DD2466 Second Advanced, Individual Course in Computer Science

Implementation of a compiler for the Stoc Programming Language

Jorge García Pueyo

KUNGLIGA TEKNISKA HÖGSKOLAN
Stockholm, Sweden 2020

Contents

List of Figures **ii**

1 Introduction **1**
1.1 Objectives 1
1.2 Stoc Programming Language 1
1.3 What is a compiler 5
1.3.1 Structure of a Compiler 5
1.3.2 Structure of the Stoc compiler 7

2 Lexical Analysis **8**
2.1 Role of the Lexical Analysis 8
2.2 Approaches to Lexical Analysis 8
2.3 Lexical Analysis in real compilers 8
2.4 Lexical Analysis in the Stoc compiler 9

3 Syntax Analysis **11**
3.1 Role of the Syntax Analysis 11
3.2 Approaches to Syntax Analysis 11
3.3 Syntax Analysis in real compilers 12
3.4 Syntax Analysis in the Stoc compiler 12

4 Semantic Analysis **14**
4.1 Role of the Semantic Analysis 14
4.2 Approaches to Semantic Analysis 14
4.3 Semantic Analysis in real compilers 15
4.4 Semantic Analysis in the Stoc compiler 15

5 Intermediate Representation Code Generation **17**
5.1 Role of the Intermediate Representation Code Generation 17
5.2 Approaches to Intermediate Representation Code Generation 17
5.3 Intermediate Representation Code Generation in real compilers . . . 18
5.4 Intermediate Representation Code Generation in the Stoc compiler . 18

6 Backend and Optimizations **21**
6.1 Optimizations in LLVM 21
6.2 Backend in the Stoc compiler 22

7 Conclusion **23**

Bibliography **24**

A Token Types in the Stoc Programming Language **I**

B Grammar of the Stoc Programming Language **III**

List of Figures

1.1 Basic structure of a compiler. 5
1.2 Basic structure of the compiler frontend. 6
1.3 Basic structure of the compiler backend. 6
1.4 Comparison between set of compilers without intermediate representation and set of compilers with intermediate representation [1]. . . . 6
1.5 Structure of the compiler implemented for Stoc. 7

2.1 Example of the input and output of the Lexical Analysis phase. . . . 8
2.2 High level overview of the lexer loop. 9

3.1 Example of the input and output of the Syntax Analysis phase. . . . 11

4.1 Example of the input and output of the Semantic Analysis phase. . . 14

5.1 Example of the input and output of the Intermediate Representation Code Generation. 17

1

Introduction

1.1 Objectives

The objective of this project is to define the syntax of a new programming language and implement a compiler for it to create an executable program. This new programming language is named *Stoc*.

1.2 Stoc Programming Language

Stoc is a statically strong typed language with C-like syntax. Because of the scope of the project, *Stoc* is a simple language whose aim is to be able to write basic programs and print the results to the screen.

Variables and Constants

Stoc only contains simple types: *bool*, *int*, *float* and *string*.
A variable is declared with the keyword *var* followed by the type and the identifier, and it always has to be initialized. It ends with a semicolon:

```
// Declaring variables
var bool a = true;
var int integer = 5;
var float b = 6.0;
var string characters = "string";
```

Identifiers must start with a letter or underscore. After the first character, any sequence of letters, numbers or underscore is allowed. If the value of the variable can not be changed, it is a constant and it is defined with the keyword *const*:

```
// Declaring constants
const bool a_boolean = true;
const int FIVE = 5;
const float b6 = 6.0;
const string characters = "string";
```

Use double forward-slashes // to define single-line comments.

Basic Operators

There are basic operators for the different types:

- Assignment operator (=): for variables of all the types.
- Arithmetic operators (+ - * /): for variables of type *int* and *float*. Note that + and - can be unary operators.
- Comparison operators (== !=): for variables of all types.
- Order operators (< > <= >=): for variables of type *int* and *float*.
- Logical operators (! && //): for variables of type *bool*. Note that ! is a unary operator and that && // are not short-circuited.

Stoc does not support casting between types, so for the binary operators, both operands must be of same type.

```
const bool a = true && true;
var bool b = false || a;
var int integer = 5 + 5 * (7 - 4) / 3;
var float f = 6.0 * 7.0 / 3.5;
var float g = 9.0 - 4.6;
var int casting = integer + f;   // error!
const string characters = "string";
const string characters2 = "string2";
var bool isEqual = f == g;
```

Control Flow

Stoc has three control-flow structures:

- *if-else* conditional

```
// Structure of if-else if-else

if testexpression {
    // body
} else if testexpression2 {
    // body
} else {
    // body
}
```

It executes instructions of the body only if the test expression is *true*. Note that unlike C/C++ or Java, parenthesis in the test expressions are not required.

```
// Example of if-else if-else
var int temp = 12;
if temp > 30 {
    println("It's hot");
    if temp > 40 {
        print("It's very very hot");
    }
} else if temp >= 15 {
```

```
    println("It's warm");
} else {
    println("It's cold");
}
// Output:
// It's cold
```

- *for* loop

```
// Structure of for loop

for init; condition; post {
    // body
}
```

It executes the body of the loop if condition is *true*. The *init* statement is executed the first time before entering the loop. The *post* statement is executed once every iteration. Note that unlike C/C++ or Java, parenthesis in the *init; condition; post* is not required.

```
// Example of for loop
for var int i = 0; i < 3; i = i + 1 {
    print("Number is: ");
    println(i);
}
// Output:
// Number is: 0
// Number is: 1
// Number is: 2
```

- *while* loop

```
// Structure of while loop
while testexpression {
    // body
}
```

It executes the body of the loop while testexpression is *true*. Note that unlike C/C++ or Java, parenthesis in the test expressions are not required.

```
// Example of while loop: calculate c = a % b
var int a = 10;
var int b = 3;
var int c = 10;
while c >= b {
    c = c - b;
}
print("10 % 3 is ");
print(c)
// Output:
// 10 % 3 is 1
```

Functions

Stoc has functions and are defined by the keyword *func* followed by the identifier, the parameter list and, optionally, the return type:
func functionident(var type paramident, const type paramident, ...) returntype { }.

To return values from a function, *Stoc* uses the keyword *return*. All programs require a **main** function where the program will start executing. *Stoc* has two builtin functions to output text: *print()* and *println()* that adds a new line. Both functions only accept one argument.

