Concepts that Allow Learning the Programming Language Rexx Much Faster than Other Languages

Flatscher, Rony G.; Winkler, Till

Published in:
MIPRO 2024 Proceedings

Published: 01/01/2024

Document Version
Publisher's PDF, also known as Version of record

Concepts that Allow Learning the Programming Language Rexx Much Faster than Other Languages

Rony G. Flatscher *, Till Winkler *
* Vienna University of Economics and Business, Vienna, Austria
rony.flatscher@wu.ac.at

Abstract — **The open object Rexx (ooRexx) programming language has been successfully used to teach business administration students programming from scratch in a single semester. Students are able to interact with Windows and Microsoft Office, but also to use Java classes for programming OpenOffice/LibreOffice and JavaFX (GUIs) on all modern operating systems. A key factor in this is the design of the Rexx programming language, on which ooRexx is based. This article introduces the most important concepts of the language, which make it possible to learn programming faster than with other languages. In particular, concepts such as single data type, decimal arithmetic, instruction types and others are introduced and demonstrated with the help of examples.**

Keywords – teaching programming; bachelor students; Rexx; ooRexx

I. INTRODUCTION

In the course of the last 35 years, a course for business administration students at the Vienna University of Economics and Business has been developed to teach programming to novices in a single semester. Over time, various programming languages have been used for this purpose, originally Microsoft Visual Basic (also VBScript), which is considered an easy-to-learn programming language. After getting to know a fairly simple programming language called Rexx on IBM mainframes in the 1980s, the main author experimented with a PC version of Rexx for one semester and was surprised at how much faster essential programming concepts could be taught. Rexx was designed at IBM [5] with the motivation to create a "human-oriented" programming language that was easier to learn and more easily understandable by humans than other languages existing at the time. At that time, Rexx was extremely successful and known in the industry – Amiga OS used Rexx as its scripting language [1] and many companies created and sold Rexx interpreters. Rexx even became an international standard via an ANSI/INCITS [10] standard of the language.

In the 1990s IBM created an object-oriented successor to Rexx, called open object Rexx (ooRexx), which in 2005 became open-source [9] and has become available for all major operating systems. On Windows ooRexx allows to directly take advantage of COM/OLE, making it possible to interact with Windows and Microsoft Office directly with ooRexx messages. This enables students to access the functions implemented in these products, for example the visualization of data in Microsoft Excel. In order to be able to use the programming skills acquired on other platforms such as Linux or MacOS, an ooRexx Java bridge [4] was created that disguises Java as ooRexx. This bridge makes it possible to use Java classes and Java objects by sending them ooRexx messages.

During "Business Programming 1" (first two month), students learn basic programming, fundamentals of object-oriented programming and gain the knowledge necessary to use COM/OLE in Windows [2]. The following two months, during "Business Programming 2", are dedicated to the Java bridge [4] and include the development of platform independent GUI applications using JavaFX [3]. Flatscher and Müller [8] provide an overview of the curricula of these two combined lectures (effectively four months, four hours per week, 8 ECTS, i.e. a total teaching load of 200 hours). In addition, seminar papers, bachelor's and master's theses by students are freely accessible [12]. At the end of the course, students are able to use an open-source Java class library, such as ASF's PDFBox for PDF processing [13], and demonstrate its use by means of ooRexx nutshell examples [14], thereby further extending the course.

While established methods such as pair programming, the presentation of short examples and the completion of student projects are used throughout the course, it is the programming language and its concepts that seem to reduce the cognitive burden for novices [15].

This article focuses on the concepts of the Rexx programming language that are responsible for making it so easy and quick for business students to learn programming. Since examples are sometimes worth a thousand words, the article will give short code examples so that readers can evaluate the main Rexx concepts.

II. REXX PROGRAMMING CONCEPTS

The Rexx programming language got devised as a "human centric" language by its designer, Mike F. Cowlishaw and became an IBM product for mainframes in 1979, more than 40 years ago [16]. At that time, C was not widely used, which accounts for many of today's programming language designs, so some of the Rexx concepts may seem unfamiliar at first.

A. *Datatype*

The Rexx language has a sole datatype, a string value. The content of a string cannot (and will not) be changed it is therefore immutable like in many programming languages. Everything is a string including numbers. Expressions yield string values. Arithmetics are possible if the involved string values represent numbers.

