BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND
COMPUTER SCIENCE
SPECIALIZATION [TODO]

DIPLOMA THESIS

Pseudocode Compiler

Supervisor
[Grad, titlu si nume coordonator]

Author
Jardan Andrei

2024

Abstract

Pseudocode compiler **(TODO: bla bla bla)**

Contents

Chapter 1

Introduction

In this work, we implement a compiler for a language that is akin to the pseudocode that is used in the Romanian Baccalaureate exam. From here onwards, whenever we use the term “Pseudocode”, it will be in reference to this language, unless specified otherwise.

The Romanian Baccalaureate is an exam that is taken as part of finishing high school in Romania. The exam typically comprises 3-4 written exams. For one of these exams, there are a few subjects from which pupils may choose, including computer science. We analyzed data available from the 2023 Baccalaureate exam, and came to the conclusion that in that year, out of the 42689 pupils taking the exam who were eligible to take the computer science thing, 8708, or around 20.39%, had chosen it for the “subject to be chosen” (Figure 1).

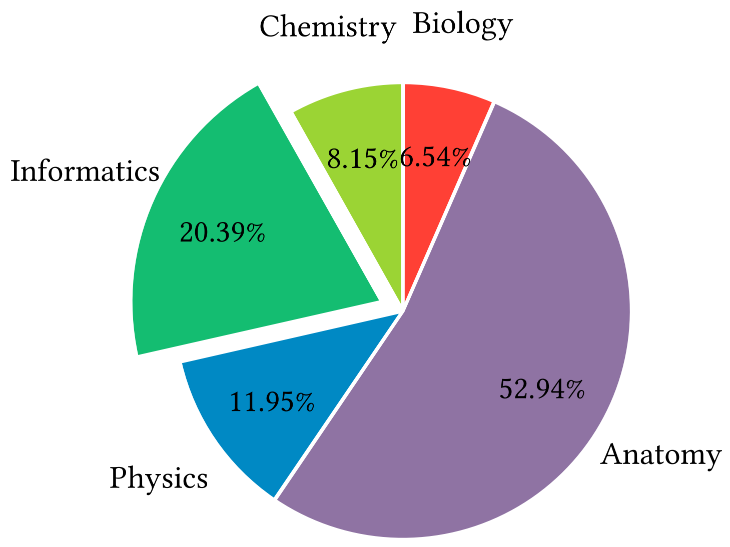


Figure 1: subjects taken by students eligible to take informatics

We believe that this work could potentially decrease the barrier to entry for pupils wanting to take the computer science exam. Because it would let them double check whether their solutions output the correct result, they would be able to debug them, etc.. It would also help teachers, and people grading the Baccalaureate exams, check the solutions faster, and with greater thoroughness.

In Section 2 we elucidate the state of the art. In Section 3 we go over a few theoretical concepts. In Section 4 we describe the pseudocode language, its implementation, and integration into an approachable web application.

Chapter 2

State of the art

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). We will enumerate a few such languages that we consider to be relevant to this work.

Scratch[1] is a programming environment that is primarily aimed at introducing children aged 8-16 to coding. Its appeal and success comes from its approachable visual editor (Figure 2), and also because it allows even people unfamiliar with programming to develop interactive, media-rich projects. It's an excellent introduction to programming, however it is not an optimal environment to learn to learn more complex aspects of computer science, such as various algorithms.

