoji 👨‍👩‍👧‍👦, formed by seven code points, namely:

1. The “Man” emoji, 👨, `"1F468"U`, UTF-8 `"F0 9F 91 A8"X`, 4 bytes.
2. A `Zero Width Joiner`, UTF-8 `"200D"U`, `"E2 80 8D"X`, 3 bytes.
3. The “Woman” emoji, 👩, `"1F469"U`, UTF-8 `"F0 9F 91 A9"X`, 4 bytes.
4. A `Zero Width Joiner`.
5. The “Girl” emoji, 👧, `"1F467"U`, UTF-8 `"F0 9F 91 A7"X`, 4 bytes.
6. A `Zero Width Joiner`.
7. The “Boy” emoji, 👦, `"1F466"U`, UTF-8 `"F0 9F 91 A6"X`, 4 bytes.

Now `var` is a `BYTES` string (all `U` strings are `BYTES strings`), and `Length(var) = 25` (because $25 = 3 \cdot (4 + 3) + 4$).

```
Say Length(var)             /* 25 (3 * (4 + 3) + 4)      */
Say StringType(var)         /* BYTES                     */
Say var[1]                  /* "F0"X -- Non printable    */
```

What happens if we convert `var` to a `CODEPOINTS` string?

```
var = CodePoints(var)       /* Now a CODEPOINTS string   */
Say Length(var)             /* 7 (7 code points)         */
Say StringType(var)         /* CODEPOINTS                */
Say var[1]                  /* 👨 The "Man" emoji         */
```

The same BIFS produce different results, depending on the string type. Let us observe the effect of converting `var` to a `TEXT` string:

```
var = Text(var)             /* Now a TEXT string         */
Say Length(var)             /* 1 (1 grapheme cluster)    */
Say StringType(var)         /* TEXT                      */
```

and `Say var[1]` will print `"👨‍👩‍👧‍👦"`.

5.1.5 Exceptions to these rules

Some few BIFS are not covered by the cases just presented. For example, one would expect that `LOWER()` and `UPPER()` implemented the `toLowercase()` and `toUppercase()` Unicode functions, instead of operating only on the `"a".."z"` and `"A".."Z"` ranges, as is the case with the Classic Rexx BIFS.

5.1.6 List of unicode-enabled built-in string manipulation functions

The following Classic Rexx built-in string manipulation functions have been enhanced to support the Unicode string types in version 0.5 of the Tutor package: `CENTER()` (`CENTRE()`), `CHANGESTR()`, `COPIES()`, `DATATYPE()`, `LEFT()`, `LENGTH()`, `LOWER()`, `POS()`, `REVERSE()`, `RIGHT()`, `SUBSTR()` and `UPPER()`. All of them implement the semantics defined above, except for the following exceptions:

- `LOWER()`. In addition to lowercasing the `"a".."z"` and `"A".."Z"` ranges, as the Classic Rexx function does, the Unicode variants of this BIF implement the full `toLowercase()` Unicode function. In practice, this reduces to the `Simple_Lowercase_Mapping` Unicode property, as defined in UnicodeData.txt, plus two exceptions, listed in SpecialCasing.txt.
- `UPPER()` In addition to uppercasing the `"a".."z"` and `"A".."Z"` ranges, as the Classic Rexx function does, the Unicode variants of this BIF implement the full `toUppercase()` Unicode function. In practice, this reduces to the `Simple_Uppercase_Mapping` Unicode property, as defined in UnicodeData.txt, plus a number of exceptions, exceptions, listed in SpecialCasing.txt.

5.2 Stream functions

Several of the stream built-in functions have been rewritten to implement a basic level of Unicode support.[36]

5.2.1 Backwards compatibility

By default, stream operations are byte-oriented, unless you specifically request otherwise. This allows existing programs to continue running unchanged.

5.2.2 Unicode-enabled streams

A stream is said to be *Unicode-enabled* when an `ENCODING` is specified in the `STREAM OPEN` command:

```
Call Stream fn, "Command", "Open Read ENCODING UTF-8"
```

[36] This section contains a re-elaboration of the *Stream functions for Unicode* markdown document, version 0.5, `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/stream.md`.

When an encoding is specified, STREAM() first checks that an encoding with that name is available in the system. The name is looked for both as an official name, and as an alias. If no encoding of that name can be found in the system, a syntax error is raised. If the encoding can be found, the stream is opened, in the mode defined by the options specified in the OPEN command, and the encoding information gets associated with the stream, until the stream is closed. The official name of the encoding can be retrieved by using the QUERY ENCODING NAME command:

```
Say Stream(fn, "C", "QUERY ENCODING NAME")  /* UTF-8 */
```

Once a stream is opened with the ENCODING option, stream I/O BIFS recognize that the stream is Unicode-enabled, and change their behaviour accordingly:

