Universität Bremen

Term Paper

in the course of studies Computer Science - Focus Compiler Construction

Topic: Development of an own programming language

Author: Rune Krauß <krauss@uni-bremen.de>
Student number: 4258388

Version from: February 6, 2019

Supervisor: Dr. Berthold Hoffmann

Contents

List of Figures

List of Tables

List of Codes

List of Abbreviations

ANTLR .... ANother Tool for Language Recognition
ASCII .... American Standard Code for Information Interchange
ASG .... Abstract syntax graph
AST .... Abstract syntax tree
BNF .... Backus-Naur Form
CFG .... Context-free grammar
CPU .... Central Processing Unit
DFA .... Deterministic finite automaton
EBNF .... Extended Backus-Naur Form
GPU .... Graphics Processing Unit
JVM .... Java Virtual Machine
NFA .... Nondeterministic finite automaton
TDD .... Test Driven Development

1 Introduction

Compilers are programs that transform a source language into a lower target language. A source program and a compiler serve as input. Afterwards, the execution takes place on a machine. On the other hand, an interpreter translates command by command and executes them directly on the machine. However, a compiler is faster because no interpreter has to be active at runtime and it can be optimized better but it is more difficult to implement. An example of a compiled language is C++[1]. Another example of an interpreted language is Python[2].

This paper deals with the implementation of a platform-independent compiler for a language called *E*. ANother Tool for Language Recognition (ANTLR), an object-oriented parser generator, is used for this purpose. Of course, there are already several compilers like C++ or Fortran[3]. However, there are several reasons why new compilers also make sense. On the one hand, a new way of thinking is to be tried out. Haskell[4] investigated e.g. laziness in the language or BASIC[5] wanted to make learning easier for programmers. Here, conventional language properties are to be removed as far as possible until a Turing machine-like language remains whereby the compiler is kept small. Furthermore, the language concept is kept very simple so that a lot can be described with as little expression as possible without limiting the universal readiness for use. Finally, concepts for application-oriented languages will be demonstrated. For example, these include loops and conditions. The goal is to explicitly promote structural thinking. During the implementation, special attention is paid to how trees are traversed efficiently so that the code can be generated optimally.

The compiler is designed according to various quality criteria. It must deliver correct results and react to any operating errors with a well-defined state. Furthermore there should be an understandable user documentation and the source code should be readable. Ultimately, another goal is portability on multiple systems. Software tests are used to detect and localize errors and contribute to quality assurance. Specifically, module tests and integration tests are used. Thus modules are also examined together.

The explanation of the compiler is based on the respective compiler phases which are subdivided into an analysis and synthesis (see chapter 2 on page 8). With regard to analysis, this concerns lexical, syntax and context analysis. The code generation is about generating assembler-like instructions. Furthermore, the module and integration tests are explained in chapter 3 on page 55 to investigate the quality of the compiler.

[1] C++, `http://www.cplusplus.com`
[2] Python, `https://www.python.org`
[3] Fortran, `http://www.fortran.de`
[4] Haskell, `http://www.haskell.org`
[5] BASIC, `http://www.media.salford-systems.com/pdf/spm7/BasicProgLang.pdf`

2 Compiler Phases

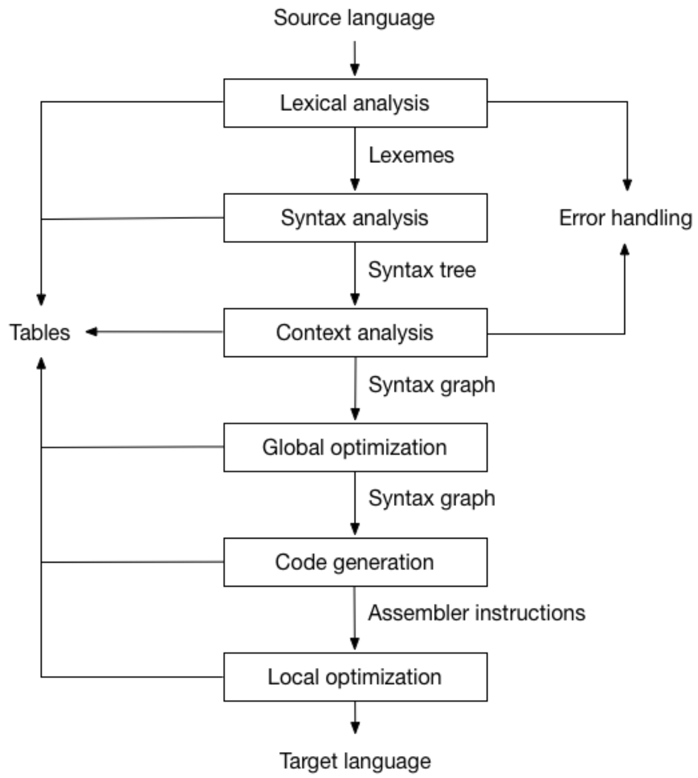

Figure 1: Compiler phases

The figure 1 shows the compilation process which starts with the input of a *source program*. The program must be written in *E*.

The lexical, syntax and context analysis are a part of the analysis phase. This is primarily about the properties of the source program. The *lexical analysis* is about recognizing words (see chapter 2.1 on page 12). The *syntax analysis* then builds a logical structure of the lexemes (see chapter 2.2 on page 18). The *context analysis*, on the other hand, includes e.g. a type test. The result is then a syntax graph. During these phases, *errors* are always handled. Accordingly, the system checks, for example, whether a variable was used before the definition. The *global optimization* makes an optimization on the source text. Thus e.g. isomorphies are summarized in the code.

The *code generation* represents the synthesis. The structure of the source program already exists there. The synthesis consists of three components:

- **Allocation**: The available components like a Graphics Processing Unit (GPU) are selected.
- **Binding**: The specific functions are mapped to the components.
- **Control flow**: The chronological sequence of the commands is displayed.

It generates assembler instructions and is therefore dependent on the respective instruction set of the processor. This refers, for example, to the allocation of registers which is generally NP complete [BDR06], i.e. there are only runtime-intensive algorithms for an exact solution. Moreover, hashtables are used for quick access to identifiers or strings. *Local optimization* is again based on basic blocks. These start with the entry into a control flow and end with the exit. However, they do not include jumps. More about this topic can be found in chapter 2.4 on page 47. The main goal here is to rearrange commands. The bytecode for the source language E then results.

Altogether, there is a difference between runtime and compile time which is shown in the tombstone diagram[6] 2:

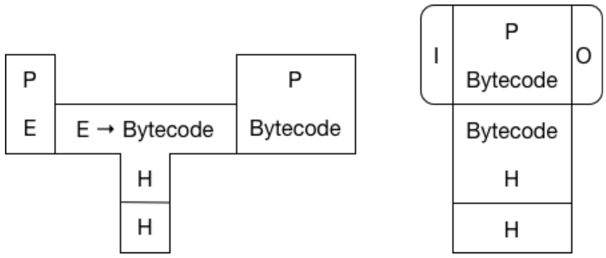

Figure 2: Compile time and runtime

The compile time refers to the compiler phases (see figure 1 on page 8). The language E is transformed by a host (H) to bytecode. The execution also happens there. The bytecode can then be interpreted at runtime where it is located in the Java Virtual Machine (JVM). Therefore it is also possible to generate code for different central processing units (CPU) like PowerPC or x86. However, the code can also be translated on a different architecture and then executed elsewhere. Accordingly, retargeting and rehosting are supported.

In the following, the respective components of the compiler will be explained. The following folder structure regarding the architecture is used as help:

```
e
   io
   ...
   std
   ...
   test
      main.e
grammar
   E.g4
```

[6] Tombstone diagram, `http://www.informatik.uni-bremen.de/agbkb/lehre/uebersetzer/Kopien/BootstrapWatt.pdf`

```
lib
   antlr.jar
   jasmin.jar
src
   main
      java
         com
            runekrauss
               compiler
                  CustomType.java
                  CustomTypeVisitor.java
                  DataType.java
                  DataTypeStack.java
                  EVisitor.java
                  FunctionDefinitionVisitor.java
                  FunktionPrototype.java
                  FunctionPrototypeList.java
                  Main.java
                  StaticVariableVisitor.java
                  TypeInformation.java
                  exception
                     AlreadyDefinedFunctionException.java
                     ...
                     UndeclaredVariableException.java
                     ...
                     WrongDataTypeException.java
               parser
                  E.tokens
                  ...
                  EBaseVisitor.java
                  ELexer.java
                  EParser.java
                  EVisitor.java
      resources
         assembly
            inline_asm.e
            ...
         branch
         comments
         function
```

```
        loop
        operators
        ...
  test
    java
      com
        runekrauss
          compiler
            CompilerTest.java
pom.xml
target
README.md
...
```

The folder *grammar* contains the defined grammar with the parser and lexer rules. In the folder *lib* you can also find the used libraries for ANTLR and Jasmin (see chapter 2.4 on page 47). In summary, a Maven[7] project is used where these are listed as dependencies in the *pom.xml* (see appendix on page 63). The actual source files are divided into the folders *compiler* and *parser*. The *compiler* folder contains the defined logic for traversing the tree with code generation (see chapter 2.3 on page 27) and the start method for starting a file such as *main.e* written in the language *E*. The folder *e* also contains the standard library of the language that can be imported. The *exception* package contains error classes for incorrect use of the described constructs regarding the context analysis of language *E*. For example, a message is generated when a variable is used although it has not yet been declared. Another example would be that an error message appears when a function is defined twice or if a data type is not compatible with an operation. In the *parser* folder there is a lexer and a parser to make analyses. The recognized tokens can also be found there. The unit and integrations tests can be found in the *test* folder (see chapter 3 on page 55). The respective tests also read test files from the *resources* folder if the corresponding source code is relatively long. Among other things, this refers to the tests for functions, branches, loops and operators. The respective compilates are created in the *target* folder.

