A Tokenizer for Rexx and ooRexx

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 4^{th}, 2024

Part I

Introduction: General concepts

[Introduction] Natural and formal languages (1/3)

Natural languages have...	Programming languages have...
An *alphabet*	An *alphabet*
Lexical elements: Words, numerals, acronyms, spaces, punctuation, ...,	*Lexical* elements: Identifiers, numbers, strings, comments, whitespace, punctuation, ...,

[Introduction] Natural and formal languages (2/3)

Natural languages have...	Programming languages have...
An *alphabet*	An *alphabet*
Lexical elements	*Lexical* elements
Syntax rules Non rigid. Especially in certain contexts: Poetry Marketing News headlines Jokes ...	*Syntax* rules Rigid Not optional Can not be bended

[Introduction] Natural and formal languages (3/3)

Rigid rules: when we operate with a formal language (logic, physics, chemistry, mathematics, music, programming languages, ...) we want to be completely sure of

- *What* we are saying.
- The *meaning* of what we are saying.

That is, we want to *eliminate the ambiguity* that is inherent to natural languages, by means of clear, unambiguous, definitions of the *syntax* and the *semantics* of the formal language.

[Introduction] Lexers, tokenizers and parsers

An application that reads programs written in a certain programming language and returns the sequence of its lexical elements is called a *lexer* or a *tokenizer*.

$\Rightarrow$ *Beware*: "Token" has *two* special, different, meanings in the Rexx language.

An application that reads programs written in a certain programming language and returns a representation of its syntax tree is called a *parser*.

[Introduction] Clauses, tokens and items (1/2)

A REXX *clause*: a sequence of whitespace, comments and tokens, ended by a (in many cases implied) semicolon. A *token* may be:

- A literal string (including hexadecimal and binary strings).
- A symbol (`Chair`, `t.`, `t.i.j`, `25AB`, .Soup, ...).
- A number ($\Rightarrow$ a special form of literal string [`"-12.3"`, `"4e-2"`, ...] or symbol [`12.34`, `5E+12`, ...]).
- An operator character (`"+"`, `"-"`, `"*"`, ...).
- A special character (`":"`, `"("`, `")"`, `"."`, ...).

[Introduction] Clauses, tokens and items (2/2)

A desirable property of a lexical analyzer is to *return all the components* of a clause, including whitespace and comments, instead of only its *tokens*.

- Our tokenizer will return *all the components*, not only the tokens.
- This allows to *reconstruct the source program* by collating these components in order.

$\Rightarrow$ Our tokenizer returns *more* than only tokens.

[Introduction] "Tokenized" programs

Colloquially, one refers to a program distributed without source as *a tokenized program*. Although this denomination has stuck, it is inexact, since "tokenized" programs are indeed full abstract syntax trees, not a mere sequence of tokens.

$\Rightarrow$ In this presentation, we will use "token" in its proper sense.

[Introduction] What is a tokenizer good for?

- A *language processor* (i.e., an interpreter or a compiler) has to "understand" a program before running it. To that purpose, it has to first break it into its constituent elements.
- Other purposes: a tokenizer it is ideally suited to *introduce transformations* into the sequence of lexical elements that compose a program [Examples: a prettyprinter, a preprocessor (like RXU, see below)],
- and also to *compile data* about that sequence [Example: a cross-referencer].

Part II

Tokenizer features

[Features] The specificity of Rexx (1/4)

The syntax of Rexx is peculiar in several aspects. One of the main ideas behind its design is to make life easy for users, not for language processor implementers.

Example 1: Rexx has *no reserved words*.

```
while = 4
Do while = 1 To (while) While (while < 7)
  Say while
End while
```

$\Rightarrow$ Parsing may be more difficult that with less peculiar languages.

[Features] The specificity of Rexx (2/4)

The syntax of Rexx is peculiar in several aspects.

Example 2: The concept of token is *counterintuitive*:

- Whitespace is not a token, but, when significant, it may be an operator.
- Some basic constructs like `"**"`, `"+="` or `'::'` are not a single token but a sequence of several tokens (and may have whitespace and/or comments in between, not that it is a great idea).

```
a1 = a2 |      /* That was a */ - /* (continued) */
        | a3 /* concatenation, after all      */
```

[Features] The specificity of Rexx (3/4)

The syntax of Rexx is peculiar in several aspects.

