The Unicode Tools Of Rexx

35th International Rexx Language Symposium

Brisbane, Australia, March 3-6 2024

Josep Maria Blasco
jose.maria.blasco@gmail.com

EPBCN – ESPACIO PSICOANALÍTICO DE BARCELONA
C/ BALMES, 32, 2º 1ª — 08007 BARCELONA

March the 4th, 2024

The Unicode Tools Of Rexx

Part I

Introduction

Introduction

TUTOR (*The Unicode Tools Of Rexx*) is a

- partial,
- prototype,
- experimental,
- pure Open Object REXX,
- procedural-first,
- level one

implementation of Unicode for the REXX language.

⇒ The meaning of the highlighted terms will be made clear in the following slides.

The Architecture Review Board (ARB)

The design and features of TUTOR have been greatly influenced by the debates held in the ARB of the Rexx Language Association.

I am very thankful for all the input, commentaries, suggestions and general feedback received in the course of these conversations.

In particular, I want to thank Jean Louis Faucher and René Vincent Jansen: our interchanges in GitHub allowed me to get up to speed in Unicode matters.

A prototype, partial implementation

TUTOR is a prototype, not a finished product. *It should not be used in production environments*. In particular, the package may exhibit *incoherent behaviour*. Example: the stream BIFS have been migrated to support Unicode, but the stream classes have not.

TUTOR is a partial implementation of Unicode,

1. because it does not implement the totality of the Unicode standard, and also
2. because not all of the existing features of the Rexx language have been revised to add Unicode support.

An experimental implementation

TUTOR is an experimental implementation: its purpose is to provide a collection of proof-of-concept REXX implementations of several aspects of the Unicode standard, in such a way that REXX users can get an opportunity to play and experiment, to self-educate ("tutor") themselves, by immersing in the standard and the intricacies of a possible future Unicode-enabled REXX implementation.

⇒ The acronym "Tutor" was suggested by Chip Davis (thanks!) [Previously, the package was called "The Unicode Tools *For* REXX"].

A pure ooRexx implementation

Tutor does not depend on any external Unicode library, since it is written in pure ooRexx. This has drawbacks, but it also exhibits some advantages.

Drawbacks. The main one is that every feature has to be written from scratch, which is very laborious.

Advantages. It offers an excellent opportunity to understand in great depth and detail the more subtle aspects of the standard.

- $\Rightarrow$ Implementations of better quality.
- $\Rightarrow$ A good example for severely memory-constrained language implementations (e.g., VM/370).

A level one implementation

An idea discussed in the ARB: Implementing Unicode in REXX in a series of passes, stages or *levels*.

- Level one: string denotation and manipulation.
- Level two: Unicode variable and constant symbols, non-ANSI numerals in numbers, ...
- ...

TUTOR is a level one implementation. $\Rightarrow$ Level one can be implemented with minimal modifications to the parser.

Other implementations

- Jean Louis Faucher's Executor, a ooRexx derivative that contains a trove of extensions to ooRexx, including a (also partial) Unicode implementation.
- Adrian Sutherland's CRexx project, an experimental, Rexx-based low level language, which is designed and built to use Unicode from the start.

Both implementations are based in sets of design decisions which are different from the ones taken in Tutor.

The procedural-first approach

Procedural-first: concentrate on defining a set of extensions *of Classic* REXX. Object-oriented extensions will come as a consequence of the procedural extensions.

Rationale: Once we define Unicode-enabled REXX as an extension of Classic REXX, it becomes very easy to define new and modified classes to implement Unicode in object-oriented versions of REXX.

$\Rightarrow$ It is not so easy to make the travel in the reverse direction.

A single, universal, string interface

- We will need to define several new types of strings.
- All these string types will be usable in the same way: all the Classic REXX built-in functions (BIFS) will work unaltered (when this makes sense!) with all the new string types.
- $\Rightarrow$ 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 leverage her experience with Classic REXX BIFS to create new, Unicode-enabled programs.
- This also makes programming a prototype much easier.

