The Rexx Parser

36th International Rexx Language Symposium

The Wirtschaftsuniversität Vienna, Austria, May 4-7 2025

Josep Maria Blasco

jose.maria.blasco@gmail.com

EPBCN – ESPACIO PSICOANALITICO DE BARCELONA

Balmes, 32, 2º 1ª — 08007 Barcelona, Spain

May the 5th, 2025

Notice

This whole document[1] is an experiment in CSS printing. It comfortably mixes normal text and programs beautified by the heavy prettyprinting produced by the Rexx Highlighter, and it can be viewed both as a standard web page and as a slide show.

If you are viewing this file as a PDF, chances are good that you are looking at it as a slide show. If you are viewing it as a web page, the suggested print settings to produce a PDF containing a slide show are: no headers or footers, and background images active. This is true for the Chrome browser at the time of this writing (Jan-May 2025).

The default style for Rexx fenced code blocks is `dark`*. You can choose the light style by adding a* `style=light` *query string to the url of this document.*

---

1. HTML version: `https://rexx.epbcn.com/publications/2025-05-05-The-Rexx-Parser/`;
   PDF version (slides): `https://www.epbcn.com/pdf/josep-maria-blasco/2025-05-05-The-Rexx-Parser.pdf`.↩

Introduction

Introduction

Extensibility

Lists and trees: The two APIs

The Tree API

Introduction

The **Rexx Parser** is a full Abstract Syntax Tree (AST) parser for the **Open Object Rexx** (ooRexx) language.

Since ooRexx is a functional superset of Classic Rexx, it can also be used to parse **Classic Rexx programs**.

The parser includes optional support for **TUTOR-flavoured**[2] **extended Unicode Rexx programs**.

Although the Rexx Parser builds over the experience of writing **the Rexx Tokenizer**,[3] it is a completely new program, developed from scratch by **Josep Maria Blasco**[4] in 2024 and 2025.

The Rexx Parser can be **extended** by using its **module system**, and comes with a sample "print" module.

Parsed programs can be manipulated using two different, complementary APIs: the Element API and the Tree API.

---

2. See `https://rexx.epbcn.com/TUTOR/`.↩
3. See `https://rexx.epbcn.com/tokenizer/`.↩
4. `https://www.epbcn.com/equipo/josep-maria-blasco/` and `https://rexx.epbcn.com/josep-maria-blasco/`.↩

Extensibility: The module system

- **Modules** are ooRexx packages containing additional methods for the base parser classes.

```
-- Adds a "MakeArray" method to the "Do.Instruction" class
::Method "Do.Instruction::MakeArray"
/* ... */
```

- The module loader uses the `define` method of the `Class` class to add the new methods to the corresponding classes.
- A sample "print" module is provided. When loaded, many internal Parser objects become printable (→ good for debugging).
- Possible modules:
  - Expression evaluation.
  - Full interpretation.
  - Transpiling.
  - Code generation.
  - ...

Lists and trees: The two APIs

The Parser receives an Open Object Rexx source program as an argument and fully parses it, including all directives, keyword instructions and resources. The Parser then constructs **two complementary representations** of the source program:

- A full Abstract Syntax **Tree representation** of the parsed program. This representation can be accessed by navigating the tree, using a set of method calls and constants collectively known as **the Tree API**. The Tree API handles entities like the *prolog* of a program, its list of *directives*, the (optional) *code body* after a directive, *instructions*, *expressions*, etc.
- A **doubly-linked list** containing all the parsed **elements**, semantically decorated to indicate their syntactic category and other properties. This complementary representation can be accessed by a set of methods and constants collectively known as **the Element API**. The Element API handles Rexx tokens, other elements in the source program that are not considered tokens, like comments and ignorable whitespace, and other convenient markers, introduced by the Parser.

The Tree API

The **Tree API** is **experimental** at the moment of this writing, and there is little documentation about it, apart from the Parser source code.

Our intention is to **stabilize the Element API** first, before starting work on the Tree API. The following example uses methods that may not appear in the final specification.

```
parser  = .Rexx.Parser~new(name, source, options)
package      = parser~package          -- This is not a Rexx:Package, but a
                                       -- different class defined by the parser
prolog       = package~prolog          -- May be empty
body         = prolog~body             -- A Code.Body object
instructions = body~instructions       -- An array
Say instructions[2]                    -- "Say 'Hi'" (maybe)
Say instructions[2]~expression         -- "'Hi'"
```

More about the Tree API[5] in the 2026 Symposium (hopefully!).