Example 3: The concept of symbol is highly *unusual*:

- It encompasses *variable symbols* (simple,compound or stems),
- *environment symbols*,
- *constant symbols*,
- and *numbers* ($\Rightarrow$ syntax rules are bended to accommodate signs in numbers with an exponent).

More "classical" languages have *identifiers* and *numbers* (as distinct syntactical constructs), but no *constant symbols* or *environment synbols*.

[Features] The specificity of Rexx (4/4)

The syntax of Rexx is peculiar in several aspects.

Example 1: Rexx has *no reserved words*.
Example 2: The concept of token is *counterintuitive*.
Example 3: The concept of symbol is highly *unusual*.

⇒ Our tokenizer will have to take into account all these peculiarities.

[Features] Simple and full tokenizing (1/2)

Simple tokenizing: we want the sequence of tokens and separators exactly as they occur in the source file.

For example, if we tokenize "a `+=` 1", we want to get:

1. `"a"` (a variable symbol),
2. `" "` (whitespace, a blank),
3. `"+"` (an operator character),
4. `"="` (another operator character),
5. `" "` (another blank), and
6. `"1"` (an integer number symbol).

[Features] Simple and full tokenizing (2/2)

Full tokenizing: we want that some tokens are combined into higher level constructs, and that non-significant separators are discarded.

Tokenizing again "a `+= 1`", we would get:

1. `"a"` (a variable symbol, with an indication that this is the start of an [extended] assignment),
2. `"+="` (an extended assignment operator),
3. `"1"` (an integer number symbol).

[Features] Tokenizing several dialects

We want to be able to recognize several variants of REXX:

- Open Object REXX (ooRexx)
- Regina REXX
- ANSI REXX (implemented by Regina)
- ... (in the future?)

Every dialect has its own, slightly different definitions. For example, whitespace in ooRexx includes only `HT` as `other_blank_characters`, but under Regina we also accept `VT` and `FF`.

[Features] Experimental support for Unicode (1/5)

When activated, we accept five new string suffixes.

Low-level Unicode strings, `"String"U`, composed of any number of

- Blank-separated hexadecimal code points (with or without a `"U+"` or `"u+"` prefix: `"61"U == "a"`, `"u+0061"U == "a"`, `"1F680"U == "🚀"`, `"U+1F680"U == "🚀"`).
- Parenthesized names, alias or labels (`"(Rocket)"U == "🚀"`, `"(End-of-line)"U == "0A"X`, `"(<Control-000A>)"U == "0A"X`).

Names, alias and labels are case-insensitive, and they ignore blanks, dashes and underscores.

[Features] Experimental support for Unicode (2/5)

Low-level BYTES strings, "String"Y, composed of bytes.

BYTES strings are explicitly declared to be equivalent to Classic Rexx strings. The "Y" suffix is useful when unsuffixed strings have been assigned non-classical semantics.

```
Options DefaultString Codepoints
/* --> Now a string is a CODEPOINTS string by default */

a = "🐄"        /* A CODEPOINTS string, 1 code point    */
b = "🐪🦒"Y /* A BYTES string, 8 bytes                  */
```

[Features] Experimental support for Unicode (3/5)

CODEPOINTS strings, "String"P, composed of Unicode code points.

"String" has to be valid UTF-8, or a syntax error will be raised.

```
zoo = "🐒🦒🐅"P
Say Length(zoo)          /* 3 (3 code points)         */
Say zoo[2]               /* 🦒                        */
```

[Features] Experimental support for Unicode (4/5)

GRAPHEMES strings, "String"G, composed of Unicode extended grapheme clusters.