- For input BIFS, the contents of the stream is automatically decoded and converted to Unicode.
- By default, or when the target type is TEXT or GRAPHEMES,[37] both LINEIN() and CHARIN() return strings of the respective type, composed of extended grapheme clusters; additionally, when the target type is TEXT, input lines and character strings are automatically normalized to the NFC Unicode normalization form. When the target type is CODEPOINTS, both BIFS return CODEPOINT strings, composed of Unicode code points.
- When you call CHARIN() and specify the *length* parameter, the appropriate number of code points (or grapheme clusters) are read and returned.
- Each encoding can specify its own set of end-of-line characters. For example, the IBM-1047 encoding (a variant of EBCDIC) specifies that "15"X, the NL character, is to be used as end-of-line. Both LINEIN() and LINEOUT() honour this requirement, i.e., when reading lines, a line will be ended by "15"X, and when writing lines, they will be ended by "15"X too, instead of the usual LF or CRLF combination
- When using Unicode semantics, some operations can become very expensive to implement. For example, a simple direct-access character substitution in a file is trivial to implement for ASCII streams, but it can become prohibitive when using a variable-length encoding. These operations have been limited in the current release.
- Similarly, when the Unicode-enabled stream has a string target of TEXT (the default) or GRAPHEMES, some operations can become pro-

[37] See section 5.2.4, *Specifying the target type*, below, on page 41.

hibitive too: a TEXT or a GRAPHEMES “character” is, generally speaking, a grapheme cluster, and a grapheme cluster can have an arbitrary length. Direct-access character substitutions become too expensive to implement.

5.2.3 Error handling

When using a Unicode-enabled stream, encoding and decoding errors can occur. By default, ill-formed characters are replaced by the Unicode Replacement Character, "FFFD"U. You can explicitly request this behaviour by specifying the REPLACE option in the ENCODING of your stream:

```
Call Stream fn, "C", "Open Read ENCODING UTF-8 REPLACE"
```

You can also specify SYNTAX as an error handling option,

```
Call Stream fn, "C", "Open Read ENCODING UTF-8 SYNTAX"
```

Finding ill-formed character sequences will raise a syntax error in this case. If the syntax condition is trapped, you will be able to access the undecoded or unencoded offending line or character sequence by using the QUERY ENCODING LASTERROR command:

```
Call Stream fn, "C", "Open Read ENCODING UTF-8 SYNTAX"
...
Signal On Syntax
...
var = LineIn(filename)    /* May raise a Syntax error  */
/* Do something with "var"                              */
...
Syntax:
  offendingLine = Stream(fn, "C", "Q ENCODING LASTERROR")
  /* Do something with "offendingLine"                  */
  ...
```

If the function causing the error was LINEIN() or CHARIN(), the result of the QUERY ENCODING LASTERROR command will be the original, undecoded, line or character sequence, as it appears in the file. If the function causing the error was LINEOUT() or CHAROUT(), the result of the QUERY ENCODING LASTERROR command is the string provided as an argument.

5.2.4 Specifying the target type

By default, Unicode-enabled streams return strings of type TEXT, composed of grapheme clusters automatically normalized to the NFC Unicode normalization form. You may prefer to manage Unicode string that are not automatically normalized; in that case, you should use GRAPHEMES as the target type. In some other occasions, you may prefer to manage CODEPOINTS strings. You can specify the target type in the ENCODING section of your STREAM OPEN command:

```
Call Stream fn, "C", "Open Read ENCODING UTF-8 TEXT"
```

When you specify TEXT (the default), LINEIN() and CHARIN() will return strings are of type TEXT, automatically normalized to NFC. When you specify GRAPHEMES, both BIFS will return GRAPHEMES strings, without any automatic normalization. When you specify CODEPOINTS, returned strings will be of type CODEPOINTS.

Note: *Some operations that are easy to implement for a* CODEPOINTS *target type may become impractical when switching to a* GRAPHEMES *or a* TEXT *type. For example,* UTF-32 *is a fixed-length encoding, so that with a* CODEPOINTS *target type, direct-access character positioning and substitution is trivial to implement. On the other hand, if the target type is* TEXT*, these operations may become prohibitive to implement.*

5.2.5 Options order

You can specify any of TEXT, GRAPHEMES, CODEPOINTS, REPLACE and SYNTAX in any order, but you can not specify contradictory options. For example, TEXT SYNTAX is the same as SYNTAX TEXT (and as Syntax text, since case is ignored), but REPLACE SYNTAX will produce a syntax error.

5.2.6 STREAM QUERY extensions

The QUERY command string of the STREAM() BIF has been extended to support Unicode-enabled streams:

```
Call Stream fn, "Command",,
  "Open Read ENCODING IMB1047 CODEPOINTS SYNTAX"
Say Stream(fn, "Command", "Query Encoding Name")
/* "IBM1047"                                        */
Say Stream(fn, "Command", "Query Encoding Target")
```

```
/* "CODEPOINTS", the name of the target type           */
Say Stream(fn, "Command", "Query Encoding Error")
/* "SYNTAX", the name of the error handling option     */
Say Stream(fn, "Command", "Query Encoding LastError")
/* "", the offending line or character sequence        */
Say Stream(fn, "Command", "Query Encoding")
/* "IBM1047 CODEPOINTS SYNTAX"                         */
```

5.2.7 Manual encoding and decoding

Although the simplicity and ease of use of Unicode-enabled streams is very convenient, in some cases you may want to resort to manual encoding and decoding operations. For maximum control, you can use the new BIFs, `ENCODE()` and `DECODE()`. Please refer to section 6.2, *Encoding and decoding functions*, on page 45, for additional details.

5.2.8 Implementation limits, and some reflections

The usual semantics of the stream BIFs can not be directly translated to the Unicode world without a lot of precautions and limitations. Some of these limitations are due to the fact that the present implementation is a prototype, a proof-of-concept. Some other limitations are of a more serious nature.