5. See https://rexx.epbcn.com/rexx-parser/doc/guide/treeapi/↩

The Element API

- The Element API
- The Element chain
- Special elements
- Element categories
- Sets of categories, and the `<` operator
- "Taken constants"
- Element subcategories, and the `<<` operator
- Compound variables
- The 'elements' utility

The Element API

The **Element API**[6] is based on **the Element class** and its subclasses, on a set of constants defining the various **syntactical categories** an element may have, and on another set of constants defining the different variants (**"subcategories"**) of symbols which are defined to be "strings or symbols which are taken as constants".

```
If element < .EL.EXPOSED_STEM_VARIABLE Then Do
  -- Things to do when "element" is an exposed stem variable
End
/* ... */
If element << .METHOD.NAME Then Do
  -- "element" is a method name (a string or a symbol)
End
```

Please note that the `<` and `<<` methods have been overloaded to simplify querying for the category and subcategory of an element.

6. See https://rexx.epbcn.com/rexx-parser/doc/guide/elementapi/.↩

The Element chain

Elements are stored in a doubly-linked list, the **element chain**. The head of the chain is returned by the `firstElement` method call of a `Rexx.Parser` instance. The `next` and `prev` methods of an element instance are used to navigate the chain; they both return `.Nil` at the extremes of the chain.

```
element = parser~firstElement
Do While element \== .Nil
  -- Do something with "element"...
  element = element~next
End
```

- Elements may be standard Rexx tokens, and other code elements which are not tokens, like comments or non-significant blanks, or
- Tokens inserted by the Rexx parsing rules, like semicolons at the end of the line, or before and after the `THEN` keyword, or
- Other **special elements**, described below.

Special elements (1/2)

The Rexx Parser inserts a number of **special elements** in the element chain. These elements help to ensure that the element chain has a number of convenient properties.

- The Parser inserts a **dummy end-of-clause marker** at the beginning of the stream. This guarantees that each clause is delimited by two end-of-clause (`"EOC"`) markers.

```
<EOC, element₁, ..., elementₙ, EOC>
      └─────── clause ───────┘
```

- The Parser inserts a **dummy `EXIT` instruction** at the end of each code body. This ensures that all code bodies contain at least an instruction.

```
<EOC, Implicit-EXIT, EOC>
└── minimal code body ──┘
```

Special elements (2/2)

- The implicit `EXIT` instruction also provides a convenient anchor point for comments after the last instruction and labels found at the end of a code body.

```
-- This comment is attached to the SAY instruction
Say "Bye!"
-- This comment is attached to the implicit exit instruction
a_label_here: -- This label points to the implicit EXIT instruction
::Requires "Some.program"
```

- The parser inserts an additional pseudo-instruction after the last implicit `EXIT` instruction, the **end-of-source marker**. This ends the element stream. As all instructions, it is flanked by end-of-clause markers, so that all streams end with the sequence

```
<EOC, Implicit-EXIT, EOC, End-Of-Source, EOC>
└──────── end of an element stream ─────────┘
```

Element categories (1/2)

Every *element* has a ***category*** method, which returns a one-byte value that describes its *syntactic category*.

```
category = element~category
If category == .EL.COLON Then Do
  /*  Things to do when 'element' is a colon */
End
```

Categories are assigned from a set of more than 100 categories, described exhaustively in the Globals.cls package. The Parser provides a set of global environment constants, to aid in the symbolic manipulation of categories. The names of all these constants share a `.EL.` (for "ELement") prefix.

```
.EL.OP.CASCADING_MESSAGE                -- Identifies the "~~" operator
.EL.RIGHT_BRACKET                       -- Identifies the right bracket "]"
```

Element categories (2/2)

Categories are very detailed and fine-grained. For example, there is a distinct category for every one of the extended assignment character sequences, and for every simple or compound operator:

```
-- Some extended assignment categories
.EL.ASG.PLUS                      -- "+="
.EL.ASG.MINUS                     -- "-="
.EL.ASG.MULTIPLY                  -- "*="
-- ...
-- Some operator categories
.EL.OP.PLUS                       -- Infix addition
.EL.OP.PREFIX.PLUS                -- Prefix "+"
.EL.OP.GREATER_THAN               -- ">" comparison
.EL.OP.REFERENCE.GREATER_THAN     -- ">" reference operator
.EL.OP.CONCATENATION              -- A compound operator
```

Compound elements may include intervening whitespace and comments. In these cases, the Parser assigns the right category to the first character, and marks the rest as ignorable.