This would be a program for printing the factorial of a number:

```
func factorial(var int n) int {
    var int res = 1;
    while n > 0 {
        res = res * n;
        n = n - 1;
    }
    return res;
}

func main() {
    const int c = 10;
    print("Factorial of number ");
    print(c);
    print(" is ");
    println(factorial(c));
}

// Output:
// Factorial of number 10 is 3628800
```

Stoc also supports function overloading depending on the number and type of parameters of the function:

```
func multiplybyfour(var int n) int {
    print("integer: ");
    return n * 4;
}

func multiplybyfour(var float n) float {
    print("float: ");
    return n * 4.0;
}

func main() {
    var int a = multiplybyfour(5);
    println(a);
    var float b = multiplybyfour(5.0);
    println(b);
```

```
16 }
17
18 // Output:
19 // integer: 20
20 // float: 20.000000
```

1.3 What is a compiler

A compiler is "a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language)" [2]. When designing a programming language, one aspect to take into account is what the source and the target language will be and where will the target language be executed.

Depending on this decision, we can differentiate several types of compilers. Traditional compilers that transform high-level languages into low-level languages (usually assembly or machine code), like GCC for C/C++ or the Golang compiler. Transpilers that transform high-level languages into other high-level languages, for instance the Typescript transpiler that takes Typescript source code and transforms it into JavaScript. Finally, interpreters, although they are not technically compilers because they read the source code and executes it directly without converting it to a target language, for instance the Python interpreter. We can also have a mixed approach like in Java, that compiles the Java source language into an imtermediate representation (Java bytecode) and later it executes the Java bytecode in the interpreter, the Java Virtual Machine (the reality is more complex but from the outside we can see the Java Virtual Machine as an interpreter).

1.3.1 Structure of a Compiler

The general structure of a compiler is divided into two parts, analysis (frontend tasks that depend on the source language), and synthesis (backend tasks that depend on the target language). We could think of a compiler as a sequence of phases that transform one representation to another. The representation used between frontend tasks and the backend tasks is the intermediate representation, or IR.

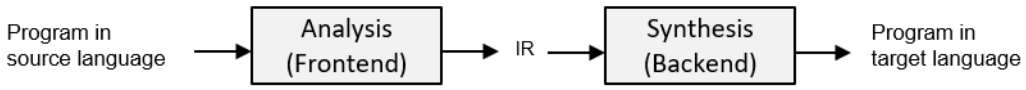

Figure 1.1: Basic structure of a compiler.

The analysis phase splits the source program into pieces and imposes a grammatical structure to produce the intermediate representation. This can be divided into smaller phases: the lexical analysis phase that reads the source program and transforms it into a sequence of tokens, the syntax analysis phase that reads a sequence of tokens and creates a tree structure describing the grammatical structure of the

program, the semantic analysis phase that uses the syntax tree structure to check the program for semantic consistency, and the intermediate code generation phase that uses the syntax tree (after being analysed semantically) to transform it to the intermediate representation.

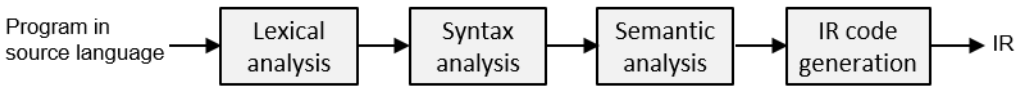

Figure 1.2: Basic structure of the compiler frontend.

The synthesis phase takes the intermediate representation and transforms it into the desired program in the target language. This can be divided into smaller phases: the optimization phase to produce better target code (where better can mean faster execution, smaller target program, or consuming less power) and the code generation phase that takes the optimized intermediate representation and transforms it into the target language. Depending on the target language, there may be additional phases like register allocation.

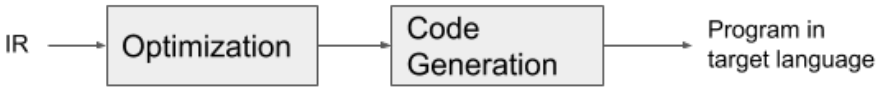

Figure 1.3: Basic structure of the compiler backend.

The problem with traditional compilers is that for every single programming language that is designed, there have to be as many compilers as execution environments, because the target language will be different. This is intractable and the solution is to define a common intermediate representation that reduces the number of different compilers to build for every new programming language.

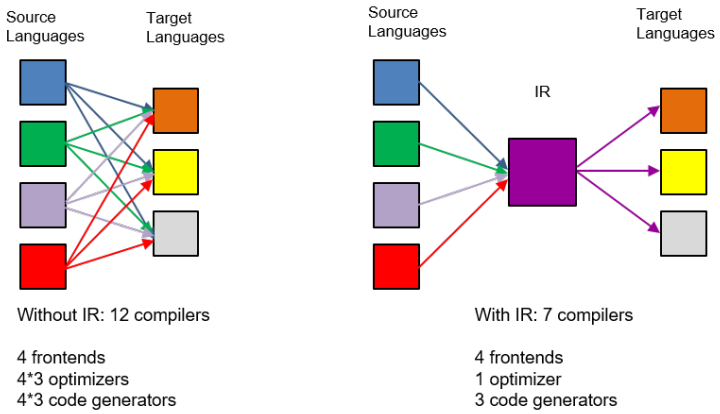

Figure 1.4: Comparison between set of compilers without intermediate representation and set of compilers with intermediate representation [1].

One well known intermediate representation, used by compilers like Clang, the official Swift compiler, and some other compiler implementations for languages like Haskell or Kotlin (Kotlin Native) is the LLVM IR [3].

1.3.2 Structure of the Stoc compiler

The objective of the project is to build a compiler. However, building all the phases of a compiler, from the frontend to the backend, would require to learn about the execution environment where the program would execute. This is an extensive task outside of the scope of the project.

The compiler for the *Stoc* language that has been implemented contains the phases related to the frontend tasks to transform *Stoc* source code into LLVM IR, and uses the LLVM tools to transform LLVM IR into native machine code. This leverages the large LLVM infrastructure for the backend tasks and extends the number of platforms where the *Stoc* Programming Language can be compiled and executed.

