

**BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
COMPUTER SCIENCE IN ROMANIAN SPECIALIZATION**

DIPLOMA THESIS

Pseudocode Compiler

Supervisor

Conf. Dr. Rareş Florin Boian

Author

Jardan Andrei

2024

Abstract

This thesis presents the design and implementation of a compiler for the Pseudocode programming language, which is utilized in Romanian high-school Computer Science education and the Romanian Computer Science Baccalaureate exam. The primary objective is to enhance the efficiency and accuracy of verifying Pseudocode programs, thereby benefiting students, educators, and exam graders. The compiler employs LLVM as a backend, transforming Pseudocode into optimized machine code. Key features include an accessible online editor and debugging support. The performance of the executables generated by the compiler is evaluated through various benchmarks. Future work aims to further improve the compiler's usability and extend its feature set.

Contents

1 Introduction	1
1.1 Pseudocode	1
1.2 Impact Estimation	2
1.3 Summary of sections	3
1.4 Use of generative AI tools	3
2 State of the art	4
2.1 Educational programming languages	4
2.1.1 Scratch	4
2.1.2 Racket	5
2.1.3 Python	6
2.2 Pseudocode	6
3 Theory	8
3.1 Shunting yard algorithm	8
3.2 LLVM	10
3.3 Language Server Protocol	12
3.4 CodeMirror	12
3.5 DWARF format	12
3.6 Pseudocode language	15
3.6.1 Syntax	16
3.6.1.1 Operations	16
3.6.1.2 Statements	17
3.6.1.3 Lists	18
3.6.1.4 EBNF Grammar	18
3.6.2 Sample programs	20
4 Application	22
4.1 Compiler implementation	22
4.1.1 Parsing	22
4.1.1.1 Error messages	23
4.1.1.2 Parsing technique	24
4.1.1.3 Abstract Syntax Tree	24
4.1.1.4 Treatment of diacritics	25
4.1.2 Compilation	25
4.1.2.1 Floating-point number equality comparison	26
4.1.2.2 Variables	26
4.1.2.3 Debug metadata	27
4.1.3 Compiler interface	28
4.2 Accessible online editor	29

4.2.1 Architecture	29
4.2.2 Showcase	30
5 Evaluation	33
5.1 Compiler benchmarks	33
5.2 Compiler tests	34
5.3 Editor backend load testing	35
6 Discussion	36
6.1 Future work	36
6.1.1 Deployment	36
6.1.2 Syntax highlighting	36
6.1.3 LSP server	37
6.1.4 Editor plugin	37
6.1.5 Improved error messages	37
6.1.6 Execution of Pseudocode from a photo	38
6.1.7 Assess impact of compiler use on academic performance	39
7 Conclusions	40
Bibliography	41

Chapter 1

Introduction

This work details the implementation of a compiler for the Pseudocode programming language, together with the requisite theoretical concepts. Here, “Pseudocode” is in reference to the programming language that is part of Romanian high-school level Computer Science education, and the Romanian Computer Science Bacalaureate exam.

In this work, “Pseudocode” (capitalized) will refer to the aforementioned meaning, whereas “pseudocode” (non-capitalized) will be used with the common definition: “a notation resembling a simplified programming language, used in program design”[1].

The primary motivation for the development of this work is to decrease the effort and time investment which is presently required for rigorous verification of Pseudocode programs. This benefits Romanian pupils, teachers, and exam graders alike. Pupils may find it easier to check their work, to discover errors, and to repair said errors, whereas teachers and exam graders may experience a decrease in their workload.

1.1 Pseudocode

When high-school pupils are initially introduced to computer science in Romania, the first programming language they learn is “Pseudocode”. As the name suggests, “Pseudocode” is supposed to be a simple language, suitable for learning basic programming, and introductory algorithms.

```
citește x
i ← 2
┌cât timp  $i*i \leq x$  execută
│ ┌dacă  $x \% i = 0$  atunci
│ │ scrie i
│ │ ┌dacă  $i \neq x/i$  atunci
│ │ │ scrie  $x/i$ 
│ │ └─┐
│ └─┐
│ └─┐
└─┐
  i ← i+1
└─┐
```

Code listing 1: Pseudocode

While this language is not as emphasized in the middle years of high school, it returns in the final year. The Pseudocode language is a significant part of the Computer Science Baccalaureate exam. Students that choose to take this exam necessarily spend a lot of time with the language, as they study and prepare.

Presently, the only viable way for a Pseudocode program to be executed on a computer, is for it to first be manually translated into a compilable/interpretable programming language (e.g., Python, C++, Java), and only then executed, using the chosen language's toolchain. This process is not only tedious, but also error prone. An implementation of Pseudocode can save people's time, while increasing the rigor of the educational system.

1.2 Impact Estimation

The Romanian Baccalaureate comprises a series of 3-4 written exams, which mark the completion of high-school level studies in Romania. One of these exams is in Romanian Language and Literature; this exam is mandatory for everyone. Another is in the compulsory subject of the profile, which the pupil has taken; this exam can be either in mathematics or history.

A third exam is in a subject of the pupil's choosing, but which is relevant to that pupil's profile. For example, pupils whose profile is "Theoretical: Sciences", may choose one of the following subjects: Computer Science, Physics, Biology, or Chemistry.

In 2023, 130,522 Romanian high-school pupils took the Baccalaureate exam. Of those pupils, 32.7% were eligible to take the computer science exam (Figure 1). Of the pupils that were eligible to take the computer science exam, 20.39% chose to do so (Figure 2). This comes out to 8708 pupils taking the computer science exam, that would have been directly impacted by this work, in the year 2023.

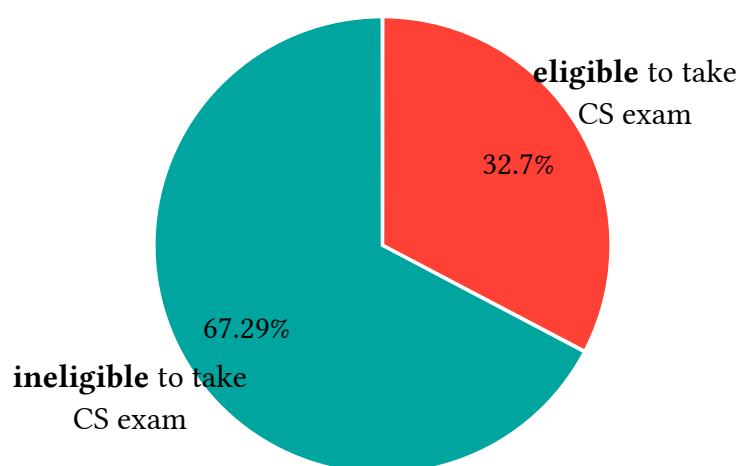


Figure 1: pupils eligible and ineligible to take Computer Science exam, 2023

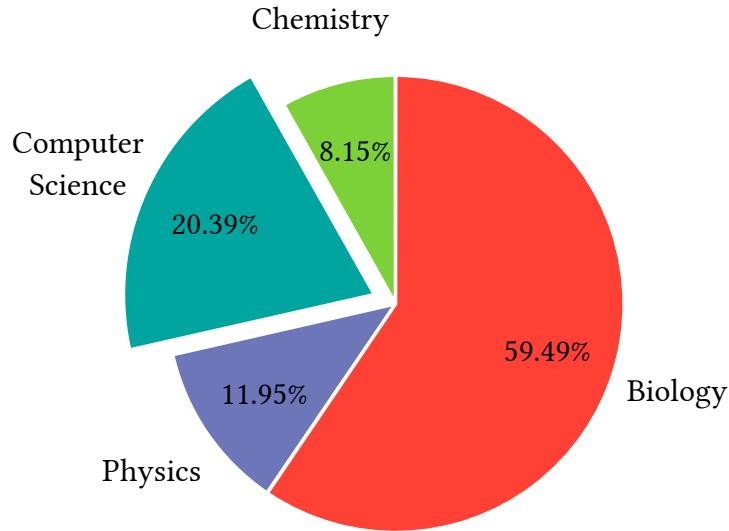


Figure 2: subjects chosen by pupils eligible to take Computer Science, 2023

1.3 Summary of sections

In Section 2 previous work in the domain of educational programming languages is recounted, and compared to this work. Section 3 goes over a series of relevant theoretical and technical concepts, including a description of the Pseudocode language. Section 4 describes the implementation of a Pseudocode compiler, consisting of a parser, and a backend which generates LLVM IR. Additionally, an implementation of an online Pseudocode editor is offered, which aims to be accessible to high-school pupils. Section 5 mainly compares the performance of Pseudocode to industry-standard programming languages. Section 6 presents advantages and disadvantages of the implemented system, as well as suggestions for future work. In Section 7 the author’s final thoughts are presented.

1.4 Use of generative AI tools

During the development of this work, the author used ChatGPT to improve the flow and clarity of the text. After having used this service, the author revised and edited the generated content. The author assumes responsibility for the content of this work.

Chapter 2

State of the art

2.1 Educational programming languages

Educational programming languages aim to assist and encourage people to learn programming and computing concepts. Frequently, though not always, the languages are designed with a particular audience in mind. The concepts that they introduce can be narrow (e.g., the fundamentals of functional programming), as well as broad (e.g., game development with 2D sprites). What follows is an enumeration of such educational programming languages, which have been considered relevant to this work.

2.1.1 Scratch

Scratch[2] is a programming environment that is primarily aimed at introducing children aged 8-16 to coding. Scratch's appeal and success comes from its focus on enabling novice programmers to easily develop interactive, media-rich projects.

Developing such projects in an industry-standard language would be quite difficult for a novice, as they would need to learn and become acquainted with a significant amount of new concepts: programming language syntax, interfacing with audio, video, and input devices using external libraries, possibly having to compile source code, etc.. With their underlying programming environment and the outward facing visual editor (Figure 3), Scratch can achieve almost the same result, but with a significantly lower upfront cost.