Experimenting with concepts

We want to create a toolbox that is useful to experiment with *language concepts* (not only *language features*).

New concepts are a way to disseminate new ideas, and an attempt to create a shared vocabulary, to develop a collective imaginary about a possible Unicode-enabled REXX implementation ($\Leftarrow$ *psychological, sociological and epistemological* reasons — not only programming reasons).

$\Rightarrow$ Some concepts will be promoted to a foreground position in a way that may not necessarily be reflected in the final design of the product.

The Unicode Tools Of Rexx

Part II

Rexx and Unicode, today

Introduction. Character encoding

Focus on REGINA and ooREXX, and on Windows and Linux.

What can be done today?

Character encoding:

$$\text{REXX alphabet} \subsetneq \text{ASCII} \subsetneq \texttt{UTF-8}$$

$\Rightarrow$ We need an editor that supports `UTF-8`.

Unicode literal strings

Using UTF-8, we can create Unicode literal strings:

```
croissant = "🥐"
```

We can also use the UTF-8 encoding of a string:

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

⇒ TUTOR defines *Unicode strings*: we can select characters by hexadecimal code point, name, alias or label.

```
Say croissant == "1F950"U       /* 1          */
Say croissant == "(Croissant)"U /* 1          */
```

Operating with Unicode strings

UTF-8 strings are normal strings. They 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 (should be 3) */
```

Unicode labels, and external programs

Labels, class and method names, etc., can be Unicode:

```
Call ("🐾🐾🐾🐾")   /* Follow the trail      */
...
"🐾🐾🐾🐾": ...
```

External programs names and command names can be Unicode too:

```
Call "🦜🦜🦜🦜🦜" /* Fire the AI assistant  */
/* (Calls "🦜🦜🦜🦜🦜.rex")                 */
/* Invoke a command called "🐄.exe" (Windows) */
Address Command "🐄"
```

Stream and console I/O

We can read and write strings containing Unicode:

```
Call LineOut dinner, "🦞🐙🦪"
```

We can even use UTF-8 strings in file names:

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

⇒ Be careful with Dropbox, though: it does not synchronize filenames with characters that are outside the BMP.

Both Linux and Windows offer console configurations with some support for UTF-8 (Windows setup is quite convoluted).

Other environments

REXX CGI scripts written using the UTF-8 encoding can be easily used

- to create HTML5 web pages (HTML5 uses the UTF-8 character encoding by default).
- to process the contents of HTML5 forms.

...

String identity in REXX, and its effects on labels

According to the standard, and assuming a UTF-8 encoding, `"a"`, `"61"X` and `"0110 0001"B` are *the same string*. This means that, if we have a label

```
"🎨": /* Do something */
```

and, since UTF-8 for "🎨" is `"D8 3C DF A8"X`, we can

```
Call ("D8 3C DF A8"X)
```

or even

```
Signal "1101 1000 0011 1100 1101 1111 1010 1000"B
```

⇒ With TUTOR, we can also `Call "(Artist palette)"U`.

The Unicode Tools Of Rexx

Part III

Unicode for Classic Rexx

The compatibility conflict (1/2)

What is the semantics of an unsuffixed string?

- Compatibility: a string composed of bytes.
- In a Unicode world: a Unicode string.
- Different semantics $\Rightarrow$ Conflict!
- A partial way out; use string suffixes to determine *the type of a string*: for example, `"string"Y` would be a bYtes string, and `"string"T` would be a Text (Unicode) string.

The compatibility conflict (2/2)

- Still: what is the type of an unsuffixed string?
- $\Rightarrow$ If Bytes, Unicode strings are second-class.
- $\Rightarrow$ If Unicode, we create compatibility problems.
- The semantics of an unsuffixed string should be *selectable*.

$\Rightarrow$ TUTOR defines a new, experimental form of the `Options` instruction:

