Cognitive Load in Programming Education: Easing the Burden on Beginners with REXX

Till Winkler, Rony G. Flatscher
Vienna University of Economics and Business
Institut für Wirtschaftsinformatik und Gesellschaft
Welthandelsplatz 1, 1020 Wien, Austria
{till.winkler, rony.flatscher}@wu.ac.at

Abstract. *To learn and teach programming is very difficult, often leads to poor results, and causes many students to drop out or turn away from the subject. Cognitive load theory can help to understand the challenges students face, improve programming education, and select an appropriate language for instruction. In this paper, we take a theoretical look at programming education and, in particular, language characteristics that reduce students' cognitive load and thus enable rapid learning and frustration-free productivity. We introduce the REXX language and some of its favorable characteristics that make it possible to teach novices programming within a single semester. In this limited time, students are empowered to program Microsoft products (Windows, Office), address the command line, grasp the basics of object-oriented programming, use Java classes, and create portable graphical user interfaces (GUIs) with JavaFX.*

Keywords. Programming education, Cognitive load theory, Human-oriented programming, REXX, ooRexx

1 Introduction

It has always been a topic of discussion as to which programming language should be taught. With Wikipedia listing 691 (“List of programming languages,” 2023) different programming languages, such a choice can be overwhelming. Moreover, when beginners ask an expert, they usually get answers based on individual preferences, which can be confusing. It can be equally confusing to rely on popularity ratings of programming languages. Until the mid-1980s, the most popular choices were Fortran, Pascal, or Ada; in the 1990s it was clearly C; in the late 2000s it was Java; and today it is, with a 28.4% certainty, Python (“Data is Beautiful,” 2019; PYPL, 2023). The fact that language popularity scores nowadays are often calculated based on the frequency of online searches, such as for tutorials, should not be ignored (PYPL, 2023). Since we believe that it is not the amount of support needed that matters, but the ease with which a language can be learned, we would like to present our experience with using REXX in programming classes. Specifically, we will discuss how language characteristic can place an unnecessary cognitive burden on students. In addition, we will give illustrative examples of REXX, a language we consider particularly suitable for teaching.

Learning a programming language can be challenging for beginners, as they need to grasp the syntax, semantic and language-specific concepts such as variables, data types, arithmetic, and others (Sands, 2019; Stachel et al., 2013). Moreover, students must quickly apply new knowledge to solve complex and often novel problems. It is well known that it is a combination of students' lack of experience, understanding new concepts, applying syntactic and semantic rules, and solving new complex problems that can be overwhelming (Sands, 2019). Programming courses are generally considered difficult, with high dropout rates and poor outcomes; some students cannot program loops even after several semesters (Robins et al., 2003). Many programming educators find that students achieve poor grades or, more importantly, become disillusioned with programming (Garner, 2002). This is in contrast to what we observe in REXX teaching. According to the course evaluation (WS22/23), 83.3% of students would definitely recommend the class to others and consider the demands to be reasonable (66.6%) or slightly taxing (33.3%).

The second author became acquainted with REXX on IBM mainframes in the 1980s and developed an experimental course using the PC version of REXX. To his surprise, it was possible to teach programming concepts much faster compared to VBScript, a language considered easy to learn at the time. REXX was developed at IBM (Cowlishaw, 1987) with the motivation of creating a "human-oriented" language—by keeping it small—that is easy to learn, code, remember, and maintain (Fosdick, 2005). At that time REXX was extremely successful; Amiga OS used it as a script language (“AmigaOS Manual: Arexx,” 2023) and several companies developed interpreters. Today, REXX is still an integral part of IBM mainframes and has been also formalized as an ANSI/INCITS X3.274 standard (ANSI, 1996). In the 1990s, IBM developed an object-oriented successor to REXX called ooREXX, which is open source and has been available for all major systems since 2005 (ooRexx, 2023). Under Windows, ooREXX allows the direct use of COM/OLE, which enables direct interaction with many Microsoft products. For applying the acquired skills on other platforms, a Java bridge called BSF4ooREXX is available, which disguises Java as ooREXX and allows for the use of Java classes (BSF4ooRexx, 2023).