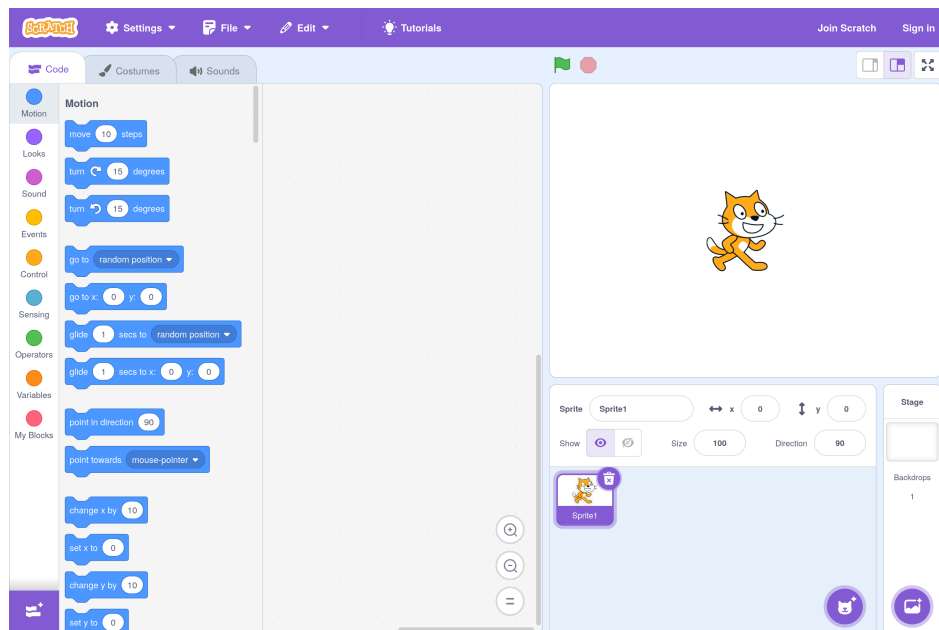


Figure 3: Scratch’s online visual editor

2.1.2 Racket

Racket[3] is a programming language and environment that was borne out of a desire to improve programming education. The language originally started out as a copy of Scheme; gradually, features were added to make it better suited to the authors’ needs: a syntax extension system, mechanisms for creating safe abstractions, etc.. The Racket environment also includes an IDE (Figure 4), with modern features such as syntax highlighting, and debugging. While Racket was designed to be easily approachable for novices, it is mature enough to be suitable for industry use.

An intriguing feature of Racket is its “language levels”. It allows for only a subset of the language to be enabled at a given moment. This permits, for example, for a programming course to start students on a simplified version of Racket, with fewer features being enabled, and for increasingly advanced features to be enabled as the course progresses.

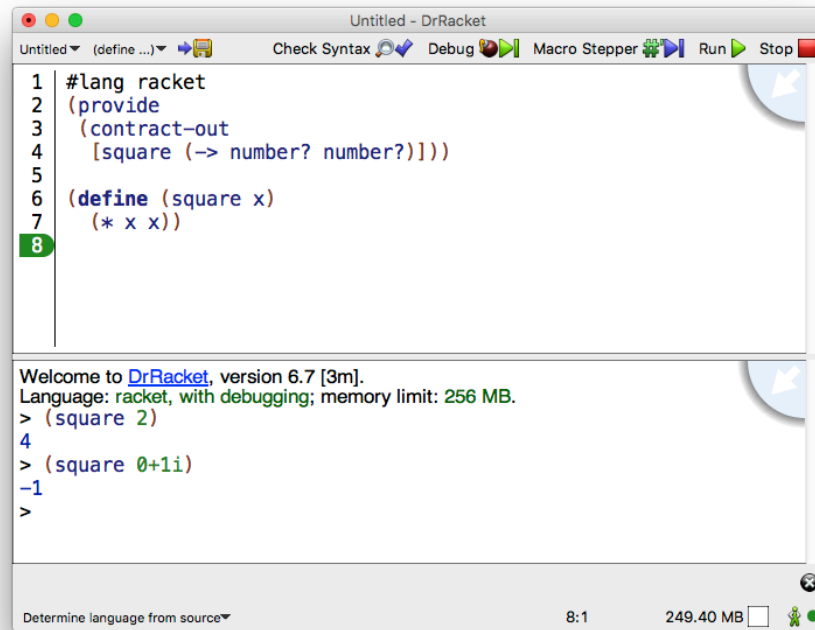


Figure 4: Racket’s IDE[4]

2.1.3 Python

Python is an industry programming language, whose educational potential was recognized as early as 2012[5]. The syntax is intuitive and easy to learn, the type system is forgiving, as the language is dynamically typed. In addition to this is Python’s renown for possessing a wealth of easily installable packages, which enable users to develop complex projects, ranging from GUI applications to web servers.

2.2 Pseudocode

The Pseudocode programming language does not appear to possess an official specification, nor an implementation. It is targeted at a specific audience – high-school level pupils in Romania, and it accomplishes a precise goal – teaching the basics of imperative computer programming, together with some algorithms.

This work takes inspiration from educational programming languages’ focus on convenience and approachability. The online editor which has been implemented (Section 4.2) takes after Scratch’s visual editor, and Racket’s IDE. The syntax of the language was influenced by Python.

Having gone over a number of educational programming languages raises the following question: why develop another language, Pseudocode, considering that so many mature educational languages already exist? The justification is twofold.

First, Pseudocode is better suited for examination purposes, i.e. the Romanian Computer Science Baccalaureate exam. This is a consequence of it being a simple language, with a limited set of features, which does not ever change, and can be quickly learned in its en-

tirety. On the other hand, the aforementioned languages are either constantly receiving updates, or only practically usable on a computer.

The reason why it is undesirable for a language to constantly receive updates, is because it either forces all exam graders to keep up with the stream of updates, lest they misunderstand a section of code which employs a new feature, and lower a pupil's grade; or otherwise, the language has to be restricted to a particular subset of features, which could be difficult to practically implement in a written exam.

Second, it is commonly accepted that governments tend to change and adopt new technology at a slow pace. This means that even if, for example, it was discovered that a particular programming language significantly improved pupils' outcomes in education, it is not guaranteed when or if it would be adopted in the educational system.

Taking these considerations into account, improving the Pseudocode developer experience by creating compilers, debuggers, editors, and other tools is a valid approach to support Romanian high-school students in their computer science education.

Chapter 3

Theory

3.1 Shunting yard algorithm

The shunting yard algorithm is a method for parsing infix expressions¹, first described by Edsger Dijkstra in 1961[6]. The following is an overview and description of the algorithm's function.

The algorithm expects to receive an expression as input; it should be possible for this expression to be split into a list of tokens. Let's consider that each token can belong to one of the following three categories: **operand** (e.g., 42, x), **operator** (e.g., +, *), and **paren** (e.g., (,)). Let's also consider that every **operator** has a certain precedence value, and that operators with higher precedence bind stronger.

Next, it is necessary to describe the apply function. This function computes the result of applying an operation to a pair of operands, for instance applying addition could be defined as $\text{apply}(+, a, b) = a + b$. The formal definition of the function is: $\text{apply} : \text{Operator} \times \text{Operand} \times \text{Operand} \rightarrow \text{Operand}$.

The outline of the algorithm is as follows: the tokens comprising the input expression are iterated, while manipulating an **output** queue, and an **operators** stack. If the current token is an **operand**, it is added to the **output** queue. If it is a left parenthesis, it is pushed onto the **operators** stack.

If the current token is an **operator**, then the following procedure will be performed in a loop: a pair of operands, operand_1 and operand_0 are extracted from the **output** queue, an operator is popped from the **operators** stack, and the result of $\text{apply}(\text{operator}, \text{operand}_0, \text{operand}_1)$ is added back into the **output** queue. This process is repeated, as long as the topmost operator's precedence is greater than the current token's precedence. When the loop finished, the current token is pushed onto the **operators** stack.

When a right parenthesis is encountered, a similar process is carried out, however with some differences: the loop only repeats as long as the topmost operator is not a left paren-

¹The algorithm accepts all valid infix expressions, as well as some expressions which are not in infix form.

thesis, the right parenthesis is not added to the `operators` stack, and its matching left parenthesis must be popped from the stack.

After all the expression's tokens have been iterated, a similar procedure is carried out once again, with some differences: the loop stops only when the operators stack has been emptied, and this time there is no token which should be pushed onto the `operators` stack.

At the end, the `output` stack must contain a single operand, and this operand is the final output of the shunting yard algorithm.

A pseudocode version of the algorithm is offered in Algorithm 1. In this implementation, `out` is the `output` queue, and `ops` is the `operators` stack.

Algorithm 1: Shunting yard algorithm

```

input: expression
output: result of parsing the expression (e.g., a number, an AST)
1  out  $\leftarrow$  queue()
2  ops  $\leftarrow$  stack()
3  for token in expression.tokens() do
4    if token.type() = "operand" then
5      out.enqueue(token)
6    if token.type() = "operator" then
7      while ops.len() > 0 and ops.top()  $\neq$  "(" and token.precedence() < ops-
8        s.top().precedence() do
9        operator  $\leftarrow$  ops.pop()
10       operand1  $\leftarrow$  out.dequeue()
11       operand0  $\leftarrow$  out.dequeue()
12       out.enqueue(apply(operator, operand0, operand1))
13     ops.push(token)
14   if token = "(" then
15     ops.push(token)
16   if token = ")" then
17     while ops.len() > 0 and ops.top()  $\neq$  "(" do
18       operator  $\leftarrow$  ops.pop()
19       operand1  $\leftarrow$  out.dequeue()
20       operand0  $\leftarrow$  out.dequeue()
21       out.enqueue(apply(operator, operand0, operand1))
22     ops.pop()
23   while ops.len()  $\geq$  1
24     operator  $\leftarrow$  ops.pop()
25     assert operator  $\neq$  "("
26     operand1  $\leftarrow$  out.dequeue()
27     operand0  $\leftarrow$  out.dequeue()

```

```
27   out.enqueue(apply(operator, operand0, operand1))
28   assert ops.len() = 0
29   assert out.len() = 1
30   result ← out.dequeue()
31   return result
```

3.2 LLVM

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies[7].

Underpinning much of the project is LLVM Intermediate Representation, commonly referred to as LLVM IR. Put shortly, LLVM IR is a high-level assembly language. It does not directly expose CPU registers; instead it is an SSA, or static single assignment form, meaning that it consists of variable assignments, where each variable is assigned to exactly once.

Many of the programming languages that target machine code can be cleanly made to target LLVM IR. Some examples are: C and C++ (**clang** compiler), Rust, Swift, Haskell, Zig, among others.

LLVM's libraries expose procedures for constructing LLVM IR, optimizing it, and converting it to an object or executable file, supporting a wide range of target platforms. Emitting LLVM IR allows a compiler developer to focus on improving and polishing their language, while having the time-consuming process of writing routines for optimization and machine code generation handled by LLVM.