- Variable-length encodings. Managing character read/write positions for variable-length encodings, like `UTF-8` and `UTF-16`, can be prohibitive to the point of becoming impractical. The same can be said when the target type is `TEXT` (a ”character”, in this case, is an [extended] grapheme cluster, and, in the limit case, an arbitrarily large cluster could substitute a one-byte, one-letter, ASCII grapheme). Operating systems do not have primitives to insert/delete bytes in the middle of a file, and, although this behaviour can certainly be simulated, it can be so, but at a extremely expensive price. It is highly dubious that such a functionality should be defined in the language, or implemented.
- In an encoding where the `LF ("0A"X)` character can be embedded in a normal character, like `UTF-16` or `UTF-32`, ooRexx line count and line positioning can not be relied upon. Tutor does not go to the lengths of actively simulating line count and positioning, and therefore, it preventively disables such operations.

5.2.9 List of unicode-enabled stream built-in functions

The following Classic Rexx built-in string manipulation functions have been enhanced to support the Unicode string types in version 0.5 of the Tutor package: `CHARIN()`, `CHAROUT()`, `CHARS()`, `LINEIN()`, `LINEOUT()`, `LINES()` and `STREAM()`.

5.3 Low-level functions

The two *low level* built-in functions are `C2X()` and `DATATYPE()`.

5.3.1 C2X

`C2X()` is well defined for `BYTES` strings, but its definition for Unicode strings has generated a lot of debate. The controversy stems from the belief that `C2X(string)` should somehow return "the internal representation" of `string`. As mentioned in section 3.3, *Changing glasses: The view metaphor*, on page 22, it is doubtful that this concept of internal representations is indeed well defined for entities like strings. For instance, when Tutor creates an instance variable of a Unicode string class, it conveniently stores, apart from several other instance variables, a minimum of *two* representations of the string, one in the UTF-8 format, and another one in UTF-32 format. What is the "internal representation" of such a string supposed to be, if not the whole collection of state variables, a collection completely unsuitable for `C2X()`?

Now, if the concept of internal representation is not well defined, it becomes obvious that we can not base the definition of a built-in function on such a concept. A possible solution to this problem would be to establish that `C2X()` is only defined for `BYTES` strings, creating an asymmetry between string types. Another way to avoid getting stuck in what we could call "the internal representation trap", is the one taken by Tutor: we extend `C2X()` to accept a second, optional, argument that indicates *a string encoding*:

```
C2X(string, encoding)
```

Currently, `encoding` defaults to `"UTF-8"`.[38] This argument does not have any effect if `string` is a `BYTES` string, but when `string` is a Unicode string, it determines the encoding of the returned value: `"UTF-8"`, for example, means: convert the string to UTF-8, and then return the hexadecimal representation of this UTF-8 string.

[38] And only this value and `"UTF8"` can be specified, to the same effect. Later releases will allow for a wider range of encodings, including UTF-16 and UTF-32.

The definition of C2X() has been chosen so that it has the following, desirable, property: conversion functions do not alter the C2X() value of a string (except for a possible normalization to NFC in the case of TEXT). This means that, if string is a BYTES string containing well-formed UTF-8 normalized to NFC, the following three strict equalities hold:

```
C2X(string) == C2X(CODEPOINTS(string))
C2X(string) == C2X(GRAPHEMES(string))
C2X(string) == C2X(TEXT(string))
```

Furthermore, if **Options Coercions** Promote is in effect (the default), then the following three strict equalities also hold:

```
string == CODEPOINTS(string)
string == GRAPHEMES(string)
string == TEXT(string)
```

5.3.2 DATATYPE

DATATYPE(string, type) accepts a new type, C, for uniCode.

DATATYPE(string,"C") returns 1 when the contents of the string would be a valid U string if suffixed with a U character.

6 New built-in functions

In addition to providing Unicode-enhanced versions of the existing Classic Rexx built-in functions, Tutor defines a number of new built-in functions.

6.1 Type conversion functions

The *type conversion functions* are `BYTES()`, `CODEPOINTS()`, `GRAPHEMES()` and `TEXT()`. They take an argument of any string type, and convert it to the type denoted by its name. `CODEPOINTS()`, `GRAPHEMES()` and `TEXT()` expect an argument that contains well-formed UTF-8; otherwise, a syntax error is raised. `TEXT()` additionally converts its argument, if necessary, to the NFC Unicode normalization form. Please refer to section 3.4, *What is a character, anyway?*, on page 24, for additional details.

6.2 Encoding and decoding functions

The *encoding and decoding functions* are `DECODE()`, `ENCODE()` and `UTF8()`.

6.2.1 DECODE

`DECODE()` can be used as an *encoding validator* when used in the

```
DECODE(string, encoding)
```

format. For example, after executing

```
validString = DECODE(string, "UTF-16")
```

the value of the `validString` variable will be `1` if `string` contains well-formed `UTF-16`, and `0` otherwise.