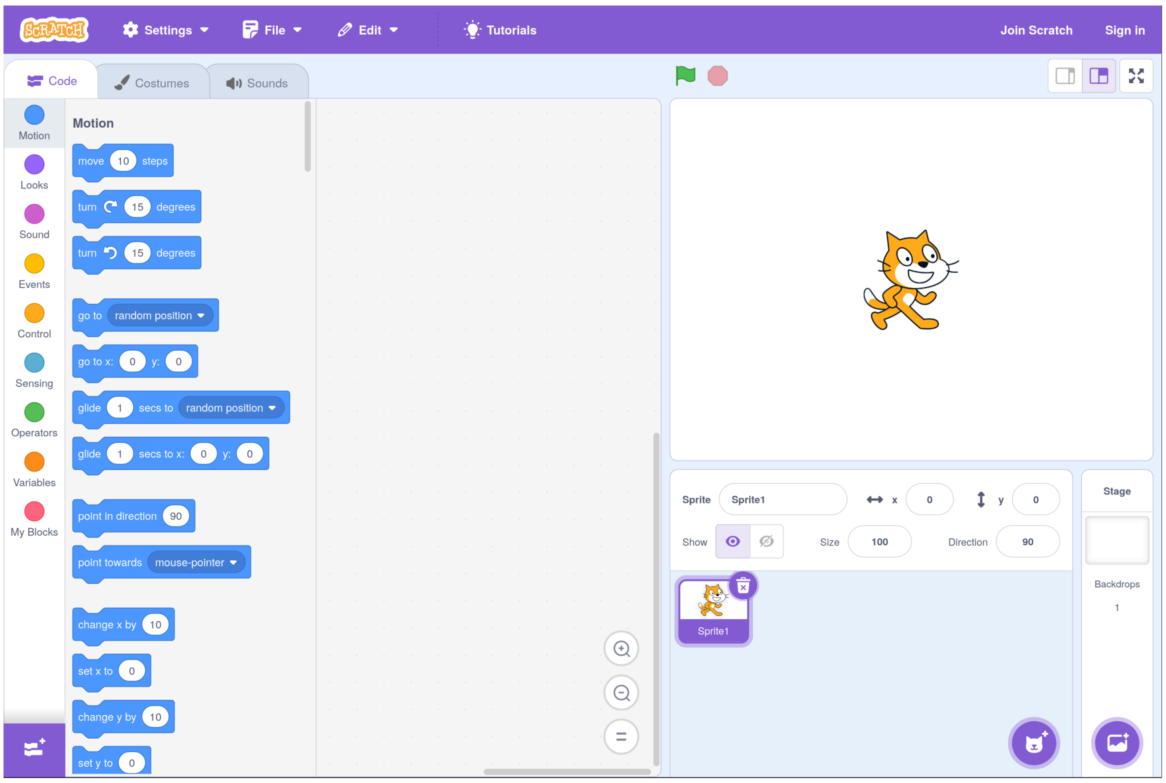


Figure 2: scratch online editor

Python is a programming language, whose educational potential was recognized as early as 2012[2]. The syntax is easy to learn, the type system is forgiving, as the language is dynamically typed, and there is also a wealth of easily installable packages, which allow users to do develop complex projects, ranging from GUI applications to web servers.

However, we posit that Python is not an optimal language for use cases such as standardized testing. Despite the language's outward simplicity, in reality it possesses a large set of features, which is constantly being expanded with updates. It would be unreasonable to ask of all test graders to learn the entirety of Python, and to regularly learn the newest features. But it would also be inconvenient to limit students to writing only a particular version of Python, or to only use a particular set of features.

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

We believe that it is unlikely that the languages we have enumerated prior would be able to replace Pseudocode in the Romanian Baccalaureate. Our reasoning is twofold. First, Pseudocode possesses properties which are advantageous for its purpose of being used in examinations: it is a simple language, which does not receive updates, and which can be very quickly learned in its entirety. Second, governments generally tend to change and take up new technologies slowly **(TODO: cite)**, which means that even if an alternative language which happens to be better than Pseudocode is developed, it is not guaranteed when or if it will be adopted.

With all this in mind, we believe it's worthwhile to improve the user experience of Pseudocode, by developing editors, compilers, debuggers, etc.. It has the potential of encouraging and helping high-school students in Romania, on their path to learning informatics, while also assisting teachers and exam graders.

Chapter 3

Theory

3.1 Shunting yard algorithm

(TODO: re-explain with parens)

The shunting yard algorithm is a method for parsing expressions specified in infix notation, and producing as result an AST. It was first invented by Edsger Dijkstra[3].

The algorithm consists of an output queue and an operator stack. An expression is read left to right, token by token. Whenever an operand is encountered, it is added to the output queue. Whenever an operator is encountered, as long as there are operators of lesser precedence in the operator stack, the following process takes place: the top operator is popped from the stack, two operands are extracted from the output queue, then the result of applying the operator on the operands is added back to the queue.