Different strings can be concatenated by listing them in an expression one after the other delimiting them with blanks. Strings can be referred to by variables or can be explicitly given within quotes (double or single quotes).

Since people sometimes use multiple spaces to align strings to a specific column to make them easier to read in code, Rexx by default reduces multiple spaces between strings to a single space before concatenating, treating the space as a string on its own. If this default behaviour (concatenation with intervening blank) is not desired, then one can either place a quoted string with a variable without interleaving blank right next to each other or the concatenation operator needs to be used explicitly: || (two vertical bars) which will allow for concatenating strings without intervening blanks. Fig. 1 demonstrates using strings.

```
1   say "hello world"          /* output: hello world          */
2   say "hello"    'world'     /* output: hello world          */
3   a="one"                    /* a string                     */
4   b=2                        /* a string                     */
5   c="3"                      /* a string                     */
6   say a   b      c           /* output: one 2 3              */
7   say a   b || c             /* output: one 23               */
8   say a   b  +   c           /* output: one 5                */
9   say a   ": hello!"         /* output: one : hello!         */
10  say a": hello!"            /* output: one: hello!          */
11  say a || ": hello!"        /* output: one: hello!          */
```

Figure 1. *Everything is a string*

B. *Arithmetics*

Arithmetics get carried out in decimal (and not binary) form so that numbers get used exactly as supplied and the results are the same as if carried out manually by humans. The decimal arithmetics of Rexx [6] as defined in the ANSI X3.274–1996 standard was used as the basis to define the IEEE 754 and ISO/IEC/IEEE 60559:2011 decimal arithmetics standards which were then used to implement decimal arithmetics in many other languages like C, C++, Java, JavaScript, Python and more. By default arithmetics get carried out with nine significant digits but it is possible to explicitly use an arbitrary number of digits that should be used for carrying out arithmetic operations. Fig. 2 demonstrates arithmetics carried out in Rexx.

```
1   say 0.9/10                 /* gives: 0.09                  */
2   say 1 - 0.9/10             /* gives: 0.91                  */
3   say 4**2                   /* gives: 16                    */
4   say 1 + 2 * 3/4            /* gives: 2.5                   */
5   say 1 + 2*3/4**2           /* gives: 1.375                 */
6   say 1/17                   /* gives: 0.0588235294          */
7   numeric digits 22          /* now use 22 digits            */
8   say 1/17                   /* gives: 0.05882352941176470588235 */
```

Figure 2. *Arithmetics*

C. *Instructions*

The ANSI Rexx standard defines three instruction types: *assignment* instruction, *keyword* instruction and *command* instruction:

- An *assignment* instruction consists of a variable name an assignment operator (the equal sign =) and an expression that yields a string that gets assigned. The assignment *a=3*4* would evaluate the expression (a multiplication) on the right hand side and assign the result *12* to the variable *a*.
- A *keyword* instruction starts with a keyword that is an English word that conveys the purpose of the instruction. This makes Rexx programs readable as if the code was formulated in pseudo code that can be understood even by people who have no knowledge of programming. Examples are: *address*, *if*, *call*, *do*, *loop*, *parse*, *say*, …
- A *command* instruction is a string that does not start with a keyword, it can be an enquoted string, a variable, or an expression that evaluates to a string. By default the command will be given to the operating system to be carried out as if it was directly typed into a command line window using the keyboard. Upon return from the command the return code will be immediately made available using a variable named RC. It is possible to explicitly address command handlers that should carry out such a command.

Fig. 3 gives examples for each type of instruction. It includes the *if* keyword instruction with a dependent *then* and an *else* keyword instruction, each on a separate line. The *if* keyword instruction with its branches can be formatted differently to accommodate the programmer's preference. As can be seen in Fig. 3 (line 15f) indenting does not change the semantics of an instruction as is the case in Python.

```
1     /* an assignment instruction:                                  */
2   a="hello world"   /* assigns "hello world" to a variable named a */
3     /* a keyword instruction:                                      */
4   say a               /* output: hello world                       */
5     /* a command instruction:                                      */
6     /* a Windows command (could be typed into a command line)      */
7   "dir a.txt"         /* command: list the file a.txt              */
8     /* variable RC contains the command's return code, 0 is success */
9   if rc=0             /* keyword instruction: if                   */
10  then                /* keyword instruction: then                 */
11  say "found!"        /* keyword instruction: say                  */
12  else                /* keyword instruction: else                 */
13  say "some problem ocurred, rc="rc      /* show return code       */
14    /*  differently formatted                                       */
15  if rc=0 then say "found!"
16     else say "some problem ocurred, rc="rc  /* show return code   */
```