```
Options DefaultString type
```

where `type` must be one of `BYTES`, `CODEPOINTS`, `GRAPHEMES` and `TEXT` (these will be defined shortly).

Implementing types in a "typeless" language

- Classic REXX is, in a sense, *typeless*: everything is a string.
- Widen the paradigm: everything is a string, yes; *but* there are several types (or kinds) of string.
- ⇒ New conversion BIFS to *change the type* of a string: `Text(string)`: Unicode; `Bytes(string)`: classic string; ...
- ⇒ New `StringType(string)` BIF to *query the type* of a string: `StringType(Bytes(string)) == "BYTES"`.

Changing glasses: the view metaphor (1/2)

Consider the following code:

```
/* 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)  /* Promotion                  */
Say StringType(string2) /* "TEXT"                     */
Say Length(string2)     /* 1 (1 Unicode codepoint)    */
```

What happens, when we execute `string2 = Text(string)`? In which sense is `string2` different from `string`?

Changing glasses: the view metaphor (2/2)

We do not need to believe that `string` and `string2` are different "inside" (assuming that this makes any sense), only that our view of these strings has changed: we have put on *new glasses*.

These new glasses consist of a whole new semantics for the very same BIFS we are used to. No need to learn new BIFS, no need to learn new paradigms. We have only changed the *level of abstraction* we are using to look at a string.

What is a character, anyway? (1/3)

User expectations when manipulating Unicode strings:

```
Length("🥐🥐🥐") == 3
SubStr("🦞🐙🦪",2) == "🐙🦪"
Left("🦞🐙🦪",1) == "🦞"
"🦞🐙🦪"[2] == "🐙"
Pos("☕","🥐🥐🥐☕") == 4
Upper("Paçà") == "PAÇÀ"
```

Also:

```
  acute = "(Combining acute accent)"U
"Jose"acute == "José"
```

What is a character, anyway? (2/3)

When we say “manipulating Unicode strings”, we assume that things like “Unicode strings” exist.

Do they? And, if they do, what are their components?

There are *two* different answers to this last question:

1. Unicode code points (*grosso modo*, numbers between zero and "10FFFF"X).
2. Extended grapheme clusters, or “user perceived characters”.

What is a character, anyway? (3/3)

Grapheme clusters are *extremely tricky*:

```
glue = "(Zero width joiner)"U /* Glues emojis together */
family = "👴"glue"👶"glue"👴"glue"👧"
Say family                  /* "👴👶👴👧", 1 grapheme cluster! */
```

⇒ We need methods to "disassemble" grapheme clusters into their constituent code points (and even to "disassemble" code points, in turn, into their constituent characters, given a certain encoding):

```
elements = CodePoints(family)
Do i = 1 To Length(elements)
  Say i": '"elements[i]"'"
End
```

Abstract and encoded characters

Unicode offers several ways to express “the same” character.