After the tokens run out, the output stack is collapsed according to the following procedure: as long as is possible, two operands are extracted from the output queue, one operator is extracted from the operator stack, and the result of applying the operator to the operands is added back to the queue.

At the end, the operator stack must be empty, and the output queue must contain a single element, representing the output. Otherwise, the algorithm throws an error.

A pseudocode version of the algorithm is offered in Algorithm 1.

(TODO: explain apply(), out, ops)

Algorithm 1: Shunting yard algorithm

```
for token in tokens do
  if token.type() = "operand" then
    out.enqueue(operand)
  if token.type() = "operator" then
    while ops.len() > 0 and ops.top().type() ≠ "lparen" and
     token.precedence() < ops.top().precedence() do
```

```
            operator $\leftarrow$ ops.pop()
            operand$_1$ $\leftarrow$ out.dequeue()
            operand$_0$ $\leftarrow$ out.dequeue()
            out.enqueue(apply(operator, operand$_0$, operand$_1$))
        ops.push(token)
    **if** token.type() = “lparen” **then**
        ops.push(token)
    **if** token.type() = “rparen” **then**
        **while** ops.len() $> 0$ **and** ops.top().type() $\neq$ “lparen” **do**
            operator $\leftarrow$ ops.pop()
            operand$_1$ $\leftarrow$ out.dequeue()
            operand$_0$ $\leftarrow$ out.dequeue()
            out.enqueue(apply(operator, operand$_0$, operand$_1$))
        ops.pop()
**while** ops.len() $\geq 1$ **and** out.len() $\geq 2$
    operator $\leftarrow$ ops.pop()
    **assert** operator.type() $\neq$ “lparen”
    operand$_1$ $\leftarrow$ out.dequeue()
    operand$_0$ $\leftarrow$ out.dequeue()
    out.enqueue(apply(operator, operand$_0$, operand$_1$))
**assert** ops.len() $= 0$
**assert** out.len() $= 1$
result $\leftarrow$ out.dequeue()
**return** result
```

3.2 LLVM

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies[4].

In this work, we use a Rust wrapper over the LLVM C library, called Inkwell[1].

LLVM can be used as the backend to a programming language. It can help turn unoptimized code into optimized code. Your code writes LLVM Intermediate Representation, commonly referred to as LLVM IR. LLVM IR sort of looks like higher-level assembly language. After your code is finished outputting the IR, you hand it off to LLVM to perform optimization, and to turn it into an object file.

For example, compiling the C program in Code listing 1, results in the LLVM IR at Code listing 2[2].

```
#include <stdio.h>
```

[1]https://github.com/TheDan64/inkwell

[2]`clang` version 17.0.6 was used

```c
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 1: Simple C program

```llvm
; ModuleID = 'main.c'
source_filename = "main.c"
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 (ptr, ...) @__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 (i32, ptr, ...) @__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++ TBAA"}
```

Code listing 2: LLVM IR generated for Code listing 1

3.3 Recursive descent

(TODO: explain recursive descent parsing, or maybe don't?)

Chapter 4

Application

4.1 Pseudocode language description

We aimed to make our implementation of the Pseudocode language as close to the version that is used in the Baccalaureate exam as possible, while keeping it easy to write. We will refer to the Pseudocode used in the Baccalaureate exam as “Baccalaureate Pseudocode”, and to our implementation as “Pseudocode” or “our Pseudocode”. Some of the main differences in syntax between our Pseudocode, and Baccalaureate Pseudocode, can be observed in Code listing 3.

While Baccalaureate Pseudocode uses box-drawing characters to delimit code blocks, we chose to simply use indentation, akin to what is done in the Python programming language. Additionally, we approximate characters such as `←` and `≤` (among others), with versions which are easier to type: `<-` and `<=`.