Figure 3. *Assignment, keyword and command instructions*

Fig. 4 depicts a Python program that does the same as the Rexx programs in Fig. 3. There is no keyword instruction for outputting strings, so Python's built-in function *print()* gets used instead. Python has no instruction type for commands. As the sample command is one directed at the operating system one needs to import a Python module capable of submitting commands to the operating system, in this case the module (an advanced concept) named *subprocess*. Its method *run()* allows for

submitting the command and fetching a result object that carries information about the executed *subprocess* such as the *returncode* attribute which is of the strict type *int*. The keyword instructions *if* and *else* must have a colon appended, followed by the statement that should be carried out. If formatting the program such that the statement for the two branches go into separate lines they must be indented (forcing a Python block). Note that it is mandatory to explicitly turn the return code's int value into a string object using the built-in function *str()* if concatenation is desired.

```
# an assignment instruction
a="hello world"    # assigns "hello world" to a variable named a
# no keyword instruction for output, using built-in function print()
print(a)
# no command instruction using module subprocess instead
import subprocess  # import subprocess module
# execute command
completedProcess=subprocess.run("dir a.txt", shell=True) # run command
rc=completedProcess.returncode  # fetch return code, an int
if rc==0:
    print("found!") # indentation mandatory (forcing a block)
else:
    print("some problem occurred, rc="+str(rc)) # turn rc into a string
```

Figure 4. *Assignment, output and issuing a command in Python*

Unlike with Rexx issuing commands and getting to their return code and concatenating it to a string for output is quite complicated and the amount of explanations (and time) needed for novice people being introduced to programming is prohibitively high.

D. Blocks

Blocks of instructions in Rexx get defined with the *do* keyword instruction and encompasses all instructions until the matching *end* keyword instruction. In addition blocks can be defined with the *loop* and *select* keyword instructions and get closed with a matching *end* instruction.

Fig. 5 demonstrates a program with three blocks, the comment at the bottom shows the output that gets produced by it. Note that any indentations are meant for better legibility but removing them would not change the semantics of the program. Due to the meaningful English keywords of Rexx the code becomes self-descriptive and can be read and understood as if it was pseudo code.

Contrast this to the equivalent Python example of Fig. 6, where the reader needs to know quite a few technical Python details before being able to fully understand what the code does. Think e.g. about the pecularities of the *range()* built-in function and the *match* switch statement as well as the importance of correct indentation which determines the boundaries of blocks and how they get nested.

```
max=5              /* number of repetitions   */
loop a=1 to max    /* loop block              */
   select          /* nested block # 1        */
      when a=1 then say a": first round"
      when a=2 then say a": second round"
      when a=3 then say a": third round"
      otherwise say "(a="a")"
   end
   if a=max then
   do              /* nested block # 2        */
      say "-> a=max"
      say "-> last round!"
      say "-> loop will end"
   end
end

/* output of the above program will be:
  1: first round
  2: second round
  3: third round
  (a=4)
  (a=5)
  -> a=max
  -> last round!
  -> loop will end */
```

Figure 5. *A Rexx program with three blocks, two of which are nested in the loop block*

```
max=5      # number of repetitions
for a in range(1,max+1): # loop with range() function, must add 1 to max
    match a:   # must be indented, "match" needs Python 3.10 or higher
        case 1: print(str(a)+": first round")      # nested block # 1
        case 2: print(str(a)+": second round")  # nested block # 1
        case 3: print(str(a)+": third round")    # nested block # 1
        case _: print("(a="+str(a)+")")           # default, nested block # 1
if a==max:  # must be indented
        print("-> a==max")        # nested block # 2
        print("-> last round!")     # nested block # 2
        print("-> loop will end")  # nested block # 2
```

Figure 6. *A Python program with three blocks, two of which are nested in the for block*

E. Built-in Functions (BIFs)

Rexx defines about 80 functions that are built into the language among them many string functions from *abbrev()* that tests whether one string is the abbreviation of another one, to *words()* which counts and returns the number of words in a string. There are functions that allow for interacting with streams (files) and a number of miscellaneous functions that supply useful functionality like the *arg()* function that returns the number of arguments by default, but can also be used to fetch arguments by position or test whether an argument got supplied or not at a certain position.

