

RPI: An Object-Oriented Interpreter for Romanian-Language Pseudocode in Computer Science Education

Valentin-Ștefan ȘANDOR, Bogdan IANCU
Bucharest University of Economic Studies, Romania
stefansandor26@gmail.com, bogdan.iancu@ie.ase.ro

The current paper presents the steps used and the development process of a custom interpreter for the Romanian pseudocode language that offers native Object-Oriented Programming (OOP) support. A custom scanner was designed to be able to tokenize the Romanian keywords, and the results were fed forward to a parser that constructs an Abstract Syntax Tree (AST). The AST is then evaluated by an interpreter that executes the instructions. Advanced features such as recursion, method overriding, and user-friendly error messages are also supported. The trials showed great potential for in-class usage and can make the interpreter a steppingstone for introducing newer paradigms to students, as is the case of OOP. Future work includes debugging support, which can be easily developed since the codebase was designed with a focus on extensibility.

Keywords: Pseudocode, Interpreter, Romanian, Computer science, Education

DOI: 10.24818/issn14531305/30.3.2026.01

1 Introduction

OOP languages are mainly used nowadays for software development due to their capabilities in better modeling real-world problems through concepts such as classes, objects, abstraction, encapsulation, inheritance, and polymorphism. The most popular OOP languages: Java, C++, and Python (according to the TIOBE index [1]) allow developers to create modular, maintainable, and reusable code. Both interpreters and compilers are used in the translation process of source code written by programmers into executable code that computers can understand and execute.

The interpreter will translate and execute code line by line, by reading the source program. Languages such as Python use interpreters, which provide more flexibility but slower execution times when compared to languages that use compilers [2].

By contrast, a compiler will translate the entire source code into machine code that can be later executed. For instance, the C++ language uses a compiler to develop platform-dependent binary files, which can be later executed [2].

Regarding OOP language compilers, C++ is one of the most notable representatives. As mentioned, a compiler is a computer program that transforms source code written in one

programming language into machine code or an interpreted language. One of the most popular compilers for C++ is GCC, part of the GNU Compiler Collection [3].

Another example is the Java compiler, which compiles Java source code into an intermediate language called bytecode. The bytecode is then executed by the Java Virtual Machine, an interpreter that executes the Java bytecode on a target platform. This hybrid model increases the code portability: the same Java program can execute on different systems without change [4].

Interpreted OOP languages, such as Python and Ruby, rely on interpreters that execute source code line by line. For example, CPython, the default and most popular Python interpreter, allows direct execution of Python code, eliminating the need for complete compilation. This approach provides ease of use and simplicity, but at the cost of slower execution speeds compared with other languages that are fully compiled [5].

2 Formal languages and automata in interpreter design

Building an interpreter requires a solid understanding of formal languages – the mathematical way to describe syntax, and automata theory – abstract machines that recognize those

languages [6]. The Romanian Pseudocode Interpreter (RPI) project was inspired by “limbajul pseudocod” (the pseudocode language) used in the Romanian education system, which, according to [7], “contains the structures specific to an ordinary programming language (like Pascal, C/C++, Basic, etc.) but is intended to be read by humans, not computers”.

Pseudocode is, in general, a natural-language description of algorithms: it may omit certain formal requirements of a real programming language (for example, variables might not need explicit declaration, and syntax can be more flexible and closer to natural language).

To create an interpreter for a pseudocode language, we must formalize it. We need to turn the algorithm's human-readable steps into a specific formal language that a machine can parse. The Chomsky Hierarchy categorizes languages into a hierarchy (see **Fig. 1**) that includes: regular languages (type-3), context-free languages (type-2), context-sensitive languages (type-1), and recursively enumerable languages (type-0). Programming languages (and the Romanian pseudocode) fall into the first two categories: the lexical structure can be described by a regular grammar, and the grammatical structure by a context-free grammar.

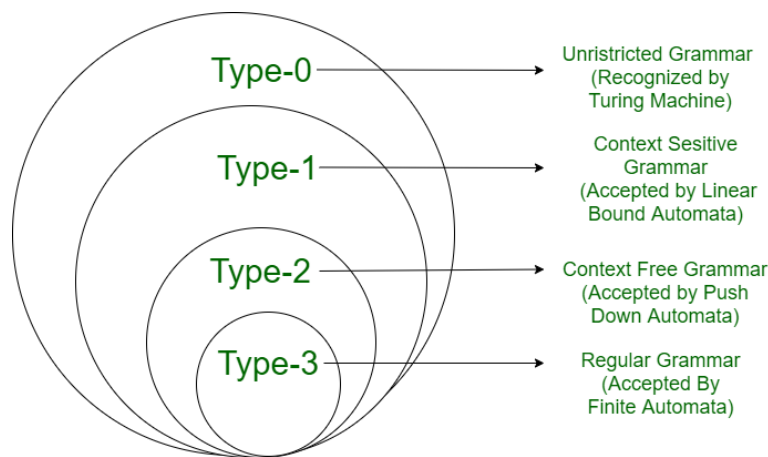


Fig. 1. Chomsky Hierarchy [8]

These two theoretical classes influenced the interpreter implementation: finite-automaton principles were used for the lexical analysis (tokenizing the input code) and pushdown-automaton principles for parsing (analyzing the hierarchical structure of code). Formal grammars (such as context-free grammars) allow the specification of a precise sequence of tokens that form a valid program. As [9] explains, context-free grammars (CFGs) “are widely used in computer science to write new programming languages... they allow a complicated system (like a compiler) to be specified in a very concise way, and there are programs that will automatically take the grammar and build the system for you”.

In other words, by writing a grammar for a language, we not only document its syntax, but also, we can easily create parser generators or methods to implement the interpreter.

In the current scenario, the grammar was manually crafted and written in the form of a custom parser.