"String" has to be valid UTF-8, or a syntax error will be raised.

```
Options DefaultString Bytes
glue   = "(Zero Width Joiner)"U
family = "👨"glue"👩"glue"👦"glue"👧"G
Say Length(family)          /* 1 (1 grapheme cluster)  */
Say family                  /* 👨‍👩‍👦‍👧                      */
```

[Features] Experimental support for Unicode (5/5)

TEXT strings, "String"T, composed of Unicode extended grapheme clusters automatically normalized to NFC.

```
jose  = "Jose"T
joseacute = jose"301"U  /* "301"U is the acute accent */
Say C2X(joseacute[4])   /* 39A9 (not 65CC81)          */
Say Reverse(joseacute)  /* ésoJ                       */
```

Part III

Using the tokenizer

[Usage] Installation

- To use the tokenizer in conjunction with all the TUTOR-defined Unicode REXX features, follow the TUTOR installation instructions, and load the Unicode libraries using:

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

- If you do not need Unicode features, you can load a standalone version of the tokenizer:

```
::Requires "Rexx.Tokenizer.cls"
```

[Usage] Choosing the right tokenizer

Choose the class that represents the tokenizer variant you want to run:

- `ooRexx.Tokenizer`, for programs written in ooRexx.
- `Regina.Tokenizer`, for programs written in Regina.
- `ANSI.Rexx.Tokenizer`, for programs written in ANSI REXX.

If you need Unicode features, choose one of

- `ooRexx.Unicode.Tokenizer`,
- `Regina.Unicode.Tokenizer` or
- `ANSI.Rexx.Unicode.Tokenizer`.

[Usage] Creating a tokenizer instance

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

```
/* Assume the source program resides in a file   */
/* 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 <u>get an instance</u> of the tokenizer for this particular program source.

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

[Usage] Load the tokenizer constants

You should load the tokenizer symbolic constants contained in the tokenizer `tokenClasses` constant by using the following code fragment:

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

This will allow you to identify the token classes and subclasses returned by the tokenizer, like `END_OF_SOURCE`, `SYNTAX_ERROR`, `VAR_SYMBOL` or `ASSIGNMENT_INSTRUCTION`.

All constants have one byte values.

[Usage] Choosing simple or full tokenizing

Depending on the characteristics of your program, you may want to choose simple tokenizing (using the `getSimpleToken` method), or full tokenizing (using `getFullToken`):

```
/* Two possible reasons to exit the loop                 */
exit_conditions = END_OF_SOURCE || SYNTAX_ERROR
Do Forever
  item = tokenizer~getSimpleToken /* Or getFullToken */
/* Exit on error or end of source                        */
If Pos(item[class], exit_conditions) > 0 Then Leave
  /* ==> Do things with the itemn                        */
End
```

[Usage] Choosing detailed or undetailed tokenizing

If you have chosen to use the full tokenizer, you will also have to decide if you want to get *detailed* or *undetailed* results from your `getFullToken` method calls. You can do that when creating your tokenizer instance, by using a second, optional, argument of the `new` class method:

```
/* A second, boolean and optional, argument of   */
/* the 'new' method determines if tokenizing     */
/* will be detailed or not.                      */

tokenizer = .ooRexx.Tokenizer~new(source, .true)
```

[Usage] Returned items are REXX stems

The result of a call to `getSimpleToken` (or `getFullToken`) is a REXX stem:

```
token. = tokenizer~getSimpleToken
```

Each stem has a number of predefined indexes (we sometimes call them "properties" or "attributes"), like `token.class`, `token.subclass`, `token.location` and `token.value`. Results of full tokenizing and special tokens like `SYNTAX_ERROR` may have additional properties.

[Usage] Class and subclass

`Token.class` and `token.subclass` describe the nature of the returned token. Examples:

- `token.class == VAR_SYMBOL & token.subclass == SIMPLE_VAR`: a variable symbol which is not a stem or a compound symbol.
- `token.class == KEYWORD_INSTRUCTION & token.subclass == CALL_INSTRUCTION`: a `Call` instruction (full tokenizing only).
- `token.class == BLANK`: whitespace.
- `token.class == STRING & token.subclass == TEXT_STRING`: a TEXT string, specified with the `"T"` suffix.

[Usage] Location

`Token.location` is a string containing four integers separated by blanks which describe the location and extent of the returned token:

```
"startLine startCol endLine endCol"
```

The token starts at line `startLine`, column `startCol`, and extends until line `endLine`, column `endCol - 1`. `StartLine` and `endLine` always have the same value, except for multi-line comments and ooRexx resources.

[Usage] Value

In most cases, `token.value` is the value of the token as it appears in the source program.

Comments and ooRexx resources return a placeholder (but you can reconstruct the original token value by resorting to `token.location` and inspecting the source code).

Some few token classes return values which are interpreted. For example, hexadecimal and binary strings are converted to character strings, and Unicode strings are replaced by their `UTF-8` representations.

[Usage] Other attributes

Some few item classes return stems with additional attributes.

As we have seen, `SYNTAX_ERROR` returns a number of additional attributes to fully describe the error.

Additionally, detailed full tokenizing may return "ignored" (or "absorbed") tokens in the `token.absorbed` array (more about that below).

[Usage] Error handling (1/2)

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 of the error in the source file
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 = secondary error message
item.line     = line number where the error occurred
```