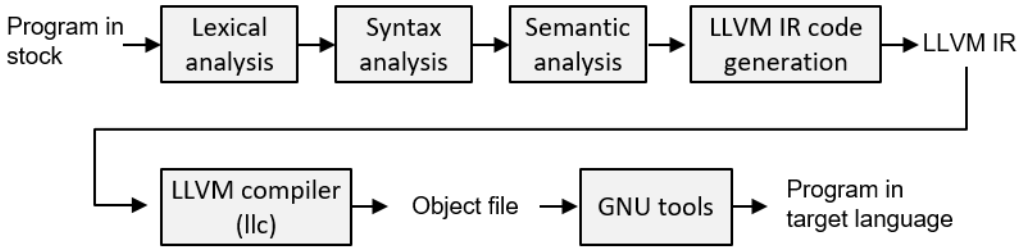

Figure 1.5: Structure of the compiler implemented for Stoc.

Although it was decided to build a traditional-like compiler using LLVM IR, other options were also considered. For instance, compiling to a custom intermediate representation and running in a custom virtual machine. This was discarded because of the scope of the project, which was to build a compiler and not a virtual machine. Other option considered was to transform the source code to Java Byte Code and execute it in the Java Virtual Machine. This was discarded because of the familiarity with the programming language of the bindings for the LLVM infrastructure, which is C/C++, as opposed to use some JVM related programming language to output Java Byte Code.

The following chapters will explain the different phases that have been implemented for the *Stoc* compiler in more detail, comparing it with the implementations of other compilers, and an additional chapter to rouglhy explain how the LLVM infrastructure works, especially the information related to the backend tasks.

The source code of the project can be found in:

`https://github.com/jgarciapueyo/stoc` .

2
Lexical Analysis

2.1 Role of the Lexical Analysis

The main task of the lexical analysis phase (also called lexing or scanning) consists on taking the program written in the source language, as a stream of characters, and produce a sequence of tokens. Usually, this also means to discard whitespaces, some punctuation marks and comments.

Figure 2.1: Example of the input and output of the Lexical Analysis phase.

2.2 Approaches to Lexical Analysis

In order to split the input into tokens, what we do is specify the pattern of the token types that compose the programming language, read the stream of characters and continuously check if a part of the stream of characters matches any pattern, and in that case assign that part to a token. A token is a tuple of the form *<token type, value>*.

The tool to specify patterns on a sequence of characters is regular expressions. Regular expressions are implemented as finite automata. There are multiple programs for this tasks, among which the most famous are Lex/Flex[4] and ANTLR[5].

2.3 Lexical Analysis in real compilers

Even though regular expressions and finite automata are taught in every course on Compilers, often using the tools mentioned previously, when we go to the source code of compilers for widespread programming languages, like Clang[8], Swift[10], Golang[12] or Rust[15], we find that the lexers are hand-coded.

This is because building a lexer is a relatively simple task and, in contrast, adding a tool to a compiler needlessly increases the complexity of the compiler. It implies to add separate files for defining the regular expressions of the token types and add

bindings between the tool and the rest of the compiler. Another reason to use custom lexers is to control the report of errors, trying to be as exact and detailed as possible. This is a very important feature of real compilers, and the key to making a compiler usable. However, it is often glossed over by theoretical courses because it is a more practical aspect of a compiler.

2.4 Lexical Analysis in the Stoc compiler

Following the reasons about how the use of external tools add complexity to a compiler and how they make more difficult custom error handling, the *Stoc* compiler uses hand-coded lexer. Another reason to build the lexer from zero is to learn the inner workings of this phase.

In order to split the source program into tokens, first we must define what a *Token* is. A *Token* in the *Stoc* compiler contains the *TokenType*, the *value*, and information about the position of the *value* in the program, for better error reports. See appendix A for all the *TokenType* of the *Stoc* Programming Language and the regular expressions that define them.

The core of a lexer (or scanner) is a loop. It starts by reading the first character, it evaluates to which token type it belongs to, and it consumes it together with all the following characters that are part of that token. When it reaches the end, usually a whitespace or a punctuation mark, it creates the token. Sometimes, it will have to read more than one character to decide to which token type it belongs to.

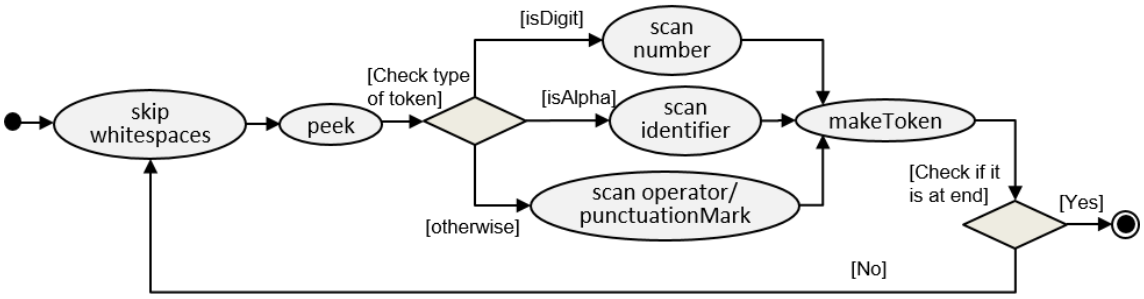

Figure 2.2: High level overview of the lexer loop.

The skipWhitespaces function discards whitespaces characters or comments, the peek function reads the character and, depending on the type of this character, if is a digit, an alphabet character or something else, chooses how to scan the following characters until finding a separator character. It creates the token and repeats the loop, until having scanned all the file. Looking at the figure, we can see that is a high level finite automata, with similarities with a finite automata for a regular expression.

Using a hand-coded scanner allows to add information when creating a token, like the line and the column of the value of the token in the source program, which it is

a valuable information for having useful error messages.

As a way to better illustrate how the *Stoc* compiler works and the input and output of every phase, we are going to take the program of the introduction to calculate a factorial and show how it is transformed along every phase.