To use `DECODE()` as a decoder, you have to specify an additional argument, its *format*:

```
DECODE(string, encoding, format)
```

Format can be `UTF8` (or `UTF-8`), `UTF-32` (or `UTF32`), or a blank-separated list of the above. If you specify both decoding *formats*, an array of two items is returned, in the format (*UTF-8 representation*, *UTF-32 representation*):

```
/* "string" is encoded using IBM-1047. Decode it and   */
/* return its UTF-32 representation.                   */
string = DECODE(string, "IBM-1047", "UTF-32")
```

6.2.2 Decoding and error handling

A fourth argument to the DECODE BIF determines the way in which ill-formed character sequences are handled:

```
decoded = DECODE(string, encoding, "UTF-8", "REPLACE")
```

When the fourth argument is omitted, or is specified as "" or "NULL" (the default), a null string is returned if any ill-formed sequence is found. When the fourth argument is "REPLACE", any ill-formed character is replaced with the Unicode Replacement Character, "FFFD"U. When the fourth argument is "SYNTAX", a syntax error is raised in the event that an ill-formed sequence is found.

6.2.3 ENCODE

ENCODE(string, encoding) first validates that string contains well-formed UTF-8. Once the string is validated, encoding is attempted using the specified encoding.

By default, ENCODE returns the encoded string, or a null string if validation or encoding failed. You can influence the behaviour of the function when an error is encountered by specifying the optional error_handling argument:

```
ENCODE(string, encoding, error_handling)
```

- When error_handling is not specified, is the null string or is NULL (the default), a null string is returned if an error is encountered.
- When error_handling has the value SYNTAX, a syntax error is raised if an error is encountered.

```
/* Encode to IBM-1047, and raise a syntax error if an */
/* error is encountered.                              */
string = ENCODE(string, "IBM-1047", "SYNTAX")
```

6.2.4 UTF-8

UTF8() is a version of DECODE() specialized in variants of the UTF-8 format. UTF8() is packaged and programmed in such a way that it can be used independently of TUTOR.[39]

[39] See the classfile utf8.cls in the TUTOR distribution.

By default, `UTF8(string)` acts as a UTF-8 validator. You can also use the `UTF8(string, format)`, and then `format` can be one of `UTF8` (or `UTF-8`), `UTF8Z` (or `UTF-8Z`), `WTF8` (or `WTF-8`), `CESU8` (or `CESU-8`), and `MUTF8` (or `MUTF-8`).[40]

To use `UTF8()` as a decoder, you have to specify a `target` encoding.

```
UTF8(string, format, target)
```

This argument accepts a single encoding, or a blank-separated set of tokens.

Each token can have one of the following values: `UTF8` (or `UTF-8`), `WTF8` (or `WTF-8`), `UTF32` (or `UTF-32`), `WTF32` (or `WTF-32`).

The `W-` forms of the encodings allow lone surrogates, while the `U-` do not.

Duplicates, when specified, are ignored. If one of the specified encodings is a `W`-encoding, the rest of the encodings should also be `W`-encodings. If `format` allows lone surrogates (i.e., if it is not `UTF-8` or `UTF-8Z`), then all the specified encodings should be `W`-encodings.

When several `target`s have been specified, a stem is returned. The stem will contain a tail for every specified encoding name (uppercased, and without dashes), and the compound variable value will be the decoded string.

An optional fourth argument, `error_handling`, determines the behaviour of the function when a decoding error is encountered.

```
UTF8(string, format, target, error_handling)
```

It is an error to specify `error_handling` without specifying `format` at the same time.

- When `error_handling` is the null string of has the value `NULL`, a null string is returned when a decoding error is encountered.
- When `error_handling` has the value `REPLACE`, any ill-formed character will be replaced by the Unicode Replacement Character, `"FFFD"U`.
- When `error_handling` has the value `SYNTAX`, a syntax condition will be raised when a decoding error is encountered.

6.3 Low-level functions

The *low-level* functions are `C2U()`, `N2P()`, `P2N()` and `STRINGTYPE()`.

[40] Please refer to `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/new-functions.md#utf8` for additional details.

6.3.1 `C2U` (Character to Unicode)

`C2U(string)` returns `string` converted to Unicode code points.

By default, `C2U` returns a list of blank-separated hexadecimal representations of the code points. An optional `format` argument allows to select different formats for the returned string:

```
C2U(string, format)
```

1. When `format` is the null string or `"CODES"` (the default), `C2U(name)` returns a list of blank-separated hexadecimal code points. Code points larger than `"FFFF"U` will have their leading zeros removed, if any. Code points smaller than `"10000"U` will always have four digits (by adding zeros to the left if necessary).
2. When `format` is `"U+"`, a list of hexadecimal code points is returned. Each code point is prefixed with the characters `"U+"`.
3. When `format` is `"NAMES"`, each code point is substituted by its corresponding name or label, between parentheses. For example,

   ```
   C2U("S") == "(LATIN CAPITAL LETTER S)"
   ```

   and

   ```
   C2U("0A"X) = "(<control-000A>)
   ```

4. When `format` is `"UTF-32"`, a UTF-32 representation of `string` is returned.

6.3.2 `N2P` (Name to code Point)

`N2P(name)` returns the hexadecimal Unicode code point corresponding to `name`, or the null string if `name` does not correspond to a Unicode code point.

`N2P()` accepts *names*, as defined in the second column of `UnicodeData.txt` (that is, the Unicode `Name` [`Na`] property), like `"LATIN CAPITAL LETTER F"` or `"BELL"`; *aliases*, as defined in `NameAliases.txt`, like `"LF"` or `"FORM FEED"`, and *labels* identifying code points that have no names, like `"<Control-0001>"` or `"<Private Use-E000>"`.

