A Tokenizer for REXX and OOREXX[*]

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 a tokenizer for the REXX language. The tokenizer is written in Open Object REXX (OOREXX), and it can process program files written in OOREXX, Regina REXX, or ANSI REXX. Experimental support for Unicode extensions is also included (Unicode support requires the use of the TUTOR package). The tokenizer has two modes of operation: *simple tokenizing*, where all items are returned as-is, and *full tokenizing*, where REXX rules are applied to discard separator items, like comments or non-significant whitespace, and certain tokens are combined into higher-level items, like compound operators or extended assignment operators. In the case of full tokenizing, *detailed* or *undetailed* tokenizing can also be selected; the former includes, as an attribute of every higher-level construct, the sequence of constituent items, while the latter discards these low-level elements. In both cases, the returned sequence contains enough information to reconstitute the whole source program file.

The tokenizer is distributed as part of the TUTOR package, which is described in an accompanying document, but, when not making use of the Unicode features, it does not depend on TUTOR, and, therefore, it can be used independently.

[*]URL of this document: `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx.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 document, we present a *tokenizer* for several dialects of the REXX language; the exact meaning of "tokenizer" will be made clear below. The dialects include Regina REXX ("REGINA") and some of its variants (e.g., ANSI REXX), as well as Open Object REXX ("ooREXX"), and several experimental extensions of REXX for Unicode.

1.1 History of the tokenizer

Work on the tokenizer started in mid May, 2023, just after the closing of the 34th International Rexx Language Symposium. The original plan was to produce a full abstract syntax tree (AST) parser, and writing a tokenizer was a necessary first step to that end. It also created an opportunity to delve into the ANSI standard (1996), and also into the fascinating, but regretfully incomplete, Object REXX Dallas Draft report (1998).

Around March, 2023, I was invited to participate in the Architecture Review Board (ARB) of the Rexx Language Association (REXXLA). Soon, discussions about a possible Unicode-enabled implementation of REXX started; I became very active in these debates. On 10 July I released "a toy ooREXX implementation of the `General_Category` Unicode property";[1] several new releases of the then called "Unicode toys" quickly followed.[2]

I started to realize that many of the concepts we were debating about would be much easier to understand (and, eventually, to improve) if we could play with them, that is, if we could manipulate them in practice. We would thus need an implementation of a Unicode extension of REXX that included these concepts. Writing a preprocessor would be a way to fulfil these needs: it would translate Unicode-extended REXX into standard REXX, and, with the help of an accompanying library, it would allow us to experiment with the new concepts.

Normally, writing a preprocessor is not a trivial task, but in the present case I could count on my tokenizer. In a few days, and after introducing some few modifications to the tokenizer so that it could optionally parse Unicode-extended REXX, I had a working prototype of the preprocessor.

On 19 July I released version 0.1d of the *Unicode Toys for REXX*: they

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

[2]See, for example, version 0.1, released on 20230716 (`https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/0.1-release-notes.md`).

included the updated tokenizer, and a new Rexx preprocessor for Unicode (rxu). At some moment in August, the Toys were renamed to *The Unicode Tools for Rexx*, and, later, following a suggestion by Chip Davis, to *The Unicode Tools Of Rexx* (a denomination which happens to have a nice acronym: Tutor).

The tokenizer is since being distributed as part of the Tutor package, but, if no use is being made of the Unicode extensions, it can be used independently of that package.[3]

1.2 Structure of this document

Section 1, *Introduction*, on page 4, contains a minimal presentation, a historical introduction, and the structure listing you are now reading.

Section 2, *General concepts*, on page 6, includes a gentle introduction to the basic concepts needed to understand what a tokenizer is. We specifically discuss the different uses of the term "token" that can be found in the Rexx literature, and we also establish our own nomenclature, to be used throughout this document. We also detail some of the possible uses of a tokenizer.

Section 3, *Tokenizer features*, on page 9, highlights the features of our tokenizer. Some of the specificities of the Rexx language are examined, and we introduce the notions of *simple* and *full* tokenizing. We highlight the fact that the tokenizer can accept several dialects of the Rexx language, and we introduce important details about the Unicode support, in particular, we list the five new kinds of string that the Tutor-flavoured variant of Unicode-enabled Rexx defines.

Section 4, *Using the tokenizer*, on page 12, covers the installation and use of the tokenizer, how to create a tokenizer instance, how to use the *simple* and the *full* tokenizers, and, in the latter case, how to request *detailed* or *undetailed* tokenizing. The section ends with the examination of a code snippet recommended for error handling.

Section 5, *Structure of the returned items*, on page 15, details the internal structure of the items returned by the tokenizer. The concept of *attribute* is introduced, and the standard attributes, *class*, *subclass*, *location* and *value*, common to all items, are defined. The fact that some special items, and full tokenization results, may include additional attributes, is examined. The concrete, symbolic names that define the repertoire of the *class* and *subclass* attributes are listed in the appendixes.

Section 6, *Testing the tokenizer: the* `InspectTokens` *program*, on page 17,

[3]Please refer to section 4.1, *Installation instructions*, on page 12, for details about the installation and use of the tokenizer under both scenarios.

describes `InspectTokens`, a Rexx program designed to test and debug the tokenizer. `InspectTokens` is also excellent as a learning tool. The diverse modes of operation of `InspectTokens` are described, and sample test runs are examined in detail.

Section 7, *An example application: the Rexx tokenizer for Unicode*, on page 22, describes, as a sample application of the tokenizer, the implementation of RXU, the Rexx Preprocessor for Unicode; RXU makes heavy use of the tokenizer. This will give us an occasion to examine in detail some of the internal workings of RXU, and the nuances of its translation process.

Section 8, *Further work*, on page 28, outlines the natural evolution in the development of the tokenizer, a full abstract syntax tree (AST) parser, and quickly explores the possibilities that the existence of such a parser would open.

Section 9, on page 28, contain the *Acknowledgements*.

Appendix A, *Class and subclass constants used in simple tokenizing*, on page 29, describes the part of `TokenClasses` related to simple tokenizing. `TokenClasses` is a constant method of the Rexx tokenizer which provides an array of symbolic names for the item classes and subclasses.

Appendix B, *Class and subclass constants used in full tokenizing*, on page 31, describes the rest of the symbolic names provided by the `TokenClasses` constant; these are used exclusively by the full tokenizer.

2 General concepts

2.1 Alphabets, lexical elements and syntax

Like natural languages, programming languages have an *alphabet*. Natural languages form *words* using *letters*, a distinguished subset of the alphabet, and normally allow also *numerals* and some forms of *punctuation marks*. Similarly, a programming language defines its own set of *lexical* elements; usual elements are *identifiers*, *numbers*, *strings*, *operators* and other *punctuation*. Natural languages define a set *syntactic* rules that stipulate how lexical elements should be combined to produce meaningful discourse, although these rules are routinely ignored or deliberately broken in certain allowed contexts, like jokes, marketing, press headlines, poetry, and many others. Syntactic rules in formal languages do not allow for such leniencies, since one of the major requirements of a formal language is to convey meaning in a completely unambiguous way. When we are using a formal language, be it the language of logic, mathematics, physics, or a programming language, we want to be absolutely sure of what we are saying, and of the meaning of what we are saying; hence, the syntactic rules will have to be relatively *rigid* and, in any case, they will not be optional, in the sense that the existence of special contexts in which they can be ignored or broken will not be allowed.

2.2 Lexers, tokenizers and parsers

Lexical analysis of a program breaks the program into its constituent lexical elements. An application that parses programs written in a certain programming language and identifies and returns its lexical elements is usually called a *lexer* or a *tokenizer*, although the term "token" has specific and slightly different meanings in the REXX language. Lexical elements or tokens are combined into higher level constructs or structures, like *expressions*, *templates*, *instructions* or *directives.* Many of these constructs are defined in a recursive way, and therefore their instances can be represented naturally using certain variants of nested, tree-like structures. A tool that is able to completely identify and extract these higher level structures from a source program is called a *parser.* The representation of a whole program using these higher level structures is sometimes known as an *abstract syntax tree* (AST).