Over the past 35 years, what was once an experimental course has evolved into two programming courses[1] that teach students the necessary skills to solve complex programming tasks in a single semester (Flatscher & Müller, 2021). Within the first two months, during "Business Programming 1" (BP1), students learn basic programming concepts, fundamentals of object-oriented programming and everything necessary to use COM/OLE in Windows (BP-1, 2023; Flatscher & Müller, 2021). The following two months, during "Business Programming 2" (BP2), are dedicated to the Java bridge (BSF4ooRexx, 2023) and include the use of Java classes including the development of platform independent GUI applications with JavaFX (BP-2, 2023; Flatscher & Müller, 2021). Some students are even so motivated that they write seminar papers, bachelor's and master's theses that go far beyond what they originally learned (WU, 2023). We believe that this learning outcome and motivation is primarily related to language characteristics of REXX that reduce students' cognitive load and minimize frustration. Before looking more closely at specific language characteristics, we will introduce the perspective of cognitive load theory on learning, problem solving and programming education.

2 Cognitive Load Theory

Human expertise and problem-solving skills, are based on knowledge stored as so-called schemata in our long-term memory (Sweller & Van Merriënboer, 2005; Garner, 2002). A schema might be anything that can be treated as a single element; for instance, a word, a mathematical formula, or a particular programming concept (Garner, 2002). During learning, multiple new or previously disconnected pieces of information are bundled together into a single, more complex element or schema (Paas et al., 2003). For example, it is almost impossible to remember all the digits of a telephone number individually unless bundled into more complex elements, such as, country code (two digits), area code (four digits) and the remainder as three-digit blocks ("141" instead of "1 - 4 – 1"). In this way, a twelve element/digit number can be remembered easily. The same basic principle applies to any kind of learning, including physics, mathematics, spoken languages and programming. The general goal of teaching is to enable the construction of increasingly complex schemata and to facilitate their automation through practice (Paas et al., 2003). A practiced stick driver has automated the procedure of shifting gears to a point where he or she no longer needs to think about it, whereas a novice driver requires active processing of each step, which can be tiring and frustrating. Similarly, a skilled programmer can easily create a "selection block", while a beginner must actively think about the necessary structure, syntax, variables and boolean symbols.

More complex schemata can only be built if the brain is actively involved in the learning process, for which free working memory capacities are needed (Sweller & Van Merriënboer, 2005). In other words, students must actively think about new programming concepts. Unlike the nearly unbound long-term memory, however, our working memory can only deal with up to four elements or schemata at a time (Sweller & Van Merriënboer, 2005). People who are able to handle complicated programming tasks do not think more sophisticatedly or process more elements, but already have complex schemata that are treated as a single element. In comparison, an inexperienced programmer must process many different details (elements) in his limited working memory. When details or new information overwhelm the capacity of working memory, problem-solving performance and learning success decline (Sweller, 1988). During programming education in particular, as in any other problem-solving area, the cognitive load on novices must, therefore, be carefully managed (Garner, 2002; Paas et al., 2003). There are three different types of cognitive load that essentially fight for the limited resources of working memory. The intrinsic cognitive load is caused by the learning content itself: the programming language with its individual and interacting elements (Sands, 2019). Extraneous cognitive load is a burden on top of the content, that may be caused by information search or inappropriate teaching methods (Sweller & Van Merriënboer, 2005). Intrinsic and extraneous cognitive load can add up to such an extent that there is no capacity left for germane cognitive load. Germane cognitive load is necessary for learning through thinking about new information and concepts (Paas et al., 2003). The chosen programming language and teaching methods must facilitate the construction of schemata without overwhelming limited cognitive capacities (Garner, 2002).

[1] In the spirit of open education, the course material is freely available (see BP-1, 2023; BP-2, 2023).

2.1 Intrinsic Cognitive Load

It is especially the degree of interactivity between novel elements that can produce high intrinsic cognitive load (Garner, 2002). Simply learning vocabulary, for example, produces a relatively low burden because each word—an individual element—can be learned separately (Garner, 2002). Learning grammar, on the other hand, produces more intrinsic cognitive load because the words in a sentence are connected and their interactivity must be taken into account (Garner, 2002). In this sense, learning a programming language is an extremely high cognitive burden, since abstract concepts and the syntax, so to speak, and the grammar of a language has to be learned (Sands, 2019; Stachel et al., 2013). It is often assumed that the intrinsic cognitive load caused by the learning content cannot be reduced (Sands, 2019; Garner, 2002). This is not entirely true for programming, as we can choose a language with fewer abstract concepts and a simpler syntax, which is an advantage that other fields do not have.