In theory, a finite automaton is an abstract machine with a finite number of states, the ability to transition between those states, and to process strings one symbol at a time. One formal definition [10] describes finite automata as “abstract machines used to recognize patterns in input sequences, forming the basis for understanding regular languages in computer science”. In simpler terms, if a set of strings can be described by a regular expression, then there exists a finite automaton (deterministic or nondeterministic) that will accept exactly those strings. According to [10], “Finite automata come in two flavors: deterministic (DFA) and nondeterministic (NFA), both having the same expressive power over regular languages. They are widely used in practical

applications such as text processing, network protocols, and, importantly, compilers and interpreters”.

The first stage of interpretation, breaking the input source code into tokens, is known as lexical analysis or scanning. This stage can be directly modeled using regular languages and finite automata. Each token type (keywords, identifiers, numbers, symbols, etc.) can be described by a regular expression, and the collection of tokens forms a regular language.

Many lexer generators (like the classic Unix tool `lex` or modern equivalents) are built on DFA theory [11]: “Lexer generators such as `lex` and `ocamllex` are built on the theory of deterministic finite automata... the input to a lexer generator is a collection of regular expressions that describe the tokens of the language. The output is an automaton... that takes characters as input and eventually produces tokens as output.”

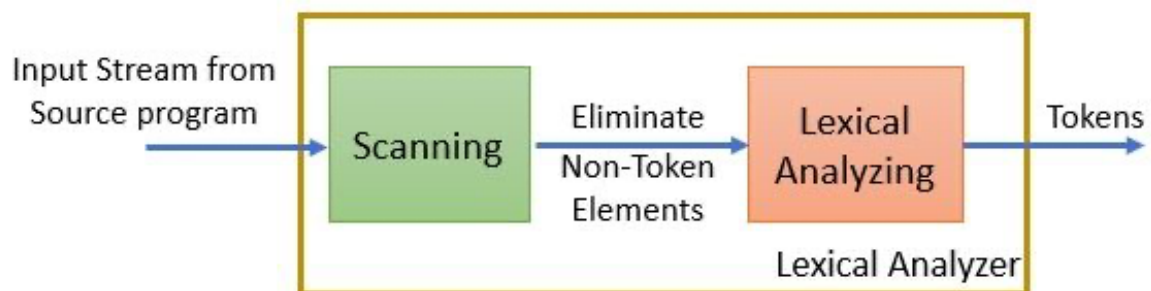


Fig. 2. Lexical analysis process [12]

In the current project, a generator was not used; a manually crafted lexer was used; nonetheless, the same principles were followed. Patterns for tokens were defined, such as:

- **Keywords:** specific reserved words in Romanian (e.g., `daca`, `atunci`, `altfel`, `pentru`, `executa`, `cattimp`, `repetă`, `panacand`, `scrie`, `citeste`); these are recognized as distinct token types.
- **Identifiers:** names for variables (e.g., `n`, `suma`, `index`); in pseudocode, variables are not pre-declared, but they must follow a pattern (starting with a letter, followed by letters, digits, or underscores); a regular expression like `[A-Za-z_][A-Za-z0-9_]*` could define an identifier.
- **Literals:** numeric constants (e.g., `123`, `3.14`) and string constants (text in quotes); each has a regex (for example, `\d+` for an integer, or a pattern for real numbers and a pattern for strings in quotes).
- **Operators and symbols:** e.g., `+`, `-`, `*`, `/`, `%`, comparison operators (`=`, `!=`, `<`, `<=`, etc.), logical operators (SI for AND, SAU for OR, NOT), assignment symbol (`<-`), punctuation like commas and

parentheses; each of these is a fixed string (or symbol) that the lexer can recognize directly.

- **Whitespace and newlines:** which are typically ignored or used only as separators (except indentation, where they might have semantic meaning).

By constructing a finite-state machine that reads characters and recognizes specific token patterns, we ensured that the source code is split in a correct manner. For instance, when the lexer reads the characters `d`, `a`, `c`, sequentially, it will match the pattern for the keyword “`daca`” (meaning “if”). Similarly, a sequence of digits such as `123` matches the pattern for a number literal. If a sequence of characters doesn’t match any token pattern, the lexer will report an error for an illegal token. Using automata theory here guarantees that the lexicon of our language is well-defined and that tokenization can be done in linear time, relative to the input size, since DFAs run in linear time in general. The result is a scanner that reads the raw Romanian pseudocode text and produces a stream of tokens for the next stage.

While the lexer deals with linear patterns, the parser must deal with hierarchical structures

in code: matching parentheses, matching an if with the corresponding else, grouping statements into blocks, and so forth. “The theoretical model for handling such nested or recursive structures is the pushdown automaton (PDA), which is essentially a finite automaton equipped with a stack” [13].

Programming languages require matching constructs and potentially infinite nesting, similarly (e.g., an if inside an if, loops inside loops, etc.), that being the reason why a PDA model is more appropriate. To quote a succinct explanation [14]: “Pushdown Automata (PDA) extend the capabilities of finite automata by using a stack, which allows them to handle a broader range of languages, particularly context-free languages.”

In practical terms, the parser of our interpreter behaves like a pushdown automaton: it reads a stream of tokens (from the lexer) and uses a recursive strategy based on a stack (implicitly the call stack, in our recursive descent implementation) to build a parse tree or abstract syntax tree (AST) representing the program structure. Each time the parser encounters a code construct that can be nested (like an expression with parentheses, or a control structure with a block of inner statements), it will push that context onto the stack (for instance, by calling a function to parse a subexpression or sub-block), and later will pop it when the construct ends.

As [11] states: “A context-free grammar is the formal device we use to specify which strings (or token sequences) a pushdown automaton should accept. A CFG consists of a set of production rules of the form $A \rightarrow \alpha$, where A is a nonterminal symbol, and α is a sequence of symbols (terminals or nonterminals) that A can be replaced with.”

In the case of our interpreter, context-free grammar was designed to describe the pseudocode language formally. For example, there are rules to describe an if statement structure, a loop structure, an assignment, etc. The grammar provides a blueprint for writing the parser: essentially, code was written to implement each grammar rule (in a top-down parser, this means writing a function for each nonterminal that consumes tokens according

to the rule). An important aspect of grammar design for a programming language is ensuring that the grammar is unambiguous (each valid program has a unique parse tree) and is compatible with the parsing technique.