2.3 Rexx clauses, tokens and items

The REXX language defines a program to be a sequence of *clauses.* A clause, in turn, is composed of sequences of whitespace, comments, and certain syntactical constructs called *tokens*, and it ends with a semicolon. In REXX, this is the main acceptation of the word "token".

In most cases, the semicolon that ends a clause is implied by the syntactic rules, and can in practice be omitted.

An application that scans REXX programs and identifies and returns the *tokens* that compose that program is a *tokenizer*; for completeness and for convenience (in particular, to be able to reconstitute the original program), a tokenizer can identify and return not only the tokens, but also these other syntactic sequences, which are part of the program but are formally not tokens, like whitespace or comments. We will use the term *items* to refer to the whole set of tokens and non-token separators.

2.4 "Tokenized" programs

In addition to the formal meaning of the term "token", in the REXX parlance we find another, different, connotation. Colloquially, one refers to a program distributed without source (for example, after being processed by the ooRexx`rexxc` utility) as a *tokenized* program. Although this denomination has stuck, in fact it is inexact, because the output of `rexxc` is indeed a full abstract syntax tree representing the source program, and not a mere sequence of tokens. In this article, we will use the term "tokenizer" in its proper sense (i.e., a tool that breaks programs into their constituent *items*, i.e., tokens, and other separators).

2.5 What is a tokenizer good for?

Tokenizers are an indispensable part of language processors like compilers and interpreters. To be able to interpret a program, an interpreter needs to "understand" it first, and to this purpose it necessarily has to begin by breaking the program into its constituent elements.

A tokenizer, however, can also be used for many other purposes. Since it essentially produces the sequence of lexical elements that compose a source program, it is ideally suited to *introduce transformations* into such a sequence, and also to *compile data* about that sequence.

2.5.1 Transforming programs

Transformations can be of a cosmetic nature, as in a program formatter or prettyprinter, or have a deeper purpose, like implementing some form of language extension.

As an example of the latter, we have implemented a set of experimental, Unicode-related, extensions to the Rexx language in our Rexx Preprocessor for Unicode (RXU), using a slight modification of our original tokenizer. The RXU preprocessor is part of the (Tutor) package, described in a separate and accompanying document,[4] where a *definition* of the functionality of RXU can be found; the *implementation* of RXU is described in some detail below, in section 7, *An example application: the Rexx tokenizer for Unicode*, on page 22.

2.5.2 Compiling data about a program

Compiling data allows to create a number of program analysis tools, like variable and cross-reference listings.

[4]See *The Unicode Tools Of Rexx*, `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx.pdf`. See also section 3.4, *Experimental support for Unicode*, on page 11, below.

A cross-referencer for the Rexx language cannot be reasonably implemented with a tokenizer, because Rexx has no reserved words, and therefore it is impossible, in the general case, to determine whether a certain symbol is or not a keyword, without resorting to a deeper syntactic analysis of the source program. In the case of languages with reserved words, however, some forms of cross-referencing can be easily implemented with a simple tokenizer.

2.5.3 Tokenizers vs. parsers

Tokenizers form the basis for a full parser, since a parser is always built upon a tokenizer. Once a full parser is available, it becomes possible, among other things, to write more ambitious language extensions and program analysis tools, but also interpreters, compilers, and translators to other languages (or to different dialects of the same language).

3 Tokenizer features

3.1 The specificity of Rexx

The syntax of the Rexx language contains a number of departures from what is usual in most other languages. There are no reserved words, for example, and therefore keywords can be used as the names of variables in many cases;

To a degree that varies, depending on the dialect and implementation. For example, ooRexx considers

```
Do i = 1 To While
```

erroneous, while it allows

```
Do i = 1 To (While).
```

Regina, on the other hand, does not accept any of these variations.

whitespace outside literal strings is only used as a separator, but it is otherwise ignored in most contexts; however, there is one of these contexts (concatenation) where it is significant; there is a single concept of “symbol”, which encompasses simple variables, stems, compound variables, numbers, environment symbols, and constant symbols;

Constants symbols, like `23abc`, which has a value of `"23ABC"`, are a very unusual construction. Similarly, the syntactical exception by which a numeric symbol can include a signed exponent is also very rare.

the very concept of "token" is somewhat counterintuitive, because some of the basic building blocks of the language, for example multi-character operators, extended assignments and the "::" construction that starts a directive, are not tokens, but combinations of tokens (which may be separated by optional non-tokens): this means that one can insert whitespace and/or comments between the different characters of an operator, for example (not that it is, generally speaking, a very good idea). A tokenizer for the REXX language has to take into account all these specificities.

```
/* This not very readable, to say the least:          */
a1 = a2 | /* comment */ - /* continued...             */
        | a3 /* That was a concatenation, after all   */
```

3.2 Simple and full tokenizing

For some applications, it may be necessary to get the items of a source program exactly as they are written. The assignment instruction

```
a += 1,
```

for example, would get us *six* items, namely:

- "a" (a variable symbol),
- " " (whitespace, a blank),
- "+" (an operator character),
- "=" (another operator character),
- " " (another blank), and
- "1" (an integer number symbol).

In other cases, we might benefit from a higher level of analysis; in our example, we might want to get only three items:

- a variable ("a"),
- an extended assignment operator ("+="),
- and a number ("1").

Our tokenizer implements variants for both possibilities. We call the verbatim tokenizing of a source file (i.e., our first example) a *simple* tokenizing, and the more elaborate version a *full* tokenizing. A tokenizer instance can be used as a simple tokenizer, or as a full tokenizer (but not as both at the same time).

3.3 Tokenizing several dialects

From a syntactic point of view, the differences between the different dialects of REXX are minimal (and, in some cases, these differences can also be extremely subtle). This is one of the reasons why the ANSI standard defines a number of optional syntactical categories that extend the basic REXX definitions that any conforming dialect has to implement. For example, `extra_letters` (5.3.2) is empty for OOREXX, but it includes `"$"`, `"#"` and `"@"` for REGINA. Similarly, OOREXX only recognizes the horizontal tab (`HT`) character as `other_blank_characters`, but REGINA additionally allows `CR`, `FF`, `LF` and `VT`.

Different dialectal variants of the same implementation also recognize different syntactical constructs. For example, the single line comment form (that is, comments starting with `"--"`), is allowed by default by REGINA, but not by the ANSI variant of REGINA itself.

The tokenizer is designed to accept several dialects of REXX. This is accomplished by creating a main, centralized, class, `Rexx.Tokenizer`, that implements the basic functionality common to all the dialects, and then writing a number of subclasses that implement the aspects specific to each dialect. Currently, OOREXX (class `ooRexx.Tokenizer`), REGINA (class `Regina.Tokenizer`) and ANSI REXX (class `ANSI.Rexx.Tokenizer`) are supported, and it would most probably be very easy to add support for other dialects, like BRexx.

3.4 Experimental support for Unicode

In addition to the above variants of REXX, the tokenizer includes an extra set of classes that implement some Unicode extensions, following the design guidelines of the TUTOR package.

For a full description of the Unicode extensions defined by TUTOR, please refer to the accompanying document, *The Unicode Tools of REXX*.[5]

The Unicode class names are formed by adding `".Unicode"` before `".Tokenizer"`, so that, for example, `ooRexx.Unicode.Tokenizer` is the Unicode variant of the OOREXX tokenizer, `ooRexx.Tokenizer`. Unicode classes extend the language by allowing five new, case insensitive, string suffixes:

- `"Y"` strings (`bYtes` strings): a string terminated with a `"Y"` suffix, `"string"Y`, is a `BYTES` string, that is, a string explicitly composed of bytes. This is equivalent to current REXX strings. The notation can

[5]`https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx.pdf`

be used to emphasize that a string is a classic REXX string, or when default, unsuffixed, strings are set to represent another kind of strings.