When specifying a `name`, case is ignored, as are certain characters: spaces, medial dashes (except for the `"HANGUL JUNGSEONG O-E"` code point) and underscores that replace dashes. Hence, `"BELL"`, `"bell"` and `"Bell"` are all equivalent, as are `"LATIN CAPITAL LETTER F"`, `"Latin capital letter F"`, `"latin-capital-letter f"`, `"latin_capital_letter_f"` and `"LAtin-CAPItal LEttERF"`.

Returned code points will be normalized, i.e., they will have a minimum length of four digits, and they will never start with a zero if they have more than four digits.

6.3.3 P2N (codePoint to Name)

P2N(codepoint) returns the name or label corresponding to the hexadecimal Unicode codepoint argument, or the null string if the codepoint has no name or label.

The argument codepoint is first verified for validity. If it is not a valid hexadecimal number or it is out-of-range, a null string is returned. If the codepoint is found to be valid, it is then normalized: if it has less than four digits, zeros are added to the left, until the codepoint has exactly four digits; and if the codepoint has more than four digits, leading zeros are removed, until no more zeros are found or the code point has exactly four characters.

Once the code point has been validated and normalized, it is uppercased, and the Unicode Character Database is then searched for the Name (Na) property.

If the code point has a name, that name is returned. If the code point does not have a name but it has a label, like <control-0010>, then that label is returned. In all other cases, the null string is returned.

Note. Labels are always enclosed between "<" and ">" signs. This allows to quickly distinguish them from names.

6.3.4 STRINGTYPE

STRINGTYPE(string) returns BYTES, CODEPOINTS, GRAPHEMES or TEXT, depending on the string type of string.

You can also use the boolean form of the function, STRINGTYPE(string, type), where type if one of BYTES, CODEPOINTS, GRAPHEMES or TEXT. The function will return 1 is the string type of string is the same as the type indicated by type, and 0 otherwise.

6.4 The UNICODE general function

UNICODE() is the Swiss-army knife of Unicode functions, since it centralizes a big (and growing) collection of Unicode functions and properties. Please refer the documentation for the UNICODE() built-in function for details.[41]

[41] See https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/new-functions.md#utf8.

6.4.1 Functional form

`UNICODE(string, function)` implements a series of Unicode-defined functions.[42] The particular function is selected by specifying its name as the case-insensitive `function` argument.

The following functions are implemented, as of the 0.5 release of TUTOR:

- `isNFC`: returns `1` when `string` is NFC-normalized.
- `isNFD`: returns `1` when `string` is NFD-normalized.
- `toNFC`: returns `string`, after normalizing it to NFC if necessary.
- `toNFD`: returns `string`, after normalizing it to NFD if necessary.
- `toLowercase`: applies the full Unicode `toLowercase` algorithm (no CLDR).
- `toUppercase`: applies the full Unicode `toUppercase` algorithm (no CLDR).

6.4.2 Property form

The `UNICODE(code, "PROPERTY", name)` returns the Unicode property identified by `name` applied to the code point `code`.[43] `Name` can be specified using the Unicode property name or any of their alias, as defined in the `UCD` file `PropertyAliases.txt`. `Code` can be a UTF-32 code point (i.e., a four byte integer), or an hexadecimal code point (with no leading `"U+"`).

Please refer to Appendix C, *Unicode properties implemented by the* `UNICODE` *built-in function*, on page 58, for a comprehensive list of the properties implemented by the `UNICODE(code, "PROPERTY", name)` built-in function, as of the 0.5 release of TUTOR.

[42] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/new-functions.md#unicode-functional-form`.

[43] See `https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/new-functions.md#unicode-property-form`.

7 Utilities

7.1 The setenv utility

The setup utility for Tutor is called `setenv.bat` (or `setenv.sh` for Unix-like systems). You should use `setenv` to set the path environment variable, and before using other Tutor programs, like `rxu`, the Rexx preprocessor for Unicode.

> The only exception to this rule is the `rxutry` utility, which, as a usability aid, is able to temporarily adjust the path by itself even when `setenv` has not been previously run.

Assume that you have unzipped the Tutor distribution in the `C:Unicode` directory and that your operating system is Windows.

```
C:\Unicode>setenv
Adding "C:\Unicode" to the PATH environment variable...

C:\Unicode>
```

If you are running under a Unix-like system, remember to invoke `setenv` using the `. ./setenv.sh` form, or your environment changes will not persist.

```
user@host:/Unicode$ . ./setenv.sh
Setting env
user@host:/Unicode$
```

7.2 The Rexx preprocessor for Unicode (rxu)

Tutor defines a series of extensions to the Rexx language, and another, dependent, set of extensions to ooRexx. To be able to experiment with these extensions, a Rexx Preprocessor for Unicode called `RXU` has been written. The purpose of `RXU` is to offer a Unicode-enhanced Rexx experience that is as seamless and as simple as possible. A Unicode-enabled Rexx program ("a `RXU` program", for short) is a program written in a language based on standard (oo)Rexx and enhanced with a set of Unicode specific *additions* and *modifications*.

$$\text{Rexx} \subsetneq \text{ooRexx} \subsetneq \texttt{RXU}\ (\text{Tutor})$$

As an example of *additions*, `RXU` programs allow for four new types of literal strings.[44] There is also a set of new built-in functions.[45]

[44] See section 3.4, *What is a character, anyway?*, on page 24, for details.

[45] See section 6, *New built-in functions*, on page 45, for details.

