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).

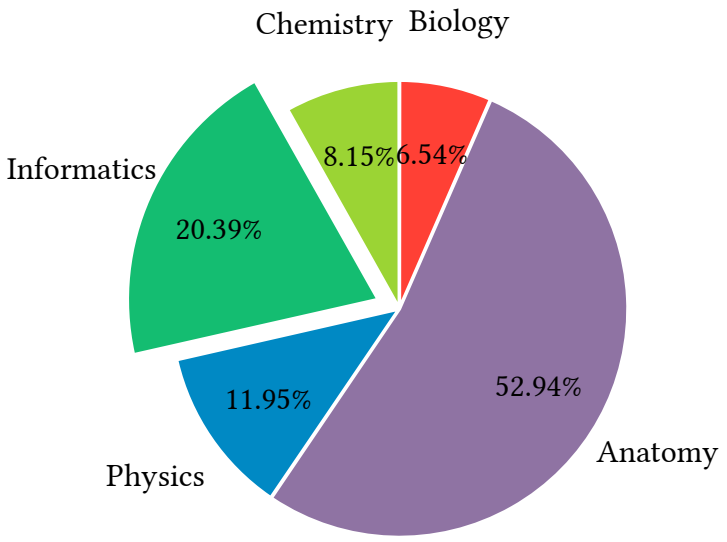

Figure 1: subjects taken by students eligible to take informatics

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.

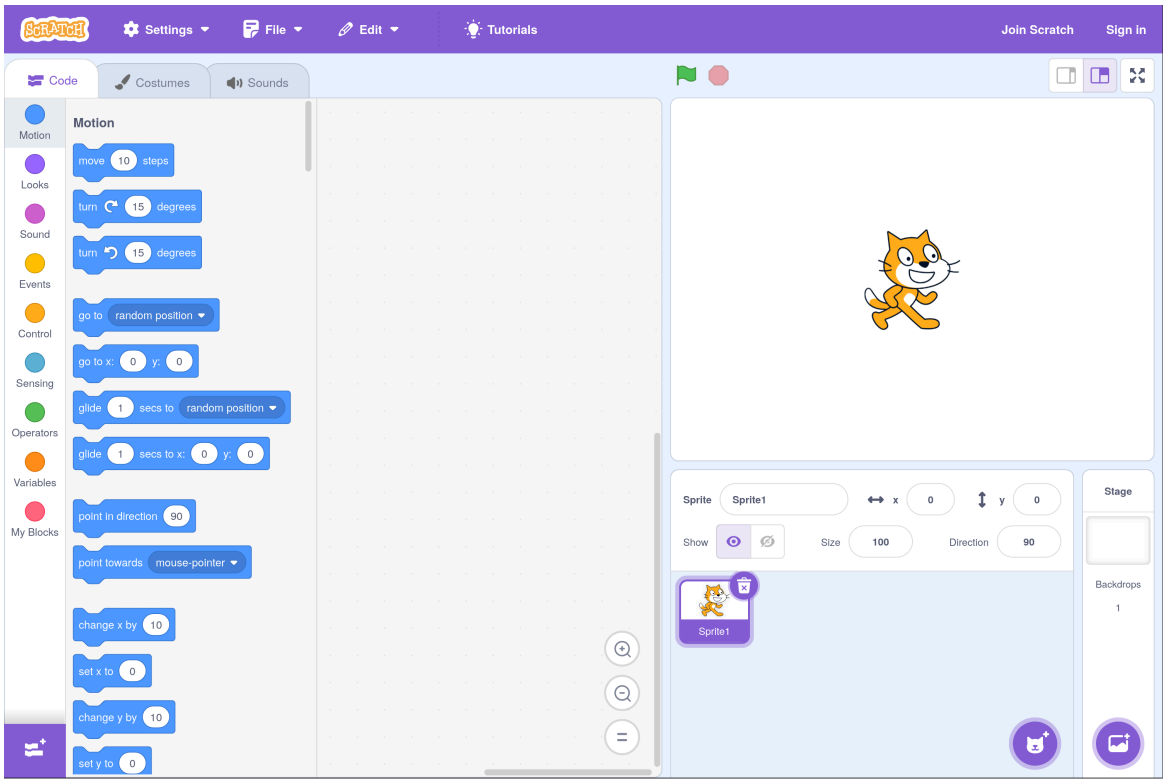

Figure 2: scratch online editor

Python is a programming language, whose educational potential had been 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.

Our work, Pseudocode, is a language that has a specific target audience – high-school pupils in Romania, a specific goal – teaching the basics of imperative computer programming, and it also has significant adoption, being put forward by the Romanian Government. Its specification has not appeared to change since its inception**(TODO: fact check?)**, and it is already part of the Romanian Baccalaureate.