A recursive descent parser was chosen (a type of top-down parser), which typically requires the grammar to be LL(1) or at least free of certain complexities like left recursion [15]. As the grammar was designed, it was adjusted for these constraints: for instance, eliminating immediate left recursion in expression rules by using equivalent formulas. Operator precedence was leveraged to structure the grammar for expressions. The result was a grammar that the parser could easily handle.

At any point, the “stack” of the PDA is mirrored by the call stack of parsing routines (e.g., if we are in the middle of parsing an expression inside an if-statement inside a loop, the call stack has something like `parseLoop` $\rightarrow$ `parseIf` $\rightarrow$ `parseExpression`). Each of these corresponds to a partially processed construct, and returning from the function corresponds to completing that construct (popping from the stack).

Creating a programming language, even a simple one for educational purposes, requires a balance between simplicity, familiarity, and clarity. The primary inspiration for RPI was the pseudocode used in the Romanian educational system, so the language should feel familiar to students and readers of that pseudocode. At the same time, since we wanted to implement a real interpreter, we needed to remove possible ambiguities and make certain implicit aspects explicit. Romanian pseudocode (like pseudocode languages in general) is meant to be human-readable in the first place. For example, one might write an algorithm in pseudocode as if they were writing instructions in plain language, mixing mathematical notation and natural language. This led to a specific design principle: preserve the human-readable nature of the language. We chose to keep keywords and most syntax in Romanian, rather than replacing the words with English or symbolic equivalents, to maintain student familiarity. This meant keywords like `daca/atunci/altfel` (if/then/else),

`pentru` (for), `cattimp` (as long as), `repetă/panacând` (repeat/until), and `scrie` (write, used for output) are exactly as one would see in educational pseudocode problems from the Romanian curriculum.

However, human-oriented pseudocode often omits details that a real language would require. For example, pseudocode typically does not require variable declarations or type annotations – one just starts using a variable. We decided to embrace this simplicity: our interpreter does not require explicit variable declaration. Any new identifier encountered is treated as a variable, which the interpreter will create on the fly (with an initial value of 0 or null, as appropriate, similarly to most dynamically typed languages). This design choice lowers the barrier for writing code, at the cost of making the interpreter more complex (it must manage a symbol table dynamically). It also implies the pseudocode language is dynamically typed (a variable can hold any type of value, though in practice our pseudocode mainly deals with numbers and strings).

We think this is an acceptable trade-off given that pseudocode algorithms and problems usually focus on a single numeric domain. Another omission in pseudocode is explicit end-of-block markers. In languages such as C or Java, we have braces to mark the beginning and end of a block of code. In Pascal, we have the `begin` and `end` keywords. In Romanian pseudocode, some examples just indent the code to show it's inside an if or a loop, similar to Python, since there is no standalone “end if” or “end for”. For our implementation, we chose to use braces for simplicity reasons.

3 The Romanian Pseudocode Interpreter implementation

The RPI project was designed with a classic compiler architecture in mind, consisting of distinct scanning, parsing, and execution stages. The source code (written in Romanian pseudocode) is processed through a pipeline of components: a *scanner* for lexical analysis, a *parser* for syntax analysis, and an *interpreter* for executing the instructions. Each stage transforms its input and feeds it to the next stage, following a common design in

compiler theory: “In the lexical analysis phase, the input (the pseudocode text) is read and converted into a sequence of tokens, with whitespace and comments discarded” [16]. The tokens are then passed to the parser, which checks their ordering against the language grammar and builds an in-memory representation of the program's structure: “To perform syntactic analysis, there's a parser that processes these tokens and parses them into a tree called the Abstract Syntax Tree (AST)” [17].

This modular separation offers clarity and maintainability: each component handles a specific set of tasks (lexical, syntactic, or semantic processing), making the system easier to develop and debug. It also mirrors the structure of many real-world language processors, which makes it appropriate for an educational pseudocode interpreter. Notably, while the interpreter is inspired by established compiler design, it was custom-built for Romanian pseudocode syntax. This meant adapting the generic architecture to support Romanian keywords and idioms. For example, the language uses the symbol `<-` for assignment (instead of `=` as in many languages) and keywords like “*daca*” and “*pentru*” (meaning “if” and “for”), which needed to be recognized. The architecture remained flexible to accommodate these specifics. The design also allowed adding features incrementally: basic expressions and control flow were implemented first, and more complex features (like functions and classes) were integrated on top.

The scanner goes through the input character by character, grouping characters into higher-level units such as identifiers, keywords, numbers, string literals, and punctuation. Each token is classified by type (for example, “NUMBER”, “IDENTIFIER”, “PLUS_SIGN”, or specific keyword categories) and retains the exact text (lexeme) and its location (line number) for error reporting. In our implementation, the *scanner* was coded in Java following a common algorithmic pattern for lexical analyzers. It maintains indices into the input string and advances through it, using a switch-case structure to identify token patterns. For instance, when it encounters a digit

character, it will read the full number sequence to produce a numeric token; when it sees a double quote, it will accumulate characters until the closing quote to produce a string literal token. Pseudocode keywords in Romanian (such as “daca”, “pentru”, “variabila”, etc.) are recognized by the *scanner* as well. The *scanner* uses a dictionary of reserved words so that any identifier matching a known keyword is output as a keyword token. This allowed easy adaptation to Romanian-specific keywords.

The output of the *scanner* is a list of tokens. Internally, a Token class holds the token's type, the exact text snippet (lexeme), any literal value (for numbers or strings), and the line number. For example, a numeric token might be represented as type NUMAR with lexeme “123” and a numeric value 123.4, whereas a keyword token for “pentru” would be type PENTRU with lexeme “pentru” and no literal value. These tokens are appended to a list as scanning progresses. At the end of the input, the *scanner* adds an end-of-file marker token (EOF) to signal completion. This token list is then passed on to the next phase, the *parser*.