```
a      = "61"X
/* UTF-8 for "Combining acute accent":          */
acute  =    "CC 81"X
aacute = "61 CC 81"X
Say aacute     /* Prints as "á", same as "C3 A1"X */
```

The single character `"C3 A1"X`, `"Latin small letter a with acute"`, also prints as `"á"`. `"C3 A1"X` and `"61 CC 81"X` are *visually indistinguishable* (but not in the windows terminal! 🙄). They represent the same abstract character, even if *they are not the same character*.

Normalization forms and string equivalence

In Unicode parlance, they are not equal, but equivalent, according to 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 were to be automatically normalized to the NFC form, then `"a"` concatenated to `acute` would indeed be *identical* to `"C3 A1"X`. This will be our definition of a default Unicode string: a string composed of extended grapheme clusters, automatically normalized to NFC.

Defining the four string types (1/2)

- A Classic REXX string is a `BYTES` string, a string composed of bytes.
- A `TEXT` string is a sequence of extended grapheme clusters, automatically normalized to NFC.
- A `GRAPHEMES` string is a sequence of extended grapheme clusters, *not* automatically normalized to NFC (sometimes we need to work with string as-is, with no automatic modifications).
- A `CODEPOINTS` string is a sequence of Unicode code points.

Defining the four string types (2/2)

Classic REXX defines *Binary* strings, with a `"B"` suffix, and *heXadecimal* strings, with a `"X"` suffix. New suffixes for the new types of string are:

- `"String"Y` is a `BYTES` string, composed of bYtes.
- `"String"P` is a `CODEPOINTS` string, composed of code Points.
- `"String"G` is a `GRAPHEMES` string, composed of Grapheme clusters.
- `"String"T` is a `TEXT` string, the default Unicode type.

The type of a unsuffixed string, `"string"`, is determined by the value specified in the `Options DefaultString` instruction (default: `TEXT`).

Conversion functions. STRINGTYPE

- `BYTES(string)` converts `string` to the `BYTES` type.
- `CODEPOINTS(string)` converts `string` to the `CODEPOINTS` type.
- ...

Converting a string to `BYTES` always succeeds. In all other cases, `string` has to contain valid Unicode (i.e., currently, `UTF-8`), or a syntax error will be raised. `TEXT(string)` will additionally normalize `string` to NFC, if necessary.

A new BIF, `STRINGTYPE(string)`, will return the name of the string type of a string. For example,

```
STRINGTYPE("string"T) == "TEXT"
```

Defining the default string type

The semantics of an unsuffixed string is determined by the value specified in the `Options DefaultString` instruction. Its format is

```
Options DefaultString string_type
```

where `string_type` must be one of `BYTES`, `CODEPOINTS`, `GRAPHEMES` or `TEXT`; the default value is `TEXT`.

This allows to experiment with "new" programs, where strings are Unicode-enabled by default and "old" programs, by using

```
Options DefaultString BYTES
```

Specifying one of the other two string types may be useful in specialized circumstances.

Coercions (1/2)

Should binary operations be allowed, when the operands are of different 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? */
/* If yes, what should StringType(c) be?  */
```

Let us define an order on the four string types, as follows:
BYTES $<$ CODEPOINTS $<$ GRAPHEMES $<$ TEXT.

Coercions (2/2)

TUTOR implements a new instruction to determine the type of a binary operation $R = A \circledcirc B$:

```
Options Coercions form
```

Form must be one of:

- PROMOTE ⇒ `StringType(R) = Max(StringType(A), StringType(B)).`
- DEMOTE ⇒ `StringType(R) = Min(StringType(A), StringType(B)).`
- LEFT ⇒ `StringType(R) = StringType(A).`
- RIGHT ⇒ `StringType(R) = StringType(B).`
- NONE ⇒ raise a syntax error if `StringType(A) \== StringType(B).`

Unicode strings

- A new string type, with a new suffix: `"U"`.
- Low level, similar to binary or hexadecimal strings. Always a `BYTES` string (convert if needed).
- Blank-separated sequences of hexadecimal code points (with or without a `"U+"` prefix), and parenthesized code point *names*, *alias* or *labels*.

Examples:

- `"61"U == "0061"U == "U+0061" == "a"`.
- `"(Latin small letter a)"U == "a"` `/* Name */`.
- `"(New line)"U == "0A"X` `/* Alias */`.
- `"(<control-000A>)"U == "0A"X` `/* Label */`.
- `"(Saxophone)(Guitar)"U == "🎷🎸"` `/* Names */`.

The Unicode Tools Of Rexx

Part IV

Unicode for (Open) Object Rexx

The four string classes. The BYTES class

The four string types are implemented by four string classes.

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

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()`.