Many of the built-in functions allow for omitting arguments in which case the most important functionality gets excercised, e.g. the *date()* function will return the current date as a string in the form of "30 Nov 2059" or if invoked with the argument "standard", e.g. as *date('s')*, will return the string with the sortable date in form of "*20591130*" (if invoked on November 30th 2059). All of the built-in functions have been devised such that arguments that control options do not need to be spelled out but could be abbreviated to their first letter allowing the programmer to save typing (and therefore minimizing the chances of mistyping options).

Over the course of 40 years the number of built-in functions supported by the Rexx language was kept stable which keeps the language small and easy to use. To allow for extending Rexx with additional functions a standard

API got defined that allows for creating and using external Rexx function libraries. Such external function libraries are usually organized around the functionality of a specific domain e.g. for access to relational database management systems [11]. So if additional functionality is needed it gets only added on a per program basis. The external functions do not become part of the language such that the number of built-in functions does not increase over time. Compare this to e.g. Visual Basic, which has gained a wealth of built-in functions over its lifetime, so much so that the number of built-in functions available could no longer be kept track of by VB programmers, effectively losing control as the language evolved.

F. *Subroutines and Functions*

A label followed by a sequence of instructions up to a return keyword instruction defines an internal subroutine or function that can be invoked by a *call* keyword instruction or a function reference. In the latter case the return keyword instruction must supply a return value. Argument lists are used to supply data to the routine which can refer to them using the *arg()* built-in function or apply the *parse* or *use* keyword instructions. Similarly external Rexx programs can be called or invoked as functions.

G. *Case Independency*

Unlike many programming languages (including Python) the case of symbols used in Rexx instructions is irrelevant, such that dog and *DOG* are regarded to represent the same symbol. The Rexx interpreter will uppercase all characters outside of quoted strings before executing the instruction. Fig. 7 displays the uppercased version of the instruction that the Rexx interpreter will execute in the block comment next to each instruction. This concept makes it easy for beginners to write programs as they do not have to pay attention to the case of symbols they use. After all, a human who reads the words dog and *DOG* would regard both words to convey the same meaning.

```
1  a="hello world"   /* becomes: A="hello world"          */
2  A=a  "..."        /* becomes: A=A "..."               */
3  say a "/" A       /* becomes: SAY A "/" A             */
4  /* outputs: "hello world / hello world"               */
```

Figure 7. *All unquoted characters get uppercased*

H. *Variables*

Rexx being a dynamically typed language does not allow to strictly type variables. Therefore one does not declare variables, but rather defines them implicitly with assignment instructions or in the context of the parse and use keyword instructions.

Variables can be symbols that start with an alphabetic letter, underscore, exclamation or question mark, followed by the same characters and additionally, numbers and dot(s). If a variable name contains dots, all variables that have the same "stem" (the characters up to and including the first dot) are called "compound variables" which can be used for representing associative arrays.

Fig. 8 demonstrates how to define four variables, three of which are compound variables with the stem *fn.* that have a whole number as their "tail" (the characters following the dot).

```
1  a_b = "hey"       /* assign a value to a variable                    */
2  say a_b cde       /* outputs: hey CDE                                */
3  fn.1="max.txt"    /* stem: "FN.", tail: "1"                          */
4  fn.2="moritz.txt"/* stem: "FN.", tail: "2"                           */
5  fn.0=2            /* stem: "FN.", tail: "0"                          */
6  loop i=1 to fn.0  /* loops two times                                 */
7     say i":" fn.i  /* outputs: "1: max.txt" and "2: moritz.txt"       */
8  end /* loops two times                                               */
9  /* output of the above program will be:
10    hey CDE
11    1: max.txt
12    2: moritz.txt  */
```

Figure 8. *Variables in Rexx*

The *SAY* keyword instruction in line 2 outputs the result of the expression that blank concatenates the value of variable *A_B* with the symbol *CDE* that could be used as a variable, but has no assigned value. Therefore the output will be the resulting string "*hey CDE*".