Input: program in *Stoc*
File: *factorial.st*

```
1  func factorial(var int n) int {
2      var int res = 1;
3      while n > 0 {
4          res = res * n;
5          n = n - 1;
6      }
7      return res;
8  }
9
10 func main() {
11     const int c = 10;
12     print("Factorial of number ");
13     print(c);
14     print(" is ");
15     print(factorial(c));
16 }
```

Output: sequence of tokens
$ stoc –token-dump factorial.st

```
LINE COL      VALUE    TOKEN TYPE
----------------------------------
   1   1       func          func
   1   6  factorial    IDENTIFIER
   1  15          (           '('
   1  16        var           var
   1  20        int           int
   1  24          n    IDENTIFIER
   1  25          )           ')'
   1  27        int           int
   1  31          {           '{'
   2   5        var           var
   2   9        int           int
   2  13        res    IDENTIFIER
   2  17          =           '='
   2  19          1       LIT_INT
   2  20          ;           ';'
   3   5      while         while
   3  11          n    IDENTIFIER
   3  13          >           '>'
   3  15          0       LIT_INT
   3  17          {           '{'
   4   9        res    IDENTIFIER
   4  13          =           '='
   4  15        res    IDENTIFIER
   4  19          *           '*'
   4  21          n    IDENTIFIER
   4  22          ;           ';'
...
  15   5      print    IDENTIFIER
  15  10          (           '('
  15  11  factorial    IDENTIFIER
  15  20          (           '('
  15  21          c    IDENTIFIER
  15  22          )           ')'
  15  23          )           ')'
  15  24          ;           ';'
  16   1          }           '}'
  16   2                    T_EOF
```

3

Syntax Analysis

3.1 Role of the Syntax Analysis

The main task of the syntax analysis phase (also called parsing) consists on taking the sequence of tokens from lexical analysis and verify that it complies with the grammar of the programming language. This is done by building a parsing tree. The grammar of a programming language is a formal definition of the structure that the programs in that language must have.

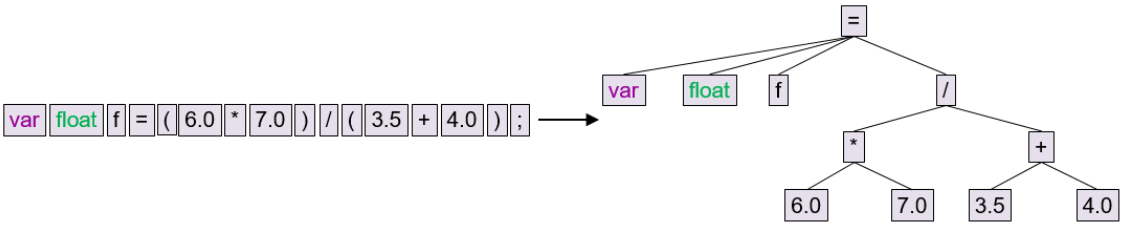

Figure 3.1: Example of the input and output of the Syntax Analysis phase.

3.2 Approaches to Syntax Analysis

The methods commonly used for parsing are top-down and bottom-up. Even though they can not parse every grammar, they are powerful enough to parse LL and LR grammars. These grammars are sufficiently expressive to describe most of the structures of general programming languages.

Defining the grammar of a programming language in a formal way is very important because it allows to ensure that the grammar does not contain any ambiguity which can cause the program to be understood in different ways.

There exist tools called parser generators, that given a formal specification of a grammar, construct a parser for it, like ANTLR[5] and Bison[6].

3.3 Syntax Analysis in real compilers

There are many parsing techniques: LL(k), LR(k), SLR, LALR, ... but recursive descent parsers (a type of top-down parser) is widely used in production compilers, like GCC[7], Clang[9], and Golang[13]. This is because, similar as what happens with a hand-coded lexer, building a hand-coded recursive descent parser is "simple" and can support custom error-handling, in addition to hacks for parsing difficult derivations in the grammar of the programming language. All this without adding extra complexity of integrating a new tool to the compiler.

Even though the compilers mentioned previously do not need to specify the formal grammar of the language for generating a parser, sometimes in the documentation of the programming language we can find the grammar of the language, Golang [14], Swift[11], showing that this is still useful. It can happen that these grammars are ambiguous and generally are accompanied with textual explanations (i.e. the explanation about operator precedence in unary and binary expressions).

3.4 Syntax Analysis in the Stoc compiler

The way *Stoc* programming language has simple constructs (variable and constant definition, function definitions, simple flow control structures, ...) allows for using a recursive descent parser, written from scratch, because seeing only the first token the parser can decide which rule of the grammar it corresponds to. The decision of using a recursive descent parser is supported by the example of production grade compilers using the same technique, as mentioned previously.

The complete grammar of *Stoc* can be found in appendix B but here we are going to go over the more important decisions. An important remark here is that the definition of the syntax of the language and how the parser is implemented are somewhat coupled, since a decision in one part could affect the other.

To define the grammar of the language (for later implement the parser), the first step is to decide what the top-level grammar rule will be and we have different options available: Golang top-level grammar rules are declarations[14] and Swift top-level grammar rules are statements[11]. For *Stoc*, top-level grammar rules are declarations and a program will be a list of declarations (variable, constants or function declarations). From there, the rest of the grammar rules follow the usual structure of a C-like syntax, and the implementation is straightforward for the recursive descent parser.

However, *Stoc* has some features to try to make the language easier to use and less error prone that C. One of this features is that an assignment is an statement and not an expression, so it does not return a value. The reason behind this is to try to avoid errors like write a single = when doing a comparison, that has double = in the condition of the if statement. This is similar to how Swift and Python im-

plement assignment, although Python has recently added, with controversy, a new assignment operator to allow for assignment expressions [18].

Other decision is to make control flow structures statements in *Stoc*, similar to C/C++ or Go [14]. There are other languages where control flow structures are expressions, like Rust [16] and Kotlin [17] (for some control flow structures). The reason for this was to be more similar to C, but both options could have worked.

Declarations and statements are easily parsed by the recursive descent parser, it just have to check the first keyword and from there decide how to parse it (if it sees *for* keyword, it is a for loop statement, if it sees *func* keyword is a function declaration, etc.) but when we get to parsing expressions, simple recursive descent parsers are not that good of a fit. For instance, given the grammar rule for parsing expressions:

```
1       EXPR → BINARYEXPR
2 BINARYEXPR → UNARYEXPR
3     | EXPR + EXPR
4     | EXPR - EXPR
5     | EXPR * EXPR
6     | EXPR / EXPR
```

The rule for binary expressions is clearly ambiguous. Usually this was solved by adding different rules depending on the precedence of the operators. This could end being very big (i.e. JavaScript has 21 precedence levels [19]). The solution implemented in *Stoc* for solving this is to use the technique of Pratt parser[20], which associates semantics with tokens instead of grammar rules, in this case the precedence of the operators.

Reporting errors during parsing is a difficult job. Usually, the more distinct syntactic errors are reported at the same time the best. No one wants to compile the same program once for every error, so usually when one error is found the compiler keeps parsing the rest of the program to report more errors. However, this could provoke cascading errors. *Stoc* solves this by synchronizing the state of the parser when it finds an error. This consists on consuming tokens until it finds a boundary token indicating the end of a declaration or statement, and after that, it can continue parsing the program with a high probability that it will not provoke cascading errors. Usually this boundaries tokens are easy to decide, like semicolon, closing brackets or closing parenthesis. Other options to parsing error recovery is to include error rules in the grammar. The idea from how to implement error recovery comes from [2][21].