The CODEPOINTS class

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

```
Options Coercions Promote
a = "a"P                    /* A CODEPOINTS string  */
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 codepoint)    */
Say Length(C2X(aacute))     /* 4 ("C3A1" [UTF8])    */
```

The GRAPHEMES class

Graphemes subclasses Codepoints, and redefines Length() and [] so that they operate on 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-       */
```

The TEXT class

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

```
Options Coercions Promote
a = "a"T                    /* A TEXT string          */
aacute = "á"T               /* A TEXT string          */
Say aacute =  a"301"U       /* 1 Equal, and..         */
Say aacute == a"301"U       /* 1 ..strictly equal     */
```

The Unicode Tools Of Rexx

Part V

Modifications to existing built-in functions

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.

BIMS and BIFS definable in terms of LENGTH and []

Many of the usual string manipulation BIMS can be defined in terms of LENGTH() and [] (or LENGTH() and SUBSTR()). The same is true of the corresponding BIMS.

```
  /* 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)
```

Methods and functions definable in terms of the corresponding String method

Some other string BIFS and BIMS can be defined in terms of the corresponding methods of the String class.

```
::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 ) )
```

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

Examples

Variations over

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

- `Var` is a `BYTES` string, and therefore `var[1]` will be the first byte of the (`UTF8`) representation of "👨", the "Man" emoji, that is, `"F0"X`.
- `Codepoints(var)` is a `CODEPOINTS` string, and therefore `Codepoints(var)[1]` will be "👨", the "Man" emoji itself.
- `Text(var)` is a `TEXT` string, and therefore `Text(var)[1]` will be "👨‍👧‍👦" (in this case, the whole string).

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.

```
string = "En compañía del Barça"
Say Upper(string)  /* EN COMPAÑÍA DEL BARÇA */
```

Unicode-enabled streams

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

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

This encoding can be queried:

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

Error handling:

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

$\Rightarrow$ Ill-formed characters are substituted by the Unicode replacement character, "FFFD"U. You can also specify

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

in which case an ill-formed sequence will raise a syntax error.

Error handling

If you have used the SYNTAX option in the OPEN command and the syntax condition is trapped, you will be able to access the offending line or character sequence:

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

Specifying the target type

By default, Unicode-enabled streams return strings of type TEXT. You may select the target type in the OPEN command:

```
Call Stream fn, "C", "Open Read ENCODING UTF-8 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 become very difficult to implement.*

Low-level functions

- `DataType(string,"C")` returns 1 when the contents of `string` is a valid Unicode string.
- `C2X()` accepts an optional second parameter, `C2X(string, encoding)`. `Encoding` defaults to `UTF-8`. This definition has interesting properties: when `string` is a `BYTES` string containing well-formed `UTF-8` normalized to NFC, then

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

The Unicode Tools Of Rexx

Part VI

New built-in functions

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.

Additionally, `TEXT()` converts its argument, if necessary, to the NFC Unicode normalization form.

DECODE (1/2)

- *Encoding validation*: `DECODE(string, encoding)` returns 1 if `string` is well-formed according to the `encoding`, and 0 otherwise.
- *Decoding*: `DECODE(string, encoding, format)` decodes `string` to the a certain `format` (`"UTF-8"` or `"UTF-32"`).

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

DECODE (2/2)

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

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

  $\Rightarrow$ ill-formed sequences are substituted by the Unicode replacement character, "FFFD"X.
- When the fourth argument is omitted, or specified as "" or "NULL" (the default), a null string is returned in the event that an ill-formed sequence is encountered.
- When the fourth argument is "SYNTAX", a syntax error is raised in the event that an ill-formed sequence is encountered.

ENCODE (1/2)

`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)
```

ENCODE (2/2)

- 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")
```

UTF8

A version of `DECODE()` specialized in variants of the `UTF-8` format. It can validate and decode the `UTF-8`, `UTF-8Z`, `WTF-8`, `CESU-8`, and `MUTF-8` formats.

`UTF8()` does not depend on other components of TUTOR, and it can be used independently.