Many educators—here referring to C and VisualBasic.NET—see "...the excessive amount of class time spent on teaching the language syntax…" (Al-Imamy et al., 2006, p. 280) as a major issue. While a C-style syntax has influenced languages such as Java, PHP, Go, or Swift, it is challenging for beginners (Denny et al., 2011; Stefik & Siebert, 2013). Since learning syntax is a common challenge, tools and languages have been developed to bypass syntax altogether, but even so, general-purpose programming languages are still predominantly used in classrooms (Stefik & Siebert, 2013).

Some syntactic choices made by language designers are perceived as easier to understand because they are more similar to knowledge or schemata from other domains (Stefik & Siebert, 2013). Unfortunately, most languages require consideration of unnecessary elements and interactions that students take time to learn. In Java, for example, a keyword specifying the data type is required before the name of a variable is stated in the declaration (Sands, 2019). While an experienced Java programmer doesn't have to think about it—having automated schemata—this can be a burden for beginners. In general, strictly typed variable declarations pose a major challenge for beginners, with dynamically typed languages being perceived as more intuitive (Stefik & Siebert, 2013). From a cognitive load perspective, omitting such declarations reduces the number of elements and interactions a novice must consider in working memory, thereby freeing up capacity.

While experienced programmers are already familiar with abstract characteristics of programming languages, beginners tend to find aspects that are not literal or are rooted neither in English or mathematics difficult to understand (Stefik & Siebert, 2013). For instance, novices are able to use statements like repeat ten times more accurately than traditional C-style looping syntax (Stefik & Siebert, 2013). The word repeat, or loop is simply more common in English and can be understood literally, as opposed to for. Moreover, the use of a single equal sign is perceived by beginners as easier to grasp than that of a double equal sign (Stefik & Siebert, 2013). The meaning of a single equal sign is a schema developed in mathematics education, while a double equal sign is rather uncommon. An intuitive language should be designed so that prior, non-programming knowledge can be applied as expected (McIver & Conway, 1996). When choosing a language to teach, we need to put ourselves in the beginner's shoes and recognize how many new aspects are necessary to understand, and what existing schemata from other fields can be useful. This is recognized by many teachers who choose a language primarily for pedagogical reasons rather than popularity or industry relevance (Mason et al., 2012). In terms of students' future and cognitive load theory, it makes sense to focus on a general-purpose language because it is much easier to transfer schemata from one general-purpose language to another. It is simply not possible to predict which language will be popular or desired when students start working in the industry. Therefore, we must enable students to master one language without frustrating them, as they can easily switch to another language later if needed.

Learning success is of course also strongly influenced by the teaching methods, which should aim to keep the extraneous cognitive load as low as possible in order to free up cognitive processing resources for learning (Sweller & Van Merriënboer, 2005).

2.2 Extraneous Cognitive Load

Extraneous cognitive load is an additional burden that is not required for learning (germane load) and is not directly related to the content (intrinsic load) (Sands, 2019). Such a cognitive burden can, for instance, be caused by redundant, unnecessary or too frequently expressed information (redundancy effect) (Sweller & Van Merriënboer, 2005). Additionally, extraneous cognitive load may be imposed by lengthy web searches for information needed to complete a task, or by spreading relevant information across multiple lessons, textbooks or reference manuals (locations or times) (Sands, 2019). Various methods such as pair programming or presenting worked or nutshell examples are known to reduce cognitive burden. During pair-programming learners can split minimally demanding tasks—typing, navigation, or file management—and highly demanding tasks—syntax development or solution search—among themselves, thereby reducing the cognitive burden (Sands, 2019). Presenting practical—worked or nutshell—examples where students are shown a solution step-by-step from start to finish, helps to break down a complex and novel problem into meaningful steps and provide scaffolding for other problems (Sands, 2019; Stachel et al., 2013).