For an example of LLVM IR, compiling the C program in Code listing 2 with **clang**², LLVM's C compiler, produces the LLVM IR at Code listing 3.

```
#include <stdio.h>

int main(int argc, char **argv) {
    int a, b;
    scanf("%d %d", &a, &b);

    int c = a+b;
    printf("%d + %d = %d", a, b, c);

    return 0;
}
```

Code listing 2: Simple C program

```
; ModuleID = 'main.c'
source_filename = "main.c"
```

²**clang** version 17.0.6 was used.

```

target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [6 x i8] c"%d %d\00", align 1
@.str.1 = private unnamed_addr constant [13 x i8] c"%d + %d = %d\00", align 1

; Function Attrs: nounwind sspstrong uwtable
define i32 @main(i32 noundef %0, ptr nocapture noundef readnone %1) local_unnamed_addr
#0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    call void @llvm.lifetime.start.p0(i64 4, ptr nonnull %3) #4
    call void @llvm.lifetime.start.p0(i64 4, ptr nonnull %4) #4
    %5 = call i32 @__isoc99_scanf(ptr noundef nonnull @.str, ptr noundef
nonnull %3, ptr noundef nonnull %4)
    %6 = load i32, ptr %3, align 4, !tbaa !4
    %7 = load i32, ptr %4, align 4, !tbaa !4
    %8 = add i32 %7, %6
    %9 = call i32 @__printf_chk(i32 noundef 1, ptr noundef nonnull
@.str.1, i32 noundef %6, i32 noundef %7, i32 noundef %8) #4
    call void @llvm.lifetime.end.p0(i64 4, ptr nonnull %4) #4
    call void @llvm.lifetime.end.p0(i64 4, ptr nonnull %3) #4
    ret i32 0
}

; Function Attrs: mustprogress nocallback nofree nosync nounwind willreturn
memory(argmem: readwrite)
declare void @llvm.lifetime.start.p0(i64 immarg, ptr nocapture) #1

; Function Attrs: nofree nounwind
declare noundef i32 @__isoc99_scanf(ptr nocapture noundef readonly, ...)
local_unnamed_addr #2

declare i32 @__printf_chk(i32 noundef, ptr noundef, ...) local_unnamed_addr #3

; Function Attrs: mustprogress nocallback nofree nosync nounwind willreturn
memory(argmem: readwrite)
declare void @llvm.lifetime.end.p0(i64 immarg, ptr nocapture) #1

attributes #0 = { nounwind sspstrong uwtable "min-legal-vector-width"="0" "no-trapping-
math"="true" "stack-protector-buffer-size"="4" "target-cpu"="x86-64" "target-
features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { mustprogress nocallback nofree nosync nounwind willreturn
memory(argmem: readwrite) }
attributes #2 = { nofree nounwind "no-trapping-math"="true" "stack-protector-buffer-
size"="4" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,
+x87" "tune-cpu"="generic" }
attributes #3 = { "no-trapping-math"="true" "stack-protector-buffer-size"="4" "target-
cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-
cpu"="generic" }
attributes #4 = { nounwind }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 2}
!3 = !{!"clang version 17.0.6"}
!4 = !{!5, !5, i64 0}

```



```
!5 = !{"int", !6, i64 0}
!6 = !{"omnipotent char", !7, i64 0}
!7 = !{"Simple C/C++ TBA"}
```

Code listing 3: LLVM IR generated for Code listing 2

3.3 Language Server Protocol

The Language Server Protocol's (LSP) primary goal, is to decouple the program that analyzes source code, and implements code editing features such as: symbol renaming, auto-complete, go to definition, etc., i.e. the language server, from the text editor, i.e. the language server's client.

Instead of every editor having to implement intricate analysis logic for every programming language that it desires to support, it is sufficient to support the Language Server Protocol, for it to be able to provide the IDE features that users expect today.

Language server implementations exist for most of the programming languages which are actively used today: C, C++, TypeScript, Python, etc..

3.4 CodeMirror

CodeMirror³ is a modern, robust, and extensible text editor library, that is created for the web. It is implemented on top of the comparatively simple building blocks provided by web standards, e.g., the `<textarea>` element, the event handling systems.

CodeMirror supports syntax highlighting, customizable keybindings, as well as extensions which bring features such as LSP support. It exposes a rich and flexible API, allowing for the editor to be custom-fit to specific use-cases.

```
1 ✓ let person = {
2   firstName: "John",
3   lastName: "Doe",
4   age: 30,
5 ✓   fullName: function() {
6     return this.firstName + " " + this.lastName;
7   }
8 };
9 console.log(person.firstName, person.fullName());
```

Figure 5: JavaScript source code in CodeMirror editor

3.5 DWARF format

DWARF⁴ is a standardized format[8] for embedding debugging metadata into an executable. It encodes various information which is helpful for debugging purposes, for example: mappings between source code lines and machine code instructions, mappings between a stack address, and the corresponding variable's name, type, and source line on which it was declared, etc..

Structurally, DWARF metadata consists of Debugging Information Entries, or DIEs[8, Section 2.1]. Each DIE is defined by a tag name (e.g., `DW_TAG_subprogram`, `DW_TAG_variable`),

³Accessible at <https://github.com/codemirror/dev/>.

⁴DWARF is an acronym for "Debugging With Arbitrary Record Formats".

and a set of attributes. The attributes[8, Section 2.2] store various information which is relevant to their tag, for example: the source line from which a particular DIE was generated (`DW_AT_decl_line`), the name of the variable, type, or function that the DIE is describing (`DW_AT_name`), etc..

Each DIE may optionally have a list of child DIEs[8, Section 2.3], which gives the metadata a tree structure. Child DIEs can serve various purposes, including but not limited to, defining the members of a struct (`DW_TAG_member`), or the variables declared in a function (`DW_TAG_variable`).

Code listing 5 is an example of DWARF metadata generated for the program in Code listing 4.

```
struct Student {
    char *name;
    int age;
    float gpa;
};

int main() {
    struct Student student = {
        .name = "John Doe",
        .age = 20,
        .gpa = 9.75,
    };

    return 0;
}
```

Code listing 4: Simple C Program

```
target/main: file format elf64-x86-64

.debug_info contents:
0x00000000: Compile Unit: length = 0x00000081, format = DWARF32, version = 0x0005,
unit_type = DW_UT_compile, abbr_offset = 0x0000, addr_size = 0x08 (next unit at
0x00000085)

0x0000000c: DW_TAG_compile_unit
           DW_AT_producer ("clang version 17.0.6")
           DW_AT_language (DW_LANG_C11)
           DW_AT_name ("main.c")
           DW_AT_str_offsets_base (0x00000008)
           DW_AT_stmt_list (0x00000000)
           DW_AT_comp_dir (".")
           DW_AT_low_pc (0x0000000000001130)
           DW_AT_high_pc (0x0000000000001133)
           DW_AT_addr_base (0x00000008)

0x00000023: DW_TAG_variable
           DW_AT_type (0x0000002a "char[9]")
           DW_AT_decl_file ("./main.c")
           DW_AT_decl_line (9)
```

```

0x0000002a: DW_TAG_array_type
              DW_AT_type (0x00000036 "char")

0x0000002f: DW_TAG_subrange_type
              DW_AT_type (0x0000003a "__ARRAY_SIZE_TYPE__")
              DW_AT_count (0x09)

0x00000035: NULL

0x00000036: DW_TAG_base_type
              DW_AT_name ("char")
              DW_AT_encoding (DW_ATE_signed_char)
              DW_AT_byte_size (0x01)

0x0000003a: DW_TAG_base_type
              DW_AT_name ("__ARRAY_SIZE_TYPE__")
              DW_AT_byte_size (0x08)
              DW_AT_encoding (DW_ATE_unsigned)

0x0000003e: DW_TAG_subprogram
              DW_AT_low_pc (0x0000000000001130)
              DW_AT_high_pc (0x0000000000001133)
              DW_AT_frame_base (DW_OP_reg7 RSP)
              DW_AT_call_all_calls (true)
              DW_AT_name ("main")
              DW_AT_decl_file ("./main.c")
              DW_AT_decl_line (7)
              DW_AT_type (0x00000056 "int")
              DW_AT_external (true)

0x0000004d: DW_TAG_variable
              DW_AT_name ("student")
              DW_AT_decl_file ("./main.c")
              DW_AT_decl_line (8)
              DW_AT_type (0x0000005a "Student")

0x00000055: NULL

0x00000056: DW_TAG_base_type
              DW_AT_name ("int")
              DW_AT_encoding (DW_ATE_signed)
              DW_AT_byte_size (0x04)

0x0000005a: DW_TAG_structure_type
              DW_AT_name ("Student")
              DW_AT_byte_size (0x10)
              DW_AT_decl_file ("./main.c")
              DW_AT_decl_line (1)

0x0000005f: DW_TAG_member
              DW_AT_name ("name")
              DW_AT_type (0x0000007b "char *")
              DW_AT_decl_file ("./main.c")
              DW_AT_decl_line (2)
              DW_AT_data_member_location (0x00)

0x00000068: DW_TAG_member
              DW_AT_name ("age")
              DW_AT_type (0x00000056 "int")
              DW_AT_decl_file ("./main.c")

```

```

DW_AT_decl_line (3)
DW_AT_data_member_location (0x08)

0x00000071: DW_TAG_member
            DW_AT_name ("gpa")
            DW_AT_type (0x00000080 "float")
            DW_AT_decl_file ("./main.c")
            DW_AT_decl_line (4)
            DW_AT_data_member_location (0x0c)

0x0000007a: NULL

0x0000007b: DW_TAG_pointer_type
            DW_AT_type (0x00000036 "char")

0x00000080: DW_TAG_base_type
            DW_AT_name ("float")
            DW_AT_encoding (DWATE_float)
            DW_AT_byte_size (0x04)

0x00000084: NULL

```

Code listing 5: DWARF metadata generated for Code listing 4

The DWARF format is supported by an extensive list of compilers (e.g. the GNU Compiler Collection, the LLVM compiler suite) and debuggers (e.g., the GNU Debugger `gdb`, the LLVM Debugger `lldb`). If one is writing a compiler, and desires for it to produce a debuggable executable, emitting DWARF metadata is an effective way to achieve this goal.

3.6 Pseudocode language

This section outlines the syntax and properties of the Pseudocode language which has been implemented as part of this work. This language is similar, but not the same, as the language used in Romanian computer science education, which will be referred to as “Baccalaureate Pseudocode”. Most of the changes serve the goal of making the language easier to type on a keyboard. Some of the principal differences between Baccalaureate Pseudocode and Pseudocode can be observed in Code listing 6.

While Baccalaureate Pseudocode relies on box-drawing characters to delimit code blocks, the version developed in this work uses indentation, similar to the Python language. Characters such as `←` and `≤` (among others) are approximated with easier to type alternatives, `<-` and `<=` respectively.

```

citește x
i ← 2
cât timp i*i ≤ x execută
|   dacă x % i = 0 atunci
|   |   scrie i
|   |   dacă i ≠ x/i atunci
|   |   |   scrie x/i
|   |   ■
|   ■
|   i ← i+1
■

```

```

citește x
i <- 2
cât timp i*i <= x execută
    dacă x % i = 0 atunci
        scrie i
        dacă i != x/i atunci
            scrie x/i
    i <- i+1

```

Code listing 6: structurally equivalent Bacculaureate Pseudocode (left) and Pseudocode (right)

The paradigm that best describes Pseudocode is imperative programming. The language supports the most common control-flow structures: if-else statements, while loops, for loops, repeat ... until loops. It does not support user-defined functions nor classes.

In Bacculaureate Pseudocode, all variables are floating point numbers. The Pseudocode implemented in this work expands the language slightly, by introducing an additional type, lists of floating point numbers (further described in Section 3.6.1.3). Lists-adjacent functionality may be disabled, so that the compiler will more closely match Bacculaureate Pseudocode.

3.6.1 Syntax

3.6.1.1 Operations

Table 1 lists all the operations present in Pseudocode, with their description, precedence (higher binds stronger), type of input and output values, and arity.