The *Stoc* compiler prints the AST with more information computed in the Semantic Analysis phase, so in the next chapter we will show how the list of tokens is transformed into an AST.

4

Semantic Analysis

4.1 Role of the Semantic Analysis

The main task of the semantic analysis phase is to compute additional information after the syntactic structure of the program is known. It is called semantic analysis because the information goes beyond what a context-free grammar can express and it is closer related with the meaning of the program. In a statically typed language like *Stoc*, semantic analysis consists on building a symbol table to connect variables definition and their use, and also doing type checking.

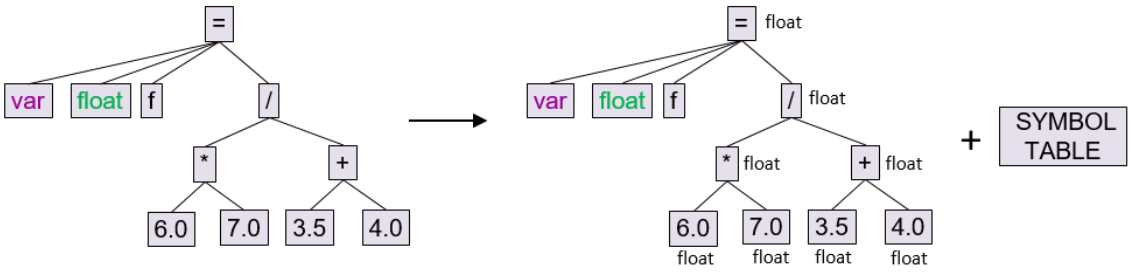

Figure 4.1: Example of the input and output of the Semantic Analysis phase.

4.2 Approaches to Semantic Analysis

We will divide semantic analysis into two parts, name resolution and type checking. Name resolution consists on associating the usage of identifiers with their definition. This is usually done with the help of an external data structure called the symbol table.

Type inference consists on the automatic detection of the data type of an expression and type checking consists on verify that the types of the operands in expressions are consistent and valid with the semantics of the programming language. Designing a consistent type system can be a very complex and formal task, including things like subtyping, recursive types, polymorphism, etc. [22].

4.3 Semantic Analysis in real compilers

The semantic analysis phase changes between the compilers for the different programming languages depending on its semantics. Depending on the programming language, the amount of information to compute varies significantly. In dynamically typed languages, there might not be any type checking at all while in statically typed languages, like Golang or C++, semantic analysis is a considerable part of the compiler. This can even be more important in languages with strong requirements like Ada.

4.4 Semantic Analysis in the Stoc compiler

Semantic analysis is performed in the abstract syntax tree (AST) constructed in the syntax analysis using the visitor pattern for every node of the AST.

Stoc has static scoping, like Golang[14] and many other languages, which means that "a variable usage refers to the preceding definition with the same name in the innermost scope that encloses the expression where the variable is used"[23]. To perform name resolution, it uses a symbol table to store all the declarations. Because of the recursive nature of static scoping (a scope can contain other scopes), the structure of the symbol table is recursive, with the inner symbol table containing a reference to its parent scope. The functioning is similar to other languages: when inserting a declaration, it is inserted in the innermost symbol table, which is the current one, if not other symbol with the same identifier already exists. When looking up a declaration, to bind it to a usage, it starts by looking at the innermost symbol table, if it is not found, it goes to the parent symbol table and recursively like this until finding the symbol. If it is not found, an error is reported.

Because *Stoc* supports function overloading, inserting a function is not so straightforward as it has been described. When inserting the function in the symbol table, it has to check not only the identifier but also the type of the parameters. This is also true for binding a call expression to the function definition. It has to look for the identifier but also check that the type of the arguments is the same as the type of the parameters in the function definition.

Stoc has a pretty simple type system with basic types and without any type conversion. This is because of the simple nature of the programming language and the scope of the project, since creating a full type system can be very complex. When performing type checking, the *Stoc* compiler starts with from the literals, which are the bottom nodes, to establish the type of the expressions. From the literal nodes, it goes up the AST and for every node it checks if the types of the children are valid (i.e. in the binary expression it checks that children have the same type and that the operator is allowed for the children types), and in that case, assigns the corresponding type to the node of the AST.

If this project was continued, a future optimisation that could be added, and that a lot of compilers implement, is constant folding. This allows to evaluate some expressions during compilation, so they do not have to be calculated in execution, improving time efficiency.

Continuing with the example program to calculate the factorial, here we show the input to the Syntax Analysis phase (a sequence of tokens) and the output of the Semantic Analysis phase (an AST with type information). The representation used between Syntax Analysis and Semantic Analysis is this same AST but without type information.

Input: sequence of tokens
$ stoc –token-dump factorial.st

```
LINE COL      VALUE    TOKEN TYPE
---------------------------------
   1   1       func          func
   1   6  factorial    IDENTIFIER
   1  15          (           '('
   1  16        var           var
   1  20        int           int
   1  24          n    IDENTIFIER
   1  25          )           ')'
   1  27        int           int
   1  31          {           '{'
   2   5        var           var
   2   9        int           int
   2  13        res    IDENTIFIER
   2  17          =           '='
   2  19          1       LIT_INT
   2  20          ;           ';'
   3   5      while         while
   3  11          n    IDENTIFIER
   3  13          >           '>'
   3  15          0       LIT_INT
   3  17          {           '{'
   4   9        res    IDENTIFIER
   4  13          =           '='
   4  15        res    IDENTIFIER
   4  19          *           '*'
   4  21          n    IDENTIFIER
   4  22          ;           ';'
...
  15   5      print    IDENTIFIER
  15  10          (           '('
  15  11  factorial    IDENTIFIER
  15  20          (           '('
  15  21          c    IDENTIFIER
  15  22          )           ')'
  15  23          )           ')'
  15  24          ;           ';'
  16   1          }           '}'
  16   2                  T_EOF
```