While extraneous cognitive load can mostly be reduced by suitable teaching methods and materials, part of the load is also caused by the programming language. For example, when a language forces to use an unfamiliar operating system or a complex IDE, basic tasks such as typing, file management, or navigation become an unnecessary additional cognitive burden.

2.3 Germane Cognitive Load

To learn, novices must actively invest free cognitive resources; that is, an appropriate germane cognitive load should be elicited (Sweller & Van Merriënboer, 2005). Freeing up working memory capacity by reducing intrinsic and extraneous cognitive load is only effective if students are motivated to actively invest cognitive effort in schemata construction (Paas et al., 2003). For this, it is important that students are not frustrated with the teacher, grading, or programming in general. In many classes, a common problem is that students use a passive elaboration strategy; they do not use free capacities to self-elaborate new concepts (Sweller & Van Merriënboer, 2005). Common teaching methods to promote active elaboration in class are to have students annotate worked examples or complete missing code from a well-structured program (Garner, 2002).

It is well-known that practicing programming concepts in variable situations has a positive effect on schemata building and educational transfer (Paas et al., 2003). In order to have enough time to practice, the chosen programming language should have characteristics that are easy to understand and therefore do not take up unnecessary time in class.

3 Language Characteristics

REXX coding can be achieved with a simple editor (e.g., Notepad, gedit) or more complex IDEs (e.g., IntelliJ). While gedit is equipped with REXX syntax highlighting by default, Intellij requires a readily available plugin (Seik, 2023). This allows students to select a tool with which they are most familiar, thereby reducing extraneous cognitive load. While for most languages a simple editor is sufficient, for Python it is advisable to use an IDE, as it is necessary to create intended blocks or include and manage packages for basic functions.

REXX was developed with the goal of creating a "human-oriented" language that is small, as well as being easy to learn, code, remember, and maintain (Fosdick, 2005). In the limited time available for teaching, large languages such as C++ or Java can only be taught by focusing on a subset of the entire language, and intentionally ignoring important aspects (McIver & Conway, 1996). This can be confusing because textbooks or online tutorials rarely adhere to the same subset, and beginners may encounter features that were intentionally not taught. In comparison, REXX is a small but powerful language that can be taught in a short time. All necessary knowledge is bundled in a single reference manual (ooRexx, 2023). This eliminates the tedious search for information and thereby reduces extraneous cognitive load (Sands, 2019). In addition, the reference manual itself provides brief and meaningful explanations, syntax diagrams and nutshell examples. Figure 1 shows a syntax diagram for the Strip method; such diagrams are used for all methods and functions in the reference manual (ooRexx, 2023). With its multimodal presentation (description and visualization) of key knowledge and its nutshell examples, the manual does a good job of reducing unnecessary cognitive load (cf. Sands, 2019; Stachel et al., 2013).

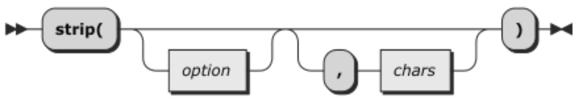

Figure 1. Syntax diagram (ooRexx, 2023, p. 206)

Many believe that making and correcting mistakes is the best way to learn. However, the inadequacy of error messages is a problem that dates back to COBOL, but is still a problematic in C++ or Java (Becker et al., 2016). Errors and related messages of a compiler or interpreter should be understandable without knowing technical jargon (McIver & Conway, 1996). Unfortunately, error messages are often "… terse, confusing, too numerous, misleading, and sometimes seemingly wrong...", this way "...they become a source of frustration and discouragement" (Becker et al., 2016, p. 21). We consider the error messages of the ooRexx interpreter to be clear and, above all, precisely pointing to the source of an error. Most importantly, REXX's free-form syntax, its case-insensitive nature, and its use of dynamic data types avoid many common errors from the outset. Avoiding such errors helps reduce frustration or disillusionment with programming, which can motivate students to invest the cognitive load required to construct schemata and automate these through practicing (Garner, 2002; Paas et al., 2003).

3.1 Free-form Syntax

REXX has a free-form syntax where the positioning of the code is irrelevant. By default, the interpreter merges multiple blanks into a single one before execution. If this behavior is not desired, quotation marks (" or ') can be placed directly next to each other or two vertical bars can be used directly as concentration operators (||). Strings in REXX can be merged by listing them one after another in a single expression and delimiting them with blanks. A string encapsulated by quotation marks is not changed. Figure 2 provides examples of this.