Note: For a better overview, several comments have been removed in the codes. However, these are visible in the *E Compiler*[8] as a Maven project. The current description of the installation of the compiler is shown in the respective *README.md*.

[7] Maven, `https://maven.apache.org`
[8] E Compiler, `https://github.com/RuneKrauss/eCompiler`

2.1 Lexical analysis

Generally speaking, lexical analysis creates lexemes from source code. The scanner is responsible for recognizing lexemes. The screener, on the other hand, performs actions after recognition. This includes e.g. removing comments. Each lexeme consists of one or more characters. These include, for example, printable characters such as letters or numbers but also control characters such as tab stops. The encoding of the characters is limited to American Standard Code for Information Interchange (ASCII). The code can be found in the appendix on page 66.

The lexical analysis is done with finite automata, i.e. the lexemes are described by regular expressions.

Definition 1 (Deterministic finite automaton)**:**
A deterministic finite automaton (DFA) is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of a finite set of states Q, a finite set of input symbols called the alphabet Σ, a transition function $\delta : Q \times \Sigma \to Q$, an initial or start state $q_0 \in Q$ and a set of accept states $F \subseteq Q$.

Definition 2 (Nondeterministic finite automaton)**:**
A nondeterministic finite automaton (NFA) is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, consisting of a finite set of states Q, a finite set of input symbols Σ, a transition relation $\Delta : Q \times \Sigma \to P(Q)$, an initial state $q_0 \in Q$ and a set of states F distinguished as accepting or final states $F \subseteq Q$. Here, $P(Q)$ denotes the power set of Q.

Definition 3 (Regular expression)**:**
Be Σ an alphabet. $\emptyset$ and ϵ are regular expressions, a is a regular expression of Σ for all $a \in \Sigma$. If R_1 and R_2 are regular expressions, then also $R_1 \cup R_2$, $R_1 \circ R_2$ and R_1^* are regular expressions.

Definition 4 (Regular language)**:**
A regular language is a language that can be expressed with a regular expression or a deterministic or non-deterministic finite automata or state machine.

Definition 5 (Grammar)**:**
A grammar is a 4-tuple, (N, Σ, P, S), consisting of a finite set N of nonterminal symbols, a finite set Σ of terminal symbols, a finite set P of production rules with $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$ and a distinguished symbol $S \in N$ that is the start symbol.

Type-3 grammars generate regular languages and have a single non-terminal on the left-hand side as well as a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal. The productions have the form $X \to a$ or $X \to aY$ where $X, Y \in N$ and $a \in \Sigma$. Moreover, the rule $S \to \epsilon$ is allowed if S does not appear on the right side of any rule.

For the language E there are different lexemes which can be distinguished as follows in code 1:

```
// --------------------------------------------------------------------------------
// Here are the built-in functions of the language E where an access is possible even
// without imports.

BUILTINFUNCTION                 : 'toInt'
                                | 'toFloat'
                                | 'toString'
                                | 'append'
                                | 'length'
                                ;

// --------------------------------------------------------------------------------
// Represents the two truth values of logic (true, false).

BOOL                            : 'true'
                                | 'false'
                                ;

// --------------------------------------------------------------------------------
// Refers to integers

INTEGER                         : (DIGIT)+
                                ;

// --------------------------------------------------------------------------------
// Refers to floating point numbers

FLOAT                           : INTEGER DOT INTEGER
                                | DOT INTEGER
                                ;

// --------------------------------------------------------------------------------
// Any number of characters, but as little as possible that the rule is still
// fulfilled

STRING                          : QMARK .*? QMARK
                                ;

// --------------------------------------------------------------------------------
// Identifiers

IDENTIFIER                      : LETTER(LETTER | DIGIT)*
                                ;

// --------------------------------------------------------------------------------
// (Multi)line comments (but as little as possible that the rule is still fulfilled)

COMMENT                         : ('//' ~[\r\n]* '\r'? '\n' | '/*' .*? '*/' | '/**'
                                  .*? '*/') -> skip
;

// --------------------------------------------------------------------------------
// Ignore control characters (the screener removes these)

WHITESPACE                      : [ \t\n\r]+ -> skip
                                ;

```

```
// ---------------------------------------------------------------------------
// Here are the repeating tokens in the grammar

SCOLON                   : ';'
                         ;

DOT                      : '.'
                         ;

COMMA                    : ','
                         ;

ASSIGN                   : '='
                         ;

QMARK                    : '"'
                         ;

OPAREN                   : '('
                         ;

CPAREN                   : ')'
                         ;

OBRACE                   : '{'
                         ;

CBRACE                   : '}'
                         ;

OBRACKET                 : '['
                         ;

CBRACKET                 : ']'
                         ;

OCBRACKET                : '[]'
                         ;

// ---------------------------------------------------------------------------
// Letters (will never be counted as a token)

fragment LETTER          : [a-zA-Z_]
                         ;

// ---------------------------------------------------------------------------
// Digits (used for data types and identifiers)

fragment DIGIT           : [0-9]
                         ;
```

Code 1: Lexer rules of grammar `E`

Conventionally, the rules for the Lexer are written in capital letters. A distinction is also made between meta and object characters. Thus, for example `+` is a metacharacter which states that an integer must consist of at least one digit. However, `'toInt'` is an object character that defines a built-in function. Repeating tokens such as brackets

`{` etc. are stored separately. The different comments are removed by the screener. In addition, there are different data types:

- Booleans like `true`
- Integers like `5`
- Floating point numbers like `0.5` or `.5`
- Strings like `"world"`
- Arrays
- Structures and the corresponding objects

The concrete composition of the arrays and structures can be seen in chapter 2.2 on page 18. A data type can be primitive or an object associated with references whereby they consist of fragments such as digits. A fragment will never be counted as a token, it only serves to simplify a grammar and makes the grammar more readable as well as easier to maintain. A special feature of the strings is that they may consist of any number of characters but only so few that the rule is still fulfilled. This ensures that strings may not apply beyond quotation marks. The same principle applies to comments. The keywords are still reserved. Furthermore, the layout is not ignored. For this reason, no lookahead is required for the automatons. Finally, the performance is increased. In addition, the compiler also knows the position in the source code so that more concrete error messages can be output (see chapter 2.3 on page 27). The generation of the automatons or tables for the lexemes is shown in the following figure 3:

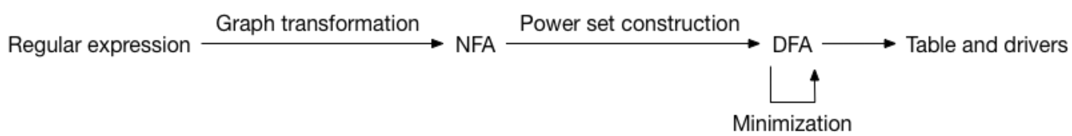

Figure 3: Workflow of the lexical analysis

The workflow of the scanner is explained below using the example `R = LETTER(LETTER | DIGIT)*` (see figure 4 on page 16). First of all, the graph transformation takes place on the basis of rules. These come from the proof of the equivalence of finite automata and regular expressions [Vis13]. The rules for graph transformation can be found in the appendix on the page 68.

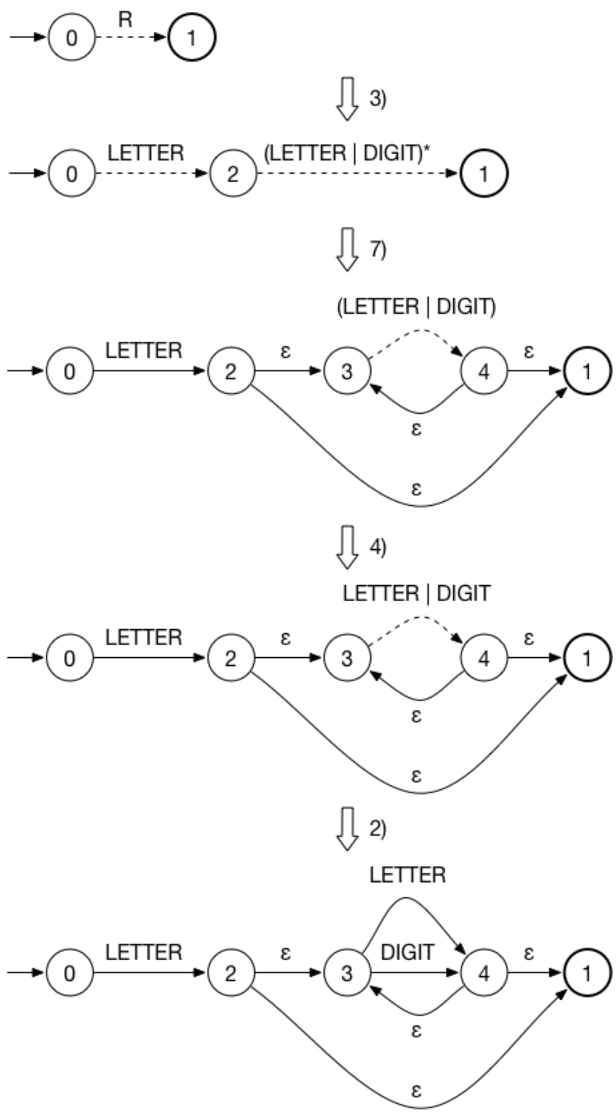

Figure 4: Graph transformation on a regular expression

Accordingly, a NFA with epsilon transitions is formed. The epsilon transitions must then be removed and the powerset construction applied[9]. First, states are marked for this purpose:

[9]Powerset construction, `http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node9.html`

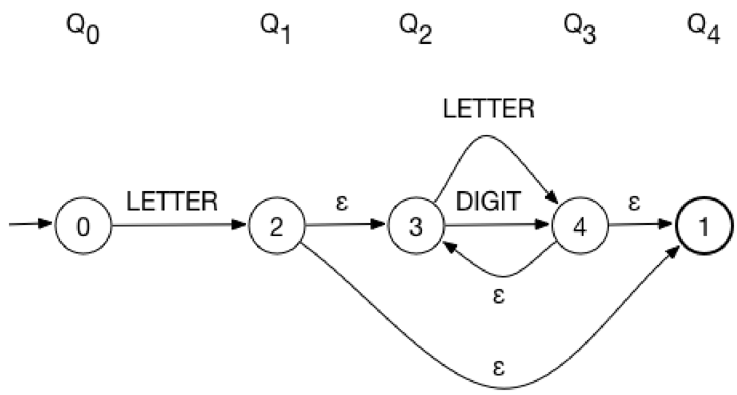

Figure 5: Powerset construction at a NFA

According to figure 5, the states $Q_0 = \{0\}, Q_1 = \{2,3,1\}, Q_2 = \{3\}, Q_3 = \{4,3,1\}$ are reached depending on the epsilon transitions. Q_4 is dropped here because there is nothing to read. Because the DFA states consist of sets of NFA states, an n-state NFA may be converted to a DFA with at most 2^n states. A converted DFA has exactly 2^n states, so giving an exponential time complexity. However, this runtime can be neglected since it is before the compile time.

When minimizing, the same states are combined or unattainable states are removed. Two states are equal if they have the same transitions. In concrete terms, equivalence classes are formed by applying the Table Filling method[10]. The resulting minimum DFA is shown in the figure 6 below.

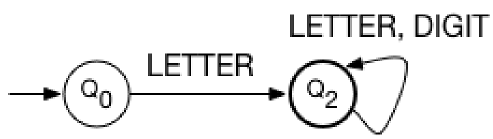

Figure 6: DFA after minimization

Afterwards, the DFA is mapped to a table 1:

Table 1: Realization of a DFA in a table

	LETTER	**DIGIT**
0	2	
2	2	2

Valid words or identifiers are thus e.g. “_test” or “t3st”. Finally, all DFAs are combined for the respective lexemes. Alternatively, GOTO or WHILE programs[11] can be used

[10]Table Filling method, `http://pages.cs.wisc.edu/~shuchi/courses/520-S08/handouts/Lec7.pdf`

[11]GOTO and WHILE programs, `https://www.informatik.hu-berlin.de/de/forschung/gebiete/algorithmenII/Lehre/ws11/einftheo/skript/einftheo-goto.pdf`

instead of tables to show computability. The corresponding code can be found in the appendix on the page 69.

2.2 Syntax analysis

The syntax analysis generates an abstract syntax tree (AST) from lexeme sequences. The basis for this is the extended Backhus-Naur Form (EBNF) which is a metalanguage for the description of context-free or Type-2 grammars (CFGs). Type-2 grammars generate context-free languages. The productions must be in the form $X \rightarrow a$ where $X \in N$ and $a \in (\Sigma \cup N)^*$. These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Definition 6 (BNF)**:**
The BNF is a formal notation for encoding grammars intended for human consumption. Every rule in BNF has the structure name ::= expansion where ::= means "may expand into" and "may be replaced with". The name stands for a non-terminal symbol. Every name in BNF is surrounded by angle brackets, <> whether it appears on the left- or right-hand side of the rule. An expansion is an expression containing terminal and non-terminal symbols, joined together by sequencing and choice. A terminal symbol is a literal or a class of literals.

Definition 7 (EBNF)**:**
The EBNF is a collection of extensions to BNF but not all of these are strictly a superset, as some change the rule-definition relation ::= to =, while others remove the angled brackets from non-terminals. Additional operations are options (<term> ::= ["-"] <factor>), repetitions (<args> ::= <arg> "," <arg>) and groupings (<expr> ::= <term> ("+" | "-") <expr>).

In contrast to the conventional BNF, the EBNF therefore requires fewer constructs to describe CFGs. For example, options [] or repetitions can be used directly to present constructs more compactly instead of using recursions [Chu09].

Essentially, a derivation tree is generated by the parser during the analysis which also states how it was derived. In this way, a differentiation was also made to a recognizer. As already mentioned in the introduction, ANTLR is used as a parser generator. ANTLR uses the top-down approach instead of a bottom-up parsing, i.e. a recursive descent parser is used. The input is processed from left to right by calculating a left derivation. This means that an attempt is made to create the desired word from a start symbol. Note that there are several language classes of CFGs that can be parsed deterministically whereby ANTLR handling $LL(k)$ grammars. The k stands for the respective lookaheads that can be made. If there are many lookaheads, there are fewer conflicts but more states. At E $k = 1$ was set. The language classes are listed in the appendix on page 71.

The parser is divided into three parts, a configuration, execution and actions which are explained in the following table 2:

Table 2: Functionality of the recursive descent parser

Part	**Description**
Configuration	$\alpha.w$
Execution	$s.w_0 \vdash^* \epsilon.\epsilon$
Actions	*match*: $a\alpha.aw \vdash_a \alpha.w$ *expand*: $A\alpha.w \vdash_p \beta\alpha.w$, where $p : A \rightarrow \beta \in P$

Here, α corresponds to the hypothesis N^* and w is the unread input. With regard to the execution, the start and end configuration are visible. The *match* function is called when a terminal is derived. With *expand*, on the other hand, a rule is generally applied. It is obvious that *expand* as oracle advises the next application of a rule in doubt. If there is an error or a match, backtracking must be used and there is an exponential runtime. In order to obtain a linear runtime, special care was taken to define a deterministic CFG for the language E.

Definition 8 (Ambiguous grammar)**:**
A grammar means ambiguous if $w \in \Sigma^*$ so that several derivation trees or left and right derivations exist.