The compound variables in Fig. 8 consist of the stem *FN.* and a tail with a whole number that serves as an index into a conceptual array which stores the number of elements in the tail with the value *0* (the value 2), the value "*max.txt*" with the tail 1 and "*moritz.txt*" with the tail 2. The loop (lines 6 through 8) uses a loop variable named *i* to loop from *1* to the value of the variable *FN.0*, which is *2*. In the loop the variable *i* gets resolved in the tail of the respective compound variable, which in the first run will resolve to variable *FN.1* and in the concluding second loop to variable *FN.2*. The output of the loop is shown in lines 11 through 12 in Fig. 8.

I. *Parsing Strings*

The *parse* keyword instruction makes it easy to parse text using a parse template and assign the parsed strings directly to variables. The parse template may define parsing by blanks, by strings, by column positions and/or applying relative lengths in the parsing process. Fig. 9 demonstrates how to parse text delimited by one or more blanks and assign the parsed values to the variables defined in the parse template.

```
1  text = " John    Doe   Vienna Austria"
2  parse var text fn ln city country
3  say "fn:" fn "ln:" ln "city:" city
4  /* outputs: "fn: John ln: Doe city: Vienna"                  */
5  text = "Mary Doe Tokyo Japan"
6  parse var text fn ln city .  /* ignore country */
7  say "fn="fn "ln="ln "city="city
8  /* outputs: "fn=Mary ln=Doe city=Tokyo"                      */
```

Figure 9. *Parsing strings into variables*

As can be seen the code almost reads like pseudo code and can be understood even if one does not know the Rexx programming language.

Fig. 10 depicts a Python version that demonstrates how one could use the split method to parse a blank delimited string into a list and assign specific elements to variables. The technique of line 11 that allows to assign individual list elements to variables in a single statement probably needs students who have acquired already quite a lot knowledge

of Python, taking advantage of regular expressions would need even more skilled persons.

```
text  = " John     Doe   Vienna Austria"
words = text.split()   # create list of words
fn    = words[0]       # assign to variable
ln    = words[1]       # assign to variable
city  = words[2]       # assign to variable
print("fn:",fn,"ln:",ln,"city:",city)
# outputs: “fn: John ln: Doe city: Vienna”
text  = "Mary Doe Tokyo Japan"
words = text.split()   # create list of words
# assign multiple elements in a single statement
fn, ln, city = [words[i] for i in (0, 1, 2)]
print("fn="+fn, "ln="+ln, "city="+city)
# outputs: “fn=Mary ln=Doe city=Tokyo”
```

Figure 10. *A Python program for parsing strings into variables*

J. Filter Programs

The Rexx *say* keyword instruction allows to output strings to the system's standard output file *stdout*. The Rexx *parse pull* keyword instruction allows for reading and parsing strings from the system's standard input file *stdin*. Therefore it is easy to write filter programs in Rexx that can be used as stages in pipes. By means of redirection it becomes already possible to read and write files in Rexx.

The program in Fig. 11 reads and parses data from *stdin* (by default the keyboard) and writes people from "Vienna" to *stdout* (by default the screen) in a loop. If an empty line gets parsed (if the user presses the <enter> key or there is no more input from the redirected input file) then the loop is ended.

```
loop until firstName=""   /* if empty we are at end of file              */
   parse pull firstName lastName city country  /* parse from stdin */
   if city="Vienna" then say city":" lastName"," firstName
end
```

Figure 11. *A simple filter program "filter.rex" in Rexx*

Given the content of the input file in Fig. 12 and the command in Fig. 13, then the resulting output file will have the content as depicted in Fig. 14.

```
John    Doe   Vienna Austria
Mary  Doe Tokyo   Japan
 Anthony   Xavier   Vienna Michigan
N. N. ? ?
```

Figure 12. *Content of "input.txt"*

```
C:\work>filter.rex < input.txt > output.txt
```

Figure 13. *Command to run filter.rex with stdin and stdout*

```
Vienna: Doe, John
Vienna: Xavier, Anthony
```

Figure 14. *Content of "output.txt"*

III. CONCLUSION

In the course of developing and improving a lecture for introducing programming to business administration students over many years, the Rexx (and later its successor ooRexx) programming language has been standing out with respect of making it easy and quick to learn programming. It is considered to be the most important success factor that allows students to learn programming quickly such that after two months they become able to program Windows and Windows applications like MS Office.