- `"P"` strings (`codePoints` strings): a string terminated with a `"P"` suffix, `"string"P`, is a `CODEPOINTS` string, that is, a string explicitly composed of Unicode code points. The source string has to contain well-formed UTF-8, or a syntax error will be raised at parse time.
- `"G"` strings (`Graphemes` strings): a string terminated with a `"G"` suffix, `"string"G`, is a `GRAPHEMES` string, that is, a string explicitly composed of Unicode extended grapheme clusters. The source string has to contain well-formed UTF-8, or a syntax error will be raised at parse time.
- `"T"` strings (`Text` strings): a string terminated with a `"T"` suffix, `"string"T`, is a `TEXT` string, that is, a string explicitly composed of Unicode extended grapheme clusters, and normalized to the Unicode Normalization Form C (NFC). The source string has to contain well-formed UTF-8, or a syntax error will be raised at parse time, and it will be automatically normalized to the NFC form if necessary.
- `"U"` strings (`Unicode` strings): a string terminated with a `"U"` suffix, `"string"U`, is a `BYTES` string composed of (the UTF-8 representation of) a sequence of Unicode codepoints specified inside the quotes. You can use character names, aliases and labels, and also codepoint numbers, to specify the contents of a ”U” string. Names, aliasis and labels should be specified between parentheses; codepoints should be separated by blanks, and they can optionally include the ”U+” prefix that many languages use.

```
"Test"Y                  /* A BYTES string           */
"🐪"Y                    /* Another BYTES string     */
                         /* 1 emoji, 4 bytes (UTF-8) */
"🐄🦒"P                  /* A CODEPOINTS string      */
                         /* 2 emojis, 2 code points  */
"(cow)(giraffe)"U        /* A BYTES string           */
                         /* 2 emojis, 8 bytes        */
```

4 Using the tokenizer

4.1 Installation instructions

Install the TUTOR package[6] and follow the instructions found in the `readme.md` file. In your program, you will need to load the Unicode library:

[6]You can download it from `https://github.com/RexxLA/rexx-repository/tree/master/ARB/standards/work-in-progress/unicode/UnicodeTools`

```
::Requires "Unicode.cls"
```

Alternatively, and only in the case that you are not going to use the new Unicode strings,[7] you can directly download `Rexx.Tokenizer.cls`, a `readme.md` file and some utility programs from the `parser` subdirectory.[8] In this case, you can directly load the tokenizer classes:

```
::Require "Rexx.Tokenizer.cls"
```

4.2 Creating a tokenizer instance

To create a tokenizer instance, you will first need to construct a Rexx array containing the source program to tokenize.

```
/* Read the whole file into an array                      */
source  = CharIn(inFile,,Chars(inFile))~makeArray
```

This array will then be passed as an argument to the `new` method of the corresponding tokenizer class, to get an instance of the tokenizer for this particular program source.

```
/* Now create a tokenizer instance                        */
tokenizer = .ooRexx.Tokenizer~new(source)
/* Or .Regina.Tokenizer, etc.                             */
```

4.3 Using the simple and full tokenizers

You will also have to decide whether you will be using the *simple tokenizer* (i.e., if you will be getting items using the `getSimpleToken` tokenizer method), or you will prefer to use the *full tokenizer* (i.e., you will be getting your items using the `getFullToken` tokenizer method).

```
Do Forever
  item = tokenizer~getSimpleToken
  /* Or tokenizer~getFullToken                             */

  /* Retrieve the class of our item                        */
  cls   = item[class]

  /* Two possible reasons to exit the loop                 */
```

[7] Unicode strings are implemented in `Unicode.cls`.

[8] `https://github.com/RexxLA/rexx-repository/tree/master/ARB/standards/work-in-progress/unicode/UnicodeTools/parser`

```
If cls == END_OF_SOURCE | cls == SYNTAX_ERROR Then Leave
  /* Do things with the itemn                            */
End
```

4.4 Detailed and undetailed tokenizing

Both kind of items will be described below. In case you have opted for the full tokenizer, you will also be able to select *detailed* or *undetailed* tokenizing. *Detailed* tokenizing will return the sequence of simple items absorbed by the full item as an attribute (`"absorbed"`) of the full item (we will define what an attribute is shortly).

```
/* A second, boolean and optional, argument of the  */
/* 'new' method determines if tokenizing will be    */
/* detailed or not.                                 */
tokenizer = .ooRexx.Tokenizer~new(source, .true)
```

Undetailed tokenizing returns only the full items, and discards the elementary, simple items, once the full items have been constructed.

```
/* Creates an undetailed tokenizer                  */
tokenizer = .ooRexx.Tokenizer~new(source, .false)
```

In any case, you will always be able to reconstitute the entirety of your source file by considering the `location` attributes of the returned items.

4.5 Error handling

When an error is encountered, tokenizing stops, and a special item is returned. Its class and subclass will be `SYNTAX_ERROR`, and a number of special attributes will be included, so that the error information is as complete as possible:

```
item.class              = SYNTAX_ERROR
item.subclass           = SYNTAX_ERROR
item.location           = location in the source file
                           where the error was found
item.value              = main error message

/* Additional attributes, specific to SYNTAX_ERROR    */
item.number             = the error number,
                          in the format major.minor
```

```
item.message          = the main error message
                        (same as item.value)
item.secondaryMessage = the secondary error message,
                        with all substitutions applied
item.line             = line number where the error
                        occurred (first word of
                        .location)
```

If you want to print error messages that are identical to the ones printed by ooRexx, you can use the following code snippet:

```
If item.class == SYNTAX_ERROR Then Do
  line = item.line
  Parse Value item.number With major"."minor

  Say
  /* "array" contains the source code                 */
  Say Right(line,6) "*-*" array[line]
  /* "inFile" is the input file name                  */
  Say "Error" major "running" inFile,
    "line" line":" item.message
  Say "Error" major"."minor": " item.secondaryMessage

  /* -major should be returned when a syntax error    */
  /* is encountered                                   */
  Return -major
End
```

5 Structure of the returned items

The elements returned by the `getSimpleToken` and `getFullToken` methods are Rexx stems. In an abuse of language, we will ofter refer to the stem tails (indexes) as “attributes” of the stem. A item `i.` has a *class*, `i.class`, a *subclass*, `i.subclass`, a *location* `i.location`, and a *value*, `i.value`, and, in some few cases, some additional attributes.

5.1 Classes and subclasses

Classes and *subclasses* are defined in the `tokenClasses` array constant (simple and full tokenizing return different subsets of the whole set of item classes).

You can find a listing of the simple tokenizer classes and subclasses in appendix A, *Class and subclass constants used in simple tokenizing*, on page 29, and a listing of the full tokenizer classes and subclasses in appendix B, *Class and subclass constants used in full tokenizing*, on page 31.

You should use the following code to replicate these constants in your own program:

```
Do constant over tokenizer~tokenClasses
  Call Value constant[1], constant[2]
End
```

You should always use this construction, instead of relying on the internal values of the constants, which can be changed without notice.

5.2 Locations

A *location* is a blank-separated sequence of four integers, in the form `"startLine startCol endLine endCol"`. `Startine` and `startCol` refer to the first character in a item; `endLine` and `endCol` refer to the first position *after* the item (this may not correspond to any character in the line if the item occupies the last position in the line). `Startine` and `endLine` will always be identical, except for multi-line comments and ooRexx resources.