Output: AST with type information
$ stoc –ast-dump factorial.st

```
-FuncDecl <l.1:c1> 'factorial' int
 |-ParamDecl <l.1:c.16> 'n' int
 `-BlockStmt <l.1:c.31> - <l.8:c.1>
   |-DeclarationStmt
   | `-VarDecl <l.2:c.5> 'res' int
   |   `-LiteralExpr <l.2:c.19> LIT_INT '1' int
   |-WhileStmt <l.3:c.5>
   | |-BinaryExpr <l.3:c.13> '>' bool
   | | |-IdentExpr <l.3:c.11> 'n' int
   | | `-LiteralExpr <l.3:c.15> LIT_INT '0' int
   | `-BlockStmt <l.3:c.17> - <l.6:c.5>
   |   |-AssignmentStmt <l.4:c.13>
   |   | |-IdentExpr <l.4:c.9> 'res' int
   |   | `-BinaryExpr <l.4:c.19> '*' int
   |   |   |-IdentExpr <l.4:c.15> 'res' int
   |   |   `-IdentExpr <l.4:c.21> 'n' int
   |   `-AssignmentStmt <l.5:c.11>
   |     |-IdentExpr <l.5:c.9> 'n' int
   |     `-BinaryExpr <l.5:c.15> '-' int
   |       |-IdentExpr <l.5:c.13> 'n' int
   |       `-LiteralExpr <l.5:c.17> LIT_INT '1' int
   `-ReturnStmt <l.7:c.5>
     `-IdentExpr <l.7:c.12> 'res' int
-FuncDecl <l.10:c1> 'main'
 `-BlockStmt <l.10:c.13> - <l.16:c.1>
   |-DeclarationStmt
   | `-ConstDecl <l.11:c.5> 'c' int
   |   `-LiteralExpr <l.11:c.19> LIT_INT '10' int
   |-ExpressionStmt
   | `-CallExpr void
   |   |-IdentExpr <l.12:c.5> 'print' (string) void
   |   `-LiteralExpr <l.12:c.11> LIT_STRING 'Factorial of number ' string
   |-ExpressionStmt
   | `-CallExpr void
   |   |-IdentExpr <l.13:c.5> 'print' (int) void
   |   `-IdentExpr <l.13:c.11> 'c' int
   |-ExpressionStmt
   | `-CallExpr void
   |   |-IdentExpr <l.14:c.5> 'print' (string) void
   |   `-LiteralExpr <l.14:c.11> LIT_STRING ' is ' string
   `-ExpressionStmt
     `-CallExpr void
       |-IdentExpr <l.15:c.5> 'println' (int) void
       `-CallExpr int
         |-IdentExpr <l.15:c.13> 'factorial' (int) int
         `-IdentExpr <l.15:c.23> 'c' int
```

5

Intermediate Representation Code Generation

5.1 Role of the Intermediate Representation Code Generation

Splitting the structure of a compiler into frontend and backend, where ideally details of the source language are treated by the frontend and details of the target language are treated by the backend, allows to create a set of compilers that share an intermediate representation, saving a considerable effort, as we saw in section 1.3.1. The objective of IR is to be a good binding between the frontend and backend of the compiler, and the main task of IR code generation is to generate IR from the abstract syntax tree.

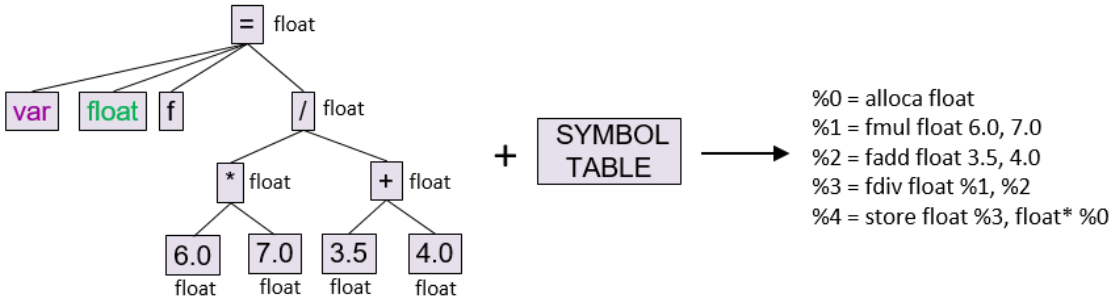

Figure 5.1: Example of the input and output of the Intermediate Representation Code Generation.

5.2 Approaches to Intermediate Representation Code Generation

There can be multiple intermediate representations and they can be high level, closer to source language, or low level, closer to target language.

We could say that the Abstract Syntax Tree constructed in parsing could be considered an intermediate representation between the frontend and the backend of the

compiler. However, it could be argued that it is not a very good IR since it is not close to any target language, particularly at representing control flow.

For this reason, other IR exists and usually represent some form of linearization of the AST. One of these common IR is three-address code [2], where there is at most one operator on the right side of an instruction. This constraint makes three-address code simple which helps optimization and transformation into the target language.

Another known IR is Static Single-Assignment Form (SSA) [24], which is similar to three-address code but differs in that all assignments are to different variables. This makes this form even more suitable to optimization.

5.3 Intermediate Representation Code Generation in real compilers

Compilers use very different IR through the different phases. However, SSA is a very used IR among real compilers [25]. For instance, the LLVM project, used in many compilers [26], defines its LLVM IR as "a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly" [27].

5.4 Intermediate Representation Code Generation in the Stoc compiler

Stoc compiler uses the LLVM IR, which allows us to forget about the target language and center on the frontend part of the compiler, as explained in section 1.3.2.

LLVM provides a C++ API to generate LLVM IR from the AST. This is done by defining a function for every node in the AST, that reads the information from the AST node (i.e. the identifier in a variable declaration), produces the needed LLVM IR instructions by calling the API with the previous information and recursively calls other functions to generate LLVM IR for the children nodes in the AST. This process is rather simple but can have some interesting cases when the LLVM IR is not a direct translation from the semantics of the source language. For instance, in *Stoc* a global variable can be defined as a complex expression including other variables or binary expressions. However, this is not allowed in LLVM IR, where the global variable has to be defined with a constant initial value. The solution to this is to define the global variable with a default value and produce the LLVM IR code to calculate the initial value inside a function that is called before any other code is executed. This function is called constructors[28].

Other problem to generate LLVM IR from the AST is the way SSA works. Usually, it is structured around the concept of basic blocks, which are a sequence of code

instructions without any branches except into the entry of the basic bloc and out of the basic block. This makes generating flow control structures, like for, while or if statements, a little more complicated, especially if inside these structures there is some branch instruction like return. This was solved by dividing every flow control structure into basic blocks (i.e. the for loop was splitted into initialization, condition, postcondition and the body of the loop). Even doing this, in the body of the loop, which corresponds to a basic block, it has to check that no other terminating instruction (like a return statement) has been added, because a basic block can only have a basic terminating instruction.