In general, it is undecidable whether a grammar is ambiguous which can be shown by a reduction of PKP to this problem [HM11, p.461-462]. In addition, a left factorization $[A \rightarrow \alpha\beta_1\gamma, A \rightarrow \alpha\beta_2\gamma] \rightarrow [A \rightarrow \alpha B\gamma, B \rightarrow \beta_1, B \rightarrow \beta_2]$ $(\alpha \neq \epsilon)$ was used for the recursive descent regarding the processing. Thus, for example, precedents in arithmetic or logical operations are taken into account. In the following, the CFG for the language E in code 2 is presented and explained:

```
// Start rule
// It is evaluated from left to right.
// A program consists of statements, functions or structures.

program                    : incls+=includes* noMains+=noMain* command+ EOF
                           ;

// ------------------------------------------------------------------------
// Allows the import of different modules for e.g. mathematical calculations.

includes                   : 'use' OPAREN (mods+=module)+ (COMMA (mods+=module)*)*
                             CPAREN
                           ;

// ------------------------------------------------------------------------
// A namespace can consist of several modules that are separated from each other by
```

```
// dots.

module                  : IDENTIFIER (DOT IDENTIFIER)*
                        ;

// ----------------------------------------------------------------------------
// This indicates that no starting point is generated for a class.

noMain                  : '#define' name='noMain'
                        ;

// ----------------------------------------------------------------------------
// This is a helper for the program because of functions and structs must not be in
// the main method during code generation. Therefore statements, structs and
// functions are separated.

command                 : statement #StatementCommand
                        | functionDefinition #FunctionDefinitionCommand
                        | structDeclaration #StructDeclarationCommand
                        ;

// ----------------------------------------------------------------------------
// Statements in the program code (also without a semicolon at the end)
// Statements can stand alone and are not dependent on an expression.

statement               : print SCOLON
                        | printLine SCOLON
                        | variableDeclaration SCOLON
                        | assignment SCOLON
                        | functionCall SCOLON
                        | branch
                        | loop
                        | includedFunctionCall SCOLON
                        | builtinFunctionCall SCOLON
                        | assembly
                        ;

// ----------------------------------------------------------------------------
// Responsible for outputs without a new line

print                   : 'print' OPAREN arg=expression CPAREN
                        ;

// ----------------------------------------------------------------------------
// Responsible for outputs with a new line

printLine               : 'println' OPAREN arg=expression CPAREN
                        ;

// ----------------------------------------------------------------------------
// Allows a declaration of variables (including an immediate initialization)

variableDeclaration     : type=dataType varId=IDENTIFIER (ASSIGN expr=expression)?
                        ;

// ----------------------------------------------------------------------------
// Assigning values to a variable or struct

assignment              : varId=IDENTIFIER ASSIGN expr=expression
                        | varId=IDENTIFIER OBRACKET index=expression CBRACKET
```

```
                                ASSIGN expr=expression
                              | structVariableAssignment
                              ;

// ------------------------------------------------------------------------
// This can be used to assign values to structures.

structVariableAssignment      : structId=IDENTIFIER DOT varId=IDENTIFIER ASSIGN
                                expr=expression
                              | structId=IDENTIFIER DOT varId=IDENTIFIER OBRACKET
                                index=expression CBRACKET ASSIGN expr=expression
                              ;

// ------------------------------------------------------------------------
// Indicates a function call (with arguments).

functionCall                  : funcId=IDENTIFIER OPAREN
                                currentParams=currentParameters CPAREN
                              ;

// ------------------------------------------------------------------------
// Identifies a branch through which decisions can be made (else block is optional).

branch                        : 'if' OPAREN cond=expression CPAREN onTrue=block
                                ('else' onFalse=block)?
                              ;

// ------------------------------------------------------------------------
// Realizes while loops whereby Turing-completeness applies.

loop                          : 'while' OPAREN cond=expression CPAREN body=block
                              ;

// ------------------------------------------------------------------------
// With this you can directly write assembler in high level language. Assembly can be
// executed or written directly. However, objects can also be initialized directly.

assembly                      : 'asm' OBRACE str=STRING CBRACE #InlineAssembly
                              | 'invoke' mod=STRING id=STRING OPAREN args+=jvmType*
                                CPAREN returnType=STRING SCOLON #InvokeAssembly
                              | 'new' type=STRING SCOLON #InitObject
                              | 'pushToStack' expression SCOLON #PushToStack
                              | 'setTopOfStack' type=STRING SCOLON #SetTopOfStack
                              ;

// ------------------------------------------------------------------------
// The supported types of the JVM

jvmType                       : STRING
                              ;

// ------------------------------------------------------------------------
// This can be used to access functions from other packages.

includedFunctionCall          : inclDir=IDENTIFIER DOT funcId=IDENTIFIER OPAREN
                                args=currentParameters CPAREN ':' type=dataType
                              ;

// ------------------------------------------------------------------------
// Can be used to call built-in functions such as castings (without any imports).
```

```

builtinFunctionCall          : funcId=BUILTINFUNCTION OPAREN
                               currentParams=currentParameters CPAREN
                             ;

// -----------------------------------------------------------------------------
// Definition of a function with parameters and a body with (several) statements
// (a return value is optional)

functionDefinition           : type=dataType funcId=IDENTIFIER OPAREN
                               formalParams=formalParameters CPAREN OBRACE
                               stmts=statements ('return' returnVal=expression
                              SCOLON)? CBRACE
                             ;

// -----------------------------------------------------------------------------
// Describes the declaration of a structure with at least one variable.

structDeclaration            : 'struct' structId=IDENTIFIER OBRACE
                               (decls+=variableDeclaration SCOLON)+ CBRACE
                             ;

// -----------------------------------------------------------------------------
// Initializes an object.

structArrayInitialization    : 'new' object=dataType OPAREN args=assignments CPAREN
                             | 'new' type=primitive OBRACKET size=expression CBRACKET
                             ;

// -----------------------------------------------------------------------------
// Helper for initialization to assign values

assignments                  : asgmts+=expression (COMMA asgmts+=expression)*
                             ;

// -----------------------------------------------------------------------------
// Mathematical operators
// Precedence (partial order): neg, (div, mul, rem), (sub, add), comp (same
// precedences), and, or, ... Labels allow access in the code.

expression                   : '-' expression #UnaryMinusExpression
                             | expression op=('/' | '*' | '%') expression
                               #DivisionMultiplicationModuloExpression
                             | expression op=('-' | '+') expression
                               #SubtractionAdditionExpression
                             | expression op=('<<' | '>>') expression #ShiftExpression
                             | expression op=('<' | '<=' | '>' | '>=' | '==' | '!=')
                               expression #RelationalExpression
                             | lExpr=expression '&&' rExpr=expression
                               #ConjunctionExpression
                             | lExpr=expression '||' rExpr=expression
                               #DisjunctionExpression
                             | expression '^' expression #ContravalenceExpression
                             | varId=IDENTIFIER OBRACKET index=expression CBRACKET
                               #ArrayExpression
                             | structId=IDENTIFIER DOT varId=IDENTIFIER
                               #StructExpression
                             | structId=IDENTIFIER DOT varId=IDENTIFIER OBRACKET
                               index=expression CBRACKET #StructArrayExpression
                             | structInitialization #StructInitializationExpression
```

```
                                | bool=BOOL #BoolExpression
                                | number=INTEGER #IntegerExpression
                                | number=FLOAT #FloatingPointExpression
                                | str=STRING #StringExpression
                                | varId=IDENTIFIER #VariableExpression
                                | builtinFunctionCall #BuiltinFunctionExpression
                                | functionCall #FunctionExpression
                                | includedFunctionCall #IncludedFunctionExpression
                                | 'topOfStack' #TopOfStack
                                ;

// ---------------------------------------------------------------------------
// The current parameter list of functions can be of any length (no visitor).

currentParameters               : exprs+=expression (COMMA exprs+=expression)*
                                |
                                ;

// ---------------------------------------------------------------------------
// Provides a basic block with an entry and exit from a control flow (no visitor).

block                           : OBRACE statements CBRACE
                                ;

// ---------------------------------------------------------------------------
// Helper for more statements regarding blocks (no visitor)

statements                      : statement*
                                ;

// ---------------------------------------------------------------------------
// The formal parameter list of functions can be of any length (no visitor).
// The number of parameters is also saved.

formalParameters                : decls+=variableDeclaration (COMMA
                                  decls+=variableDeclaration)*
                                |
                                ;

// Represents the different data types (including lists and objects)

dataType                        : primitive(OCBRACKET)?
                                | IDENTIFIER(OCBRACKET)?
                                ;

// ---------------------------------------------------------------------------
// The supported (primitive) types

primitive                       : 'bool'
                                | 'int'
                                | 'float'
                                | 'String'
                                | 'void'
                                ;
```

Code 2: Parser rules of grammar `E`

It should be mentioned that first as well as follow sets $F_i(N)$ and $F_o(N)$ were created to promote a left factorization which is why double entries are not allowed. The first set includes the start of a derivative of the variable in the grammar and the follow set the subsequent derivatives of the symbols that are possible or where the variable can be used. It applies, for example, $F_i(variableDeclaration) = \{var\}$ and $F_o(variableDeclaration) = \{'=', ';'\}$. Conventionally, the parser rules consist only of minuscules. The start symbol of the grammar is `program`. The evaluation takes place from left to right. In the grammar different constructs of the language E are listed. A program consists of commands, i.e. statements, structures and functions. `EOF` means in this context that it definitely parses to the end of the file, even if errors occur before it. This results in a total output of all errors made. Statements must be in the main program or in blocks of functions or structures. They can also contain expressions or appear again in statements. An example is `if-else`. It may contain expressions, but may appear in a loop. Functions as well as structures are outside in this context. They are separated from the main program. This means that definitions and declarations can also be made after use. There are commands such as:

- Imports like `use(e.io.reader)`
- Import calls like `String r = reader.read(): String;`
- Macro Instructions like `#define noMain`
- Outputs like `print("world");`
- Variable declarations or assignments like `int a;` as well as `float a = 5.3;`
- Function definitions like `void foo(int a, int b) { print(a*b); }`
- Function calls like `foo(7, 8);`
- Branches like `if (x) { a = 5; } else { a = 7; }`
- Loops like `int i = 0; while (i < 3) { println(i); i = i + 1; }`
- Built-in functions like `toInt(3.7);`
- Inline assembler like `invoke "static" "java/lang/System/nanoTime"() "J";`
- Structures like `struct Point { int x; int y; }`
- Struct initializations like `Point p = new Point(1, 2);`
- Arrays like `int[] a = new int[5];`
- Arithmetic operators like `a + b`
- Shift operators like `a >> b`
- Relational operators like `a <= b`
- Logical operators like `a && b`

The use of while loops make E Turing complete [Zei18]. For this reason, it can be used to simulate any Turing machine. This means that this system is able to recognize or decide other data-manipulation rule sets. It is also possible, for example, to specify expressions such as multiplication or a number directly within the output. In order to increase readability, various labels such as `arg=expression` has been assigned to `print`. This makes it easier to access the respective expression such as a number or string by `arg` via the context within the visitor pattern during traversing (see chapter 2.3 on page 27). The parser rules therefore directly access the lexer rules. Labels were also awarded for a rule that allows several derivatives. An example of this is the subtraction or addition that has the label `#SubtractionAdditionExpression`. Thus this rule can later be addressed directly by `visitSubtractionAdditionExpression` during traversing and also has its own context `SubtractionAdditionExpressionContext`. The same applies analogously to the other commands. Labels must not comply with the rules or be duplicated. The respective precedents are guaranteed by the order in which ANTLR4 follows sequentially. Different expressions can also be used in blocks such as branches. This allows statements to exist that do not have a semicolon at the end. In addition, symbols such as `+=` ensure that parameters can be counted which is particularly important for functions and later type analysis. Basically, functions are divided into call and definition by `functionDefinition` as well as `functionCall`. Functions are defined in the header and there does not have to be a declaration of variables. Furthermore, they can also be overloaded, i.e. the same name with different but arbitrarily long parameters is possible. The same principle applies to self-defined types such as structures.