In teaching, the free form of REXX allows for the creation of readable, consistent, and intentionally eye-catching syntax that helps to convey new concepts to novices who typically have difficulty grasping the signals of novel concepts (McIver & Conway, 1996).

```
say  "Hello World!"            /* output: Hello World!    */
say "  This"   'is'   "REXX!"  /* output:   This is REXX! */
say "Good""bye" || '!'         /* output: Goodbye!        */
```

Figure 2. Free-from syntax and string merge

Such flexibility is also important for learning success, since fewer syntactic rules, details, or elements are relevant, and this therefore imposes a lower intrinsic cognitive load. In contrast to this flexibility, spaces or indentations have a semantic meaning in Python (e.g., for conditional statements). Although it was probably a good intention to eliminate grouping constructs, which reduces the number of elements (e.g., parentheses, do, end, …) and enforces structure, students do not seem to be able to master the concept of consistent indentation (McIver & Conway, 1996). This Python characteristic is contrary to non-programming knowledge that novices typically have, and can be considered to be a case of excessive cleverness (McIver & Conway, 1996). A text written in a natural language is understandable even with random indentations, and this is how beginners implicitly expect a programming language to behave. REXX's free-form syntax ensures that this reasonable expectation is met. Such violations of non-programming expectations, as committed by Python, are probably the "worst pedagogical sin" a programming language can commit (McIver & Conway, 1996, p. 4).

3.2 Case-insensitivity

Unlike in most programming languages, the case of symbols used in REXX is irrelevant. It does not "bother" the interpreter whether a beginner writes *do*, *Do*, *dO* or *DO* by mistake or on purpose. The REXX interpreter will uppercase all characters outside of quoted strings before executing them. This applies equally to all aspects of the language, including variable names, statements, functions, methods, method options, and so on. Figure 3 provides an example of this. While the strip method (see Figure 1) removes leading and trailing characters by default this behavior can be changed by an option, and a character can also be specified to replace blanks. For example, if you write “*Leading”*, “*leading*”, “*LeaDing*”, “*l*” or “*L*”, which all give the same result, only the leading blanks will be removed. This example shows that in addition to being case-insensitive, an option can also be spelled out, which makes its effect literally understandable. Such literal comprehensibility, further reduces the amount of learning (intrinsic cognitive load) for novices. A Python beginner, on the other hand, must first learn the meaning of *strip()*, *rstrip()*, or *lsrtip()* and build up the schema that an "l" here is an abbreviation for “*leading*”. While it is obviously clever to use abbreviations, forcing such behavior is another case of excessive cleverness.

```
a = "  This"   'is'   "REXX!"   /* a merged string         */
Say A                           /* output:   This is REXX! */
SAY a~Strip("LEADing")          /* output: This is REXX!   */
say A~strip("l")                /* output: This is REXX!   */
```

Figure 3. Case-insensitivity

Considering variables, a Python novice must be careful when naming or referring to these, because a single case difference makes them distinct; ***Oranges*** and ***oranges*** in this case are in fact two different things (variables). Such a distinction between cases is an additional element or rule that novices must learn, which unnecessarily increases the intrinsic cognitive load and may lead to frustrating syntax errors. Case dependence also violates the expectation of natural language schemata that an Orange remains an orange regardless of its case.

If someone new to a natural language makes a grammatical error—analogous to a syntax error—he or she can still accomplish the intended task of communication if the other person has a basic level of generosity and flexibility. However, a typical compiler or interpreter is by no means generous or flexible, but will mercilessly reject any slight deviation. This can be frustrating for students because they cannot achieve their goal of creating an executable program. The free-form and case-insensitive nature of REXX makes the interpreter more generous and flexible, and allows students to write a form of pseudo-code without frustration.

3.3 Data Type and Arithmetic

The REXX language has a single data type, a string value, which is immutable. Arithmetic is possible if the string contains numbers. The REXX interpreter defines the datatype implicitly with assignment or in the context of instructions. Compared to strictly typed languages, this eases the intrinsic cognitive load on students (Stefik & Siebert, 2013). When assigning variables, REXX students must consider fewer elements (e.g., no declaration of integer, float, ...) and their interaction with the rest of the program. It is not necessary to think about the required calculation precision in advance, as is the case in mathematics classes.