Most of the operations are common to all modern programming languages. There are two that stand out, however: $[x]$ which computes the whole part of x (e.g., $[3.14] = 3$), and $x|y$ which checks whether x divides y (e.g., $3|6$ is true, but $3|7$ is false).

Operation	Description	Precedence	Input	Output	Arity
$[x]$	whole part	7	float	float	unary
$+x$	identity	7	float	float	unary
$-x$	negation	7	float	float	unary
$x*y$	multiplication	6	float	float	binary
x/y	division	6	float	float	binary
$x\%y$	remainder	6	float	float	binary
$x+y$	addition	5	float	float	binary
$x-y$	subtraction	5	float	float	binary
$x=y$	check equality	4	float	bool	binary
$x\neq y$	check inequality	4	float	bool	binary

$x < y$	check less-than	4	float	bool	binary
$x > y$	check greater-than	4	float	bool	binary
$x \leq y$	check less-than or equal	4	float	bool	binary
$x \geq y$	check greater-than or equal	4	float	bool	binary
$x y$	check x divides y	4	float	bool	binary
$x \text{ și } y$	logical "and"	3	bool	bool	binary
$x \text{ sau } y$	logical "or"	2	bool	bool	binary

Table 1: pseudocode operations

3.6.1.2 Statements

Assignment is denoted with `{variable} <- {value}`, and swapping of two variables with `{left} <-> {right}` (Code listing 7). Console input and output is achieved with `citește {variable}` and `scrie {value}` respectively (Code listing 8).

```
x <- 1
y <- 2
x <-> y
```

Code listing 7: variable assignment and swapping

```
citește x
scrie "x=", x
```

Code listing 8: console input and output

If-else statements are denoted with `dacă {condition} atunci ... altfel ...` (Code listing 9), while loops with `cât timp {condition} execută ...` (Code listing 10), repeat ... until loops with `repetă ... până când {condition}` (Code listing 11), and for loops with `pentru i<-{start},{stop},{increment} execută ...` (Code listing 12).

```
x <- 10
dacă x < 5 atunci
    scrie "x<5"
altfel
    scrie "x>=5"
```

Code listing 9: if-else statement

```
x <- 0
cât timp x < 10 execută
    x <- x+1
```

Code listing 10: while loop

```
x <- 10
repetă
```

```
x <- x-1
până când x <= 0
```

Code listing 11: repeat ... until loop

```
pentru i <- 1,10,2 execută
    scrie i

pentru i <- 1,10 execută
    scrie i
```

Code listing 12: for loop; specifying an increment is optional, and its default value is 1

3.6.1.3 Lists

List syntax and behavior was developed as part of this work, and is not part of Baccalaurate Pseudocode. The decision to implement lists was motivated by the desire to enable pupils to study a wider variety of algorithms (e.g., list sorting), using the already mostly familiar syntax of Pseudocode.

A list variable can be defined using `{variable} <- {value0}, {value1}, ...`, and an empty list uses the special syntax of `{variable} <- ,`. Lists can be indexed using square brackets, i.e. `list[{index}]`, and a list's length can be determined using the `lungime(list)` built-in function. Values may be inserted into lists using `inserează list, {index}, {value}`, and removed using `șterge list, {index}`.

All the aforementioned syntax is demonstrated in Code listing 13 and Code listing 14.

```
list <- 1,2,3,4,5
pentru i<-0, lungime(list)-1 execută
    list[i] <- list[i]+1
    scrie list[i]
```

Code listing 13: creating a list; getting the list's length; getting and setting elements of a list

```
list <- ,
pentru i<-1,10 execută
    inserează list,i-1,i
cât timp lungime(list) > 0 execută
    șterge list, lungime(list)-1
```

Code listing 14: creating an empty list; inserting elements into, and removing elements from the list

3.6.1.4 EBNF Grammar

Listing 1 describes the grammar of the Pseudocode language in Extended Backus-Naur Form. INDENT and DEDENT are special symbols, signifying an increase and decrease in indentation level by one, respectively. The rest of the symbols are defined in Table 2.

Symbol	Regular Expression ⁵	Description
ID	<code>[^\d\W][\w]*</code>	identifier
FLOAT_LIT	<code>\d+(\.\d*)?</code>	floating-point literal
STRING_LIT	<code>"[^"]*"</code>	string literal
CHAR_LIT	<code>'.'</code>	character literal
NEWLINE	<code>\n</code>	newline

Table 2: regular expressions for symbols

```

FloatBinop = "+" | "-" | "*" | "/" | "%".
FloatUnop = "+" | "-".
FloatExpr =
    FLOAT_LIT
    | ID
    | ID "[" FloatExpr "]"
    | FloatUnop FloatExpr
    | FloatExpr FloatBinop FloatExpr
    | "[" FloatExpr "]"
    | "(" FloatExpr ")"
    | ID "(" FloatExpr ")".

BoolFloatBinop = "=" | "!=" | "<" | ">" | "<=" | ">=" | "|".
BoolBoolBinop = "sau" | "și".
BoolExpr =
    FloatExpr BoolFloatBinop FloatExpr
    | BoolExpr BoolBoolBinop BoolExpr.

ListLitRest = FloatExpr [ "," [ ListLitRest ] ].
ListLit = "," | FloatExpr "," ListLitRest.

InstrAtribuireParam = FloatExpr | ListLit.
InstrAtribuire = ID "<-" InstrAtribuireParam.

ScrieParam =
    FloatExpr
    | STRING_LIT
    | CHAR_LIT.
InstrScrie = "scrie" ScrieParam { "," ScrieParam }.

Lvalue = ID | ID "[" FloatExpr "]"
InstrInterschimbare = Lvalue "<->" Lvalue.
InstrCiteste = "citește" Lvalue { "," Lvalue }.

InstrInsereaza = "inserează" ID "," FloatExpr "," FloatExpr.
InstrSterge = "șterge" ID "," FloatExpr.

```

⁵The regular expression syntax is Rust-flavored. Hence, `\d` matches any digit, `\w` matches any word character (including any unicode letter, and also digits), and `\W` matches any non-word character.


```

Bloc = NEWLINE INDENT { InstrLine } DEDENT.
InstrDaca = "dacă" BoolExpr "atunci" Bloc [ "altfel" Bloc ].
InstrCatTimp = "cât timp" BoolExpr "execută" Bloc.
InstrPentru =
    "pentru" ID "<-" FloatExpr "," FloatExpr
    [ "," FloatExpr ] "execută" Bloc.
InstrRepetă = "repetă" Bloc "până când" BoolExpr.

InstrRepeatable =
    InstrAtribuire
    | InstrInterschimbare
    | InstrScrie
    | InstrCiteste
    | InstrCiteste
    | InstrSterge.
Instr =
    InstrRepeatable { ";" InstrRepeatable }
    | InstrDaca | InstrCatTimp | InstrPentru | InstrRepetă.
InstrLine = Instr NEWLINE.

```

Listing 1: grammar of Pseudocode language in EBNF

3.6.2 Sample programs

Despite being a comparatively simple language, Pseudocode possesses enough complexity to support the implementation of algorithms which are relevant to pupils' education. Two sample programs are presented to illustrate this: Code listing 15 performs bubble sort on a list of numbers, and Code listing 16 computes $\sin(x)$ using Taylor expansion.

```

list <- 53, 34, 12, 665, 34, 23, 54, 65, 123, 65

pentru i<-0, lungime(list)-1-1 executa
    pentru j<-0, lungime(list)-i-1-1 executa
        daca list[j] > list[j+1] atunci
            list[j] <-> list[j+1]

pentru i<-0, lungime(list)-1 executa
    scrie list[i]

```

Code listing 15: bubble sort implementation in Pseudocode

```

citește x
gata <- 0
r <- 0
t <- x
n <- 1
cât timp gata = 0 execută
    r <- r + t

    s <- -1 * ((x*x) / ((2*n)*(2*n+1))) * t
    dacă s = t atunci

```

```
gata <- 1
t <- s
n <- n + 1
scrie r
```

Code listing 16: approximation of $\sin(x)$ by way of Taylor expansion, implemented in Pseudocode

Chapter 4

Application

4.1 Compiler implementation

The Rust programming language was chosen for the implementation of the compiler. This decision was made based on a number of Rust's features, which were deemed to be desirable for this task.

First, Rust's strong memory safety guarantees allow for safe and performant manipulation of strings, without needing to resort to excessive copying. This benefits the implementation of the parser, which involves a significant amount of string processing.

Second, Rust's type system helps improve the code structure, and to catch and prevent bugs, especially in a large codebase. This contributes toward ensuring the correctness of the code, which is paramount in a compiler.

Third, sum types, which are present in the Rust language, significantly aided certain parts of the implementation. Parsing was made easier by `Option<>` and `Result<>`, AST nodes could be conveniently defined and represented using sum types, etc..

4.1.1 Parsing

The entire source file is parsed in a single pass, using a hand-written parser, without any tokenization being performed beforehand. The primary reason for this decision, is that the parsing of some tokens is context-sensitive. For instance, the combination of characters `<-` could be tokenized either as a `LEFT_ARROW`, or as a `LESS_THAN` and `MINUS`, depending on the source code that precedes it. This ambiguity can be seen in Code listing 17.

```
a <- 42
```

```
dacă a<-42 atunci  
    scrie "Salut, Lume!"
```

Code listing 17: tokenization of `<-` is context-sensitive

Other aspects of the Pseudocode language, namely its use of indentation for code blocks, as well as the presence of spaces in some of its keywords (e.g., `cât timp`, `până când`), also influenced the decision to hand-write the parser.

4.1.1.1 Error messages

Error messages can be considered the user interface of a compiler. Their purpose is to assist developers with transitioning their program from an invalid state into a valid one. They appear to play an especially important role for novice programmers: it takes fewer tries for them to bring their program into a valid state, when they're using a compiler which produces higher quality error messages[9].

The desire for the compiler to produce good error messages was another factor that contributed to the decision of hand-writing the parser. The additional flexibility that a hand-written parser affords, allows for the cause of an error to be more accurately determined, which in turn leads to better error messages.

In the implementation, when a parsing error is encountered, compilation is completely halted. Thereafter, a message specifying the error, along with the line and column on which it occurred, is emitted.

The rest of this section will present a few sample error messages, along with the source code that produced them. The sample error messages will be accompanied by comments, pointing out noteworthy design decisions, implementation details, etc..

Error sample 1 presents a case where an error message could be enhanced, thanks to the parser being hand-written. According to the grammar (Section 3.6.1.4), "dacă" should be followed by a boolean expression.

In this specific case, however, it is clear that the user was trying to assign the value 41 to a variable named "dacă". If the compiler merely produced an error stating that it had failed to parse a boolean expression, it would be of little help for users trying to resolve the issue.

The parser possesses special-case code to handle such situations gracefully, by checking whether the user is trying to assign a value to a keyword.

Code

```
dacă <- 41
```

Error

```
[1:7] Eroare la parsare: nume de  
variabilă nevalid „dacă”.
```

When a left parenthesis is missing its matching right parenthesis (Error sample 2), or a right parenthesis is missing its matching left parenthesis (Error sample 3), the compiler helpfully indicates the location of the parenthesis in question, aiding the user with locating and resolving the issue.