The top-down parser uses a (simplified) abstract syntax to create the commands shown in the figure 7:

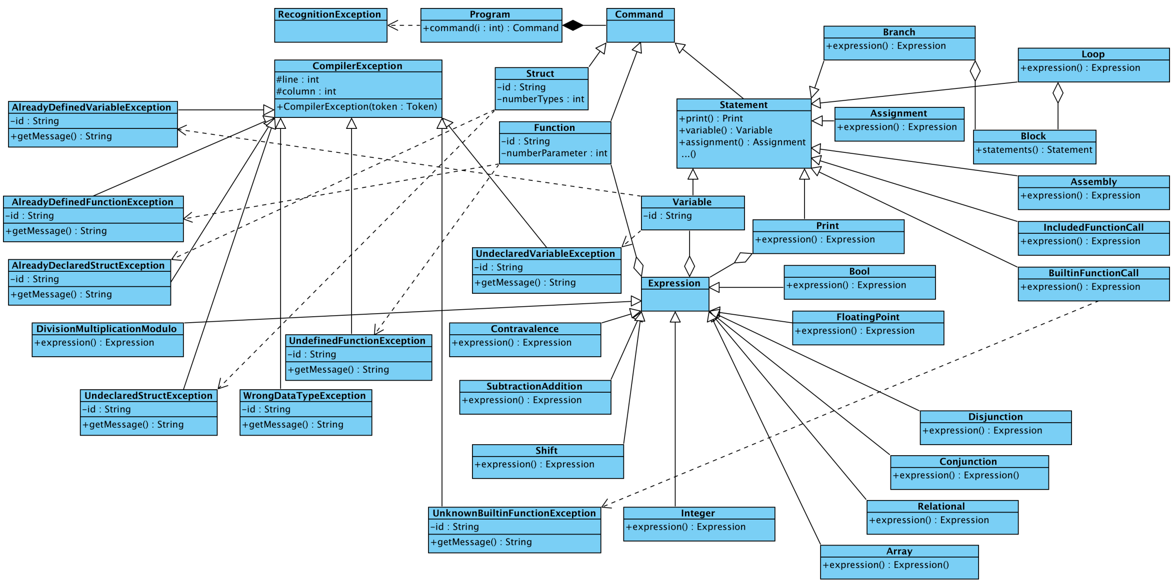

Figure 7: Abstract syntax

The AST is therefore described as an object structure by the abstract syntax. For simplicity, further edge markers, constructors and query functions have been omitted. Among other things, it can be recognized that commands cannot exist without a program. Furthermore, variables or functions can be used in expressions that are evaluated. Expressions can also be used, for example, in branches. Certain associated error classes ensure that users of language E are notified of incorrect operation. The corresponding instructions for creation are in this document (simplified) parser and can be found in the appendix on page 72. There is a procedure for each parser rule where the next lexeme l is treated. Error handling is also performed during parsing. For example, the character `@` does not belong to any expression and in this case a `RecognitionException` would be thrown because this object character is not known. If a procedure is called, then $l \in F_i$ applies before. Then $l \in F_o$ is valid.

In the following, an AST is generated from the program code `print(3+2*4)` as an example:

$$\begin{aligned}
program &\vdash command;\\
&\vdash statement;\\
&\vdash print;\\
&\vdash print(expression);\\
&\vdash print(expression\ +\ expression);\\
&\vdash print(3 + expression);\\
&\vdash print(3 + expression * expression);\\
&\vdash print(3 + 2 * expression);\\
&\vdash print(3 + 2 * 4);
\end{aligned}$$

Thus, a derivation tree could be successfully generated whereby the precedents are controlled by the ranking order. This also means that an ambiguous grammar must be checked. The corresponding AST is shown in figure 8 on page 27:

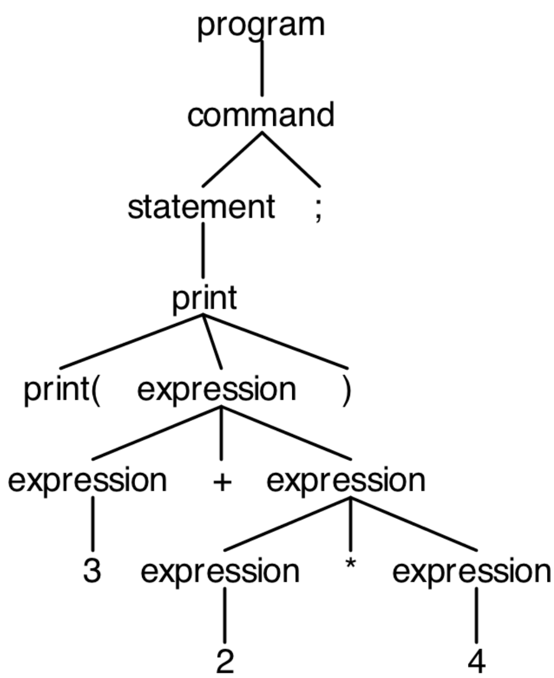

Figure 8: Abstract syntax tree

Due to the recursive definitions, a tree with theoretically infinite depth is created and can be traversed post-order to generate the instructions which will be explained in the next chapter.

2.3 Context analysis

In general, an AST is marked with attributes during context analysis. These stand for cross-references with regard to definitions or usage. The AST then becomes an abstract syntax graph (ASG) which can also contain cycles due to recursions. The general steps are as follows:

1. **Declaration analysis**: Checks, for example, whether a variable was declared before use.
2. **Type analysis**: Checks, for example, whether formal and current parameters match for functions.

Thus, all rules that cannot be covered with a scanner (see chapter 2.1 on page 12) or parser must be checked. In the context, this includes both static (variable must have been defined) and dynamic conditions (a variable must be in the index area).

To describe the context conditions, a syntax-directed transduction is used instead of a two-level grammar, i.e. recursive functions are used on the AST.