After tokenization, the *parser* takes the stream of tokens and performs syntax analysis. The goal of this phase is to understand the structure of the program according to the grammar of the pseudocode language and to produce an internal representation of this structure. In compiler terminology, this is where we build the Abstract Syntax Tree (AST). The AST is a tree data structure where each node represents a language construct (an operation, a control structure, a literal value, etc.), organized hierarchically to reflect the nesting and order of execution in the source code [2]. Our *parser* is implemented as a recursive descent parser, which is a kind of top-down parser where the set of methods corresponds to the grammar rules of the language. Each method calls sub-methods to handle sub-rules, mirroring the code's grammar structure. For example, one method might handle an <if-statement> rule: it expects a DACA (if) token, then an expression token sequence for the condition, then the ATUNCII/ALTFEL (then/else)

parts, etc. (in our pseudocode grammar, daca-atunci-altfel would be the structure of an if-then-else statement).

The *parser* methods use the current position in the token list and advance as they recognize tokens belonging to each grammar rule. If an unexpected token is encountered, the parser reports a syntax error indicating something like “Expected X but found Y at line Z,” helping the user locate mistakes in the pseudocode. The output of the parser is an AST, represented by a set of Java classes corresponding to different statement and expression types. Rather than using a generic tree node, the design defines specific classes for each kind of syntax element. For instance, there are classes like Expr.Binary for a binary arithmetic expression, Expr.Literal for a literal value, Expr.Variable for a variable reference, and similarly, Stmt.If for an if-statement, Stmt.While for a loop, Stmt.Function for a function definition, and so on. These classes are organized into class hierarchies (one for expressions and one for statements), with a base abstract class (Expr or Stmt) and subclasses for each variant. This design makes the AST typed and easier to work with in code: instead of a one-size-fits-all node, each node carries the exact fields it needs (e.g., a Binary node holds references to its left and right sub-expression nodes and the operator token).

The AST is essentially a tree of these objects. When parsing is done, we obtain the root of this tree (or a list of Stmt nodes representing the top-level statements in the program). Manually constructing the parser and the AST was one of the most complex parts of the project. One challenge was operator precedence and associativity. The *parser*'s recursive descent implementation had to ensure that, for example, multiplication and division bind tighter than addition, or that the = (equality) comparisons are evaluated after arithmetic operations. This was solved by structuring the parser methods in a typical precedence climbing fashion: methods like expression() delegate to term() for addition/subtraction, which delegates to factor() for multiplication/division, etc., all the way down to primary expressions. Another challenge was correctly parsing

nested constructs (such as an if inside a while or vice versa) and matching the correct “end” tokens. In Romanian pseudocode, blocks are indicated by words or braces (depending on the dialect; our implementation used braces as block delimiters), and ensuring that each opened block was closed properly was important to avoid syntax tree inconsistencies. The parser code includes error recovery mechanisms in a simple form – if a syntax error is encountered, it tries to synchronize by discarding tokens until it reaches a likely statement boundary, to prevent one error from causing a cascade of further errors.

Despite the complexity, the result of the parsing stage is a robust AST ready for execution. Having the AST as a separate data structure also made it easier to implement certain features, like static analysis before execution. In fact, an additional step called *the resolver* runs after parsing and before execution (still in the parsing stage conceptually). This *resolver* traverses the AST to resolve variable names to the correct scope (to differentiate global vs local variables, check for misuse like reading an uninitialized variable, etc.). It effectively builds a symbol table of definitions and uses. The resolver was implemented using the same AST visitor approach. Once parsing and resolving are done, the system proceeds to the interpretation phase, confident that the code is syntactically correct and symbols are properly resolved.

The final phase of the pipeline is interpreting the AST to produce program behavior. *The interpreter* module implements a tree-walk interpreter: it recursively traverses the AST, evaluating expressions and executing statements in a depth-first manner. In essence, the *interpreter* is a visitor that “walks” the tree and performs the appropriate action for each node type. This design cleanly separates the definition of the language syntax (in the AST classes) from the execution logic, following the Visitor design pattern.

Here, the algorithm is the execution of pseudocode, and the objects are the nodes of the AST. The Interpreter class implements the Expr.Visitor and Stmt.Visitor interfaces (as defined by the AST classes), meaning it

provides concrete methods like visitLiteralExpr, visitBinaryExpr, visitIfStmt, etc. Each of these methods knows how to execute that kind of AST node. For example, the visitBinaryExpr method in the Interpreter will evaluate the left and right sub-expressions, then apply the operator (such as addition or comparison) to produce a result. A visitIfStmt method will evaluate the condition expression, check if it is true or false, and then execute either the then-branch or the else-branch accordingly by recursively calling back into the interpreter for those branch statement subtrees. To keep track of program state during execution, the interpreter uses a runtime environment. The environment is essentially a symbol table that stores the values of variables and other identifiers at runtime. It was implemented via an Environment class that contains a map of variable names and values. When the interpreter encounters a variable declaration (e.g., `variabila x <- 5` in pseudocode), it defines a new entry in the current environment (key “x” with value 5). When it sees a variable usage (e.g., `x + 2` or an assignment `x <- 10`), it looks up the variable name in the environment chain to get or update its value. The environment is structured to support lexical scoping: there can be a chain of environment objects, representing nested scopes. A new environment is pushed (by creating a child linked to an enclosing environment) whenever entering a new scope, such as when entering a function or a block. This way, variables defined inside a function or block can shadow outer variables and will be automatically removed when the block/function ends (the interpreter will discard the inner environment, restoring the previous one). This approach provides a clear and memory-safe model for variable lifetimes. Function calls and procedures are supported by the interpreter as well. When a pseudocode function is defined (using the “functie” keyword in Romanian), the interpreter creates a function object (in our implementation, RpdFunction) that contains the function’s AST (the parameter list and body) along with a reference to the environment where it was declared (this is to support closures, i.e., the function retaining access to

external variables from its defining scope). Calling a function involves creating a new environment for its execution (with its own local variables and parameters), executing the function's body AST in that environment, and then returning any result.