Code

```
scrie (40+1
```

Error

```
[1:6] Eroare la parsare:  
paranteza stângă nu are paranteză  
dreaptă corespunzătoare.
```

Code

```
scrie 40+1)
```

Error

```
[1:10] Eroare la parsare:
paranteza dreaptă nu are
paranteză stângă corespunzătoare.
```

Despite the language being dynamically typed, some type errors can be checked at compile-time, as can be observed in Error sample 4 and Error sample 5.

Code

```
daca 41 atunci
  scrie "da!"
```

Error

```
[1:5] Eroare la parsare: tipul
acestei valori ar fi trebuit să
fie unul dintre următoarele: o
condiție.
```

Code

```
x <- 41 + (1 = 1)
```

Error

```
[1:10] Eroare la parsare: tipul
acestei valori ar fi trebuit să
fie unul dintre următoarele: un
număr.
```

4.1.1.2 Parsing technique

Most of the parsing is done in an “incremental” style, i.e. using small functions, where each function parses a specific element of the syntax: an ident, a “dacă” statement, an arrow, etc..

Expressions are parsed using an altered version of the shunting yard algorithm (Section 3.1). Where the unaltered version accepts all infix expressions (e.g., `1 + 2`), as well as some expressions which are not infix (e.g., `1 2 +`), the altered version only accepts infix expressions.

This was accomplished by introducing a variable that keeps track of the types of tokens that the parser expects to encounter next. In the beginning an operand is expected; after an operand is read, an operator is expected, and so on. If the parser encounters a token that does not match its expectations, an error is thrown. This alteration ensures the parser accepts only valid infix expressions, while rejecting sequences such as `1 2 +`.

4.1.1.3 Abstract Syntax Tree

Parsing Code listing 18 produces an AST resembling the one in Figure 6 (with minor simplifications for clarity and brevity).

```
citește x
scrie "Am citit: ", x
dacă x > 42 atunci
  scrie "da!"
```

Code listing 18: simple code sample

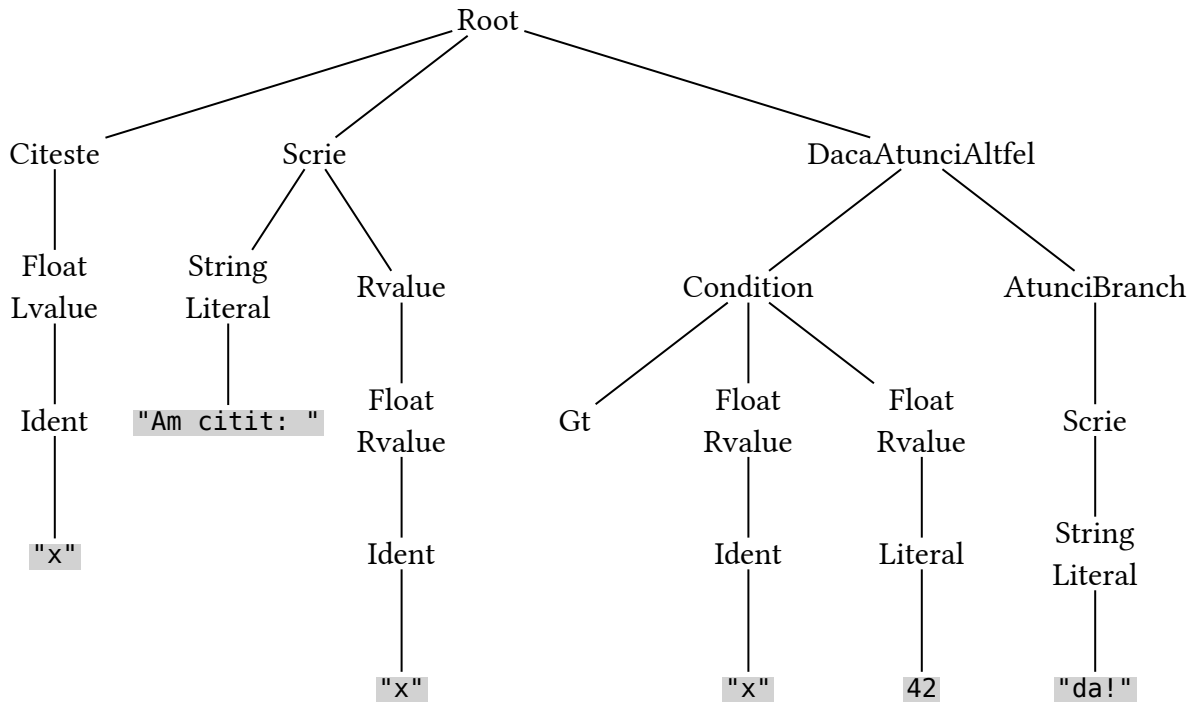


Figure 6: AST (slightly simplified) generated for Code listing 18

4.1.1.4 Treatment of diacritics

To ease the process of writing Pseudocode programs, the parser considers diacritics in keywords to be optional. Diacritics are symbols placed above or below a character, for example: the comma in “ș”, the breve in “ă”, or the circumflex in “â”. Both code samples in Code listing 19 will produce the same AST.

```

citește x
cât timp x > 0 execută
x <- x-1
  
```

```

citeste x
cat timp x > 0 executa
x <- x-1
  
```

Code listing 19: diacritics in keywords are optional

4.1.2 Compilation

LLVM (Section 3.2) was chosen for implementing the compiler’s backend⁶. Hence, it is responsible for optimizing the source code, and translating it into machine code.

Once the parsing process is finished, it outputs an AST of the entire program. This AST is recursively traversed, and converted to LLVM IR. Once the conversion is complete, the IR is optimized, and used to generate an object file. Subsequently, the `clang` command is used to link the object file with system libraries (e.g., `libc`, `libm`), and with `libpseudo_sys`, a library which was purpose-built to implement features of the Pseudocode language. The end result is an executable file.

⁶More specifically, Inkwell (<https://github.com/TheDan64/inkwell>), a Rust wrapper over the LLVM C library, was used.

The `libpseudo_sys` library primarily exposes procedures that manipulate lists: creating a list, cloning it, freeing its memory, getting, setting, inserting, and removing elements, as well as retrieving its length.

4.1.2.1 Floating-point number equality comparison

One issue that arises when the only numeric type is floating-point numbers, is verifying whether two numbers are equal. An experienced programmer will know to use an ε threshold value, and to perform a comparison between two floating-point numbers x and y using a formula such as $|x - y| < \varepsilon$.

However Pseudocode is not aimed at experienced programmers. On the contrary, it is supposed to assist high-school students, who have just started to learn to program. Taking this into consideration, the compiler was made to always automatically generate an epsilon-thresholded check, when an equality comparison is performed.

4.1.2.2 Variables

At any particular point in a program's execution, a variable may have one of two possible types: either a floating point number, or a list of floating point numbers. There is a number of ways in which such behavior can be implemented.

In this work, a “discriminated union” approach was taken. Variables are represented by structs, which are composed of two fields: a discriminator enum, and a union over all the possible values the variable could take. A version of such a data structure, implemented in Rust pseudocode, is offered in Code listing 20.

```
enum Discriminator {  
    Float,  
    List,  
}  
union Variant {  
    float: f64,  
    list: List,  
}  
struct Variable {  
    kind: Discriminator,  
    variant: Variant,  
}
```

Code listing 20: structure of discriminated union, described with Rust pseudocode

At runtime, the type of a value can be determined by inspecting its discriminator. When necessary, type checks can be generated, by comparing the discriminant with the expected type, and throwing an error in case it does not match (such as can be observed in Error sample 6).

Code

```
list <- ,  
x <- list+1
```

Error

```
[2:5] Eroare: valoarea are tipul  
„număr”, însă ar fi trebuit să fi  
avut tipul „listă”.
```

Error sample 6: Runtime type error

4.1.2.3 Debug metadata

As it generates LLVM IR, the compiler also annotates it with debugging metadata. The metadata includes details such as: mapping from line of source code to line of generated LLVM IR, the names and types of variables, etc.. This metadata enables LLVM to generate and embed DWARF (Section 3.5) debugging information into the object file it generates. This allows for the final executable to be debugged at source level, using any debugger that supports the DWARF format.

To permit for the compiler to generate debug metadata matching source code lines to generated LLVM IR, it is necessary for some of the nodes in the AST to be annotated with the source line of code which generated them. This information is computed and included into the nodes during the parsing process.

An example of a Pseudocode executable being debugged using the `gdbgui` debugger is presented in Figure 7. The debugger utilizes the DWARF metadata, to display the current line of source code (instead of displaying the current machine code instruction), in addition to showing the local variables, and their values.

One noteworthy aspect which can be observed in Figure 7, is that the user must manually determine which variant of a variable is the one that is currently active, based on the discriminant. This is undesirable, as it is unintuitive, introduces additional work, and is error-prone. Discriminated union annotations are supported by DWARF, and by LLVM's underlying implementation, however LLVM's C library, on which the compiler depends, unfortunately does not expose this functionality.

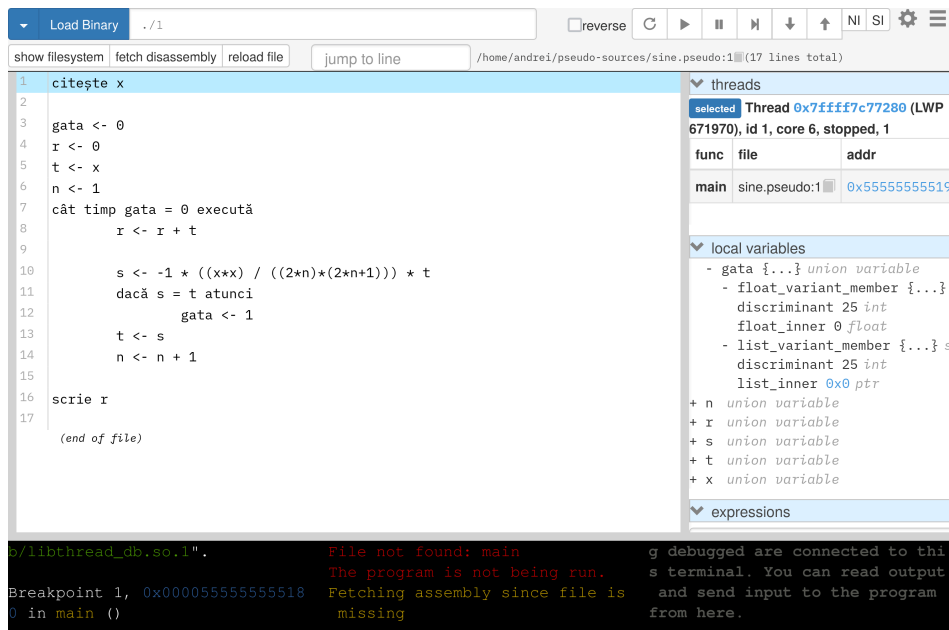


Figure 7: Pseudocode executable loaded in `gdbgui` debugger

After loading an executable in a debugger, the user may inspect the values of variables, and step through the code.