REXX's arithmetic, defined in ANSI/INCITS X3.274, formed the basis for the definition of ISO/IEC/IEEE standards that have been used to implement decimal arithmetic in languages such as Java, Python, and others (ANSI, 1996; Cowlishaw, 2022). By default, nine significant digits are used for the calculation, but this precision can be adjusted if desired. In REXX, variable names can start with a letter, an underscore, an exclamation mark or a question mark, followed by the same set of symbols and additional numbers and dots. All variables that contain a dot become compound variables, which can be used to represent associative arrays. In this way, associative arrays can be declared without much effort and can be used like a typical variable. An example of this can be seen in Figure 4.

```
1  var = 6 * 7             /* assign and evaluate 6 * 7            */
2  say var                 /* output: 42                           */
3  stem.1 = 4              /* assign 4 to compound variable        */
4  say var - stem.1 / 0.7  /* output: 36.2857143                   */
5  numeric digits 20       /* now use 22 digit precision           */
6  say var - stem.1 / 0.7 /* output: 36.285714285714285714         */
```

Figure 4. Basic arithmetic and stem variables

3.4 Instructions

The ANSI/INCITS REXX standard defines assignment, keyword and command as three distinct instruction types (ANSI, 1996). An assignment instruction in REXX consists of a variable name, a single equal sign (=) as assignment operator, and an expression that contains the string that is assigned. The assignment *"var = 6 * 7"* would evaluate the expression (a multiplication) and assign the result 42 to the variable var (line 1 in Figure 4).

A keyword statement begins with a keyword; for example, *address*, *say*, *if*, *call*, *do*, *loop*, *parse* and others. Note that these keywords reflect their meaning in literal English. In this way, students can further draw on the schemata they have acquired in English classes. For illustration, Figure 5 shows a REXX program and Figure 6 shows a Python program with the same functionality.

```
1    /* an assignment instruction:                                */
2  a = "Hello World!"           /* assigns "Hello World!" to a    */
3    /* an assignment instruction:                                */
4  say a                        /* output: Hello World!           */
5    /* an command instruction:                                   */
6  "dir a.txt"                  /* command: list the file a.txt   */
7   /* variable RC contains the command's return code             */
8  if rc = 0 then say "found!" /* 0 means success                 */
9      else say "some problem occurred, rc="rc   /* shows rc      */
```

Figure 5. Instructions in REXX

A quoted string, including a variable or an expression evaluated as a string, is recognized by the REXX interpreter as a command instruction (line 6 in Figure 5). By default, the command is executed as if it were typed in a command line. The return code is made available immediately via the *rc* variable (line 8 in Figure 5). This feature made REXX popular on mainframes as it facilitates addressing the operating system, editors and utilities. If experience with the command line is available, solutions can be found with the existing system functions even without great programming knowledge.

Figure 5 contains the *if* keyword instruction with a dependent then and an else keyword instruction (line 8f in Figure 5). Depending on the programmer's preference, these instructions can be on separate lines. The indentation here is a preference decision and does not change the semantics of an instruction. In comparison, indentions in Python (see Figure 6) have semantic meaning and are mandatory, which limits flexibility and dictates programmer preferences. To understand or even write the Python program in Figure 6, many more details must be considered. For example, a module called *subprocess* must be imported (line 6 in Figure 6), its *run()* method called to submit the command to the system (line 8 in Figure 6), and the strictly int-typed return code fetched (line 9 in Figure 6). It should also be noted that two equal signs (==) represent an equality and one sign (=) represents an assignment operator (line 10 in Figure 6). Also, the built-in function *str()* must be known if concentration is desired (line 11 in Figure 6). Only if the students then also manage to put the colons (:) and the indentation correctly do they achieve a working program.

```
1  # an assignment instruction
2  a="Hello Word!"  # assigns "Hello World!" to a
3  # no keyword instruction, using built-in function()
4  print(a)
5  # no command instruction, using module subprocess instead
6  import subprocess
7  # execute command
8  completedProcess=subprocess.run("dir a.txt", shell=True)
9  rc=completedProcess.returncode  # fetch return code, an int
10 if rc==0:
11     print("found!")                 # indentation mandatory
12 else:
13     print("some problem occurred, rc="+str(rc)) # to string
```

Figure 6. Instructions in Python

The amount of time required to explain all the necessary Python concepts in class before students can productively write such a program is enormous. This is not only problematic given the limited time in class, but also puts a strain on student cognitive capacity and motivation. From the perspective of cognitive load theory, a much greater intrinsic cognitive load is generated, straining the limited resources of working memory for the necessary germane cognitive load.