Sets of categories, and the `<` operator

Since categories are very numerous, it is convenient, in many cases, to manage them collectively, as **sets**. Many sets are defined by the Parser; they have symbolic names which start with `.ALL.`. As an example, here is the definition of the `.ALL.SYMBOLS` set, extracted from `Globals.cls`:

```
Call NewSet ALL.SYMBOLS, .ALL.VAR_SYMBOLS, .ALL.CONST_SYMBOLS, .ALL.NUMBERS
```

Since categories are one-byte values, sets can be conveniently represented as byte strings, so that checking for set membership becomes simple and efficient, as it reduces to a simple `contains` method call. As a convenience, the Parser overloads the `<` operator of the *element* class to work with both categories and sets.

```
element < category                       -- means "element~category = category"
element < set                            -- means "set~contains(element~category)"
```

The code of the Rexx Parser makes heavy use of this special notation.

"Taken constants"

In many places in the syntactic definition of Rexx and ooRexx, we find tokens that are defined to be **"literal strings or symbols taken as a constant"**, or some equivalent expression. Examples are the routine name after a `CALL` instruction keyword; labels; method names; etc. These tokens received the unfortunate name `taken_constant` in the ANSI standard (6.3.2.22), and this denomination, for lack of a better one, has stuck.

A taken constant which is a symbol has to be parsed differently than a standard symbol. For example, even if the syntactical form of the symbol is that of a compound variable, no tail variable substitution takes place.

```
  Say  stem.with.a.large.tail.55AA.12   -- A compound variable
  Call stem.with.a.large.tail.55AA.12   -- An internal routine name
  Exit

stem.with.a.large.tail.55AA.12:         -- An (admittedly bizarre) routine name
  /* ... */
```

Element subcategories, and the `<<` operator

All *taken constants* have a category of `.EL.TAKEN_CONSTANT`. These elements, and only these, have an additional method, *subcategory*, which retrieves their **subcategory**. Similarly to categories, subcategories are one-byte values identified by special environment variables which, in this case, share a `.NAME` *suffix* (some special subcategory names end with `.VALUE` instead).

```
.METHOD.NAME                        --
```

Similarly to `<`, the Parser overloads the `<<` operator of the *element* class.

```
If element << .RESOURCE.NAME Then Do
  -- Means
  --   If element~category    == .EL.TAKEN_CONSTANT, -
  --      element~subcategory == .RESOURCE.NAME      Then Do...
  /* [Things to do when "element" is a ::RESOURCE name] */
End
```

Compound variables (1/2)

Compound variables are a unique feature of Rexx. They have a dual nature: they are, at the same time, *variables* and *terms* (i.e., expressions). In some cases, we may be interested in handling them as atomic variables, and in some other cases, we may be interested in handling each of their components separately.

The Rexx Parser addresses this duality by tagging every compound variable with a category of `.EL.COMPOUND_VARIABLE` (or `.EL.EXPOSED_COMPOUND_VARIABLE` if the variable has been exposed), and simultaneously allowing to retrieve its components by using the *parts* instance method, which returns an array of components.

The first element of the array is always the stem name, that is, it is of class `.EL.STEM_VARIABLE` or `.EL.EXPOSED_STEM_VARIABLE`, and it includes the first dot in the compound variable name. The rest of the components are a sequence of either simple variables, of class `.EL.SIMPLE_VARIABLE` or `.EL.EXPOSED_SIMPLE_VARIABLE`; signless integers, of class `.EL.INTEGER_NUMBER`; pure dotless constant symbols, of class `.EL.SYMBOL_LITERAL`; or separator dots, of class `.EL.TAIL_SEPARATOR`.