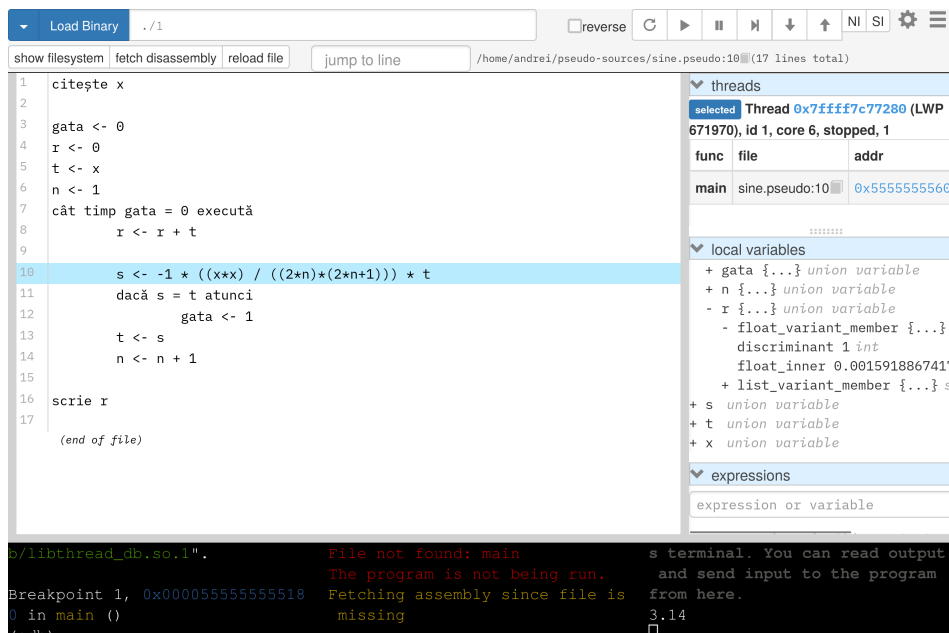


Figure 8: stepping through the executable from Figure 7

4.1.3 Compiler interface

The compiler is invoked from the command line. Parameters such as the program source file, the path to which the compiler's output will be written, the optimization level, etc. are all specified with command-line arguments.

```
pseudo-cli \
--lib-path $PATH_TO_LIBPSEUDO_SYS_S0 \
```

```
--opt default \  
./program.pseudo --executable ./program
```

Code listing 21: compiling the `program.pseudo` source file into the `program` executable, using the compiler's command-line interface

4.2 Accessible online editor

Having to invoke the compiler from the command-line would deter and prevent a significant amount of novices from actually using Pseudocode. It would also present potential distribution challenges; the compiler and its installer would have to be thoroughly tested and reliably packaged for the Windows operating system.

On the other hand, a specialized online Pseudocode editor would be free of such issues. It could wrap the compiler in a sleek, simple interface, perfectly understandable by novices. It would also resolve the potential distribution problems, as the compiler would only have to function on a predetermined set of servers, which would be running a single operating system, whose state would be known.

Taking into account the aforementioned considerations, the decision was made to implement such an online editor for Pseudocode.

4.2.1 Architecture

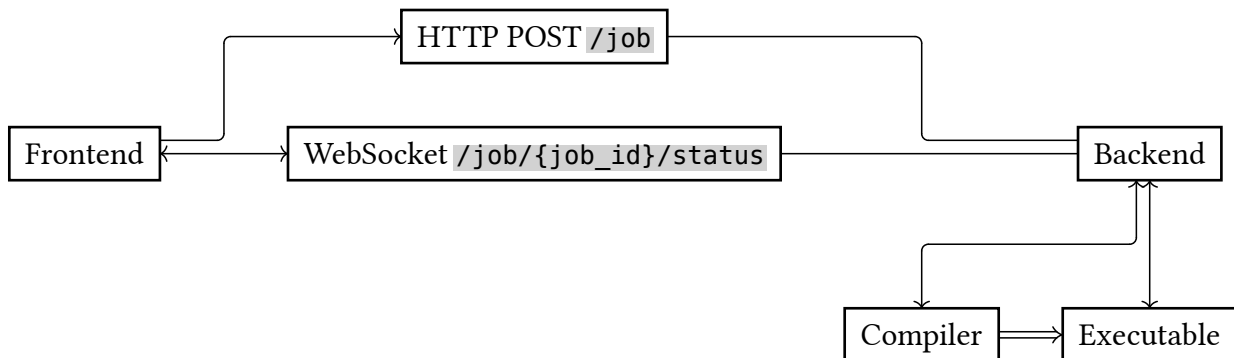


Figure 9: architecture of the online editor

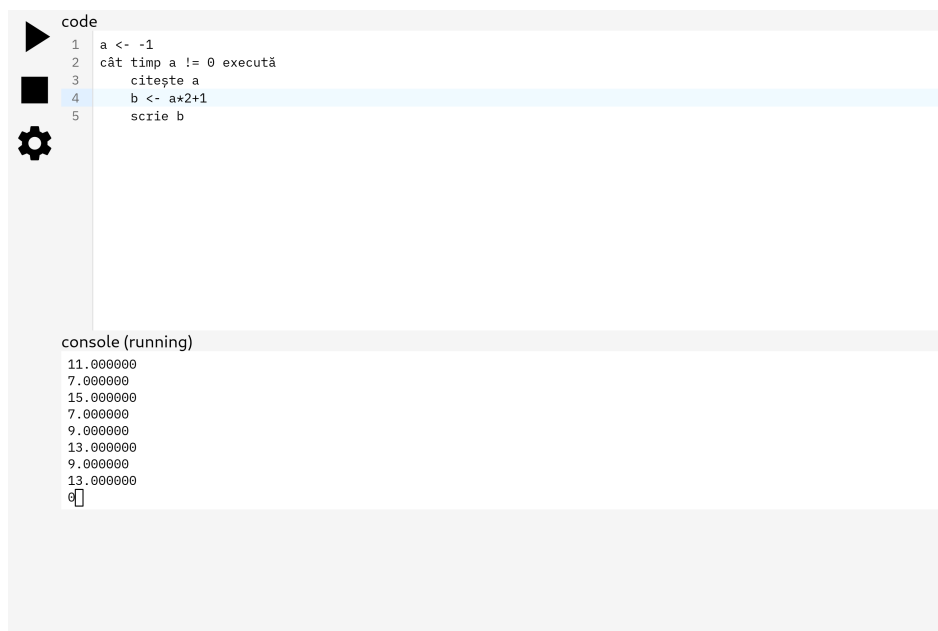
The editor consists of a frontend and backend component. The frontend is a web page containing a pair of stacked panels: a code editor, followed by an output “terminal”; it also contains buttons for running the code that has been written, for stopping the executable that is currently running, as well as a settings menu. This component was implemented using React, and the CodeMirror (Section 3.4) library, which provided the implementation of the text editor.

The backend is an HTTP server written in Python. This server receives code sent by the frontend, compiles it, runs the resulting executable, and monitors the running process. As long as the process has not stopped, the backend proxies the communication between it and the frontend, through a WebSocket. That is, everything the process writes to standard output is sent through the socket, and everything the frontend sends to the socket is written to the process' standard input.

As the editor is meant to be released to and used by the public, it is necessary for it to limit the resources afforded to the code that it executes. Otherwise, a few `while (true) { ... }` type programs could quickly consume the server's entire capacity, while achieving little. To combat this, the server imposes configurable CPU time and memory limits on users' programs⁷.

4.2.2 Showcase

In Figure 10, a small program has been written, and is running. The user can communicate with it, using the input at the bottom of the console.



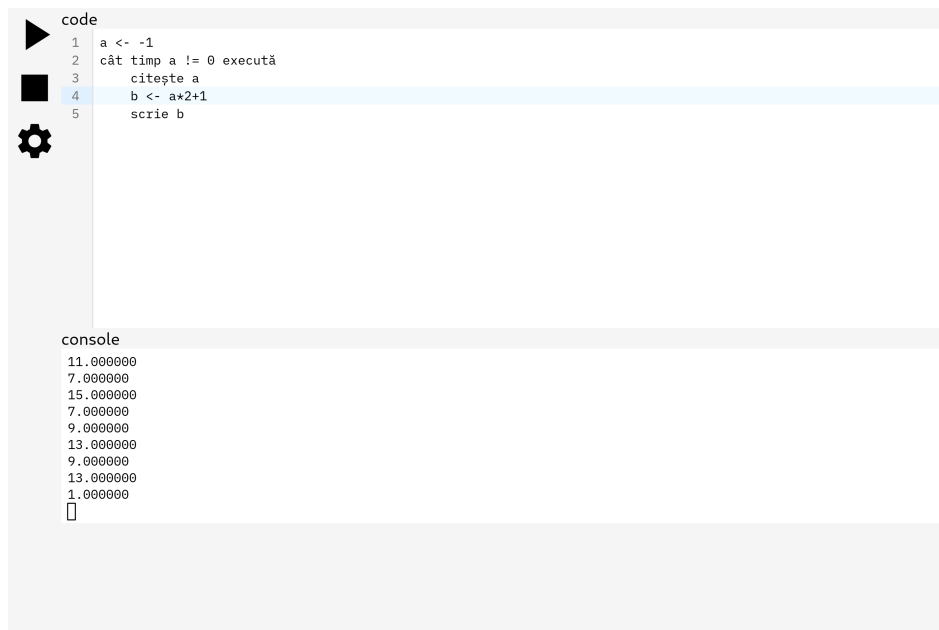
```
code
1 a <- -1
2 cât timp a != 0 execută
3 citește a
4 b <- a*2+1
5 scrie b

console (running)
11.000000
7.000000
15.000000
7.000000
9.000000
13.000000
9.000000
13.000000
0
```

Figure 10: running a program in the editor

Once the program has finished running, no further input may be sent.

⁷This was achieved using the `prlimit` Linux command.