The concept of class definitions was also implemented ("clasa" in Romanian) so that the pseudocode language supports simple object-oriented features. When a class is declared, the interpreter creates an `RpdcClass` object containing its methods. Instantiating a class (via a special `new` operation or calling the class as if it were a function) produces an `RpdcInstance` object that has its own field environment. Method calls are handled by binding the instance to the method (so that within the method, `acesta` – meaning "this" – refers to the instance).

Error handling during execution is another aspect of the runtime. The interpreter can encounter runtime errors, such as dividing by zero, using an undefined variable, or calling a function with the wrong number of arguments. In our implementation, when a runtime error is detected, the interpreter throws a `RuntimeError` exception (a custom exception that carries an error message and token information). The main execution loop catches these exceptions and reports an error to the user with the problematic line. For flow control, we also took advantage of exceptions: for example, a return statement in pseudocode is implemented by throwing a special `Return` exception behind the scenes, which is caught by the function-call mechanism to unwind the call stack and produce a return value. This technique simplified the interpreter logic for breaking out of deep execution contexts without deeply nested conditionals. The design of the interpreter was influenced by the well-known Interpreter design pattern in software engineering. In fact, the whole system can be seen as an instance of that pattern. By structuring the problem this way, each piece of the language (each AST node type) knows how to interpret itself (via the Visitor's methods), which made the code easier to extend. For example, to add a new statement type (say, a `RepeatUntil` loop), we would add a new AST

class for it and a corresponding `visitRepeatUntilStmt` method in the Interpreter.

The clean separation of concerns (parsing vs execution, and within execution, each node type separated) meant that new features could often be added without breaking existing ones. One design pattern that was considered in the architecture is the Observer pattern. While not directly used in the core interpreter logic, the Observer pattern could be useful for the overall system, especially when integrating the interpreter into a user interface or development environment.

In the context of this interpreter, an Observer pattern could allow the runtime to notify external components about certain events – for instance, when a new line is executed, when a variable's value changes, or when the program finishes execution. This would be particularly relevant for a graphical IDE or a debugger where the UI might need to update in response to program execution (highlighting the current line, updating a variable watch list, etc.). By decoupling the interpreter from the UI using an Observer-like mechanism, the core logic can remain focused on execution, and any number of observers (GUI panels, debuggers, loggers, etc.) can subscribe to receive notifications of events. Although this was not fully implemented in the current command-line prototype, the design was kept flexible enough to add such features. For example, the interpreter could easily be extended to issue callbacks or events at certain milestones (each statement execution, each output produced, etc.), and a debugger could register listeners to those events.

4 Code demonstrations of The Romanian Pseudocode Interpreter

In the following example, we demonstrate variables and assignment. The pseudocode language allows declaring variables with the `variabila` keyword and using the left-arrow `<-` for assignment. The following code sequence shows how to define three variables: a name (string), an age (integer), and a price (floating-point number):

```
variabila nume <- "Ana";
variabila varsta <- 25;
```

```
variabila pret <- 12.5;
```

The interpreter handles these variable declarations dynamically – there is no need to specify a type. This dynamic typing feature was described in the previous chapter, and it allows `nume`, `varsta`, and `pret` to hold values of different types (string, integer, and double) without explicit type annotations. Running the code above doesn't produce output by itself (it just stores values), but we can easily use `scrie` to display these variables or the results of expressions involving them:

```
scrie nume; // Output: Ana
scrie varsta + 5; // Output: 30
scrie pret * 2; // Output: 25.0
```

In these lines, the interpreter retrieves the values of `nume`, `varsta`, and `pret` and evaluates the expressions. It correctly prints Ana (the string stored in `nume`), 30 (the result of adding 5 to the `age`), and 25.0 (the result of doubling the `price`). This demonstrates that basic arithmetic operations and variable references work as expected. The pseudocode syntax for arithmetic (+, -, *, /) and the order of operations mirror those of standard mathematics and programming, making it intuitive. For instance, the interpreter respects precedence, so an expression like `5+3*2` yields 11 (since multiplication is done before addition). We can also mix types in expressions in sensible ways. String concatenation is achieved with the `+` operator as well. For example:

```
scrie "Salut, " + nume + "!"; // Output:
Salut, Ana!
```

Here, the interpreter concatenates the string literal `"Salut, "` with the content of `nume` (which is `"Ana"`) and another literal `!"`. The result is the greeting `"Salut, Ana!"`. This dynamic handling of types (treating numbers and strings appropriately when using `+`) shows the interpreter's flexibility. However, it will also detect type errors in invalid cases: for instance, trying to add a number to a boolean would result in a runtime error with a clear message, rather than a confusing behavior. We should also note that the language includes the boolean literals `adevarat` (true) and

`fals` (false), as well as the `nimic` (null/void) value. Basic comparison and logical operations are supported, using symbols that align with mathematical notation. For example, we can write equality and inequality comparisons:

```
scrie 3 = 4; // Output: fals (3 equals 4
is false)
scrie 3 != 4; // Output: adevarat (3 not-
equal-to 4 is true)
scrie 5 < 10; // Output: adevarat (5 is
less than 10)
scrie 20 >= 15; // Output: adevarat (20
is greater or equal to 15)
scrie !adevarat; // Output: fals (logical
NOT of true is false)
scrie adevarat si fals; // Output: fals
(true AND false)
scrie adevarat sau fals; // Output: ade-
varat (true OR false)
```

The interpreter evaluates each expression and prints the boolean results (`adevarat` or `fals`). The examples above confirm that relational comparisons and logical connectives function correctly.

One of the key goals of the interpreter is to support fundamental control structures found in typical algorithms. In the next example, we demonstrate how branching (if-else) and loops (while, for) work in the Romanian pseudocode language, and how the interpreter executes them.

Conditional statements use the `daca` keyword (for "if") and an optional `altfel` ("else") block. The syntax is similar to other languages, but uses Romanian words and punctuation that make it feel like pseudocode in natural language.

```
variabila a <- 7;
variabila b <- 5;

daca (a < b) {
  scrie "A este mai mic decât B";
} altfel {
  scrie "A este mai mare sau egal cu B";
}
```

In this example, since we assigned `a = 7` and `b = 5`, the condition `a < b` evaluates to false. The interpreter will skip the first block and execute the code in the `altfel` branch. Thus, it will output `"A este mai mare sau egal cu B"` ("A is greater than or equal to B"). We could easily

change the values of *a* and *b* to test the other path; for instance, with *a* = 3 and *b* = 5, the condition would be true, and the first message would be printed instead. The interpreter correctly handles these branches, only executing the block that matches the condition. This confirms that the branching mechanism operates as expected, enabling the pseudocode to make decisions.