```
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 3: structurally equivalent Pseudocode, Baccalaureate (left) and ours (right)

The Pseudocode language accomodates an imperative programming style. In Baccalaureate Pseudocode, variables may only be of floating point type. In our implementation, variables may also be lists of floating point numbers (further described in Section 4.1.2). This functionality may be disabled, so that the compiler’s behavior will match the Baccalaureate more closely.

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 x and y using a formula such as $|x - y| < \varepsilon$, however Pseudocode is not aimed at experienced programmers.

On the contrary, its supposed to assist high school students, who have just started to learn to program. Taking this into consideration, we decided to make the compiler automatically generate comparisons this way.

4.1.1 Supported statements

In the following examples, an indented block of statements will be denoted with `...`.

(TODO: expressions)

The language supports console input and output, with `citește` and `scrie` respectively (Code listing 4), variable assignment with `{variable} <- {value}`, and swapping of variables with `{left} <-> {right}` (Code listing 5).

```
citește x
scrie "x=", x
```

Code listing 4: citește and scrie statements

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

Code listing 5: variable assignment and swapping

It supports various control-flow statements, such as if-else statements (Code listing 6), while loops (Code listing 7), repeat … until loops (Code listing 8), and for loops (Code listing 9, Code listing 10).

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

Code listing 6: if-else statement: `dacă {condition} atunci ... altfel ...`

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

Code listing 7: while loop: `cât timp {condition} execută ...`

```
x <- 10
repetă
  x <- x-1
până când x <= 0
```

Code listing 8: repeat … until loop: `repetă ... până când {condition}`

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

Code listing 9: for loop: `pentru {index} <- {start}, {stop}, {increment} execută ...`

```
pentru i <- 1,100 execută
  scrie i
```

Code listing 10: specifying an `increment` for a for loop is optional

4.1.2 Lists

We extended the language with lists, so that pupils may use Pseudocode to study more complex algorithms involving lists.

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

Code listing 11: **(TODO:)**

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

Code listing 12: **(TODO:)**

4.1.3 Sample programs

Despite being a simple language, it posesses enough complexity so as to be used for educational purposes, for instance teaching students an algorithm.

The following sample program performs bubble sort on an unordered list of numbers:

```
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 13: **(TODO: write smth)**

The following program approximates the value of $\sin(x)$, by way of Taylor polynomial, x being read from the command line:

```
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 14: **(TODO: write smth)**

4.1.4 EBNF Grammar

Grammar of pseudocode language in EBNF.

(TODO: add lists and clear up writing and variables)

`IDENT_GRAPHEME` is any unicode grapheme, with the exception of: `+-*/%|=!<>()[]`.

`INDENT` and `DEDENT` are special symbols representing the increase of the indentation level by one, and the decrease of the indentation level by one, respectively.

```
Digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

IdentRest = IDENT_GRAPHEME | Digit.
Ident = IDENT_GRAPHEME { IdentRest }.

FloatBinop = "+" | "-" | "*" | "/" | "%".
FloatUnop = "+" | "-".
FloatLit = Digit { Digit } [ "." { Digit } ].
FloatExpr =
  FloatLit
  | Ident
  | FloatUnop FloatExpr
  | FloatExpr FloatBinop FloatExpr
  | "[" FloatExpr "]"
  | "(" FloatExpr ")".

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

```
InstrAtribuire = Ident "<-" FloatExpr.
InstrInterschimbare = Ident "<->" Ident.
ScrieParam =
  FloatExpr
  | D_QUOTE UNICODE_GRAPHEME_EXCEPT_D_QUOTE D_QUOTE
  | QUOTE UNICODE_GRAPHEME_EXCEPT_QUOTE QUOTE.
InstrScrie = "scrie" ScrieParam { "," ScrieParam }.
InstrCiteste = "citește" Ident { "," Ident }.

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

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

4.2 Compiler implementation

Rust was chosen as the implementation language, for its performance and memory-safety characteristics. Rust's safety guarantees are especially useful in the implementation of the parser, because they enable the developer to safely process strings, without excessive copying, which would harm performance.

Also, Rust's strong type system made it easier to catch mistakes while writing the code.