Taking into account that governments typically adopt new technology at a slow pace, we believe it unlikely that Pseudocode will be replaced soon. Therefore, it is worthwile to improve the user experience of Pseudocode, developing compilers, editors, debuggers for it, with the goal of helping pupils to learn, and assisting teachers and 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

```
1 for token in tokens do
2     if token.type() = “operand” then
3         out.enqueue(operand)
4     if token.type() = “operator” then
5         while ops.len() > 0 and ops.top().type() ≠ “lparen” and
            token.precedence() < ops.top().precedence() do
```

```
6           operator ← ops.pop()
7           operand₁ ← out.dequeue()
8           operand₀ ← out.dequeue()
9           out.enqueue(apply(operator, operand₀, operand₁))
10        ops.push(token)
11    if token.type() = “lparen” then
12        ops.push(token)
13    if token.type() = “rparen” then
14        while ops.len() > 0 and ops.top().type() ≠ “lparen” do
15            operator ← ops.pop()
16            operand₁ ← out.dequeue()
17            operand₀ ← out.dequeue()
18            out.enqueue(apply(operator, operand₀, operand₁))
19        ops.pop()
20 while ops.len() ≥ 1 and out.len() ≥ 2
21    operator ← ops.pop()
22    assert operator.type() ≠ “lparen”
23    operand₁ ← out.dequeue()
24    operand₀ ← out.dequeue()
25    out.enqueue(apply(operator, operand₀, operand₁))
26 assert ops.len() = 0
27 assert out.len() = 1
28 result ← out.dequeue()
29 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].

```c
#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

```
; 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

The implemented pseudocode language is aimed to be as close as possible to the pseudocode language used in the Romanian informatics Baccalaureate exam. We have not found an official specification of this language, therefore we have devised our own, based on code samples from previous Baccalaureate exams.

The language is imperative. All variables are of floating point type. Code blocks are designated with indentation, akin to the approach taken in the Python programming language.

The language is designed so as to be easy to use for a novice who is learning to program. To reduce surprise caused by floating point arithmetic, equality comparisons are performed using an ε tolerance value. For an $\varepsilon > 0$, two floating point numbers x and y are considered to be equal, if and only if $|x - y| < \varepsilon$.

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 3), variable assignment with `{variable} <- {value}`, and swapping of variables with `{left} <-> {right}` (Code listing 4).

```
citește x
scrie "x=", x
```

Code listing 3: citește and scrie statements

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

Code listing 4: variable assignment and swapping

It supports various control-flow statements, such as if-else statements (Code listing 5), while loops (Code listing 6), repeat … until loops (Code listing 7), and for loops (Code listing 8, Code listing 9).

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

Code listing 5: if-else statement: `dacă {condition} atunci ... altfel ...`

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

Code listing 6: while loop: `cât timp {condition} execută ...`

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

Code listing 7: repeat … until loop: `repetă ... până când {condition}`

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

Code listing 8: for loop: `pentru {index} <- {start}, {stop}, {increment} execută ...`

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

Code listing 9: specifying an `increment` for a for loop is optional

4.1.2 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 calculates all the divisors for a number that is read from the command line:

```
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 10: **(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 11: **(TODO: write smth)**

4.1.3 EBNF Grammar

Grammar of pseudocode language in EBNF.

`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 Implementation of pseudocode language

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.

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

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

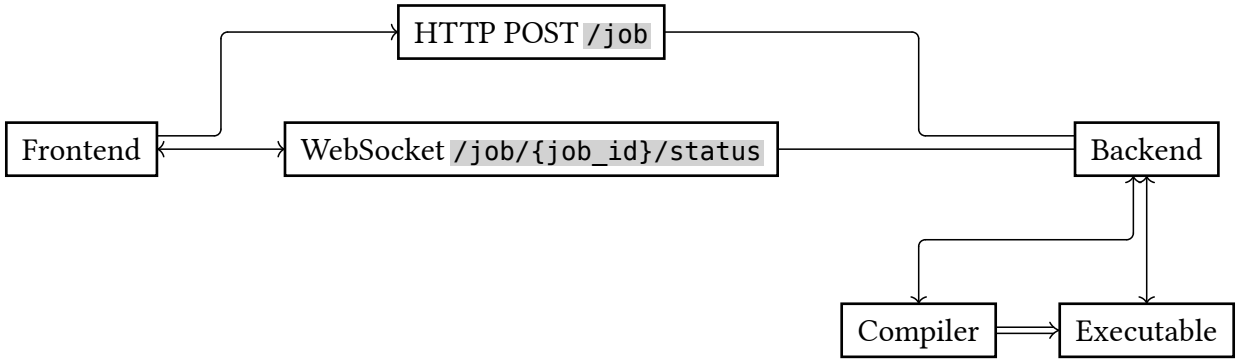

Figure 3: online editor architecture

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.

Figure 4: online editor frontend

Chapter 5

Conclusions

a

Chapter 5

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/