Locations are guaranteed to be *sequentially adjacent* and *total*: two consecutive locations (that is, the locations of two items obtained one after the other), "l_1 c_1 l_2 c_2" and "l_3 c_3 l_4 c_4", are *adjacent* when the end of the first is the start of the second, that is, because $l_2 = l_3$ and $c_2 = c_3$, and then they can be *composed* into a super-item "l_1 c_1 l_4 c_4"; the composition of all the items in the source file is *total* because it always is `"1 1 line col"`, where `line` is the number of lines in the source program, and `col` is the length of the last line, plus 1. The source file can always be reconstituted by collating the substrings indicated by the sequence of locations.

5.3 Values

The *value* of an item is normally the item itself, except in some special cases. When *comments* or *resources* are tokenized, they are not returned as the values of the `value` attribute, but only a placeholder is (you can always use the `location` of the item to retrieve the original value from the source file array). *Strings* are interpreted, so that the actual string value is returned; this means that separator blanks are removed (for binary, hexadecimal and Unicode strings), double quotes are eliminated, and Unicode code points, names, aliases and labels are substituted by their UTF-8 representations.

5.4 Other attributes

Some few item classes return stems with additional attributes (apart from `class`, `subclass`, `location` and `value`).

> For example, items with the `SYNTAX_ERROR` class return additional information, like the secondary error message, major and minor error codes, and more. See section 4.5, *Error handling*, on page 14 for details.

6 Testing the tokenizer: the `InspectTokens` program

The tokenizer distribution includes a sample test program called `InspectTokens.rex`, located in the `parser` subdirectory. `InspectTokens` takes a Rexx program file name as its argument, tokenizes the program source, and prints the results of the tokenization in the terminal. It is an excellent tool to play with the tokenizer, and to understand how it processes a source file. It is also a very useful tool to debug the tokenizer itself.

`Inspecttokens` recognizes a set of options that allow the selection of all possible variants and options of the tokenizer. Calling `InspectTokens` without arguments (or with the `-h` or `-help` options) produces the following output:

```
InspectTokens.rex -- Tokenize and inspect a .rex source file
------------------------------------------------------------

Format:

  [rexx] InspectTokens[.rex] [options] [filename]

Options (starred descriptions are the default):

  -h,  -help                   Print this information
  -d,  -detail, -detailed      Perform a detailed tokenization (*)
  -nd, -nodetail, -nodetailed  Perform an undetailed tokenization
  -f,  -full                   Use the full tokenizer (*)
  -s,  -simple                 Use the simple tokenizer
  -u,  -unicode                Allow Unicode extensions (*)
  -nu, -nounicode              Do not allow Unicode extensions
  -o,  -oorexx                 Use the Open Object Rexx tokenizer (*)
  -r,  -regina                 Use the Regina Rexx tokenizer
  -a,  -ansi                   Use the ANSI Rexx tokenizer
```

We will see how `InspectTokens` works by examining the results of a small number of tests runs. We will have to create a very simple one-line test file, called `test.rex`. Its contents will be the following (we have added a scale for your convenience):

```
i = i + 1
....+....1
```

6.1 Simple tokenizing

If we run `InspectTokens` against `test.rex` with the `-simple` option,

```
InspectTokens -simple test.rex
```

we will get the following output:

```
 1 [1  1 1  1] END_OF_CLAUSE (BEGIN_OF_SOURCE): ''
 2 [1  1 1  2] VAR_SYMBOL (SIMPLE_VAR): 'i'
 3 [1  2 1  3] BLANK: ' '
 4 [1  3 1  4] OPERATOR: '='
 5 [1  4 1  5] BLANK: ' '
 6 [1  5 1  6] VAR_SYMBOL (SIMPLE_VAR): 'i'
 7 [1  6 1  7] BLANK: ' '
 8 [1  7 1  8] OPERATOR: '+'
 9 [1  8 1  9] BLANK: ' '
10 [1  9 1 10] NUMBER (INTEGER): '1'
11 [1 10 1 10] END_OF_CLAUSE (END_OF_LINE): ''
```

6.1.1 Output format

- The first column is *an item counter.*
- The second column is the *location* of the item, prettyprinted and enclosed [between brackets].
- The third column contains one or two values. When there are two, the second one is separated by a blank and enclosed between parentheses. These are the *class* and the *subclass* of the item, as defined above. They give a lot of information about the nature of the item (e.g., this is a `NUMBER` [class], subclass `INTEGER`; or this is a `VAR_SYMBOL` [class], subclass `SIMPLE_VAR` [i.e., not a stem or a compound variable]).
- The fourth column, after a colon and between simple quotes, is the *value* of the item. Generally speaking, this is the item itself, as it appears in the source file, but in some few cases (classic comments,

resources), only an indicator is returned (you can always reconstitute the original comment or resource by referring to the *location* attribute of the item). In some other cases, the value contains *an elaboration* of the original item: for example, an `X`, `B` or `U` string will be interpreted, so that its value can be substituted in the source file. The tokenizing of `"(Steam locomotive)"U`, for instance, will generate a value of `"🚂"`.

6.1.2 Analysis of the returned items

1. The first item returned by the tokenizer is always a dummy `END_OF_-CLAUSE`, subclass `BEGIN_OF SOURCE`. This allows considerable simplication of many internal algorithms. For example, if you want to keep track of the item number in a line, you can safely reset your counter to zero when you receive an `END_OF_CLAUSE` item. In the absence of this dummy item, it would become necessary to dual-path: “if we are at the beginning of the program, *or* we just got an `END_OF_CLAUSE` item, then...”.
2. The second item returned is a proper token, a `VAR_SYMBOL` (a variable symbol), and its subclass is `SIMPLE_VAR` (i.e., it is not a stem or a compound variable). The name of the variable, `"i"`, is returned as the value of the item.
3. The third item returned by the tokenizer is a separator, `BLANK`, i.e., some whitespace. In this case, `BLANK` represents a single blank character, `" "`, as its value attribute shows, but it could also have been a tab character, or any combination of whitespace characters (what constitutes a whitespace character is dependent on the dialect).
4. The fourth item returned by the tokenizer is another proper token, an `OPERATOR` (an operator character). The operator character, `"="`, is returned as the value of the item. Simple tokenizing does not offer more detail about this character; as we will see below, full tokenizing will return more detailed discriminations, by using the subclass attribute of the item.
5. Another blank.
6. Another variable symbol.
7. Still another blank.
8. An operator more (in this case, `"+"`).
9. One blank more.
10. The tenth item returned by the tokenizer is a proper token, a `NUMBER`, subclass `INTEGER`, with a value of `"1"`.
11. The eleventh item returned by the tokenizer is an end-of-clause, generated by the end-of-line. It has the same effect as an explicit semicolon,

as it terminates the clause.

The tokenizer will also return a final `END_OF_FILE` item, but `InspectTokens` will not print it.

6.2 Undetailed full tokenizing

We will now invoke `InspectTokens` once more against `test.rex`, but this time we will be asking for an undetailed full tokenizing:

```
InspectTokens -full -nodetailed test.rex
```

The program output will be the following (colours are ours and not part of the program output):

```
1 [1  1 1  1] END_OF_CLAUSE (BEGIN_OF_SOURCE): ''
2 [1  1 1  2] ASSIGNMENT_INSTRUCTION (SIMPLE_VAR): 'i'
3 [1  2 1  5] OPERATOR (ASSIGNMENT_OPERATOR): '='
4 [1  5 1  6] VAR_SYMBOL (SIMPLE_VAR): 'i'
5 [1  6 1  9] OPERATOR (ADDITIVE_OPERATOR): '+'
6 [1  9 1 10] NUMBER (INTEGER): '1'
7 [1 10 1 10] END_OF_CLAUSE (END_OF_LINE): ''
```

What are the changes, relative to simple tokenizing? Well, both the `"="` operator and the `"+"` operator seem to have "grown", while several items have disappeared (indeed, all the items with a `BLANK` class). Indeed, the operators seem to have "eaten" or "absorbed" the corresponding blanks; we have changed the colour of their `location` value to magenta to highlight this fact. This absorption strictly follows the rules of Rexx: blanks before and after operator characters are always ignored. The tokenizer ignores the blanks, but, at the same time, it does not want to lose information — to that effect, it "expands" the absorbing items, by making them wider, so that they can (so to speak) "accommodate" the ignored blanks: the `"="` on line 3, for example, runs now from [1 2 1 3] to [1 4 1 5], that is, it encompasses all the characters between, and including, the previous and following blanks, `" "`, `"=" and " "`.