Modifications become necessary when the behaviour of already existing mechanisms of REXX has to be altered. In our case, for instance, we will expect that `RXU` programs know how to manage Unicode strings, and thus bring the rich set of features of REXX to the Unicode world. But this will mean that *existing* BIFS will have to operate with *new* entities (i.e., Unicode strings) and, of course, they will most probably have to produce new and different results when processing these new entities.

We are then confronted to the task of enhancing, and in this sense *redefining*, existing BIFS. But to redefine BIFS in Rexx happens to be a quite involved task.

7.2.1 Ways to substitute built-in functions. Necessity of a preprocessor

As it is well known, built-in functions are *second* in the Rexx search order:

> *Functions are searched in the following sequence: internal routines, built-in functions, external functions* (rexxref, 7.2.1, "Search Order").

As a consequence, when one wants to redefine a BIF, the only possible way is to write an *internal* function with the same name:

> *If the call or function invocation uses a literal string, then the search for internal label is bypassed. This bypass mechanism allows you to extend the capabilities of an existing internal function, for example, and call it as a built-in function or external routine under the same name as the existing internal function. To call the target built-in or external routine from inside your internal routine, you must use a literal string for the function name* (ibid).

If, as we stated above, we want to offer an experience that is "as seamless and as simple as possible", the only way to achieve our goal is to implement a *preprocessor*. The alternative would be to define a kind of "epilogue" that contained all the redefined functions, and ask the programmers to copy it at the bottom of their programs: a maintenance nightmare, and nothing that could be called "seamless" or "simple".

7.2.2 Ways to substitute built-in functions, part II

A preprocessor could add such an epilogue to `RXU` programs in an automated way. But, if we counted on the idea of a (sufficiently powerful) preprocessor, we could also opt for a different strategy. Instead of writing an internal routine for each BIF that we wanted to modify or enhance, we could *substitute* the name of each BIF in every BIF call, and call a different function instead. Now, that different function would have a new name, *an external function name*. Clashes with existing BIF names would disappear, and, with them, the need to define internal routines. That's a much neater solution. Indeed, if working with ooRexx, all the external routines can be grouped in some few packages, and the task of the preprocessor will practically be reduced, beyond the substitution of names and the implementation of new string types, to the trivial addition of a `::Requires` directive or a function call to enable the new external functions.

The `RXU` preprocessor for Unicode follows this approach. It substitutes calls to an arbitrary Rexx BIF, say `F`, with calls to `!F`, i.e., an exclamation mark, `"!"`, is prepended to the BIF name. For example, the preprocessor would translate `Length(var)` to `!Length(var)`.

7.2.3 Subtleties of substitution

The *basic idea* of such a substitution is very easy to explain, but, as it often happens with basic ideas, its concrete realization is nothing but trivial. You cannot simply pick every occurence of, say, `"LENGTH"` and blindly substitute it with `"!LENGTH"`: that would unintendedly transform method calls, like `var~length`, into `var~!length`, for example. Clearly, we do not want that.

Ok, you could say; let's reduce ourselves to the case where a BIF name is followed by a left parentheses. But this leaves out `CALL` statements. And there are methods that have arguments, anyway.

The `RXU` Rexx Preprocessor for Unicode handles all these complexities, and many more, except one: if there is an *internal* routine with the same name as a BIF, it substitutes names anyway. It should not, but it's beyond its power, in the current version. This limitation will be addressed in a future release.

7.2.4 The RXU command

The preprocessor is implemented by a command written in Open Object Rexx, `rxu.rex`.

```
C:\Unicode>rxu
rxu: A Rexx Preprocessor for Unicode

Syntax:
  rxu [options] filename [arguments]
```

```
Default extension is ".rxu". A ".rex" file with the same name
will be created, replacing an existing one, if any.

Options (case insensitive):

  -help, -h   : display help for the RXU command
  -keep, -k   : do not delete the generated .rex file
  -nokeep     : delete the generated .rex file (the default)
  -warnbif    : warn when using not-yet-migrated to Unicode BIFs
  -nowarnbif  : do not warn when using not-yet-migrated-to-Unicode
                BIFs (the default)

C:\Unicode>
```

The `rxu` command takes as its argument a filename identifying a `.rxu` file, and attempts to translate it to standard `.rex` code. A `.rxu` file can contain all the extensions defined by Tutor; the translated file is a pure ooRexx program, but it makes ample use of the Tutor Unicode library, `Unicode.cls`, and of other auxiliary files. If the translation phase succeeds, the translated `.rex` file is called, and then deleted (unless the `-keep` option has been specified). The net effect is that `rxu filename` interprets (a translation) of `filename.rxu`, written in the extended dialect of Rexx defined by Tutor: the `rxu` command runs `.rxu` files, as one would expect, and `.rxu` files can be written in Unicode-enabled Rexx, as defined by Tutor.

7.3 The Rexx Tokenizer

To translate `.rxu` programs to standard ooRexx `.rex` programs, the `rxu` preprocessor uses the Rexx tokenizer. The tokenizer breaks the `.rxu` program in its constituent parts (“tokens”), attaching to each part some semantic information. The preprocessor inspects these tokens and then emits the new `.rex` program by transforming the token sequence.