Compound variables (2/2)

As an example, the following code fragment defines an exposed variable, and shows the contents of the returned *parts* array.

```
::Method aMethod
  -- Define an exposed variable
  Expose exp
  stem.var.exp.12.2a
  -- Invoking the 'parts' method will return an array containing:
  -- [1]: an EL.STEM_VARIABLE (i.e., "stem."; please note that the first
  --      period is always part of the stem name.
  -- [2]: an EL.SIMPLE_VARIABLE, "var".
  -- [3]: an EL.TAIL_SEPARATOR, namely the second period, ".".
  -- [4]: an EL.EXPOSED_SIMPLE_VARIABLE, "exp".
  -- [5]: another EL.TAIL_SEPARATOR.
  -- [6]: an EL.INTEGER_NUMBER, "12".
  -- [7]: another EL.TAIL_SEPARATOR.
  -- [8]: an EL.SYMBOL_LITERAL, "2a".
```

Please note that, since exposed compound variable detection is based on a static analysis of the source program, it may produce incorrect results in certain cases.

The 'elements' utility

The `elements` utility displays the whole element chain, including compound variable details.

```
C:\rexx-parser\samples>elements hi.rex
elements.rex run on 26 Apr 2025 at 15:53:45

Examining hi.rex...

Elements marked '>' are inserted by the parser.
Elements marked 'X' are ignorable.
Elements marked 'A' have isAssigned=1.
Compound symbol components are distinguished with a '->' mark.

[   from  :    to   ] >XA 'value' (class)
 --------- ---------  --- --------------------------
[    1   1:    1   1] >   ';' (an EL.END_OF_CLAUSE)
[    1   1:    1   4]     'SAY' (an EL.KEYWORD)
[    1   4:    1   5]  X  ' ' (an EL.WHITESPACE)
[    1   5:    1   9]     'Hi' (an EL.STRING)
[    1   9:    1   9] >   ';' (an EL.END_OF_CLAUSE)
[    1   9:    1   9] >   '' (an EL.IMPLICIT_EXIT)
[    1   9:    1   9] >   ';' (an EL.END_OF_CLAUSE)
[    1   9:    1   9] >   '' (an EL.END_OF_SOURCE)
[    1   9:    1   9] >   ';' (an EL.END_OF_CLAUSE)
Total: 9 elements and 0 compound symbol elements examined.
```

Error handling

Parser errors and program errors

Early checks

The rxCheck utility

Error handling

The Parser may encounter two different kind of errors:

- **Internal errors**, that is, errors present in the Parser itself, and
- **Parsed program errors**, that is, errors present in the source program that is being parsed.

Both kinds of errors have to be reported, and they have to be reported differently.

When the parsed program contains a **parse-time syntax error**, the parsing process is unconditionally terminated, and the Rexx Parser informs the caller by producing a specially-crafted 98.900 ("Execution error") syntax error. The `ADDITIONAL` array returned as part of the condition object contains the original syntax error code, the offending line number, and also the original `ADDITIONAL` array used to raise the error, which contains all the error message substitution instances.

By default, the Rexx Parser attempts to mimic the behaviour of the Open Object Rexx interpreter, by detecting exactly the same errors in the same circumstances. The current distribution contains more than 600 tests that compare the errors produced by the Parser and by ooRexx and guarantee that they are the same.

Early checks (1/3)

When requested, **the Parser is also able to detect, at parse time, some errors that ooRexx only detects at execution time**.

`SIGNAL` instructions.

When a non-calculated `SIGNAL` instruction if found (that is, one that is not using the `VALUE` option), the label name is inspected and compared to all the labels present in the current code body. If the label is not found, a `SYNTAX` error 16.1 is raised ().

```
debug = 0
If debug Then SIGNAL Next      -- ==> SYNTAX 16.1: Label "NEXT" not found.
-- Do something
Exit

Naxt: -- Typo!
```

All early checks are **optional**.

Early checks (2/3): GUARD instructions

`GUARD` instructions are only allowed inside a method call, but the ooRexx interpreter only detects these errors at execution time.

The Rexx Parser can optionally detect incorrect `GUARD` instructions at parse time. When a code body is not a method body (that is, when it is either the prolog or a routine body) and a `GUARD` instruction is found,

```
Say "It works in ooRexx"
Exit

-- The following instruction is illegal a prolog, irrespective
-- of the fact that it will never be executed.

Guard On                      -- ==> SYNTAX 99.911:  GUARD can only be issued in an object method invocation.
```