There are some other, not immediately apparent, changes in the returned results. The class of `"i"` has changed, for example: it is no longer `VAR_-SYMBOL`, but `ASSIGNMENT_INSTRUCTION`. The full tokenizer "knows" that `i = i + 1` is an assignment instructions, and it passes this knowledge to us. Similarly, the subclass of `"="` has changed. Previously, it was `OPERATOR`: all the tokenizer knew was that `"="` was an operator character. Now it is

`ASSIGNMENT_OPERATOR`, which is much more informative. Finally, `"+"` has now a subclass of `ADDITIVE_OPERATOR`.

6.3 Detailed full tokenizing

As a last test, we will run `InspectTokens` against `test.rex` one last time, asking again for a full tokenizing, but it this case a detailed one (the default):

```
InspectTokens -full test.rex
```

Here is the program output (lines which are new are highlighted in magenta):

```
1 [1  1 1  1] END_OF_CLAUSE (BEGIN_OF_SOURCE): ''
2 [1  1 1  2] ASSIGNMENT_INSTRUCTION (SIMPLE_VAR): 'i'
3 [1  2 1  5] OPERATOR (ASSIGNMENT_OPERATOR): '='
   ---> Absorbed:
   1 [1 2 1 3] BLANK: ' '
   2 [1 3 1 4] OPERATOR: '=' <==
   3 [1 4 1 5] BLANK: ' '
4 [1  5 1  6] VAR_SYMBOL (SIMPLE_VAR): 'i'
5 [1  6 1  9] OPERATOR (ADDITIVE_OPERATOR): '+'
   ---> Absorbed:
   1 [1 6 1 7] BLANK: ' '
   2 [1 7 1 8] OPERATOR: '+' <==
   3 [1 8 1 9] BLANK: ' '
6 [1  9 1 10] NUMBER (INTEGER): '1'
7 [1 10 1 10] END_OF_CLAUSE (END_OF_LINE): ''
```

The *unindented* lines are exactly the same as in our previous run. New in the listing are the *indented* lines, grouped in different sets, and highlighted: each set lists the simple items absorbed by the parent items in the full tokenizing process. As it is easily visible, only the items that have absorbed some items list their absorptions (in the other cases, since there is nothing absorbed, nothing is listed).

Every sequence of absorbed items starts with an `"---> Absorbed:"` marker. A number of lines then follows, listing the original items present in the simple tokenizing of the source file. The original absorbing item has a `"<=="` marker to the right of its value.

7 An example application: the Rexx tokenizer for Unicode

In this section, we briefly describe *the implementation* of an example application of our tokenizer: RXU, the REXX preprocessor for Unicode, part of *The Unicode Tools Of REXX* (TUTOR), a package described in a separate document.[9]

> An ampler definition of *the specification* of the preprocessor and its role in the totality of TUTOR can be found in the accompanying document.

The main purpose of the TUTOR package is to facilitate the understanding and the implementation of prototypes of Unicode-enabled REXX. To this purpose, the package defines a set of Unicode extensions to the (ooREXX variant of the) REXX language. We will say that this extended language is (TUTOR-flavored) *Unicode REXX*. A program written in Unicode REXX is a (TUTOR-flavored) *Unicode REXX program*, or a *RXU program* for short.

> There are other flavors of Unicode-enabled REXX, most notably Jean Louis Faucher's Executor, a derivative of ooREXX 4.x, and Adrian Sutherland's CRexx, a REXX inspired low-level language which is built from the ground up to support Unicode.

In the same way that REXX programs that are invoked directly by REXX are normally created with a `.rex` extension, Unicode REXX programs usually have a `.rxu` extension.

The RXU preprocessor translates a .rxu program into a .rex program. It then calls the translated program, after which it deletes it. The net result of this process is that the user is able to *write* RXU programs and to *execute* them, effectively providing an implementation of Unicode REXX.

> In the rest of this section, we will be using the terminology defined by the TUTOR package, as described in the referred document.

7.1 An example run of the `RXU` command

We will study the inner workings of the RXU preprocessor by taking a look at a sample translation.

[9]`https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-The-Unicode-Tools-Of-Rexx.pdf`

7.1.1 Creating a test file

Let us create a `test.rxu` file containing the following RXU program:

```
1      Options DefaultString Text
2      var = "🦀" || "(Lobster)"U
3      Say '"'var'" is a' StringType(var) "string of length" Length(var)
```

Line number 1 states that a string lacking a suffix (an *unsuffixed* string) shall be a `TEXT` string, i.e., a string composed of Unicode extended grapheme clusters automatically normalized to NFC (instead of a string of bytes, like in the current implementations of REXX).

Line number 2 creates a new variable, called `var`, which contains a concatenation of two strings. The first string is the *crab* emoji; since it is unsuffixed, this string will be a `TEXT` string. The second string is a *Unicode string*, specified by listing the constituent code points by name, alias, label or (hexadecimal) number — in this case, the only code point is the *lobster* emoji.

Line number 3 prints the contents of `var`, followed by its *string type*, followed by its length.

7.1.2 A test run

If we now run the RXU preprocessor against our newly created file,

```
rxu test,
```

we will get the following output

```
"🦀🦞" is a TEXT string of length 2.
```

7.1.3 The translated file

To understand what has happened (and how the RXU preprocessor works), we will run RXU again, but this time we will be using the `-keep` option, so that the translated `.rex` file is not erased after interpretation.

```
rxu -keep test.
```

Here is the contents of the `test.rex` file (translated lines are very large in this case, which has forced us to use a small font size and avoid indentation):

```
Do; !Options = DefaultString Text; Call !Options !Options; Options !Options; End
var = (!DS("🦀")) || (Bytes("🦞"))
Say (!DS('"'))var||(!DS('" is a')) StringType(var) (!DS("string of length")) !Length(var)

::Requires 'Unicode.cls'
```

Contemplation of this listing shows us a large number of instructive things about the translation process:

- The preprocessor performs a line-by-line translation, that is, the translated `.rex` file has the same line numbers as the source `.rxu` file. This is very convenient, for example, when debugging.
- The preprocessor adds a blank line, and a line containing the directive `::Requires 'Unicode.cls'`, to the end of the translated program. This ensures that the main TUTOR class, `Unicode.cls`, is loaded, in effect adding support for Unicode.
- The `Options` instruction is translated into a complex construction. The contents of the *expression* following the `Options` keyword is stored in an intermediate variable, `!Options`, and then the `!Options` routine is called with that variable as an argument. `!Options`, which resides in `Unicode.cls`, implements the Unicode-specific aspects of the `Options` instruction. Finally, the collected expression value is passed to the standard `Options` instruction: this guarantees that we can mix Unicode-oriented options and interpreter-specific options in the same instruction.
- All unsuffixed strings are enclosed in a call to the `!DS` routine. `!DS` (which stands for *Default String*) is located in `Unicode.cls`, and it implements the semantics of the `Options DefaultString` instruction. In our example, the routine will guarantee that all unsuffixed strings are `TEXT` strings.
- The Unicode string `"(Lobster)"U` is interpreted and translated by the preprocessor into its corresponding emoji, and coerced to the `BYTES` string type.
- *New* built-in functions (BIFS), like `StringType`, appear in the translated program as-is, while *existing* BIFS, like `Length`, get a `"!"` character prepended to their name: `!Length`. This allows to assign new semantics to existing BIFS.