This article attempts to briefly and informally introduce and demonstrate in form of nutshell examples the most important concepts. This way the reader can assess the concepts and compare the resulting programs. As Python has become a very popular programming language that is regarded to be easy, a few nutshell examples are realized in Python making a direct comparisons become possible.

It should be possible to estimate that for beginners the knowledge required to become productive in Rexx is manageable. In addition, the syntax of the programming language is decisive for the fact that students of business administration can acquire more programming knowledge in one semester than in other programming languages.

ACKNOWLEDGMENT

The authors wish to thank DI Walter Pachl for his feedback and proofreading.

REFERENCES

[1] ARexx. "Amiga Rexx". Wiki.Amigaos.net. Accessed: November 17, 2022. [Online]. Available: March 18, 2024. Available: https://wiki.amigaos.net/wiki/AmigaOS_Manual:_ARexx

[2] R. G. Flatscher, “Introduction to Programming with ooRexx and BSF4ooRexx 1. 1-7.” [PDF slides]. Accessed: March 18, 2024. Available: https://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/

[3] R. G. Flatscher, “Introduction to Programming with ooRexx and BSF4ooRexx 2. 8-14.” [PDF slides]. Accessed: March 18, 2024. Available: https://wi.wu.ac.at/rgf/wu/lehre/autojava/material/foils/

[4] BSF4ooRexx. “BSF4ooRexx.” Sourceforge.net. Accessed: March 18, 2024. [Online]. Available: https://sourceforge.net/projects/bsf4oorexx/

[5] M. F. Cowlishaw, “The design of the REXX language.” in *IBM Systems Journal 23 (4)*. 1984, pp. 326-335.

[6] M. F. Cowlishaw, “General Decimal Arithmetic.” Accessed: March 18, 2024. [Online]. Available: https://speleotrove.com/decimal/

[7] R. G. Flatscher, R.G. “Resurrecting REXX, Introducing Object Rexx,” in *ECOOP*. Nantes, France, 2006. Available: https://wi.wu.ac.at/rgf/rexx/misc/ecoop06/ECOOP2006_RDL_Workshop_Flatscher_Paper.pdf

[8] R. G. Flatscher and G. Müller, “'Business Programming' – Critical Factors from Zero to Portable GUI Programming in Four Hours,” in *6th BEE-Conference*, Plitvice Lakes, Croatia, 2021, pp. 76-82.

[9] ooRexx. “ooRexx (Open Object Rexx).” Sourceforge.net. Accessed: March 18, 2024. [Online]. Available: https://sourceforge.net/projects/oorexx/

[10] RexxLA, “ANSI X3.274-1996 Information Technology - Programming Language REXX,” RexxLa.org, Accessed: March 18, 2024. [Online]. Available: https://www.rexxla.org/rexxlang/standards/

[11] Rexx/SQL, “Rexx/SQL.” RexxSql.Sourceforge.net. Accessed: March 18, 2024. [Online]. Available: https://rexxsql.sourceforge.net/

[12] WU, “Selected Seminar, Diploma, Bachelor and Master Theses.” Accessed: March 18, 2024. [Online]. Available: https://wi.wu.ac.at/rgf/diplomarbeiten/

[13] Apache PDFBox, “A Java PDF Library.” Accessed: March 18, 2024. [Online]. Available: https://pdfbox.apache.org/

[14] T. Wang, “An Introduction to Apache PDFBox Library: Nutshell Examples.” Thesis. Accessed: March 18, 2024. Available: https://wi.wu.ac.at/rgf/diplomarbeiten/#bakk_202304_01

[15] T. Winkler and R. G. Flatscher, “Cognitive Load in Programming Education: Easing the Burden on Beginners with REXX.” *In Central European Conference on Information and Intelligent Systems.* 2023, pp. 171-178

[16] M. F. Cowlishaw, *The REXX Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.