For example, unsuffixed literal strings, `"string"`, are substituted by a parenthesized function call, `(!DS("string"))`. `!DS`, a routine located in `Unicode.cls`, implements the `Options DefaultString` instruction semantics. Some care has to be taken so that not all unsuffixed strings are blindly translated; for example, the string `"Label"` in

```
"Label": /* Do something                             */
```

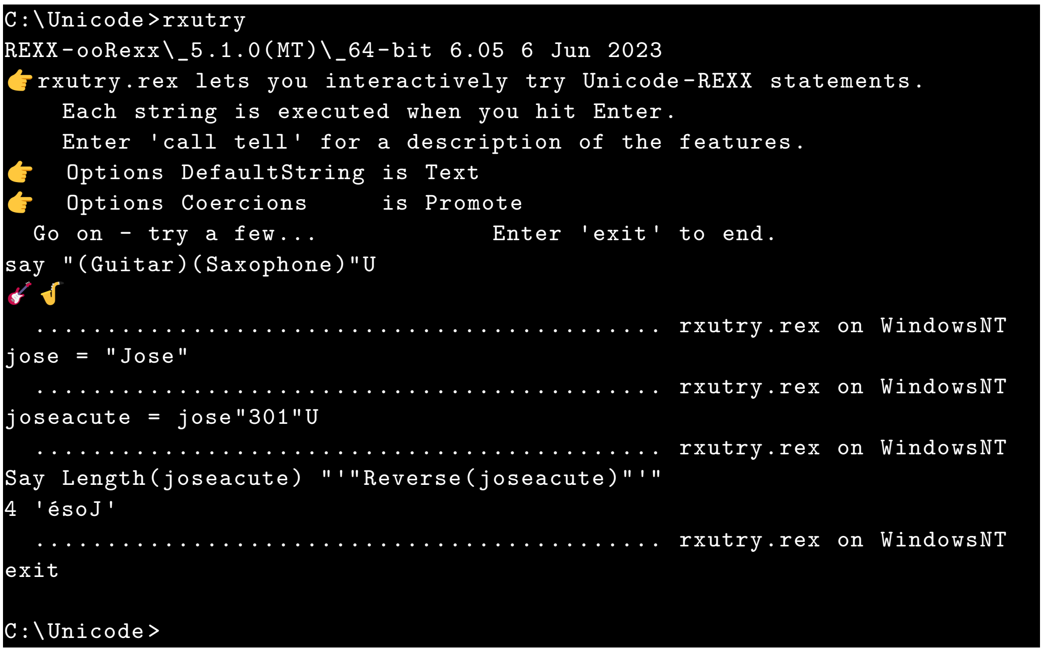
does not have to be translated, since this would generate a syntax error.

The Rexx tokenizer, located in the `parser` directory, is a work in progress towards a full abstract syntax tree parser; it is described in detail in a separate document, *A Tokenizer for Rexx and ooRexx*.[46] The tokenizer can be used independently of Tutor (unless one makes use of Unicode extensions, of course). It defines a main tokenizer class, and a set of specialized subclasses implementing parsers for Regina, ooRexx, ANSI Rexx, and Unicode variants thereof.

7.4 The rxutry.rex utility

The 0.5 release of Tutor includes a new utility called `rxutry.rex`. This program is a derivative of the standard `rexxtry.rex` utility, distributed with ooRexx, and it offers a similar functionality, adapted to Tutor and to `RXU`, the Rexx preprocessor for Unicode.

The `rxutry` utility automatically preprocesses every input line by using `RXU`. `RXU` tokenizes and translates each line to standard ooRexx code, and then this code is executed by using an **`Interpret`** instruction.



```
C:\Unicode>rxutry
REXX-ooRexx\_5.1.0(MT)\_64-bit 6.05 6 Jun 2023
👉rxutry.rex lets you interactively try Unicode-REXX statements.
    Each string is executed when you hit Enter.
    Enter 'call tell' for a description of the features.
👉  Options DefaultString is Text
👉  Options Coercions     is Promote
  Go on - try a few...              Enter 'exit' to end.
say "(Guitar)(Saxophone)"U
🎸🎷
  ........................................... rxutry.rex on WindowsNT
jose = "Jose"
  ........................................... rxutry.rex on WindowsNT
joseacute = jose"301"U
  ........................................... rxutry.rex on WindowsNT
Say Length(joseacute) "'"Reverse(joseacute)"'"
4 'ésoJ'
  ........................................... rxutry.rex on WindowsNT
exit

C:\Unicode>
```

[46] See https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx.pdf.

8 Further work

The most obvious way to extend and enhance our work is to continue implementing parts of the standard, and to reformulate existing features of Rexx, adapting them for Unicode.

As an example of the former we could choose the implementation of the Unicode collation algorithm: this would allow the assignment of Unicode semantics to the non-strict sorting comparison operators for the Unicode types.

Features that should be reformulated include, as a major example, a revision of the `Parse` instruction.

There are, however, some other, maybe not so obvious, avenues to improve and extend Tutor. One of them consists in rewriting parts of the package in pure Classic Rexx; as an example, many of the Unicode properties could easily be rewritten so that they can be run under Regina.

> A sample program, called `testgc.rex` and located in the `classic` subdirectory, is distributed with the 0.5 release of Tutor. It is a proof-of-concept Regina implementation of the (extended) `general_category` property.

This would allow to experiment with Unicode concepts using implementations very different from ooRexx: not only Regina, but probably also CMS or TSO Rexx, and maybe even BRexx for VM/370.

A further step in a similar direction would be to rewrite part of Tutor in languages like C or C++, in effect creating a growing Unicode library for Rexx users to experiment with. This would be a welcome initiative, and it could very quickly produce useful results, but, unfortunately, it is not a path that I can follow alone, since I am not a decent enough C/C++ programmer.