3.5 Built-in and External Functions

REXX defines about 80 built-in functions, the number of which has been kept stable over the last 40 years. Even though the number of built-in functions may seem limited, they are powerful and more than enough to be productive. For example, functions that require additional packages in Python, such as root calculations (e.g., *sqrt()*), are already integrated. REXX can be extended with external functional libraries using the *::requires* directive. Such libraries are easy to write and are usually organized around domain-specific functions and are only included on a per-program basis.

As with the strip method (see Figures 1 and Figure 3), most built-in functions have a default behavior that can be changed by options. The *date()* function, for instance, returns the date as a “6 Mar 2023” string, while *date(‘s’)* returns the string suitable for sorting as “20230306”. In teaching, we consider it helpful that built-in functions and methods work without specifying options, so that you can use them from the beginning without worrying about details.

Mastering Java syntax is often seen as a major obstacle even for good students (Denny et al., 2011). The BSF4ooREXX Java bridge (BSF4ooRexx, 2023) enables students to use these functions without having to deal with the demanding Java syntax. Figure 7 shows a simple example, where BSF4ooREXX is included as if it were an external function (line 6 in Figure 7) and the *javax.swing.JFrame* class is invoked and the message show sent to it (line 1 in Figure 7).

```
1  frame=.bsf~new("javax.swing.JFrame", "Hello, my beloved
     world - from ooRexx!")
2  frame~setSize(410,20)   /* set width and height        */
3  frame~visible=.true     /* make JFrame visible         */
4  call SysSleep 10        /* sleep for ten seconds       */
5
6  ::requires "BSF.CLS"    /* get access to Java bridge   */
```

Figure 7. Invoke javax.swing.JFrame class

The result is a user interface frame titled “Hello, my beloved world - from ooRexx!”. The output can be seen in Figure 8, which shows how easy it can be to create GUI programs for any modern operating system. The ease with which external functions can be written and included, and the simplicity of how the operating system can be addressed and COM/OLE objects or Java classes can be used, makes REXX more than "just" a language for beginners.

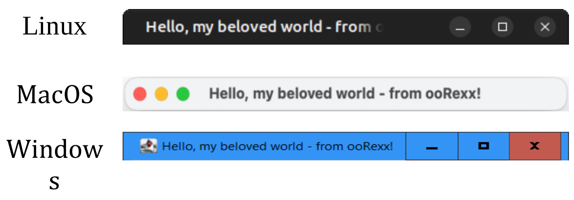
Linux
Hello, my beloved world - from c
MacOS
Hello, my beloved world - from ooRexx!
Windows
Hello, my beloved world - from ooRexx!

Figure 8. Result of code in Figure 7

3.6 Object-oriented

As an object-oriented language, useful base classes, data encapsulation, polymorphism, class hierarchy, method inheritance and concurrency are provided in ooRexx (ooRexx, 2023). ooRexx, the object-oriented paradigm of REXX, uses the tilde (~) as an explicit message operator. The programmer communicates with objects by sending them messages that name a method with potential options (or arguments). The receiving object looks for this method, invokes it on behalf of the programmer, and returns all the results that this method and its options may lead to. This explanation suffices to have students understand the concepts of insulation and inheritance. Without introduction, the object-oriented paradigm was already used in Figure 3, where the String object received the message *~strip("leading")*, which returned the string without leading spaces. Even with object-oriented programming, the ooRexx concepts manage to help beginners get started without unnecessary teaching time and cognitive load.