Moving on to loops, the interpreter supports both **cattimp** (a “while” loop for indeterminate repetition) and **pentru** (a “for” loop for fixed-count iteration). We illustrate each with an example:

```
variabila x <- 0;
cattimp (x < 5) {
  scrie x;
  x <- x + 1;
}
```

This pseudocode initializes *x* with 0, then enters the loop. The interpreter checks the condition *x* < 5 before each iteration. As long as the condition is true, it executes the loop body: printing the current value of *x* and then incrementing *x* by 1. Once *x* reaches 5, the condition becomes false and the loop ends. If we run this code, the interpreter will output:

```
0
1
2
3
4
```

The correctness of this output demonstrates that the while-loop control flow works properly: the interpreter repeatedly executes the block and updates *x* until the condition fails. This behavior is essential for expressing algorithms that require repetition, such as traversing through array elements or accumulating a sum (even though our pseudocode language doesn’t have arrays in this version, the pattern is still educationally useful for understanding loops).

For a fixed-count loop, the **pentru** construct is used, similar to the for loop from the C language, but with a more readable syntax. A **pentru** loop in our pseudocode takes an initializer, a loop condition, and an increment expression, all in one line:

```
pentru (i <- 0; i < 5; i <- i + 1) {
  scrie i;
}
```

The outcome of this code is the same sequence of numbers as the previous example (0, 1, 2, 3, 4, each on a new line). The interpreter’s handling of the **pentru** loop involves initializing the variable, checking the condition before each iteration, and updating the variable after each iteration, just as one would expect. The syntax uses semicolons to separate the three parts (initializer, condition, increment) within the parentheses, reflecting common pseudocode conventions and imperative language structures. This indicates that the interpreter can manage more complex loop semantics internally (managing loop scope and continuation) while still presenting a clean pseudocode interface to the user. Educationally, this is a big advantage: students can experiment with changing the loop bounds or conditions in Romanian, and immediately see different outcomes, reinforcing their understanding of how loops and conditions work. The fact that the interpreter prints the results instantly in each case helps to connect the abstract concept (e.g., “while loop”) with concrete behavior.

Functions are vital for structuring code and avoiding repetition. Our Romanian pseudocode interpreter supports user-defined functions using the **functie** keyword, with parameters and return values. In the next examples, we demonstrate how to define and call functions, including the use of recursion to illustrate the interpreter’s capabilities with respect to the call stack and scope. The following function adds two numbers:

```
functie aduna(a, b) {
  intoarce a + b;
}

// Using the function:
variabila rezultat <- aduna(3, 4);
scrie rezultat; // Output: 7
```

When this code is interpreted, the function definition is stored, and later the call **aduna(3, 4)** invokes the function. The interpreter creates a new environment for the function, assigns *a* = 3 and *b* = 4, evaluates *a* + *b*, and then

returns the result. The variable `rezultat` receives the returned value (7), which is then printed. This example confirms that function calls and returns work correctly: the interpreter can jump into a function's body and back out with a result. The keyword `intoarce` is used to return a value (it literally means "return" in Romanian). If a function does not explicitly return something, it returns the special `nimic` value by default – similar to null or void, which is useful for procedures that do not have side effects (like printing).

One strength of our interpreter (inherited from the tree-walk design) is that it supports recursive functions. To illustrate recursion, consider the classic example of a factorial function:

```
functie factorial(n) {
  daca (n <= 1) {
    intoarce 1;
  } altfel {
    intoarce n * factorial(n - 1);
  }
}

// Example usage:
scrie factorial(5); // Output: 120
```

In this code, the factorial function calls itself (`factorial(n - 1)`) to compute the factorial of smaller numbers. The interpreter handles this recursion by pushing a new call frame each time the function calls itself, just like a typical programming language runtime. When we call `factorial(5)`, the sequence of recursive calls will eventually reach `factorial(1)`, which returns 1, and then unwind: $2 * 1$, $3 * 2$, $4 * 6$, and $5 * 24$, resulting in 120. The interpreter manages the recursion depth and each function's local variables correctly, as demonstrated by the correct final result. This confirms that even more complex control flow (multiple nested function calls) is executed properly. It also suggests that the interpreter could be used to evaluate recursive algorithms, a common topic in computer science education.

Another aspect worth noting is that functions in this interpreter are first-class values in the sense that they can be passed around or assigned to variables (because under the hood, a function is an object of type `RpdcFunction`).

While we have not deeply used that feature in these examples, the language could support paradigms like callback functions or storing functions in data structures, increasing its expressiveness. For the scope of a teaching tool, however, the primary use case is simply defining and calling functions to encapsulate logic. This aligns with the educational goal of teaching structured programming: the interpreter not only executes the code but also encourages good practices like breaking problems into functions.

One of the more advanced features of our pseudocode interpreter is the support for object-oriented programming (OOP) concepts, namely classes, objects, polymorphism, and inheritance. This was an ambitious addition for a pseudocode system and was inspired by the needs of teaching more complex programming concepts in a simplified environment. In the next section, we demonstrate how to define classes with methods (and attributes), how to create objects, and how inheritance allows one class to extend another. These examples highlight the interpreter's power in handling OOP despite the dynamic nature of the language. To define a class in Romanian pseudocode, we use the `clasa` keyword. Inside the class, we can define methods using the `procedura` keyword (or we could use `functie` as well – the two are treated similarly by the interpreter, with `procedura` being more semantic for "method"):

```
clasa MicDejun {
  procedura gateste() {
    scrie "Painea se prajeste!";
  }
}

// Create an instance of the class and
call its method:
variabila micDejun <- MicDejun();
micDejun.gateste(); // Output: Painea se
prajeste!
```

In this example, class `MicDejun` has one method `gateste()` that, when called, prints a message "Painea se prajeste!" ("The bread is toasting!"). After defining the class, we instantiate it by calling `MicDejun()` as if it were a function. The interpreter handles this by creating a new object (an instance of `MicDejun`).