Stoc supports function overloading by parameters, however LLVM IR does not support this. To overcome this difference, we look at how the Clang compiler works and found a very interesting technique called name mangling [29], which consists on modifying the identifier of functions depending on the parameters type (C++ defines this modifications in the official ABI [30]). In *Stoc*, name mangling is done in the Semantic Analysis phase, because is when the resolution between an identifier usage and declaration is done. It only modifies the function identifiers and consists on adding the parameters types to the function name.

LLVM also allows to use C functions from static libraries. This is very useful and *Stoc* takes advantage of this using the C string library to compare strings under the hood. It also uses the C stdio library to implement *print* and *println* builtin functions.

Finally, this is the LLVM IR code that the *Stoc* compiler emits for the factorial program:

```
; ModuleID = 'report_factorial.st'
source_filename = "report_factorial.st"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@0 = internal constant [3 x i8] c"%s\00"
@1 = internal constant [21 x i8] c"Factorial of number \00"
@2 = internal constant [3 x i8] c"%d\00"
@3 = internal constant [3 x i8] c"%s\00"
@4 = internal constant [5 x i8] c" is \00"
@5 = internal constant [4 x i8] c"%d\0A\00"

declare i8* @printf(i8*, ...)

declare i64 @strcmp(i8*, i8*)

define i64 @factorial_1p_int_rint(i64 %n) {
entry:
  %0 = alloca i64
  %return = alloca i64
  store i64 %n, i64* %0
  %res = alloca i64
  store i64 1, i64* %res
```

```
24   br label %conditionwhile
25
26 conditionwhile:                                    ; preds = %
       bodywhile, %entry
27   %tempload = load i64, i64* %0
28   %greatertmp = icmp sgt i64 %tempload, 0
29   br i1 %greatertmp, label %bodywhile, label %continuationwhile
30
31 bodywhile:                                         ; preds = %
       conditionwhile
32   %tempload1 = load i64, i64* %res
33   %tempload2 = load i64, i64* %0
34   %multemp = mul i64 %tempload1, %tempload2
35   store i64 %multemp, i64* %res
36   %tempload3 = load i64, i64* %0
37   %subtemp = sub i64 %tempload3, 1
38   store i64 %subtemp, i64* %0
39   br label %conditionwhile
40
41 continuationwhile:                                 ; preds = %
       conditionwhile
42   %tempload4 = load i64, i64* %res
43   store i64 %tempload4, i64* %return
44   br label %exit
45
46 exit:                                              ; preds = %
       continuationwhile
47   %1 = load i64, i64* %return
48   ret i64 %1
49 }
50
51 define void @main() {
52 entry:
53   %c = alloca i64
54   store i64 10, i64* %c
55   %calltmp = call i8* (i8*, ...) @printf(i8* getelementptr inbounds
       ([3 x i8], [3 x i8]* @0, i32 0, i32 0), i8* getelementptr
       inbounds ([21 x i8], [21 x i8]* @1, i32 0, i32 0))
56   %tempload = load i64, i64* %c
57   %calltmp1 = call i8* (i8*, ...) @printf(i8* getelementptr
       inbounds ([3 x i8], [3 x i8]* @2, i32 0, i32 0), i64 %tempload)
58   %calltmp2 = call i8* (i8*, ...) @printf(i8* getelementptr
       inbounds ([3 x i8], [3 x i8]* @3, i32 0, i32 0), i8*
       getelementptr inbounds ([5 x i8], [5 x i8]* @4, i32 0, i32 0))
59   %tempload3 = load i64, i64* %c
60   %0 = call i64 @factorial_1p_int_rint(i64 %tempload3)
61   %calltmp4 = call i8* (i8*, ...) @printf(i8* getelementptr
       inbounds ([4 x i8], [4 x i8]* @5, i32 0, i32 0), i64 %0)
62   ret void
63 }
```

6
Backend and Optimizations

Using LLVM infrastructure in *Stoc* allows us not to worry about how to generate the machine code from the LLVM IR. Also, because of the scope of the project, the *Stoc* compiler does not perform any optimization in any of the phases. In this chapter, we will go over how LLVM performs optimizations and how it transforms LLVM IR into machine code.

6.1 Optimizations in LLVM

Optimization is often considered the most difficult part in compilers because it requires an understanding about how the program will be represented in the target language and how it will be executed. Optimization is necessary because translating high-level language constructs into low-level language can introduce inefficiencies in the code.

We can classify optimizations by different criteria: machine-independent optimizations, which is optimization of code that is not related with registers or memory locations (i.e. common subexpression elimination, dead code elimination, constant propagation, ...) or machine-dependent optimizations, which is optimization of code depending on the target machine architecture (i.e. register allocation, using special instructions of the target machine, ...). Other criteria is the scope of the optimization: local optimization is done inside of a basic block (i.e. local common subexpression elimination, use of algebraic identities, ...) or global optimization is done across basic blocks, or even across functions (i.e. strength reduction, code motion, ...) [2]. A very important tool to perform optimizations is Data Flow Analysis.

The way LLVM addresses all these types of optimizations is through passes [31]. These passes are independent and work by traversing some portion of a program to transform it. The person implementing the compiler can decide which of these optimization passes is going to apply and in which order. This is important because passes in LLVM are divided into analysis passes, transform passes and utility passes, and depending on the order, different passes can do opposite transformations, hurting the efficacy of the optimizations.

6.2 Backend in the Stoc compiler

After transforming a program in *Stoc* (source language) to LLVM IR (intermediate representation), the compiler still has to transform the intermediate representation into machine. This is done with the help of the LLVM static compiler tool, llc, which generates object code from LLVM IR.

This object code has to be passed through a linker to generate native executable. For the *Stoc* compiler, the first try was to use the GNU linker, ld, but it needed to define too many options to make it work and it would break easily. For this reason, in the end the GNU compiler, gcc, was used. The GNU compiler knows more about the machine where the code is being compiled and is capable of passing the correct options to the linker to generate the native executable.

7
Conclusion

In conclusion, we have implemented a compiler for *Stoc*, which is a simple programming language with C-like syntax. The compiler is composed of a custom frontend (lexing, parsing, semantic analysis and intermediate code generation) and uses the LLVM infrastructure for the backend.

Implementing every phase of the frontend allows to have a deeper understanding about the inner workings of real compilers and how incredibly complex they can be, because building the compiler for a rather small language was a challenge. This also shows that software engineering is a team effort and that when the projects start to be slightly big, only one person is not enough. It was even clearer when going through the source code of different real compilers for information about how different compilers work, like the Clang, Golang or Swift compiler.