C2U (Character to Unicode)

```
Say C2U("Yes")           == "0059 0065 0073"            /* 1 */
Say C2U("Yes", "U+")     == "U+0059 U+0065 U+0073"      /* 1 */
Say C2U("Y", "Names")    == "(LATIN CAPITAL LETTER Y)"  /* 1 */
Say C2U("0A"X, "Names") == "(<control-000A>)"           /* 1 */
Say C2U("Y", "UTF-32")   == "0000 0059"X                /* 1 */
```

N2P (Name to codePoint)

Returns the hexadecimal code point corresponding to name, or the null string if name does not correspond to a Unicode code point.

```
Say N2P("LATIN CAPITAL LETTER A")  /* "0041" (Name)  */
Say N2P("Latin Capital Letter A")  /* "0041" (Name)  */
Say N2P("Latin-Capital-Letter-A")  /* "0041" (Name)  */
Say N2P("Latin_Capital_Letter_A")  /* "0041" (Name)  */
Say N2P("Bell")                    /* "1F514" (Name) */
Say N2P("LF")                      /* "000A" (Alias) */
Say N2P("form feed")               /* "000C" (Alias) */
Say N2P("<control-0001>")          /* "0001" (Label) */
Say N2P("<Private use-E000>")      /* "E000" (Label) */
Say N2P("potatoes")                /* "" (Not found) */
```

P2N (codePoint to Name)

Returns the name or label corresponding to the hexadecimal code point argument.

```
say P2N(61)              /* "LATIN SMALL LETTER A"  */
Say P2N("1F957")         /* "GREEN SALAD"           */
Say P2N("A")             /* "<control-000A>"        */
Say P2N("10ffff")        /* "<noncharacter-10FFFF>" */
```

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` is 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.

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.

Functional form

`UNICODE(string, function)` implements a series of Unicode-defined functions. The particular function is selected by specifying its name as the string `function` argument. Currently, `isNFC`, `isNFD`, `toNFC`, `toNFD`, `toLowercase` and `toUppercase` are available.

Property form

The `UNICODE(code, "PROPERTY", name)` returns the Unicode property identified by `name` applied to the code point `code`. `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` codepoint (i.e., a four byte binary integer), or an hexadecimal code point (with no leading `"U+"`).

The Unicode Tools Of Rexx

Part VII

Utilities

The setenv utility

Sets the path before using other TUTOR tools.

Under Windows, use `setenv.cmd`:

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

C:\Unicode>
```

Under Linux, use `. ./setenv.sh`:

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

The REXX preprocessor for Unicode (rxu) (1/2)

The preprocessor is implemented by a ooREXX command, `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 REXX preprocessor for Unicode (rxu) (2/2)

Effect of `rxu filename` (without error handling logic):

1. Translate `filename.rxu` to `filename.rex`, a pure ooREXX program.
2. Run `filename.rex`.
3. Delete `filename.rex`.

The translation phase makes heavy use of the REXX tokenizer, described in a separate presentation.

⇒ You can experiment with the `-keep` option to see how the translator works.

The `rxutry.rex` utility (1/2)

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.

The rxutry.rex utility (2/2)

```
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>
```

The Unicode Tools Of Rexx

Part VIII

Conclusions

Further work

- Implement more Unicode features. Example: the Unicode collation algorithm.
- Adapt more language features to Unicode. Example: the `Parse` instruction.
- Reimplement parts of TUTOR in pure Classic REXX. ⇒ Parts of TUTOR might be run under CMS, TSO, VM/370, etc.
- Somebody could rewrite parts of TUTOR in C/C++ (not me: I don't do C/C++). ⇒ A growing Unicode library for many REXX implementations (I would think of ooREXX and REGINA as a minimum).

Acknowledgements (1/2)

- 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.

Acknowledgements (2/2)

- 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.

Resources

- This file: https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx-slides.pdf.
- Related article: https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx.pdf.
- Accompanying article, *A Tokenizer for* REXX *and* ooREXX: https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx.pdf. Slides: https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx-slides.pdf.

Questions?

Thank you!

Questions?