On the described object structure from chapter 2.2 on page 18 a general visitor can be defined who traverses the tree from which can be inherited later. A visit function is used to visit all connected elements. From this general visitor different visitors can be derived who define different operations on the AST. An excerpt of this visitor can be seen in the following code 3:

```
public class EVisitor extends EBaseVisitor<String> {
    /** A list with all already defined functions */
    private final FunctionPrototypeList functions;

    /** Structs with declared variables and types (with references) */
    private final LinkedHashMap<String, CustomType> structs;

    /** Static variables in the main program (references to structs) */
    private final Map<String, TypeInformation> staticVarsNamesTypes;
...
    /** The namespace of the program you want to compile currently */
    private final String namespace;

    /** Descriptor for own declared types as reference */
    private final String typeNamespace;

    /** Parent directory (path to file for legacy) */
    private final File parentDir;

    /** Saves the module name as well as the path */
    private final Map<String, String> includedModules;

    /** Should a main method be created? */
    private boolean noMain;

    /** The variables in the table are accessed numerically. */
    private Map<String, TypeInformation> vars;

    /** Logs the used data types at compile time. */
    private DataTypeStack dataTypeStack;

    /** Is one in a function or the main program? */
    private boolean isGlobalScope;

    /** Counts the branches because no label may be ambiguous. */
    private int branchCounter;
...
    /** Needed when initializing a structure */
    private String lookupStoreCommand;

    /** Also required when initializing a structure */
    private String lookupLoadCommand;

    /** Holds the current id of a struct */
```

```
    private int lookupStructId;

    public EVisitor(FunctionPrototypeList definedFunctions,
        LinkedHashMap<String, CustomType> declaredStructs,
        Map<String, TypeInformation> declaredStaticVars, String namespace,
        ...) {
        ...
    }

    @Override
    public String visitProgram(ProgramContext ctx) {
        // Generate potential code for the imports
        final int importNumber = ctx.incls.size();
        for (int i = 0; i < importNumber; ++i) {
            visit(ctx.getChild(i));
        }
        final int noMainNumber = ctx.noMains.size();
        for (int i = importNumber; i < noMainNumber; ++i) {
            visit(ctx.getChild(i));
        }
        final StringBuilder mainInstructions = new StringBuilder();
        final StringBuilder functionInstructions = new StringBuilder();
        // Regarding structs
        final StringBuilder extraClassesInstructions = new StringBuilder();
        for (int i = importNumber; i < ctx.getChildCount() - 1; ++i) {
            final ParseTree child = ctx.getChild(i);
            // Visit a parse tree and return a result of the operation
            final String instructions = visit(child);
            if (child instanceof StatementCommandContext) {
                mainInstructions.append(instructions).append('\n');
            } else if (child instanceof StructDeclarationCommandContext) {
                // The structures are marked accordingly because afterwards
                // own files are created for them
                extraClassesInstructions.append("\n*").append(instructions)
                    .append('\n');
            } else {
                functionInstructions.append(instructions).append('\n');
            }
        }
        final StringBuilder instructions =
            new StringBuilder(".class public ")
            .append(namespace)
            .append("\n.super java/lang/Object\n\n")
            .append(staticVars.toString())
            .append('\n')
            .append(functionInstructions.toString())
            .append('\n');
        if (!noMain) {
```

```
            instructions
                .append(".method public static main([Ljava/lang/String;)V\n")
                .append(".limit stack ")
                .append(dataTypeStack.getMaxStackSize())
                .append("\n.limit locals 1\n\n")
                .append(mainInstructions.toString()).append("\nreturn\n\n")
                .append(".end method")
                .append(extraClassesInstructions.toString());
        } else {
            instructions.append(mainInstructions.toString()).append('\n')
                .append(extraClassesInstructions.toString());
        }
        return instructions.toString();
    }
...
    @Override
    public String visitIncludes(IncludesContext ctx) {
        for (int i = 0; i < ctx.mods.size(); ++i) {
            final String namespace = ctx.mods.get(i).getText()
                .replace('.', '_');
            final File file = new File(parentDir, namespace);
            final String address =
            file.getAbsolutePath().substring(parentDir
                .getAbsolutePath().length() + 1);
            if (!Main.programs.contains(address) &&
                !Main.compiledPrograms.contains(address)) {
                // Add the respective module to the compilation list
                Main.programs.add(address.replace("_", "/"));
            }
            final String fileName =
                file.getName().split("_")[file.getName()
                    .split("_").length - 1];
            // Stores the filename ("math") and the path ("e_std_math")
            // to the file (to compile)
            includedModules.put(fileName, address);
        }
        return "";
    }

    @Override
    public String visitPrint(PrintContext ctx) {
        final String argumentInstructions = visit(ctx.arg);
        // Get type to output
        final TypeInformation typeInfo = dataTypeStack.pop();
        // Get the associated JVM type
        String jvmType = typeInfo.getJvmType();
        if (typeInfo.getDataType() == DataType.OBJREF) {
            // Get the address (position) of the struct
```

```
            jvmType = 'L' + typeNamespace + jvmType + ';';
        }
        // Get the type that "visit" put on the stack
        return "getstatic java/lang/System/out Ljava/io/PrintStream;\n"
            + argumentInstructions
            + "\ninvokevirtual java/io/PrintStream/print("
            + jvmType + ")V\n";
    }
...
    @Override
    public String visitVariableDeclaration(VariableDeclarationContext
        ctx) {
        String instructions = "";
        final Token varIdToken = ctx.varId;
        final String varId = varIdToken.getText();
        final String type = ctx.type.getText();
        // Get type object by type name
        final DataType dataType = DataType.getType(type);
        // Command to store a variable
        String storeCommand = "";
        final ExpressionContext exprContext = ctx.expr;
        if (isGlobalScope) {
            // When you are in the main method
            // Get type information by variable name
            final TypeInformation typeInfo = staticVarsNamesTypes
                .get(varId);
            lookupStoreCommand = "putstatic " + namespace + "/v" + typeInfo
                .getId();
            lookupLoadCommand = "getstatic " + namespace + "/v" +
                typeInfo.getId();
            storeCommand = "putstatic " + namespace + "/v";
            // Descriptor for field access
            String descriptor;
            if (dataType == DataType.OBJREF) {
                descriptor = "L" + typeNamespace + structs.get(type)
                    .getId() + ';';
                vars.put(varId, new TypeInformation(vars.size(),
                    dataType, structs.get(type).getId()));
            } else {
                descriptor = dataType.getJvmType();
                vars.put(varId, new TypeInformation(vars.size(), dataType));
            }
            staticVars.append(".field public static v")
                .append(getVariableIndexByVariableIdToken(varIdToken))
                .append(' ').append(descriptor).append('\n');
            if (exprContext != null) {
                instructions = visit(exprContext) + '\n' + storeCommand +
                    getVariableIndexByVariableIdToken(varIdToken) + ' ' +
```

```
                    descriptor;
                if (dataTypeStack.peek().getDataType() != dataType) {
                    throw new WrongDataTypeException(varIdToken);
                }
                dataTypeStack.pop();
                return instructions;
            }
            return instructions;
        } else {
            // When you are in a function
            if (vars.containsKey(varId)) {
                throw new AlreadyDeclaredVariableException(varIdToken);
            }
            if (dataType == DataType.OBJREF) {
                vars.put(varId, new TypeInformation(vars.size(), dataType,
                    structs.get(type).getId()));
            } else {
                vars.put(varId, new TypeInformation(vars.size(), dataType));
            }
            final TypeInformation typeInformation = vars.get(varId);
            if (typeInformation.isArray() || dataType == DataType.STRING ||
                dataType == DataType.OBJREF) {
                storeCommand = "astore";
                lookupStoreCommand = storeCommand + ' ' + typeInformation
                    .getId();
                lookupLoadCommand = "aload " + typeInformation.getId();
            } else if (dataType == DataType.INT) {
                storeCommand = "istore";
                lookupStoreCommand = storeCommand + ' ' + typeInformation
                    .getId();
                lookupLoadCommand = "iload " + typeInformation.getId();
            } else if (dataType == DataType.FLOAT) {
                storeCommand = "fstore";
                lookupStoreCommand = storeCommand + ' ' + typeInformation
                    .getId();
                lookupLoadCommand = "fload " + typeInformation.getId();
            }
            if (exprContext != null) {
                instructions = visit(exprContext) + '\n';
                instructions += storeCommand + ' ' +
                     getVariableIndexByVariableIdToken(varIdToken);
                if (dataTypeStack.peek().getDataType() != dataType) {
                    throw new WrongDataTypeException(varIdToken);
                }
                dataTypeStack.pop();
                return instructions;
            }
            return instructions;
```

```
        }
    }

    @Override
    public String visitAssignment(AssignmentContext ctx) { ... }

    @Override
    public String visitFunctionCall(FunctionCallContext ctx) { ... }

    @Override
    public String visitBranch(BranchContext ctx) { ... }
...
    @Override
    public String visitLoop(LoopContext ctx) { ... }

    @Override
    public String visitIncludedFunctionCall(IncludedFunctionCallContext
        ctx) { ... }

    @Override
    public String visitBuiltinFunctionCall(BuiltinFunctionCallContext
        ctx) { ... }

    @Override
    public String visitInlineAssembly(InlineAssemblyContext ctx) { ... }
...
    @Override
    public String visitFunctionDefinition(FunctionDefinitionContext
        ctx) { ... }

    @Override
    public String visitStructDeclaration(StructDeclarationContext
        ctx) { ... }

    @Override
    public String visitStructArrayInitialization(
        StructArrayInitializationContext ctx) { ... }

    @Override
    public String visitAssignments(AssignmentsContext ctx) { ... }

    @Override
    public String visitUnaryMinusExpression(UnaryMinusExpressionContext
        ctx) { ... }

    @Override
    public String visitDivisionMultiplicationModuloExpression(
        DivisionMultiplicationModuloExpressionContext ctx) { ... }
```

```java
    @Override
    public String visitSubtractionAdditionExpression(
        SubtractionAdditionExpressionContext ctx) {
        String instructions = visitChildren(ctx);
        final TypeInformation typeInfo = dataTypeStack.peek();
        final DataType leftOperandType = dataTypeStack.pop().getDataType();
        final DataType rightOperandType = dataTypeStack.pop()
            .getDataType();
        // Only integer and floating point numbers are allowed
        if (leftOperandType != DataType.INT && leftOperandType != DataType
            .FLOAT || leftOperandType != rightOperandType) {
            throw new WrongDataTypeException(ctx.start);
        }
        final String arithmeticOperator = ctx.op.getText();
        switch (arithmeticOperator) {
            case "-":
                switch (typeInfo.getDataType()) {
                    case INT:
                        instructions += "\nisub";
                        break;
                    case FLOAT:
                        instructions += "\nfsub";
                        break;
                    default:
                        throw new WrongDataTypeException(ctx.start);
                }
                break;
            case "+":
                switch (typeInfo.getDataType()) {
                    case INT:
                        instructions += "\niadd";
                        break;
                    case FLOAT:
                        instructions += "\nfadd";
                        break;
                    default:
                        throw new WrongDataTypeException(ctx.start);
                    }
                    break;
            default:
                throw new IllegalArgumentException("Unknown arithmetic
                    operator: " + arithmeticOperator);
        }
        dataTypeStack.push(typeInfo);
        return instructions;
    }
```

```java
    @Override
    public String visitShiftExpression(ShiftExpressionContext ctx) { ... }

    @Override
    public String visitRelationalExpression(RelationalExpressionContext
        ctx) { ... }

    @Override
    public String visitConjunctionExpression(ConjunctionExpressionContext
        ctx) { ... }

    @Override
    public String visitDisjunctionExpression(DisjunctionExpressionContext
        ctx) { ... }

    @Override
    public String visitContravalenceExpression(
        ContravalenceExpressionContext ctx) { ... }

    @Override
    public String visitArrayExpression(ArrayExpressionContext ctx) { ... }

    @Override
    public String visitStructExpression(StructExpressionContext
        ctx) { ... }

    @Override
    public String visitStructArrayExpression(StructArrayExpressionContext
        ctx) { ... }

    @Override
    public String visitBoolExpression(BoolExpressionContext ctx) { ... }

    @Override
    public String visitIntegerExpression(IntegerExpressionContext
        ctx) {
        dataTypeStack.push(new TypeInformation(DataType.INT));
        return "ldc " + ctx.number.getText();
    }

    @Override
    public String visitFloatingPointExpression(
        FloatingPointExpressionContext ctx) { ... }


    @Override
    public String visitStringExpression(StringExpressionContext
        ctx) { ... }
```

```
    @Override
    public String visitVariableExpression(VariableExpressionContext
        ctx) { ... }
...
    @Override
    protected String aggregateResult(String aggregate,
        String nextResult) {
        if (aggregate == null) {
            return nextResult;
        }
        if (nextResult == null) {
            return aggregate;
        } else {
        return aggregate + '\n' + nextResult;
    }
}
```

Code 3: Implementation of `EVisitor.java`

Accordingly, a semantic analysis can be realized in a visitor. Thus, new operations can be easily added by defining new visitors and related operations can be centrally managed in the visitor. The elements of the tree are visited which e.g. extends the symbol table information of variables. Expressions can also be checked to see if they are well typed. A visitor is used who also takes over the synthesis at the same time which will be explained in the next chapter.