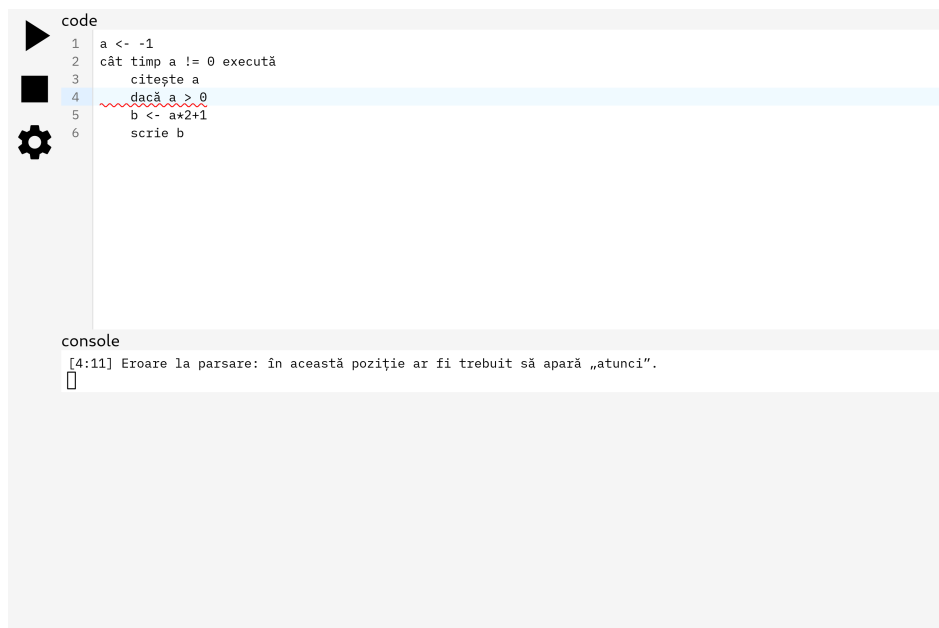


```
code
1 a <- -1
2 cât timp a != 0 execută
3   citește a
4   b <- a*2+1
5   scrie b

console
11.000000
7.000000
15.000000
7.000000
9.000000
13.000000
9.000000
13.000000
1.000000
█
```

Figure 11: program has finished running

If the code which has been submitted by the user for execution happens to contain an error, the relevant source code line is highlighted (Figure 12).



```
code
1 a <- -1
2 cât timp a != 0 execută
3   citește a
4   dacă a > 0
5     b <- a*2+1
6   scrie b

console
[4:11] Eroare la parsare: în această poziție ar fi trebuit să apară „atunci”.
█
```

Figure 12: error reporting within the editor

The editor's settings window permits for users to easily toggle between using, and not using, the lists feature of the compiler (Figure 13).

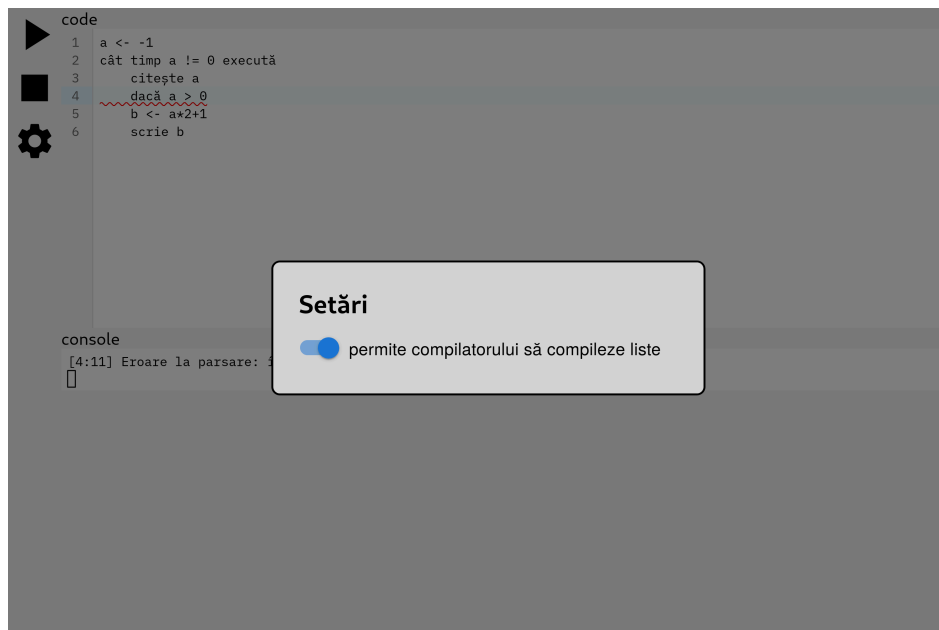


Figure 13: editor settings

Chapter 5

Evaluation

5.1 Compiler benchmarks

To measure the performance of the executable files produced by the compiler, and compare it to the performance achieved by industry-standard programming languages, a benchmark suite was written. A set of common programming tasks were chosen, and programs performing these tasks were written in Pseudocode, C, and Python. The language implementations which were benchmarked, along with other noteworthy details, are presented in Table 3.

Programming Language	Version	Notes
Pseudocode	Pseudocode compiler developed as part of this work	LLVM's default level of optimization was requested, by passing the <code>--opt default</code> argument to the compiler
C	Clang version 17.0.6	A moderate level of optimization was requested, by passing the <code>-O2</code> argument to the compiler
Python	CPython 3.11.8	-

The benchmarks were performed as follows: each program was executed on the same computer, for 5 consecutive times. After every execution, the output was verified to be correct. For every execution, the CPU time and maximum resident set size⁸ (RSS) were recorded using the `time` Linux command. The final CPU time and RSS for each program was taken to be the mean of all 5 executions.

The results of the benchmarks are presented in Figure 14 and Figure 15.

⁸Resident set size is the size of non-swapped physical memory that a process is using[10]. In this case, it can be interpreted as total memory usage, since the system on which the benchmarks were performed had sufficient memory for swapping to not be necessary.

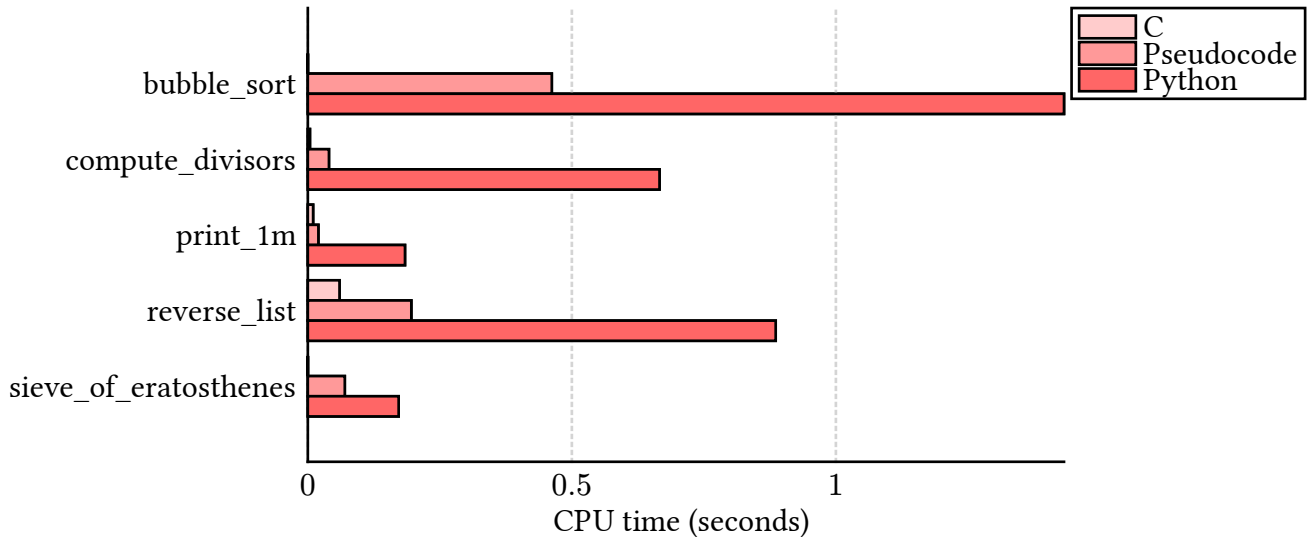


Figure 14: CPU time taken to execute the benchmarked programs

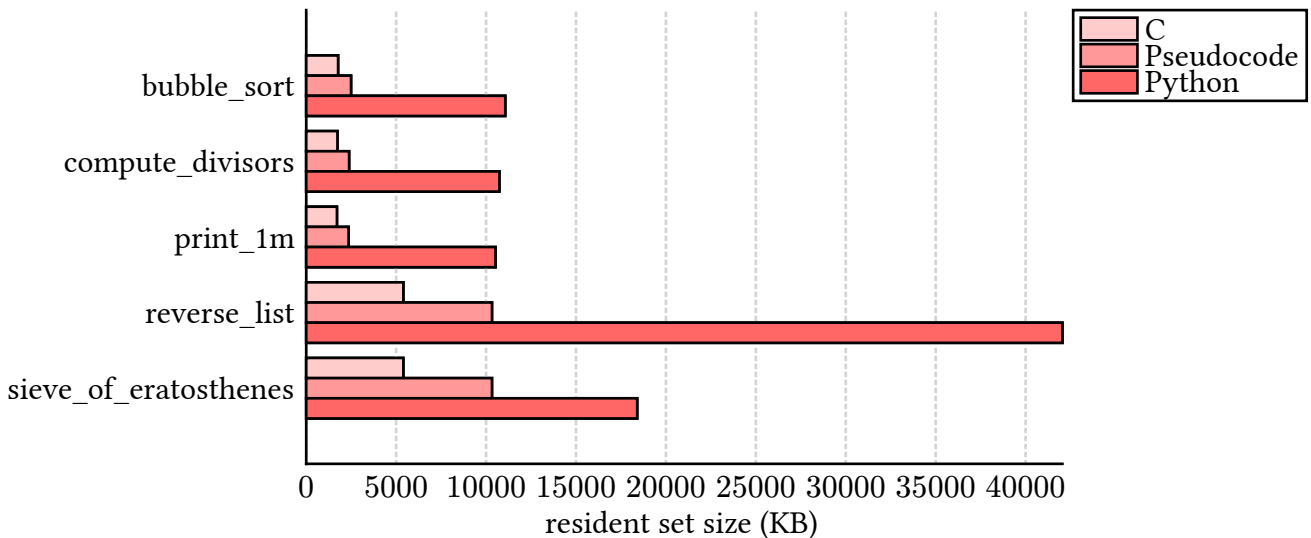


Figure 15: maximum resident set size of benchmarked programs

For both of the recorded metrics, Pseudocode placed in the middle, with C being first, and Python last. The author suspects that Pseudocode's results for the CPU time metric were significantly impacted by it being dynamically typed. Always needing to check a variable's type, and whether it is set or not, impacts the performance, and increases the size of the executable. This issue could be improved with static analysis.

On the whole, Pseudocode demonstrates a sufficient level of performance, to be fit for educational purposes.

5.2 Compiler tests

The compiler possesses a moderately sized test suite, achieving a line coverage of 82.22%. A functional testing approach was taken. That is, the tests verify whether particular features of the compiler work as expected, i.e., that a certain source code snippet produces a specific error, that a program which is compiled and executed produces a certain result, etc..

5.3 Editor backend load testing

The editor backend's capacity to prevent misuse of the host machine's resources was tested. The testing was performed as follows: 100 compilation and execution requests were sent to the backend concurrently. For each request, one of three possible source code snippets was chosen, with equal probability. One of the snippets was benign (Code listing 22), another sought to use excessive CPU resources (Code listing 23), and a third sought to use excessive memory resources (Code listing 24).

```
pentru i<-1,10 executa  
scrie "ok"
```

Code listing 22: benign code snippet

```
cat timp l=1 executa  
scrie "ok"
```

Code listing 23: code snippet using excessive CPU resources

```
list <- ,  
cat timp l=1 executa  
insereaza list,lungime(list),1
```

Code listing 24: code snippet using excessive memory resources

As can be observed in Figure 16 and Figure 17⁹, the test was successfully passed. The CPU usage spikes while the backend is processing the requests, but eventually returns to baseline. The memory usage appears to remain constant. Hence, the backend permitted for benign programs to execute until completion, while halting programs that sought to consume excessive resources.