7.2 How does the preprocessor work

We just had a glance at *what are* (some of) *the transformations* that RXU applies to translate Unicode REXX programs into standard OOREXX programs. Now we would like to understand *how are these transformations accomplished*. Since our tokenizer is not a full parser, when we are examining an item, our knowledge of its context will be, unavoidably, limited. On the other hand, the full tokenizer does indeed return a considerable amount of context, so that, by maintaining limited forms of state, we will be able to infer the syntactic category of most of the constructions we are required to process.

7.2.1 An example: translating `LENGTH()`

Let us focus, as an example, on the translation of existing built-in functions. As we have seen, they are modified by prepending a `"!"` character to their name: `Length` has to be translated to `!Length`, and so on. A naïve algorithm to implement this translation would be the following: "every time that you see a symbol that uppercases to `"LENGTH"`, prepend an exclamation mark to it". Such an approach would —erroneously— translate the name *of any variable* called `Length`:

```
l = Length(v)                 /* Should be translated     */
l = length                    /* Should NOT be translated */
```

We could try to improve our algorithm and say, "ok, let us modify the BIF name if and only if it is immediately followed by a left parentheses: this will then guarantee that it is a function call". But, in this case, we would be forgetting that there are also *methods* called `length`.

```
l = Length(v)                 /* Should be translated     */
l = v~length()                /* Should NOT be translated */
```

A still better approach would be to understand that we should be keeping some state (in our case, the last item inspected), and transform our algorithm into the following: "prepend an exclamation symbol if and only if the item is immediately followed by a left parentheses, *and, additionally*, the last item is *not* a twiddle or a double twiddle". This algorithm would almost be complete, but we would still be forgetting the (admittedly infrequent, but perfectly valid) case where the `Length()` bif is called *as a subroutine*, i.e., using the **`Call`** instruction and then taking care of the `Result` special variable (besides, there are a number of BIFS which are usually called as a subroutine).

```
Call Length v                 /* Should be translated     */
l = result
```

To handle this case, we are forced to add a second path to our algorithm, and keep, as additional state, the position of an item inside an instruction (we can do that because the tokenizer is systematically returning all the `END_-OF_CLAUSE` items to us). Then we can add to the last algorithm a second case: "*or* when the context (i.e., the item class) is a `KEYWORD_INSTRUCTION` and the subcontext (the item subclass) is `CALL_INSTRUCTION` and the called routine name is one of the recognized BIFS ".

> Have we reached a perfect algorithm? Not still. There is still a case that we are not able to handle: when there is an *internal* routine with the same name as a built-in function. In such cases, the RXU preprocessor produces the wrong result (i.e., it prepends an exclamation mark, when it should not). It would be possible to implement such discriminations using the tokenizer alone, but it would also be very costly: we would need to recognize and differentiate the diverse *code sections* (code fragments between directives), because internal labels are local to code sections. And then we would still need to apply *two passes* to our tokenizing, because a label can be used before its definition — or we should have to maintain an arbitrarily large context. We prefer to wait for the day that our tokenizer will be upgraded to a full parser: there is where such efforts belong.

7.2.2 Another example: translating strings

Another relatively complex problem is the translation of strings: we have to handle (1) the default string type (i.e., the type of unsuffixes strings); (2) the new `"Y"`, `"P"`, `"G"` and `"T"` strings, where the last three have to be validated for well-formed UTF-8; and (3) the new `"U"` strings, which can use names, aliases, labels or code points.

(1) The `Options DefaultString` setting has to be honored; this is not completely trivial, because it is a setting that may be changed at execution time. As we have already seen, we solve this problem by translating every occurrence of an unsuffixed string `"string"` to a function call `!DS("string")`, and then leaving to the `!DS` routine the task to dynamically assign the right type to the string.

```
l = "string"                /* Translated to !DS(string) */
/* The !DS routine knows how to return a string of the */
/* right type                                           */
```

Here is a simplified version of the source code for the `!DS` routine (extracted from `Unicode.cls`; the `.Unicode.DefaultString` environment variable stores the value of the `Options DefaultString` setting):

```
::Routine !DS Public
  Use Strict Arg string
  Select Case Upper(.Unicode.DefaultString)
    When "BYTES"      Then Return Bytes(string)
    When "CODEPOINTS" Then Return Codepoints(string)
    When "GRAPHEMES"  Then Return Graphemes(string)
    When "TEXT"       Then Return Text(string)
    Otherwise              Return String
  End
```

(2) On the other hand, we have to handle the full collection of new string suffixes that our flavour of Unicode Rexx defines, namely, `"Y"` (`BYTES` strings), `"P"` (`CODEPOINTS` strings), `"G"` (`GRAPHEMES` strings), `"T"` (`TEXT` strings) and `"U"` (Unicode strings). The first four of these cases can be implemented by translating the string to a function call containing the unsuffixed string, where the function name is the the corresponding built-in function: `"string"G` will be translated to `GRAPHEMES("string")`, for example; strings should also be checked for UTF-8 well-formedness.

(3) Unicode strings may require the lookup of names, aliases or labels to translate them to Unicode code points, and then the transformation of a sequence of code points into its UTF-8 encoding.

```
c = "(Duck)"U          /* Translated to c = BYTES("🦆") */
```

Translation of strings is context-dependent. For example, the string `"(Duck)"U` has to be translated to `BYTES("🦆")`, but when `"(Duck)"U` appears in certain contexts, like in the position of a label, it has to be translated to simply `"🦆"`:

```
/* A label                                          */
"(Duck)"U:                   /* Do something        */

/* If we translate to                               */
BYTES("🦆"):                 /* --> Syntax error    */
/* We should instead translate to                   */
"🦆":                        /* OK                  */
```

A single instruction may include several different contexts, so that different translation strategies may be necessary in every context. For example, in the instruction

```
Parse Arg x1 "(Duck)"U x2 ( fun("(Crab)"U) ) x3
```

the first Unicode string has to be translated to `"🦆"`, and the second to `BYTES("🦀")`.

8 Further work

The next natural step for the tokenizer is to grow and expand until it becomes a full Abstract Syntax Tree (AST) parser. A full parser will be able to assign the correct grammatical category to every token. For example, in

```
Do while = 1 to 3 While (while < 4)
  Say while
End while
```

four out of six `while`s are variables (simple `VAR_SYMBOL`s); the second one is a keyword, and the last one is a name.[10]

A full AST parser will also have an impact on several aspects of RXU, the REXX preprocessor for Unicode. For example, it will be able to know when a static function or procedure call is referring to an internal label, and, in that case, no built-in name substitution will be attempted by RXU.[11]

Generally speaking, the existence of a full AST parser will open the door to many new tools and programs. For instance, it will be almost trivial to write a powerful cross-referencer: it would reduce to traversing the AST to collect the variables in each code section, plus some sorting and some prettyprinting. The possibility to experiment with language extensions much more ambitious than the ones defined by TUTOR will also be greatly facilitated.

9 Acknowledgements

TUTOR, and the REXX tokenizer, could not have been developed without the intense debates, general creativity and overwhelming feedback of the RexxLA Architecture Review Board (ARB), for which I am deeply indebted.

I also want to extend my gratitude to Laura Blanco, Mireia Monforte, David Palau and Amalia Prats, students of my Psychoanalysis and Logic course at EPBCN, where I also teach some REXX programming, for their persistence, unwavering interest, and candid feedback.

Finally, I have to thank my colleagues at EPBCN, for being loving, caring and supportive, and for bearing with me during the long periods where I immersed myself in REXX matters, disappearing from the common world. Special thanks should go to Silvina Fernández and Olga Palomino, who have attended several essay sessions. Silvina Fernández has also taken care to operate our ElGato Stream Deck during my talks.