Additionally, in the implementation, all diacritics which are part of keywords are considered to be optional, so as to ease the process of writing pseudocode programs. In the following example, both lines will be parsed as the `citește` statement.

```
citește x
citeste x
```

4.2.1 Parsing

(TODO: explain that no tokenizing is done, and the whole thing is parsed in one pass; that is in part because of `<-`, which can either show up in an assignment (e.g. `a <- 42`), or in a boolean expression, signifying "less than a negative of something" `daca x<-42 atunci`)

Use recursive descent because of the particularities of this programming languages, such as having spaces inside keywords, like `cât timp` and `până când`. The code consists of small functions, each function having a very specific purpose, such as parsing a boolean operator, or an if statement.

For parsing expressions, a modified version of the shunting yard algorithm was employed, as described in Section 3.1.

The result of the parsing process is an AST (Abstract Syntax Tree) of the program that is being parsed.

(TODO: figure with graph of AST for some code)

4.2.2 Compilation

The LLVM (Section 3.2) library is used to assist with compilation. The AST is used to generate LLVM IR, which is subsequently compiled by LLVM into an object file. Then the object file is linked into the final executable using the `clang` command, which also takes care of linking other libraries into the executable, such as `libc`.

During the LLVM IR generation process, we also make sure to include debugging information, which allows the compiled executable to be debugged at source-level, using any debugger that supports the DWARF debugging information format, such as `gdb`.

(TODO: add picture of code in debugger)

4.3 Accessible online editor

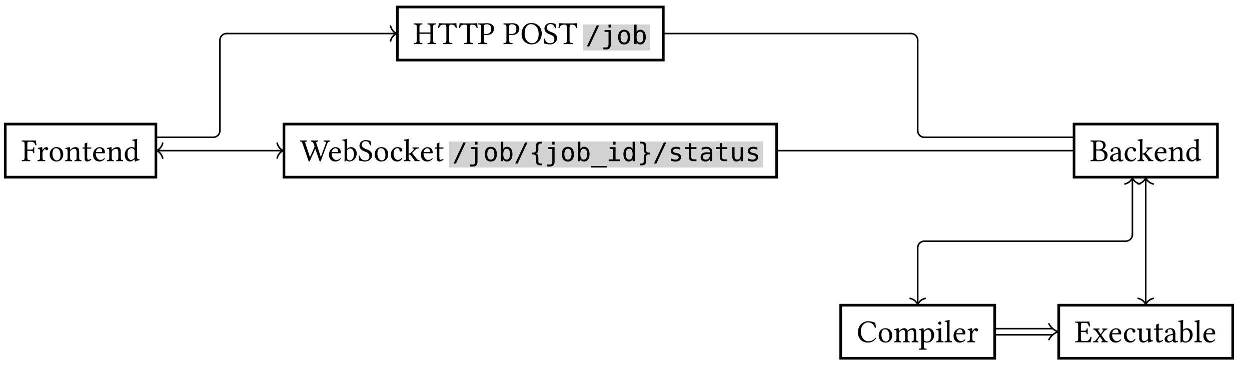
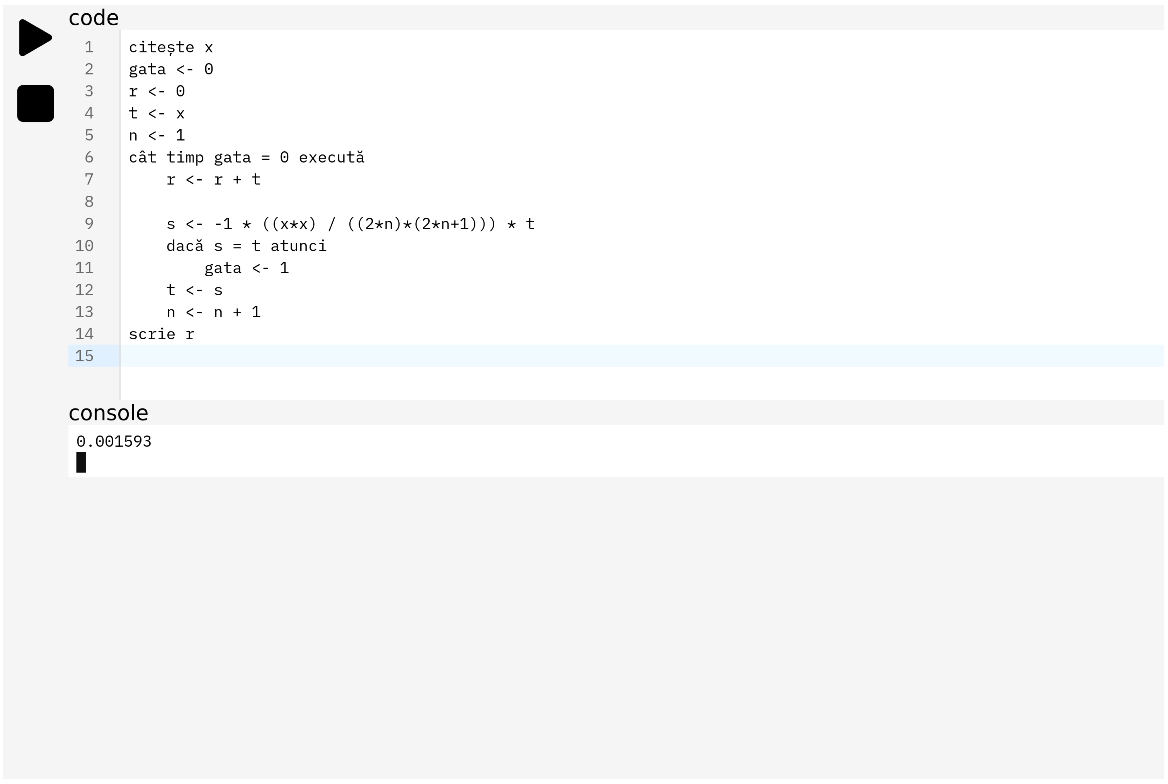