To demonstrate how the tree is traversed, the example shall serve in the form of the figure 8 on page 27. The tree is traversed post-order due to the later code generation. Thus, the children are treated first before the parents are processed. Finally, the terminals are determined with `getText`. In the example, $3, 2, 4, *, +$ would be determined. As already mentioned, there is a corresponding method for each desired operation. Besides the methods of the base class for `program` and `statement`, `visitPrint` is called with the corresponding context. Then, `visit` is called with an expression as argument (here the addition) for producing the respective code. Now, `visitChildren` is called in `visitSubtractionAdditionExpression` where the context is passed. The method `visitChildren` visits the children of a node and returns a user-defined result of the specific operation. In contrast to this, the method `visit` visits a parse tree and return a user-defined result of the operation. On the one hand, `visitIntegerExpression` is called where the first number is determined. On the other hand, the right child is a multiplication where `visitDivisionMultiplicationModuloExpression` (analogous to addition) is called. Then, the further numbers are determined. Afterwards, the parents are determined, namely `*` and `+`. The corresponding aggregation of the string is explained in chapter 2.4 on page 47. Data types have also been implemented in the compiler. So only the integers and floating point numbers work in the arithmetic operations which is queried via conditions. Otherwise an exception is thrown. In order to manage the data types, a stack is used as data structure which is visible in code 4:

```java
public class DataTypeStack {
    /** A linked list of types */
    private final Deque<TypeInformation> typesStack;

    /** Maximum stack size for variables and so on */
    private int maxStackSize;

    DataTypeStack() {
        typesStack = new LinkedList<>();
        maxStackSize = 0;
    }

    public final void push(TypeInformation type) {
        if (type.getDataType() != DataType.VOID) {
            typesStack.push(type);
        }
        if (typesStack.size() > maxStackSize) {
            ++maxStackSize;
        }
    }

    public final TypeInformation pop() {
        return typesStack.pop();
    }

    public final TypeInformation peek() {
        return typesStack.peek();
    }

    public final int getMaxStackSize() {
        return maxStackSize + 1;
    }
}
```

Code 4: Implementation of `DataTypeStack.java`

The stack manages overall type information whereby this class is visible in code 5:

```java
public class TypeInformation {
    /** Id (the position in the symbol table regarding variables) */
    private int id;

    /** Type of the construct (function, variable, ...) */
    private final DataType dataType;
```

```
    /** Address of a type (structure) regarding references */
    private int address;
...
    public final String getJvmType() {
        if (dataType != DataType.OBJREF) {
            return dataType.getJvmType();
        } else {
            return Integer.toString(address);
        }
    }

    public final boolean isArray() {
        switch (dataType) {
            case IARRAY:
            case FARRAY:
            case SARRAY:
                return true;
            default:
                return false;
        }
    }
}
```

Code 5: Implementation of `TypeInformation.java`

This class stores a type and also an address if, for example, a reference to a structure exists. Accordingly, a variable can later be assigned to a structure. A distinction is made between objects and primitive data types that are visible in code 6:

```
public enum DataType {
    /** A specific type regarding the JVM and E. */
    BOOL("bool", "Z"),
    INT("int", "I"),
    IARRAY("int[]", "[I"),
    FLOAT("float", "F"),
    FARRAY("float[]", "[F"),
    STRING("String", "Ljava/lang/String;"),
    SARRAY("String[]", "[Ljava/lang/String;"),
    VOID("void", "V"),
    OBJREF("", "");

    /** Type in E */
    private final String type;

    /** Type in the JVM */
    private final String jvmType;
...
```

```
    public static DataType getType(final String type) {
        switch (type) {
            case "bool":
                return BOOL;
            case "int":
                return INT;
            case "int[]":
                return IARRAY;
...
            default:
                return OBJREF;
        }
    }
...
}
```

Code 6: Implementation of `DataType.java`

For example, if an integer is called, the corresponding data type `INT` is pushed onto the stack. With an addition, this procedure is repeated. The addition can then pop these two data types from the stack and make decisions according to the code generation. The data type for the result is then pushed onto the stack again so that `print` uses the correct data type for the output. The individual abbreviations stand for the data type in binary code and are explained in the next chapter.

As already mentioned, the symbol table can also be extended in the context of variables. So in case of a variable declaration like `int a` the method `visitVariableDeclaration` is called and the position as well as the type is saved to the identifier. If a reference exists, the corresponding address for the object is also saved. If the variable already exists for an assignment, the position is determined and the corresponding value is saved. Otherwise, the procedure is analogous to the declaration. Later, the value of the variable can be retrieved if `visitVariableExpression` is called during traversing. Such expressions can also occur in branches or loops with blocks where the evaluation then takes place in `visitBranch` or `visitLoop`. Since labels must be unique, a counter `branchCounter` respectively `loopCounter` is incremented accordingly.

The same principle is not applied to functions because a function may be called if the definition is further down in the program. For this reason, functions have their own visitor which first collects the signatures (name and parameter types of the functions) and pass them to the visitor from code 3 on page 28. This also ensures total visibility. As already mentioned, a function can also have a body of any length. So that local variables can also be used, there are different local and one global scope. For example, the corresponding realization is visible in the method `visitFunctionPrototypes` where a new reference to an own symbol table is made. The visitor for functions is shown in code 7:

```java
public class FunctionDefinitionVisitor {
    public static FunctionPrototypeList findFunctionPrototypes(
        final ParseTree tree) {
        // Remember all prototypes of defined functions
        final FunctionPrototypeList definedFunctionPrototypes =
            new FunctionPrototypeList();
        // Notice the built-in functions
        setBuiltInFunctionDefinitions(definedFunctionPrototypes);
        // An anonymous class
        new EBaseVisitor<Void>() {
            @Override
            public Void visitFunctionDefinition(FunctionDefinitionContext
                ctx) {
                // Get return type
                final DataType returnType = DataType.getType(ctx.type
                    .getText());
                // Get function identifier
                final String functionId = ctx.funcId.getText();
                final int paramNumber = ctx.formalParams.decls.size();
                final TypeInformation[] paramTypes =
                    new TypeInformation[paramNumber];
                // Get parameters
                for (int i = 0; i < paramNumber; ++i) {
                    final DataType paramType =
                        DataType.getType(ctx.formalParams.decls.get(i)
                        .type.getText());
                    if (paramType == DataType.VOID) {
                        throw new WrongDataTypeException(ctx.start);
                    }
                    paramTypes[i] = new TypeInformation(paramType);
                }
                // Has a function regarding the signature already been
                // defined?
                if (definedFunctionPrototypes.contains(functionId,
                    paramTypes)) {
                    throw new AlreadyDefinedFunctionException(ctx.funcId);
                }
                definedFunctionPrototypes.add(new
                    TypeInformation(returnType), functionId, paramTypes);
                return null;
            }
        }.visit(tree);
        return definedFunctionPrototypes;
    }

```

```java
    private static void setBuiltInFunctionDefinitions(
        FunctionPrototypeList definedFunctions) {
        definedFunctions.add(new TypeInformation(DataType.VOID), "print",
            new TypeInformation[] {new TypeInformation(DataType.INT)});
        ...
        definedFunctions.add(new TypeInformation(DataType.INT), "length",
            new TypeInformation[] {new TypeInformation(DataType.SARRAY)});
    }
}
```

Code 7: Implementation of `FunctionDefinitionVisitor.java`

Among other things, the built-in functions that a user can call later without importing are also defined here. In contrast to the main visitor, the implementation of `aggregateResult` is omitted. Thus states are omitted whereby the method for collecting the respective functions is pure which contributes to the increase of performance. This is stored in an additional list (see code 8):

```java
public class FunctionPrototypeList {
    /** Contains all function prototypes from the respective program. */
    private final List<FunctionPrototype> functionPrototypes;
...
    public final boolean contains(final String functionId, final
        TypeInformation[] params) {
        for (FunctionPrototype prototype : functionPrototypes) {
            // Get parameters of the current function in the list
            final TypeInformation[] functionParameters = prototype
                .getParams();
            // The function may only exist if the number of parameters and
            // the function name match
            if (functionParameters.length == params.length &&
                functionId.equals(prototype.getFunctionId())) {
                if (functionParameters.length == 0 && params.length == 0) {
                    // There are no parameters available =>
                    // The function signatures are the same
                    return true;
                }
                boolean match = true;
                // Look more closely at the individual data types of
                // the parameters
                for (int i = 0; i < functionParameters.length; ++i) {
                    if (functionParameters[i].getDataType() != params[i]
                        .getDataType()) {
                        // Data types do not match =>
                        // function signatures cannot be the same
                        match = false;
                        break;
                    }
```

```
            }
            if (match) {
                return true;
            }
        }
    }
    return false;
}

public final void add(final TypeInformation typeInfo, final String
    functionId, final TypeInformation[] params) {
    functionPrototypes.add(new FunctionPrototype(typeInfo, functionId,
        params));
}
...
public final FunctionPrototype get(final String functionId, final int
    parameterNumber) {
    for (FunctionPrototype prototype : functionPrototypes) {
        if (prototype.getFunctionId().equals(functionId) &&
            prototype.getParamNumber() == parameterNumber) {
            return prototype;
        }
    }
    return null;
}
}
```