Early checks (3/3): Built-in functions

Some of the checks done against BIFs (please note that the following example progam is synthetic: the Parser would stop immediately after the first error).

```
/* Maximum number of arguments                              */
LENGTH( a, b   )    -- ==> 40.4:   Too many arguments in invocation of LENGTH; maximum expected is 1.
/* Minimum number of arguments                              */
COPIES(        )    -- ==> 40.3:   Not enough arguments in invocation of COPIES; minimum expected is 2.
/* Missing required arguments                               */
LEFT(    , 2   )    -- ==> 40.5:   Missing argument in invocation of LEFT; argument 1 is required.
/* Arguments that have to be whole numbers                  */
LEFT(   a, 2.5 )    -- ==> 40.12:  LEFT argument 2 must be a whole number; found "2.5".
/* Whole numbers that have to be positive (or non-negative) */
SUBSTR( a, 0   )    -- ==> 93.924: Invalid position argument specified; found "0".
/* Closed choice literals                                   */
STRIP(  a, "X" )    -- ==> 93.915: Method option must be one of "BLT"; found "X"
                    --            (--> a ooRexx bug: there is no method involved here).
/*                            ...                           */
```

Please refer to `https://rexx.epbcn.com/rexx-parser/doc/ref/classes/rexx.parser/early-check/` for detailed information.

The rxCheck utility

To parse and check a program file or single line of Rexx code, use:

```
rxcheck [options] ( filename | -e "rexx code" )
```

By default, all early check options are enabled, but you can individually disable all of them.

```
C:\rexx-parser>rxcheck -e "Exit; Say Left(a,b,c,d)"
Syntax error 40.004 at line 1. Additional: 1: 'LEFT', 2: '3':
     1 *-* Exit; Say Left(a,b,c,d)
Error 40 running INSTORE line 1:  Incorrect call to routine.
Error 40.4:  Too many arguments in invocation of LEFT; maximum expected is 3.
```

Additional options:

- `emptyassignments`: allow empty assignments (i.e., instructions like `a =`, meaning `a = ""`).
- `extraletters`: specify a list of characters that are to be considered letters by the tokenizer. Please note that this can work for `"@"`, `"#"` and `"$"`, which are ASCII characters, but not for `"¢"`, which is Unicode `"C2A2"U`, since the Parser does not support multi-byte characters.

Further work

Further work

- Stabilize the Element API, after a period of public discussion.
- Work towards a first public version of the Tree API (to be presented in the 2026 Symposium).
- Improve and extend the optional early, parse-time, detection of errors that are only detected at run-time by ooRexx.
- Explore the possibilities associated with parse-time constant expressions, for example, dead code elimination.
- Add optional support for other Rexx dialects, like Regina Rexx, BREXX, Jean Louis Faucher's Executor, etc.
- Explore some form of transpiling, or even Rexx code generation (i.e., Rexx implemented in Rexx), by creating the appropriate optional modules. Maybe create a toy implementation of Classic Rexx for the Rexx Architecture Review Board (full ooRexx is probably too ambitious, but maybe a limited subset could be implemented).
- Play with language extensions.
- ...

Acknowledgements

Acknowledgements

Jean Louis Faucher has integrated TUTOR into ooRexxShell, and Rony Flatscher has included TUTOR and the Rexx Parser in the net-oo-rexx distribution.

Some preliminary versions of the Rexx Parser and its accompanying subproject, the Rexx Highlighter, have been tested by different members of the RexxLA community, including, but not limited to Gilbert Barmwater, Jean Louis Faucher, Rony Flatscher, Ruurd-Jan Idenburg, René Jansen and Till Winkler; I want to thank them all for their observations and enhancement proposals.

I also want to thank my colleagues at EPBCN, Laura Blanco, Silvina Fernández, Mar Martín, David Palau, Olga Palomino and Amalia Prat, who have read several drafts of this presentation and helped to improve it with their comments and suggestions.

Questions?

References

References

The Rexx Parser can be downloaded at:

- `https://rexx.epbcn.com/rexx-parser/` (preferred: better Rexx highlighting)
- `https://github.com/JosepMariaBlasco/rexx-parser`

TUTOR can be downloaded at:

- `https://rexx.epbcn.com/TUTOR/` (preferred: better Rexx highlighting)
- `https://github.com/JosepMariaBlasco/TUTOR`

Executor can be downloaded at:

- `https://github.com/jlfaucher/executor`

The net-oo-rexx bundle can be downloaded at:

- `https://wi.wu.ac.at/rgf/rexx/tmp/net-oo-rexx-packages/`