A compiler is the central piece of a programming language, but when implementing the code generation for *Stoc*, especially implementing the string type and the print builtin functions, it became apparent that standard libraries play a very important role in the wide adoption of a programming language. Despite being things not so theoretical like lexing or parsing, concepts like good error handling or a good designed standard library are critical things to make the use of a programming language easier.

Stoc still has a huge space for future improvements, for instance adding features to the language, like structured types, or creating a standard library.

Bibliography

[1] https://nptel.ac.in/content/storage2/courses/106108113/module5/Lecture17.pdf

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Lattner. C, and Adve. V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, CA, USA, 2004.

[4] https://github.com/westes/flex/

[5] https://www.antlr.org/index.html

[6] https://www.gnu.org/software/bison/

[7] http://gcc.gnu.org/wiki/New_C_Parser

[8] https://github.com/llvm/llvm-project/tree/master/clang/lib/Lex

[9] http://clang.llvm.org/features.html#unifiedparser

[10] https://github.com/apple/swift/blob/master/lib/Parse/Lexer.cpp

[11] https://docs.swift.org/swift-book/ReferenceManual/zzSummaryOfTheGrammar.html

[12] https://github.com/golang/go/tree/master/src/go/scanner

[13] https://github.com/golang/go/blob/master/src/go/parser/parser.go

[14] https://golang.org/ref/spec

[15] https://github.com/rust-lang/rust/tree/master/src/librustc_lexer/src

[16] https://doc.rust-lang.org/stable/reference/

[17] https://kotlinlang.org/docs/reference/grammar.html#statements

[18] https://www.python.org/dev/peps/pep-0572/#relative-precedence-of

[19] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

[20] https://en.wikipedia.org/wiki/Pratt_parser

[21] https://craftinginterpreters.com/parsing-expressions.html#syntax-errors

[22] Pierce, Benjamin C., Types and Programming Languages. 2002, The MIT Press.

[23] https://craftinginterpreters.com/resolving-and-binding.html

[24] Barry Rosen; Mark N. Wegman; F. Kenneth Zadeck (1988). "Global value numbers and redundant computations". Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

[25] https://en.wikipedia.org/wiki/Static_single_assignment_form#Compilers_using_SSA_form
[26] https://en.wikipedia.org/wiki/LLVM
[27] https://llvm.org/docs/LangRef.html#abstract
[28] https://llvm.org/docs/LangRef.html#the-llvm-global-ctors-global-variable
[29] https://en.wikipedia.org/wiki/Name_mangling
[30] https://itanium-cxx-abi.github.io/cxx-abi/abi.html#demangler
[31] https://llvm.org/docs/Passes.html

A

Token Types in the Stoc Programming Language

Kind	Token Type	Regular Expression	Examples
Operators	ADD	'+'	
	SUB	'-'	
	STAR	'*'	
	SLASH	'/'	
	LAND	'&&'	
	LOR	'||'	
	NOT	'!'	
	ASSIGN	'='	
	EQUAL	'=='	
	NOT_EQUAL	'!='	
	LESS	'<'	
	GREATER	'>'	
	LESS_EQUAL	'<='	
	GREATER_EQUAL	'>='	
Delimiters	LPAREN	'('	
	RPAREN	')'	
	LBRACE	'{'	
	RBRACE	'}'	
	SEMICOLON	';'	
	COMMA	','	
Keywords	VAR	'var'	
	CONST	'const'	
	IF	'if'	
	ELSE	'else'	
	FOR	'for'	
	WHILE	'while'	
	FUNC	'func'	
	RETURN	'return'	
	BOOL	'bool'	
	INT	'int'	
	FLOAT	'float'	
	STRING	'string'	

Type Literals	LIT_TRUE LIT_FALSE LIT_INT LIT_FLOAT LIT_STRING IDENTIFIER	'true' 'false' [0-9]+ [0-9]+'.'[0-9]* ".*" [a-zA-Z_][a-zA-Z0-9_]*	 123, 513134 1.54 15. "astring" variable i
Special Tokens	ILLEGAL T_EOF		

If a character is enclosed between two single quotes ' it means that it is that literal character. The point . means any character.

B

Grammar of the Stoc Programming Language

Grammar of the Stoc Programming Language in Extended Backaus-Naur Form:

```
1        PROGRAM → TOPLEVELDECL+
2   TOPLEVELDECL → VARDECL
3                 | CONSTDECL
4                 | FUNCDECL
5
6        VARDECL → var Type Identifier = EXPR ;
7      CONSTDECL → const Type Identifier = EXPR ;
8       FUNCDECL → func Identifier ( PARAMLIST ) Type? BLOCKSTMT
9      PARAMLIST → PARAMDECL
10                | PARAMDECL (, PARAMDECL)+
11     PARAMDECL → var Type Identifier
12
13     BLOCKSTMT → { LISTSTMTS }
14     LISTSTMTS → STMT*
15          STMT → SIMPLESTMT
16                | IFSTMT
17                | FORSTMT
18                | WHILESTMT
19                | RETURNSMT
20    SIMPLESTMT → DECLSTMT
21                | EXPRSTMT
22                | ASSIGNMENTSTMT
23      DECLSTMT → DECL
24      EXPRSTMT → EXPR
25 ASSIGNMENTSTMT → VARIABLE = EXPR ;
26        IFSTMT → if EXPR BLOCKSTMT
27       FORSTMT → for (VARDECL | ASSIGNMENTSTMT)? ; EXPR? ;
     SIMPLESTMT? BLOCKSTMT
28      WHILESTMT → while EXPR BLOCKSTMT
29     RETURNSTMT → return EXPR ;
30
31          EXPR → BINARYEXPR
32    BINARYEXPR → UNARYEXPR
33                | EXPR BinaryOperator EXPR
34     UNARYEXPR → + EXPR
35                | - EXPR
36                | ! EXPR
37                | PRIMARYEXPR
38   PRIMARYEXPR → OPERAND
```

```
39                | CALLEXPR
40        OPERAND → TypeLiteral
41                | ( EXPR )
42                | Identifier
43       CALLEXPR → Identifier ( ARGS )
44           ARGS → ARG
45                | ARG (, ARG)+
46            ARG → EXPR
```

The terminals are represented in lowercase or in camelcase for the literals representing several values, like *Type, Identifier, TypeLiteral* or *BinaryOperators*.