[Usage] Error handling (2/2)

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
  /* inFile is the input file name, and array contains the source */
  Say Right(line,6) "*-*" array[line]
  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
```

Part IV

Testing the tokenizer

[Testing] The InspectTokens program

InspectTokens.rex resides in the parser subdirectory.

```
C:\Unicode>InspectTokens
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

C:\Unicode>
```

[Testing] Simple tokenizing: an example

Assume that `test.rex` contains a single line, `i = i + 1`.

```
C:\Unicode>InspectTokens -simple test.rex
     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): ''
Took 0.002000 seconds.

C:\Unicode>
```

[Testing] Undetailed full tokenizing: an example

```
C:\Unicode>InspectTokens -full -nodetailed test.rex
     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): ''
Took 0.002000 seconds.

C:\Unicode>
```

Lines that have changed are marked with a 👉 emoji.

[Testing] Detailed full tokenizing: an example

```
C:\Unicode>InspectTokens -full -detailed test.rex
    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): ''
Took 0.002000 seconds.

C:\Unicode>
```

Lines that are new are marked with a 👉 emoji.

Part V

RXU, the REXX Preprocessor for Unicode

[RXU] An example run of RXU (1/3)

Let us create a `test2.rxu` file with the following content:

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

If we now run the preprocessor against this file, we will get the following output:

```
C:\Unicode>rxu test2
"🦀🦞" is a TEXT string of length 2

C:\Unicode>
```

This worked as expected!

But *how, and why*?

[RXU] An example run of RXU (2/3)

Let us now run the preprocessor with the `-keep` option: this keeps a copy of the generated `.rex` file (instead of deleting it):

```
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'
```

- A line-by-line translation
- A blank line and `::Requires 'Unicode.cls'` are added at the end of the translated program.
- The `Options` instruction gets a complex translation. [../..]

[RXU] An example run of RXU (3/3)

```
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'
```

- Unsuffixed `"string"` ⇒ `(!DS("string"))`. `!DS` implements **`Options DefaultString`**.
- `"(Lobster)"U` ⇒ `(Bytes("🦞"))`
- *New* built-in functions, like `StringType()`, appear as-is.
- *Existing* built-in functions, like `Length()`, have a `"!"` character prepended to their name.

[RXU] How does the preprocessor work? (1/3)

Example 1: Translating `Length()`.

- We should translate function and procedure calls *only*, including

```
Call Length
```

  instructions, but not variable names, method calls or internal routines.
- We can do (mose of) that with only a few symbols of context.
- (*But* we can not handle internal routines called `Length`).

[RXU] How does the preprocessor work? (2/3)

Example 2: Translating strings [1/2]. An unsuffixed string `"string"` gets translated to `(!DS("string"))`. When an **`Options DefaultString`** instruction is found, the setting is stored in `.local~Unicode.DefaultString` (default is `"TEXT"`).

```
::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
```

[RXU] How does the preprocessor work? (3/3)

Example 2: Translating strings [2/2]. `P`, `G`, and `T` strings have to be checked for `UTF-8` well-formedness, and `T` strings have to be additionally normalized to `NFC`, if needed.

The translation of a Unicode `U` string has to be enclosed in a call to `Bytes()`, but *only in certain contexts*:

```
"(Duck)"U: Say "(Duck)"U

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

Part VI

Conclusions

[Conclusions] Further work

- Evolve the tokenizer into a full abstract syntax tree parser.
- Improve RXU, the REXX preprocessor for Unicode, to take advantage of the tokenizer enhancements (for example, calls to internal functions with the same name as built-in functions will not be translated).
- Explore the development of new tools, like a cross-referencer for REXX and ooRexx.
- Possibility of new, most probably more powerful, language extensions.
- ...

[Conclusions] 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.

[Conclusions] Resources

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

[Conclusions] Questions?

Thank you!

Questions?