Figure 3: online editor architecture

We have implemented an online editor that should be accessible for non-technical people, such as high school pupils learning informatics.

The frontend of the editor was implemented using React, along with CodeMirror providing the actual text editor implementation. The backend consists of a Python HTTP server, which receives the code from the frontend, compiles it, and runs the resulting executable.

After the executable is run, the backend proxies all communication between the frontend and the running executable, through a WebSocket. Everything the executable writes to standard output is sent through the socket, and every string a user writes and submits into the frontend console is forwarded to the process' standard input.



```
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
```

```
0.001593
```

Figure 4: online editor frontend

Chapter 5

Future work

5.1 Syntax highlighting

Syntax highlighting could be added to the editor. It would not even be that difficult: as it's parsing, the parser basically extracts all the information necessary for syntax highlighting. It would have to be slightly modified for this task, but it would not be a very significant modification.

Then, this modified parser could be compiled to WebAssembly, and included into the final editor website. If you ignore indentation, then if you parse the entire document, then change one line, only that line will have to be re-parsed. Therefore, syntax highlighting could be done quite efficiently, by re-parsing and re-highlighting only lines of code that have been changed.

5.2 Improved error message source location

Currently, when a parsing error happens, the parser only outputs the line and column where the error happened. Due to this, all the editor can do, is highlight the line where the error happened. It would be better if the parser returned a starting and ending pair of (line, column). This way, the exact location where the error occured would be easier to find, and errors would be easier to diagnose and fix.

5.3 Execution of source code from a photo

A potentially useful feature for pupils, teachers, and exam graders alike, would be the possiblity to easily execute Pseudocode source code, that has been written on a piece of paper. This could be achieved by integrating our compiler with a handwriting detection OCR system[3].

As Pseudocode is very commonly written on paper, this would make it easier for everyone to execute and check it.

[3]For example, Google's Cloud Vision API (https://cloud.google.com/vision/docs/handwriting) can extract handwriting from an image.

Chapter 6

Conclusions

a

Chapter 6

Bibliography

[1] 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.

[2] 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.

[3] E. Dijkstra, “Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60,” no. MR34/61, Jan. 1961.

[4] “The LLVM Compiler Infrastructure,” [Online]. Available: https://llvm.org/