4 Conclusion

We consider cognitive load theory as a useful perspective to improve programming education and to choose an appropriate language. We see REXX's language characteristics to be the most important success factor in enabling students to learn productive programming quickly—within a few months—by minimizing unnecessary cognitive burden. These characteristics prevent troublesome errors and reduce the frustration associated with teaching and learning programming. Our experience has shown that students who have learned REXX subsequently learn other languages considered relevant by the industry, such as Visual Basic, Python, and especially Java, much more quickly and efficiently (Flatscher, 2023). We hope that this article will encourage future research on cognitive load in programming education and a consideration of REXX as an introductory language.

References

AmigaOS Manual: Arexx. (2023, May 27). In *AmigaOS Wikipedia.* https://wiki.amigaos.net/wiki/AmigaOS_Manual:_Arexx.

ANSI. (1996). *ANSI X3.274-1996—Programming Language REXX.* Retrieved from https://www.rexxla.org/rexxlang/standards/.

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, *26*(2-3), 148-175.

BP-1. (2023, February, 16). *Business Programming 1.* Retrieved from https://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/.

BP-2. (2023, February, 16). *Business Programming 2.* Retrieved from https://wi.wu.ac.at/rgf/wu/lehre/autojava/material/foils/.

BSF4ooRexx. (2023, May, 26). *Makes all of Java directly available to ooRexx and vice versa.*

Retrieved from https://sourceforge.net/projects/bsf4oorexx/.

Cowlishaw, M. (2022). *General Decimal Arithmetic*. Retrieved from https://speleotrove.com/decimal/.

Cowlishaw, M. (1987). The design of the REXX language. *ACM SIGPLAN Notices*, *22*(2), 26-35.

Data is Beautiful. (2019, October 7). *Most Popular Programming Languages 1965 – 2019* [Video file]. Youtube. https://www.youtube.com/watch?v=Og847HVwRSI.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011, June). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (pp. 208-212).

Flatscher, R.G., (2023). Proposing ooRexx and BSF4ooRexx for Teaching Programming and Fundamental Programming Concepts. *In Proceedings of ISECON 2023*. Forthcoming. Plano.

Flatscher, R. G., & Müller, G. (2021). " Business Programming"–Critical Factors from Zero to Portable GUI Programming in Four Hours.

Fosdick, H. (2005). *Rexx programmer's reference*. John Wiley & Sons.

Garner, S. (2002). *Reducing the cognitive load on novice programmers* (pp. 578-583). Association for the Advancement of Computing in Education (AACE).

List of programming languages. (2023, May 27). In *Wikipedia*. https://en.wikipedia.org/wiki/List_of_programming_languages.

Mason, R., Cooper, G., & de Raadt, M. (2012, January). Trends in Introductory Programming Courses in Australian Universities–Languages, Environments and Pedagogy. In *Proceedings of the Fourteenth Australasian Computing Education Conference* (Vol. 123, pp. 33-42).

McIver, L., & Conway, D. (1996, January). Seven deadly sins of introductory programming language design. In *Proceedings 1996 International Conference Software Engineering: Education and Practice* (pp. 309-316). IEEE.

ooRexx. (2023, April 19). ooRexx (Open Object Rexx) Files. Retrieved from https://sourceforge.net/projects/oorexx/files/oorexx/.

Paas, F., Tuovinen, J. E., Tabbers, H., & Van Gerven, P. W. (2003). Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist*, *38*(1), 63-71.

PYPL. (2023). *PYPL PopularitY of Programming Language*. Retrieved from https://pypl.github.io/PYPL.html.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137-172.

Sands, P. (2019). Addressing cognitive load in the computer science classroom. *Acm Inroads*, *10*(1), 44-51.

Seik, A., (2023, February, 12). *ooRexx Plugin for IntelliJ IDEA and ooRexxDoc*. Retrieved from https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/aseik/ooRexxIDEA/GA/2.1.0/.

Stachel, J., Marghitu, D., Brahim, T. B., Sims, R., Reynolds, L., & Czelusniak, V. (2013). Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool. *Journal of Integrated Design and Process Science, 17*(1), 37-54.

Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, *13*(4), 1-40.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science, 12(2)*, 257-285.

Sweller, J., & Van Merriënboer, J. J. G. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review, 53*(3), 147-177.

WU (2023, February, 16). *Selected Seminar, Diploma, Bachelor and Master Theses.* Retrieved from https://wi.wu.ac.at/rgf/diplomarbeiten/.