We store this instance in the variable `micDejun`. Then `micDejun.gateste()` calls the `gateste` method on that object, resulting in the presented output.

This demonstrates several things:

1. The interpreter correctly recognizes class definitions and stores their methods.
2. It allows the instantiation of classes, returning an object reference.
3. It supports method dispatch: when `micDejun.gateste()` is executed, the interpreter finds the `gateste` method associated with `MicDejun` and executes it in the context of the `micDejun` instance.

Notably, our interpreter also supports an initializer method, which is a method named `init` that acts like a constructor (runs automatically on object creation). In the above example, we did not define `init`, so the object is created with no special initialization logic. If an `init` was present, the interpreter would call it during `MicDejun()`. This feature is similar to constructors in languages like Java or Python's `__init__`. We won't delve deeper into constructors here, but it is worth mentioning, as it adds to the completeness of the OOP support.

Perhaps the most significant OOP feature in our pseudocode language is inheritance. A class can inherit from a base class, indicated by a `<` symbol after the class name. This means the new class (subclass) will have all the methods of the base class, plus any new ones or overrides it defines. The interpreter handles inheritance by linking the classes behind the scenes and enabling method overriding and super calls. The following code sequence provides an example with two classes to illustrate inheritance and method overriding:

```
clasa Animal {
procedura vorbeste() {
scrie "Animalul scoate un sunet.";
}
}

clasa Caine < Animal {
procedura vorbeste() {
super.vorbeste();
scrie "Cainele face: Ham-ham!";
}
}
```

```
// Create instances and test:
variabila animal <- Animal();
variabila caine <- Caine();
animal.vorbeste(); // Output: Animalul
scoate un sunet.
caine.vorbeste(); /* Output: Animalul
scoate un sunet. Cainele face: Ham-ham!
*/
```

In this snippet, we define class `Animal` with a method `vorbeste()` that prints a generic message ("The animal makes a sound."). Then we define class `Caine` (Dog) as a subclass of `Animal` using the syntax `clasa Caine < Animal`. Inside `Caine`, we also define a method `vorbeste()`, intended to override the parent version. In the body of `Caine`'s `vorbeste`, we include a call to `super.vorbeste()`, this invokes the `Animal` version of the method, so that a dog will do the generic animal behavior before its own specific behavior. After calling `super.vorbeste()`, we then write a dog-specific message ("Cainele face: Ham-ham!", meaning "The dog goes: Woof-woof!").

When we create an `Animal` instance and call `animal.vorbeste()`, we get the generic animal sound message. But when we create a `Caine` instance and call `caine.vorbeste()`, the output combines both lines: first the general sound (from the parent class via `super`), and then the dog's own sound. This confirms that method overriding and super calls are working. The interpreter was able to resolve `super.vorbeste()` correctly: it looked up the superclass (`Animal`) of `Caine` and found the `vorbeste` method there, executing it on the current `caine` instance. After that, it continued with the rest of the `Caine.vorbeste` method. Additionally, although not explicitly shown in this snippet, the interpreter supports the `acesta` keyword (literally "this" in Romanian) inside methods, referring to the current instance.

For example, if we had a field or wanted to refer to the object itself, `acesta` would be used just like this in other languages. Our design ensured that when a method is called, `acesta` is bound to the instance on which the method was invoked. This is how the interpreter knows what `super.vorbeste()` should act on (it uses the same instance). We could also add attributes to classes (since our interpreter

allows dynamic fields on objects). For instance, `acesta.nume <- "Rex"` inside an `init` method could set a name attribute on a `Dog`. Though we haven't deeply demonstrated class attributes here, the ability to set and get `micDejun.orice` (any attribute) on an object exists, similar to dynamic properties in languages like JavaScript or Python.

The object-oriented features showcased in the previous examples significantly boost the expressiveness of the pseudocode language. From an educational perspective, this means the interpreter can be used not just for procedural algorithms but also to illustrate principles of OOP in a simpler setting. Students can experiment by creating class hierarchies in Romanian pseudocode, which might make the concept of classes and inheritance more relatable (because the syntax uses familiar words like `clasa`, `procedura`, etc.). The fact that our interpreter correctly handles inheritance and polymorphism (including scenarios like a class inheriting and then overriding methods and using `super`) is a strong indicator that it has matured beyond a basic scripting tool into something that can model real-world programming paradigms.

5 Conclusions

In this paper, we presented an implementation of a Romanian Pseudocode Interpreter with OOP support. In terms of capabilities, the interpreter has proven itself able to handle a wide range of language constructs: from simple expressions and I/O to complex features like recursion and inheritance. Each example ran as expected, and the outputs matched the anticipated results, indicating that the implementation is robust for the tested scenarios. There were no incorrect behaviors observed in our trials, which suggests that the language grammar and the runtime (tree-walk evaluator) were implemented according to the specification.

We found that writing these examples in Romanian pseudocode felt natural and even empowering. For instance, the if-else example reads like a normal sentence, and the loop constructs are very clear in meaning. This suggests that the tool could be effective in

classroom settings. Students could write pseudocode to solve a problem and run it to verify correctness, bridging the gap between designing an algorithm and running it. "Normally, pseudocode is not run on a computer," as [18] mentions. In essence, the interpreter can serve as a steppingstone: once students are comfortable formulating solutions in pseudocode and seeing them work, they can transition to learning a full programming language with more confidence.