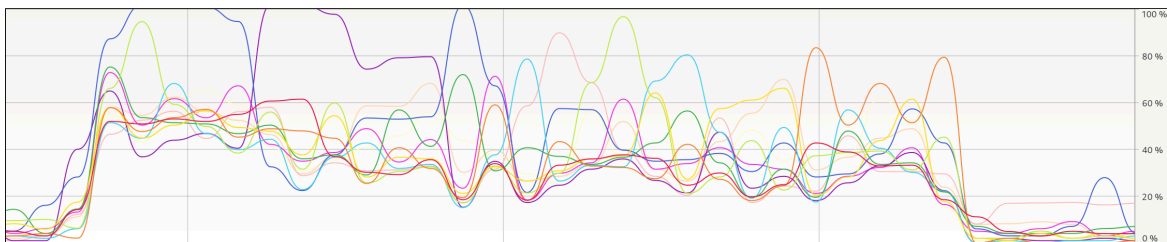


Figure 16: host machine's CPU usage; a total of 12 logical cores were available

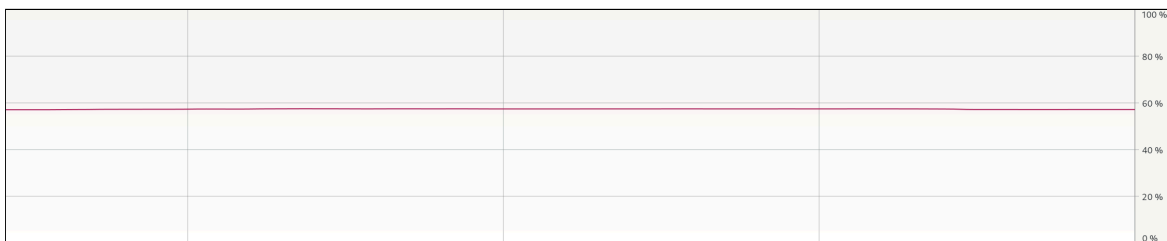


Figure 17: host machine's memory usage; a total of 64GB of memory was available

⁹Host machine CPU and memory usage was captured using GNOME System Monitor.

Chapter 6

Discussion

The systems implemented in this work achieve the goal which was set out at the beginning: transforming Pseudocode from a “static” programming language, that may only be executed through mental simulation, or manual translation into another language, into a “dynamic”, compilable and executable programming language.

The compiler is able to produce fast and debuggable executables thanks to LLVM. On the other hand, due to its dependence on LLVM, it is excluded from being able to be executed in a browser, by being first compiled to WebAssembly¹⁰. This means that the online Pseudocode editor implemented in this work can only function with the presence of an external server, which handles compilation and execution of programs. This necessarily results in increased latency, while also incurring server maintenance and scaling costs.

While the compiler’s strongly-typed AST presents certain advantages, such as being able to ensure that the compiler’s source code attains a certain level of correctness, it occasionally also results in deeply nested, repetitive, and difficult to understand code. It is possible that the aforementioned issues could be remediated, if the AST was not as strongly typed.

6.1 Future work

6.1.1 Deployment

The Pseudocode editor should be deployed onto a cloud server with a publically accessible URL, so that users may easily visit it to write, compile, and execute Pseudocode. All things considered, the system should be ready for public release: the principal features have been implemented, together with safeguards which should prevent the server from failing, in case it is made to execute particularly resource-intensive programs.

6.1.2 Syntax highlighting

Syntax highlighting would be a useful and desirable feature for the online editor. There is a number of ways in which it could be implemented. A specialized parser could be generated using Lezer¹¹, CodeMirror’s own parser generator tool. Or a TextMate[11, Section 12] grammar could be written, which is a common approach for implementing syntax highlighting today.

¹⁰Unless, of course, an additional backend is written for the compiler, which does not depend on LLVM.

¹¹ Accessible at <https://lezer.codemirror.net/>.

Alternatively, a potentially easier way may exist. If indentation is ignored, it turns out that Pseudocode parsing is line independent; that is, if a particular line of code is modified, the outcome of parsing any other line is unaffected. Hence, the parser implemented in this work could be adapted to process singular lines. Thereafter, sufficiently efficient syntax highlighting could be achieved, by only reparsing source code lines that are edited by the user.

Thereafter, this system would be compiled to WebAssembly, and integrated with CodeMirror (Section 3.4).

6.1.3 LSP server

Implementing an LSP (Section 3.3) server for Pseudocode would greatly improve the developer experience, by bringing desirable features such as auto-complete and go-to definition.

On the one hand, Rust possesses a performant and well-tested module that implements the server side of the Language Server Protocol¹². This should greatly aid the implementation of an LSP server.

On the other hand, the compiler developed as part of this work could be ill-suited as a foundation for an LSP server. An LSP server must typically correctly and efficiently handle code that is in a partially invalid state. However, the compiler written as part of this work halts immediately when it encounters an error, parses code straight from beginning to end, and uses a somewhat rigid AST structure. It could be possible to adapt it into an LSP server, but a better option may just be to write one from scratch.

As CodeMirror (Section 3.4) possesses an LSP client extension, the editor written as part of this work would also be able to benefit from LSP features.

6.1.4 Editor plugin

A code editor plugin would be another way in which Pseudocode could be made more accessible to users. It appears that novices tend to prefer Integrated Development Environments[12], due to their productivity enhancing features.

The future work directions presented in Section 6.1.2 and Section 6.1.3 would lay the groundwork for the development of such a plugin. Additionally, it would be necessary to resolve the problem of packaging and distributing the compiler, together with its dependencies, to various platforms.

6.1.5 Improved error messages

The compiler presently only outputs the starting line and column of the source code that has caused an error to occur. It would be preferable, and more informative, if the compiler was able to highlight the entire span of code that caused the error, not merely where it started. The Rust compiler successfully implements such error highlighting (Error sample 7).

¹²The `lsp_server` crate (<https://github.com/rust-lang/rust-analyzer/tree/master/lib/lsp-server>), which is used in Rust's own LSP server implementation, `rust-analyzer`.

Code

```
fn main() {  
    let for = 12;  
}
```

Error

```
error: expected identifier, found  
keyword `for`  
--> test.rs:2:6  
    |  
2  |      let for = 12;  
    |              ^^^ expected  
    |              identifier, found keyword  
    |  
help: escape `for` to use it as  
an identifier  
    |  
2  |      let r#for = 12;  
    |              ++  
  
error: aborting due to previous  
error
```

Even better, the error handling could be further improved, so as to be able to highlight multiple spans of code, which may have led to an error. This would greatly benefit the “mismatched parentheses” error, by allowing both parentheses that caused the error to occur to be highlighted (Error sample 8).

Code

```
srie [40+1)
```

Error

```
[1:11] Eroare la parsare:  
parantezele nu corespund.
```

Since error messages are typically designed and written by a compiler developer, they run the risk of using overly-precise and technical language, which is likely to confuse novices who are not familiar with the theory and terminology[13]. Hence, it would be beneficial for the error messages to be experienced and tested by users, and ultimately better adapted to their needs.

6.1.6 Execution of Pseudocode from a photo

Presently, Pseudocode is most commonly written with pen and paper. Although transcribing source code onto a computer should not be a particularly time consuming process, it could potentially be entirely automated, with the use of OCR software¹³. This could be integrated into a system where a picture of source code is taken, then automatically converted to text, compiled, and executed.

It would also perhaps be possible to develop an OCR system which would be fine-tuned for recognizing handwritten Pseudocode source code, as it typically has a noticeably distinctive form.

¹³For example, Google’s Cloud Vision API (<https://cloud.google.com/vision/docs/handwriting>) can extract handwriting from an image.

6.1.7 Assess impact of compiler use on academic performance

The system developed as part of this work could well be used to study whether it is more effective for pupils to learn computer science by hand-writing programs, and mentally simulating their execution, as opposed to writing programs on a computer, and letting it execute them.

The study could be conducted as follows: pupils studying computer science that volunteer to participate could be split into two groups, a control group, and a group with access to the Pseudocode compiler and online editor. Then, the following metrics could be measured and compared: the pupils' academic performance over the course of the school year, how many pupils from each group chose to take the Computer Science Baccalaureate Exam, what were the results they obtained in the aforementioned exam.

Chapter 7

Conclusions

This work describes the design and development process of a compiler for a programming language which closely resembles the Pseudocode language used in Romanian high-school level computer science education. Thanks to LLVM, the compiler can produce performant executables, as well as executables enhanced with debugging metadata. It is expected that pupils struggling to understand the behavior of their code will benefit from being able to debug their code.

In addition to this, the work contributes an accessible, web-based Pseudocode editor. It is likely that the majority of pupils would interact with the Pseudocode language through this system.

All things considered, this work lays a foundation for improvement of pupils' Pseudocode development experience. The systems developed up to this point are in a usable state, however further testing and refinement is recommended, so as to bring them to the highest level of quality.

Bibliography

- [1] O. E. Dictionary, “pseudocode, n.” [Online]. Available: <https://doi.org/10.1093/OED/8149907375>
- [2] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, “The scratch programming language and environment,” *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, pp. 1–15, 2010.
- [3] M. Felleisen *et al.*, “The racket manifesto,” in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, 2015.
- [4] “DrRacket: The Racket Programming Environment,” [Online]. Available: <https://docs.racket-lang.org/drracket/index.html>
- [5] V. Kruglyk and M. Lvov, “Choosing the first educational programming language,” in *ICT in Education, Research and Industrial Applications: Integration, Harmonization and Knowledge Transfer: Proceedings of the 8th International Conference ICTERI 2012*, 2012, pp. 188–189.
- [6] E. Dijkstra, “Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60,” no. MR34/61, Jan. 1961.
- [7] “The LLVM Compiler Infrastructure,” [Online]. Available: <https://llvm.org/>
- [8] D. D. I. F. Committee, “DWARF Debugging Information Format Version 5.” [Online]. Available: <https://dwarfstd.org/dwarf5std.html>
- [9] B. A. Becker, “An effective approach to enhancing compiler error messages,” in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 126–131.
- [10] “PS(1) User Commands.” 2020.
- [11] MacroMates, “TextMate 1.5.1 Documentation.” [Online]. Available: <https://macromates.com/manual/en/>
- [12] E. C. Dillon Jr, *Which environment is more suitable for novice programmers: Editor/command line/console environment vs. integrated development environment?*. The University of Alabama, 2009.

- [13] G. Marceau, K. Fisler, and S. Krishnamurthi, “Mind your language: on novices' interactions with error messages,” in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, 2011, pp. 3–18.