9 Acknowledgements

I want to express my gratitude to all the members of the Architecture Review Board, for their support and encouragement, and their invaluable discussions and suggestions.

To Jean Louis Faucher and René Vincent Jansen, for our conversations in GitHub: these were somewhat chaotic, but, at the same time, very productive. And they allowed me to get up to speed in Unicode matters.

To Jean Louis Faucher (again) for his pioneer Executor extension, a real trove of ideas, and to Adrian Sutherland, for his CRexx effort.

To my colleagues at EPBCN, for bearing with me during my prolonged Rexx raptures.

To the students of my Psychoanalysis and Logic course, where I also happen to teach some Rexx, for their interest and unfaltering persistence.

To Silvina Fernández, Mireia Monforte, David Palau and Olga Palomino, for attending several presentation rehearsals and providing essential feedback.

To Silvina Fernández, for deftly managing our Stream Deck, contributing to make my presentations much more interesting and agile.

Thanks to all, for your help, support and contributions.

Appendices

Appendix A Alphabetical list of Unicode-enabled Classic Rexx built-in functions

The following BIFS have been reimplemented to offer Unicode support, as of the 0.5 release of TUTOR: C2X, CHARIN, CHAROUT, CHARS, CENTER, CENTRE, CHANGESTR, COPIES, DATATYPE, LEFT, LENGTH, LINEIN, LINEOUT, LINES, LOWER, POS, REVERSE, RIGHT, STREAM, SUBSTR and UPPER.

Appendix B Alphabetical list of new Unicode built-in functions

The following new BIFS have been defined by TUTOR in release 0.5: BYTES, CODEPOINTS, C2U, DECODE, GRAPHEMES, N2P, P2N, STRINGTYPE, TEXT, UNICODE and UTF8.

Appendix C Unicode properties implemented by the UNICODE built-in function

Release 0.5 of TUTOR implements the following properties (properties without an associated comment are boolean):

- Alphabetic.
- Alpha (an alias for Alphabetic).
- Canonical_Combining_Class (an integer in [0..254]).
- Canonical_Decomposition_Mapping (one or two normalized hex code points).[47]
- Case_Ignorable.
- Cased.
- CCC (alias of Canonical_Combining_Class).
- Changes_When_Casefolded.
- Changes_When_Casemapped.
- Changes_When_Lowercased.

[47] Non-standard property: this corresponds to the Decomposition_Mapping column (number 6, 1-based, in UnicodeData.txt), when the mapping is not a compatibility mapping (i.e., it does not start with a "<" character).

- Changes_When_Titlecased.
- Changes_When_Uppercased.
- CI (alias of Case_Ignorable).
- Comp_Ex (alias of Full_Composition_Exclusion).
- CWCF (alias of Changes_When_NFKC_Casefolded).
- CWCM (alias of Changes_When_Casemapped).
- CWL (alias of Changes_When_Lowercased).
- CWT (alias of Changes_When_Titlecased).
- CWU (alias of Changes_When_Uppercased).
- Full_Composition_Exclusion (boolean).
- Lowercase (boolean).
- Lower (alias of Lowercase).
- Math (boolean).
- Na (alias of Name).
- Name (the name or label corresponding to the argument).[48]
- NFC_Quick_Check (one of Y, N or M).
- NFC_QC (alias of NFC_Quick_Check).
- NFD_Quick_Check (one of Y or N).
- NFD_QC (alias of NFD_Quick_Check).
- NFKC_Quick_Check (one of Y, N or M).
- NKFC_QC (alias of NFKC_Quick_Check).
- NFKD_Quick_Check (one of Y or N).
- NKFD_QC (alias of NFKD_Quick_Check).
- OAlpha (alias of Other_Alphabetic).
- OLower (alias of Other_Lowercase).
- OUpper (alias of Other_Uppercase).
- Other_Alphabetic (boolean).
- Other_Lowercase (boolean).
- Other_Uppercase (boolean).
- SD (alias of Soft_Dotted).
- Simple_Lowercase_Mapping (the lowercase version of the argument, or the argument itself when the character has no explicit lowercase mapping).[49]
- Simple_Uppercase_Mapping (the uppercase version of the argument, or the argument itself when the character has no explicit uppercase mapping).[50]

[48] This corresponds to the (1-based) column number 2 of UnicodeData.txt. This is a modified property, since it returns labels when there is no name to return. If you want only names, discard returned values that start with a "<" character.

[49] This corresponds to the (1-based) column number 14 of UnicodeData.txt.

[50] This corresponds to the (1-based) column number 13 of UnicodeData.txt.

- slc (alias of Simple_Lowercase_Mapping).
- Soft_Dotted (boolean).
- suc (alias of Simple_Uppercase_Mapping).
- Uppercase (boolean).
- Upper (alias of Uppercase).

Appendix D Resources

- A copy of this article can be downloaded from https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx.pdf.
- The presentation slides can be downloaded from https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx-slides.pdf.
- An accompanying article, titled *A Tokenizer for REXX and OOREXX*, can be found at https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx.pdf; the corresponding slides can be found at https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx-slides.pdf.
- This last article contains, as an appendix, a modified scripting.py file, class RexxLexer fragment, for the LuaLaTeX minted package, so that it supports OOREXX 5.0 and the TUTOR-defined Unicode extensions.