In terms of performance, our interpreter is a straightforward tree-walk interpreter. It is not optimized for speed, but all our example runs were near-instantaneous given the small input sizes. For the educational use case, efficiency is a minor concern – even an algorithm that is quadratic or worse in pseudocode will typically be run on small inputs for learning purposes. The interpreter's response time for these examples is well within acceptable ranges. Memory usage is also low for these short runs. We have not encountered runtime errors in the above demonstrations, except where we intentionally noted they would occur for incorrect usage (which we avoided). This suggests that the interpreter is stable for correct programs and provides meaningful error messages for incorrect ones (for example, if one tried to use an undefined variable or call a non-existent method, the interpreter would throw a runtime error with a message pinpointing the issue). This reliability is important for maintaining user trust – a learner should feel that if their program fails, it's because of their code logic and not a bug in the tool.

In summary, the code demonstrations from the previous chapter validate that the Romanian Pseudocode Interpreter is functional, feature-complete for its scope, and user-friendly. The interpreter executes a variety of pseudocode programs exactly as expected, which is a strong indicator that the implementation phase was successful. Just as important, the examples highlighted the interpreter's educational potential: by writing algorithms in their native language and seeing them run, students can more easily grasp programming fundamentals. The combination of human-readable

pseudocode and actual execution makes this tool a novel contribution that could enrich programming education, especially in Romanian settings.

Acknowledgments

The current paper is based on the bachelor's degree thesis titled "Custom Compiler for an Object-Oriented Language" authored by Valentin-Ștefan ȘANDOR under the supervision of Assoc. Prof. PhD. Bogdan IANCU, defended in 2025 at the Bucharest University of Economic Studies, Faculty of Cybernetics, Statistics and Economic Informatics.

The source code of the developed interpreter is public and can be found at [19].

During the preparation of this manuscript, Grammarly was used specifically for language rephrasing and structure enhancement. The authors reviewed and revised the suggestions generated by the tool and take full responsibility for the content and conclusions of this publication.

References

- [1] "TIOBE Index for May 2026," TIOBE, May 2026. [Online]. Available: <https://www.tiobe.com/tiobe-index/>. [Accessed May 2026].
- [2] J. L. Clinton, Build Your Own Programming Language: A programmer's guide to designing compilers, interpreters, and DSLs for solving modern computing problems, Packt Publishing Ltd, 2021.
- [3] "GCC, the GNU Compiler Collection," [Online]. Available: <https://gcc.gnu.org/>. [Accessed May 2026].
- [4] L. Tim, Y. Frank, B. Gilad, B. Alex and S. Daniel, "The Java® Virtual Machine Specification," February 2025. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se24/html/index.html>. [Accessed May 2026].
- [5] "This is Python," Python Software Foundation, [Online]. Available: <https://github.com/python/cpython>. [Accessed May 2026].
- [6] R. Nystrom, Crafting Interpreters, Gen-eve Benning, 2021.
- [7] C. Silviu, "Limbajul pseudocod," pbinfo, October 2021. [Online]. Available: <https://www.pbinfo.ro/articole/23972/limbajul-pseudocod>. [Accessed May 2026].
- [8] "Chomsky Hierarchy in Theory of Computation," GeeksforGeeks, February 2026. [Online]. Available: <https://www.geeksforgeeks.org/theory-of-computation/chomsky-hierarchy-in-theory-of-computation/>. [Accessed May 2026].
- [9] "Formal Languages - Grammars and Parsing," Computer Science Field Guide, [Online]. Available: <https://www.csfieldguide.org.nz/en/chapters/formal-languages/grammars-and-parsing/>. [Accessed May 2026].
- [10] "Introduction of Finite Automata," GeeksforGeeks, March 2026. [Online]. Available: <https://www.geeksforgeeks.org/theory-of-computation/introduction-of-finite-automata/>. [Accessed May 2026].
- [11] "Functional Programming in OCaml - Lexical analysis and parsing," Cornell University, 2021. [Online]. Available: https://courses.cs.cornell.edu/cs3110/2021sp/textbook/interp/lex_parse.html. [Accessed May 2026].
- [12] "Lexical Analysis in Compiler," [Online]. Available: <https://binaryterms.com/lexical-analysis.html>. [Accessed May 2026].
- [13] "Pushdown automaton," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Pushdown_automaton. [Accessed May 2026].
- [14] "Introduction of Pushdown Automata," GeeksforGeeks, March 2026. [Online]. Available: <https://www.geeksforgeeks.org/theory-of-computation/introduction-of-pushdown-automata/>. [Accessed May 2026].

- [15] N. A. A. Roselan, T. Ibrahim and H. Noor, "Grammarcheck: a dynamic LL (1) grammar checker using recursive descent parsing with chatbot assistance," *Progress in Computer and Mathematics Journal (PCMJ)*, vol. 3, pp. 189-197, 2025.
- [16] "Lexical Analysis and Syntax Analysis," *GeeksforGeeks*, January 2023. [Online]. Available: <https://www.geeksforgeeks.org/compiler-design/lexical-analysis-and-syntax-analysis/>. [Accessed May 2026].
- [17] B. P. C, "Abstract Syntax Tree (AST) - Explained in Plain English," December 2022. [Online]. Available: <https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38>. [Accessed May 2026].
- [18] T. Taylor, "The basics of working with pseudocode," February 2025. [Online]. Available: <https://www.tech-target.com/searchapparchitecture/tip/The-basics-of-working-with-pseudocode>. [Accessed May 2026].
- [19] V.-S. Sandor, "Romanian Pseudocode Interpreter," *GitHub*, 2025. [Online]. Available: <https://github.com/Creator-Stefann/Romanian-Pseudocode-Interpreter>.



Ștefan ȘANDOR graduated from the Faculty of Cybernetics, Statistics and Economic Informatics at the Bucharest University of Economic Studies in 2025. He is currently pursuing a master's degree in Distributed Systems from the University of Bucharest. He works as a Software Engineer at Procter & Gamble Romania.



Bogdan IANCU graduated from the Faculty of Cybernetics, Statistics and Economic Informatics at the Bucharest University of Economic Studies in 2010. He has a master's degree in Economic Informatics (2012) and a PhD in Economic Informatics starting from 2015 in the field of Ontologies and eLearning. He is an Associate Professor in The Department of Economic Informatics and Cybernetics at the Bucharest University of Economic Studies. His current research focuses on semantic technologies, cybersecurity, artificial intelligence, embedded systems, and IoT.