[10]This example is ooREXX-specific.

[11]Please refer to section 7.2.1, *An example: translating* `LENGTH()`, on page 25 for additional details.

Appendices

Appendix A Class and subclass constants used in simple tokenizing

Subclasses are listed by inserting them as nested itemizations, except when there is only one subclass that is identical to the class, in which case nothing is listed. So, for example, `BLANK` has as its only subclass `BLANK` itself, but `CONST_SYMBOL` has three subclasses, `ENVIRONMENT_SYMBOL`, `LITERAL_SYMBOL` and `PERIOD_SYMBOL`.

- `BLANK`. Whitespace, like tabs and blanks.
- `CLASSIC_COMMENT`. Infinite nesting is allowed.
- `COLON`. The `":"` character.
- `CONST_SYMBOL`. A constant symbol, like a literal symbol, a dot, or an environment symbol.
  - `ENVIRONMENT_SYMBOL`, of the form `".symbol"`.
  - `LITERAL_SYMBOL` that are not environment symbols and diferent from `"."`.
  - `PERIOD_SYMBOL`. The special case of `"."`.
- `END_OF_CLAUSE`. Semicolons and ends of lines. The tokenizer inserts a dummy `END_OF_CLAUSE(BEGIN_OF SOURCE)` clause at the beginning of the program.
  - `BEGIN_OF_SOURCE`. Dummy, inserted. Very convenient for simplification.
  - `END_OF_LINE`. Implied semicolon.
  - `INSERTED_SEMICOLON`. In simple tokenizing, this is generated only for ooRexx `RESOURCE`s, at the end of the line holding the final resource delimiter.
  - `SEMICOLON`. An explicit semicolon.
- `END_OF_SOURCE`. Signals the end of the source file. When a call to `getSimpleToken` or `getFullToken` returns `END_OF_SOURCE`, no further items should be retrieved.
- `LBRACKET`. The `"["` character.
- `LINE_COMMENT`. Up to but not including the end of line.
- `LPAREN`. The `"("` character.
- `NUMBER`. A Rexx number. This can be
  - `EXPONENTIAL`. A number with a (potentially signed) exponent.
  - `INTEGER`. An integer.

  - FRACTIONAL. A number with a decimal point, but no exponent.
- OPERATOR. An operator character. Please note that simple tokenizing does not recognize multi-character operators, like "**" or "||" (these are returned instead as multiple items, irrespective of whether they are internally separated by whitespace or comments).
- RBRACKET. The "]" character.
- RESOURCE. The resource itself, i.e., the array of lines.
- RESOURCE_DELIMITER. The end delimiter, which ends the resource.
- RESOURCE_IGNORED. Characters after ::Resource name; or ::END are ignored (this is undocumented behaviour[12]).
- RPAREN. The ")" character.
- SPECIAL. At present, only the "," character (all the other characters from ANSI 6.2.2.2 are handled separately).
- STRING. All kinds of strings, namely:
  - BINARY_STRING.
  - BYTES_STRING (Unicode-only): a string with a "Y" suffix.
  - CHARACTER_STRING, an unsuffixed string.
  - CODEPOINTS_STRING (Unicode-only), a string with a "P" suffix, checked for valid UTF-8.
  - GRAPHEMES_STRING (Unicode-only), a string with a "G" suffix, checked for valid UTF-8.
  - HEXADECIMAL_STRING.
  - TEXT_STRING (Unicode-only), a string with a "T" suffix, checked for valid UTF-8.
  - UNOTATION_STRING (Unicode-only), a string with a "U" suffix (syntax checked).
- SYNTAX_ERROR. This is the class of an item signaling a syntax error in the tokenizer program. If a call to getSimpleToken or getFullToken returns SYNTAX_ERROR, no further items should be retrieved.
- VAR_SYMBOL. It may be a variable symbol, a stem name, or a compound variable name. It may also be a keyword: please note that the simple tokenizer does not recognize any keyword, except for the ::RESOURCE construction.
  - SIMPLE_VAR.
  - STEM_VAR.
  - COMPOUND_VAR.

[12]Reported in https://sourceforge.net/p/oorexx/documentation/307/.

Appendix B Class and subclass constants used in full tokenizing

The full tokenizer builds upon the simple tokenizer. It recognizes some REXX keywords, but not all of them. In addition to the constants for item class and subclass used in simple tokenizing, the full tokenizer uses the following constants:

- `ASSIGNMENT_INSTRUCTION`. Variable assignments, not message assignments.
- `COMMAND_OR_MESSAGE_INSTRUCTION`. Cannot determine without arbitrarily large context.
- `DIRECTIVE`. All directives include and absorb the `"::"` marker.
  - `ANNOTATE_DIRECTIVE`.
  - `ATTRIBUTE_DIRECTIVE`.
  - `CLASS_DIRECTIVE`
  - `CONSTANT_DIRECTIVE`
  - `METHOD_DIRECTIVE`
  - `OPTIONS_DIRECTIVE`
  - `REQUIRES_DIRECTIVE`
  - `RESOURCE_DIRECTIVE`
  - `ROUTINE_DIRECTIVE`
- `LABEL`. Includes and absorbs the colon; it also inserts an implied semicolon.
- `KEYWORD_INSTRUCTION`. All `KEYWORD_INSTRUCTION`s include the first blank after the keyword, if present.
  - `ADDRESS_INSTRUCTION`.
  - `ARG_INSTRUCTION`.
  - `CALL_INSTRUCTION` (excluding `Call On` and `Call Off`).
  - `CALL_OFF_INSTRUCTION`. Includes the `Off` subkeyword.
  - `CALL_ON_INSTRUCTION`. Includes the `On` subkeyword.
  - `DO_INSTRUCTION`.
  - `DROP_INSTRUCTION`.
  - `ELSE_INSTRUCTION`. Inserts a semicolon after the keyword.
  - `END_INSTRUCTION`.
  - `EXIT_INSTRUCTION`.
  - `EXPOSE_INSTRUCTION`.
  - `FORWARD_INSTRUCTION`. ooREXX only.
  - `GUARD_INSTRUCTION`. ooREXX only.
  - `IF_INSTRUCTION`.

  - `INTERPRET_INSTRUCTION`.
  - `ITERATE_INSTRUCTION`.
  - `LEAVE_INSTRUCTION`.
  - `LOOP_INSTRUCTION`. ooRexx only.
  - `NOP_INSTRUCTION`.
  - `NUMERIC_INSTRUCTION`.
  - `OPTIONS_INSTRUCTION`.
  - `OTHERWISE_INSTRUCTION`. Inserts a semicolon after the keyword.
  - `PARSE_INSTRUCTION`. Includes the `CASELESS`, `LOWER` and `UPPER` keywords, if present, and adds corresponding tail values too.
  - `PROCEDURE_INSTRUCTION`.
  - `PUSH_INSTRUCTION`.
  - `PULL_INSTRUCTION`.
  - `QUEUE_INSTRUCTION`.
  - `RAISE_INSTRUCTION`. ooRexx only.
  - `REPLY_INSTRUCTION`. ooRexx only.
  - `RETURN_INSTRUCTION`.
  - `SAY_INSTRUCTION`.
  - `SELECT_INSTRUCTION`.
  - `SIGNAL_INSTRUCTION`. Excluding **`Signal On`** and **`Signal Off`**.
  - `SIGNAL_OFF_INSTRUCTION`. Includes the **`Off`** subkeyword.
  - `SIGNAL_ON_INSTRUCTION`. Includes the **`On`** subkeyword.
  - `THEN_INSTRUCTION`. Inserts a semicolon before and after the keyword.
  - `TRACE_INSTRUCTION`.
  - `UPPER_INSTRUCTION`. Regina only (but not in ANSI Rexx).
  - `USE_INSTRUCTION`. ooRexx only.
  - `WHEN_INSTRUCTION`.
- `OPERATOR`. Simple and compound operators, and extended assignment item sequences. The full tokenizing process knows how to combine adjacents items (discarding whitespace and comments) to form compound operators and other item sequences.
  - `ADDITIVE_OPERATOR` (`"+"` and `"-"`).
  - `CONCATENATION_OPERATOR` (`"||"`).
  - `COMPARISON_OPERATOR` (`"="`, `"\="`, `">"`, `"<"`, `"><"`,`"<>"`, `">="`, `"\<"`, `"<="`, `"\>”`, `"=="`, `"\=="`, `">>"`, `"<<"`, `">>=”`, `"\<<”`, `"<<="` and `"\>>"`).
  - `EXTENDED_ASSIGNMENT` (`"+="`, `"-="`, `"*="`, `"/="`, `"%="`, `"//="`, `"||="`, `"&="`, `"|="`, `"&&="`, and `"**="`).
  - `LOGICAL_OPERATOR` (`"&"`, `"|"` and `"&&"`).
  - `MESSAGE_OPERATOR` (`"~"` and `"~~"`).

– `MULTIPLICATIVE_OPERATOR` ("*", "/", "//" and "%").
– `POWER_OPERATOR` ("**").

Appendix C Resources

- A copy of this article can be downloaded from `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx.pdf`.
- The presentation slides can be downloaded from `https://www.epbcn.com/pdf/josep-maria-blasco/2024-03-04-A-Tokenizer-for-Rexx-and-ooRexx-slides.pdf`.

Modified `scripting.py` file, `class RexxLexer` fragment, for the LuaLaTeX `minted` package:

```python
class RexxLexer(RegexLexer):
    """
    Rexx is a scripting language available for a wide range of
    different platforms with its roots found on mainframe systems.
    It is popular for I/O- and data based tasks and can act as glue
    language to bind different applications together.

    .. versionadded:: 2.0

    Modified by Josep Maria Blasco <josep.maria.blasco@epbcn.com>
    to support ooRexx 5.0 and Tutor-defined Unicode extensions,
    Jan-Mar 2024.
    """
    name = 'Rexx'
    url = 'http://www.rexxinfo.org/'
    aliases = ['rexx', 'arexx']
    filenames = ['*.rexx', '*.rex', '*.rx', '*.arexx']
    mimetypes = ['text/x-rexx']
    flags = re.IGNORECASE

    tokens = {
        'root': [
            (r'\s+', Whitespace),
            (r'/\*', Comment.Multiline, 'comment'),
            (r'"', String, 'string_double'),
            (r"'", String, 'string_single'),
            (r'[0-9]+(\.[0-9]+)?(e[+-]?[0-9])?', Number),
            (r'([a-z_]\w*)(\s*)(:)(\s*)(procedure)\b',
            bygroups(Name.Function, Whitespace, Operator, Whitespace,
            Keyword.Declaration)),
            (r'([a-z_]\w*)(\s*)(:)',
            bygroups(Name.Label, Whitespace, Operator)),
            include('function'),
            include('keyword'),
            include('operator'),
            (r'[a-z_]\w*', Text),
        ],
        'function': [
            (words((
            'abbrev', 'abs', 'address', 'arg', 'b2x', 'bitand', 'bitor',
            'bitxor', 'bytes', 'c2d', 'c2x', 'c2u', 'center', 'centre',
            'changestr', 'charin', 'charout', 'chars', 'codepoints',
            'compare', 'condition', 'copies', 'd2c', 'd2x', 'datatype',
            'date', 'decode', 'delstr', 'delword', 'digits', 'encode',
            'errortext', 'form', 'format', 'fuzz', 'graphemes', 'insert',
            'lastpos', 'left', 'length', 'linein', 'lineout', 'lines',
            'lower', 'max', 'min', 'n2p', 'overlay', 'p2n', 'pos',
            'queued', 'random', 'reverse', 'right', 'sign', 'sourceline',
            'space', 'stream', 'stringtype', 'strip', 'substr', 'subword',
            'symbol', 'text', 'time', 'trace', 'translate', 'trunc',
            'u2c', 'unicode', 'upper', 'utf8', 'value', 'verify', 'word',
            'wordindex', 'wordlength', 'wordpos', 'words', 'x2b', 'x2c',
            'x2d', 'xrange'), suffix=r'(\s*)(\()'),
            bygroups(Name.Builtin, Whitespace, Operator)),
        ],
        'keyword': [
            (r'(address|arg|by|call|coercions|command|defaultstring|do|drop|'
            r'else|end|exit|expose|for|forever|if|interpret|iterate|leave|'
            r'method|nop|numeric|off|on|options|parse|procedure|pull|push|'
            r'queue|requires|resource|return|routine|say|select|signal|'
            r'strict|to|then|trace|until|use|while)\b', Keyword.Reserved),
        ],
        'operator': [
            (r'(-|//|/|\(|\)|\*\*|\*|\\<<|\\<|\\==|\\=|\\>>|\\>|\\|\||\||'
            r'&&|&|%|\+|<<=|<<|<=|<>|<|==|=|><|>=|>>=|>>|>|¬<<|¬<|¬==|¬=|'
            r'¬>>|¬>|¬|\.|,)', Operator),
        ],
        'string_double': [
            (r'[^"\n]+', String),
            (r'""', String),
            (r'"', String, '#pop'),
            (r'\n', Text, '#pop'),  # Stray linefeed also terminates strings.
        ],
        'string_single': [
            (r'[^\'\n]+', String),
            (r'\'\'', String),
            (r'\'', String, '#pop'),
            (r'\n', Text, '#pop'),  # Stray linefeed also terminates strings.
        ],
        'comment': [
            (r'[^*]+', Comment.Multiline),
            (r'\*/', Comment.Multiline, '#pop'),
            (r'\*', Comment.Multiline),
        ]
    }

    _c = lambda s: re.compile(s, re.MULTILINE)
    _ADDRESS_COMMAND_PATTERN = _c(r'^\s*address\s+command\b')
    _ADDRESS_PATTERN = _c(r'^\s*address\s+')
    _DO_WHILE_PATTERN = _c(r'^\s*do\s+while\b')
    _IF_THEN_DO_PATTERN = _c(r'^\s*if\b.+\bthen\s+do\s*$')
    _PROCEDURE_PATTERN = _c(r'^\s*([a-z_]\w*)(\s*)(:)(\s*)(procedure)\b')
    _ELSE_DO_PATTERN = _c(r'\belse\s+do\s*$')
    _PARSE_ARG_PATTERN = _c(r'^\s*parse\s+(upper\s+)?(arg|value)\b')
    PATTERNS_AND_WEIGHTS = (
        (_ADDRESS_COMMAND_PATTERN, 0.2),
        (_ADDRESS_PATTERN, 0.05),
        (_DO_WHILE_PATTERN, 0.1),
        (_ELSE_DO_PATTERN, 0.1),
        (_IF_THEN_DO_PATTERN, 0.1),
        (_PROCEDURE_PATTERN, 0.5),
        (_PARSE_ARG_PATTERN, 0.2),
    )

    def analyse_text(text):
    """
    Check for initial comment and patterns that distinguish Rexx from other
    C-like languages.
    """
    if re.search(r'/\*\**\s*rexx', text, re.IGNORECASE):
        # Header matches MVS Rexx requirements, this is certainly a Rexx
        # script.
        return 1.0
    elif text.startswith('/*'):
        # Header matches general Rexx requirements; the source code might
        # still be any language using C comments such as C++, C# or Java.
        lowerText = text.lower()
        result = sum(weight
                     for (pattern, weight) in RexxLexer.PATTERNS_AND_WEIGHTS
                     if pattern.search(lowerText)) + 0.01
        return min(result, 1.0)
```