Code 8: Implementation of `FunctionPrototypeList.java`

The same principle also applies to structures that can be defined individually. There is also a separate visitor which first determines the names of the structures and maps them to positions. The following code 9 shows this procedure:

```
public class CustomTypeVisitor {
    public static LinkedHashMap<String, CustomType> findTypes(final
        ParseTree tree) {
        // Saves the names of the structures and their positions (from top
        // to bottom).
        final Map<String, Integer> structIds = new LinkedHashMap<>();

        // Holds the structures with variables and types as well as
        // references.
        final LinkedHashMap<String, CustomType> structs =
            new LinkedHashMap<>();

        // An anonymous class (first round)
        new EBaseVisitor<Void>() {
            /** Position of the custom data type */
```

```
        int typeId = 0;

        @Override
        public Void visitStructDeclaration(StructDeclarationContext
            ctx) {
            // The name of the structure must not appear twice
            if (structIds.get(ctx.structId.getText()) != null) {
                throw new AlreadyDeclaredStructException(ctx.structId);
            } else {
                structIds.put(ctx.structId.getText(), typeId++);
            }
            return null;
        }
    }.visit(tree);

    // An anonymous class (second round)
    new EBaseVisitor<Void>() {
        /** Position of the custom data type */
        int typeId = 0;

        @Override
        public Void visitStructDeclaration(StructDeclarationContext
            ctx) {
            // Store the variables types
            final List<TypeInformation> varTypes = new ArrayList<>();
            // Maps the variable identifiers to its positions
            final Map<String, Integer> varIds = new HashMap<>();
            // Iterate over the variables in the structure
            for (int i = 0; i < ctx.decls.size(); ++i) {
                final String varType = ctx.decls.get(i).type.getText();
                // Get Data type object for the desired data type
                final DataType type = DataType.getType(varType);
                TypeInformation newType;
                if (type == DataType.OBJREF) {
                    final Integer structId = structIds.get(varType);
                    if (structId == null) {
                        throw new UndeclaredStructException(ctx.decls
                            .get(i).type.start);
                    } else {
                        // There is a reference => Map the variable type
                        // to the struct position
                        newType = new TypeInformation(type, structId);
                    }
                } else {
                    newType = new TypeInformation(type);
                }
                varTypes.add(newType);
                final String varId = ctx.decls.get(i).varId.getText();
```

```
                    if (!varIds.containsKey(varId)) {
                        varIds.put(varId, i);
                    } else {
                        throw new AlreadyDeclaredVariableException(ctx.decls
                            .get(i).varId);
                    }
                }
                structs.put(ctx.structId.getText(), new CustomType(typeId++,
                    varTypes, varIds));
                return null;
            }
        }.visit(tree);
        return structs;
    }
}
```

Code 9: Implementation of `CustomTypeVisitor.java`

Then, references can also be determined within `visitStructDeclaration`. It is possible that types are defined in types which is why a double traversing occurs. Afterwards, the names of the types as well as the attributes and references are stored in a map. In the last traversing before the main traversing the static variables are determined which is shown in the following code 10:

```
public class StaticVariableVisitor {
    public static Map<String, TypeInformation> findStaticVariables(
        final ParseTree tree, final LinkedHashMap<String, CustomType>
        structs) {
        // Saves the names and types (also references) of the static
        // variables.
        final Map<String, TypeInformation> staticVars = new HashMap<>();

        // An anonymous class
        new EBaseVisitor<Void>() {
            int typeId = 0;

            @Override
            public Void visitProgram(ProgramContext ctx) {
                for (ParseTree child : ctx.children) {
                    if (child instanceof StatementCommandContext) {
                        visit(child);
                    }
                }
                return null;
            };

            @Override
            public Void visitVariableDeclaration(VariableDeclarationContext
```

```
                ctx) {
                final String varType = ctx.type.getText();
                final DataType dataType = DataType.getType(ctx.type
                    .getText());
                final String varId = ctx.varId.getText();
                if (staticVars.containsKey(varId)) {
                    // A variable must not be declared twice
                    throw new AlreadyDeclaredVariableException(ctx.varId);
                } else if (dataType == DataType.OBJREF) {
                    final CustomType struct = structs.get(varType);
                    if (struct == null) {
                        throw new UndeclaredStructException(ctx.start);
                    } else {
                        // There is a reference => Map the variable type to
                        // the struct position
                        staticVars.put(varId, new TypeInformation(typeId,
                            dataType, struct.getId()));
                    }
                } else {
                    // Primitive data type
                    staticVars.put(varId, new TypeInformation(typeId,
                        dataType));
                }
                ++typeId;
                return null;
            }
        }.visit(tree);
        return staticVars;
    }
...
}
```

Code 10: Implementation of `StaticVariableVisitor.java`

If there is a reference to a structure, the respective address including the type is inserted, otherwise only the data type. Later, a decision is made between a global or local scope (for functions) about the variable `isGlobal` which specific code is generated. So if, for example, the construct *struct A { int a; } A a = new A(0);* is present, then `Point 0 {int, int}` and `p : 0` would apply with regard to a debug mode, i.e. `p` refers to `Point`. In the case of further structures, the number would be increased. For example, the numbers are then used later during code generation as indexes for variables.

Due to the increased complexity there is an exception handling. In concrete terms, type and declaration analysis are implemented here in the form of exceptions for which a base class 11 is used:

```
public class CompilerException extends RuntimeException {
```

```java

    protected int line;

    protected int column;

    public CompilerException(Token token) {
        line = token.getLine();
        column = token.getCharPositionInLine();
    }
}
```

Code 11: Implementation of CompilerException.java

If a construct of language E is used incorrectly, the row, column and reason are specified within the traversal. For example, there are the following exceptions:

- If a variable was not declared before use
- If a variable has been defined twice with respect to the identifier
- If a function was not defined before use
- If a function has been defined twice
- If a structure was not defined before use
- If a structure has been defined twice
- If a wrong data type was used inside an illegal operation
- If an unknown built-in function was used
- If an unknown imported module was used

Note: The exceptions refer to the specific scope.

In the case of print(a); the exception 1:6 <Undeclared variable: <a>; is thrown into visitVariableDeclaration (see chapter 3 on page 55) if the identifier is not in the hash table. The same applies to variables that have already been defined. With regard to functions the signature decides whether a function is duplicated or not. As already mentioned, this check is already carried out in the FunctionDefinitionVisitor (see code 7 on page 40). If, for example, a float get_val() { return 1.0; } float get_val() { return 2.0; } was programmed, the message 2:4 Already defined function: <get_val>; would appear. The respective body and a correct function call are still checked in the main visitor (see code 3 on page 28). In addition, all operations are checked to see whether the data types are compatible with them. For example, if an attempt is made to call the code String a = 5.0 + 3;, a WrongDataTypeException is thrown. This procedure can also be transferred to the other application scenarios.

The corresponding derived classes can be found in the appendix on page 75.

2.4 Code generation

During code generation, components of the grammar are transformed into code by required instructions, depending on the respective architecture. Here, *Jasmin*[12] is used as assembler for the JVM where bytecode is generated from assembler-like instructions. This can then be interpreted or executed in the JVM. The instructions are described in ASCII format. The basic structure is as follows (see Code 12):

```
.class public E
.super java/lang/Object
...
.method public static main([Ljava/lang/String;)V
   .limit stack 100
   .limit locals 100
...
   return
.end method;
```

Code 12: Basic structure of `Jasmin`

Jasmin statements are comments, directives and instructions. A comment begins with a semi-colon and the assembler ignores everything after that to the end of line. Directives like `.super` begin with a period and are executed by the assembler rather than the JVM. An instruction can consists of a label, an operator (also called a mnemonic) and operands. For example, these can be arithmetic or logical. The `.super` construct indicates an inheritance in which the corresponding packages must also be specified. The `[L` command specifies an argument with an array of a certain type (also called the descriptor). The symbol `V` stands for no return (also called Void). The command `.limit` can be used to specify upper limits for e.g. the stack or local variables. In this case, the `return` construct determines the end of the main method. Before this are all instructions, such as outputs or variable declarations which are determined during traversing by the AST.

Altogether, the individual programs written in *E* are processed in a queue and compiled individually as shown in the following code 13:

```
public class Main {
   /** Temporary directory for the compiled files */
   private static Path tempDir;

   /** Parent directory (path to file for legacy) */
```

[12] Jasmin, `https://cs.au.dk/~mis/d0vs/jvmspec/ref-Java.html`

```java
    private static File parentDir;

    /** The namespace of the program you want to compile currently */
    private static String namespace;

    /** Holds the programs (including path) to compile. */
    public static Queue<String> programs;

    /** Holds the compiled programs (including path). */
    public static Queue<String> compiledPrograms;

    public static void main(String[] args) throws Exception {
        // Path to the main program
        final String fileName = "e/test/main";
        // Create a temporary directory for programs compiled later
        tempDir = createTempDir(fileName);
        parentDir = tempDir.toFile();
        programs = new LinkedList<>();
        // Add the program you want to compile
        programs.add(fileName);
        compiledPrograms = new LinkedList<>();
        // Process the programs
        while (!programs.isEmpty()) {
            final String program = programs.poll();
            compiledPrograms.add(program);
            // Set the namespace for later references
            namespace = program.replace('/', '_');
            // The current file to compile
            final File currentFile = new File(parentDir.getPath(),
                namespace + ".e");
            final CharStream srcCode = CharStreams.fromFileName(program +
                ".e");
...
            // Separate the main program from the structures
            final String[] compiledCode = compile(srcCode).split("\\*");
            if (!compiledCode[0].isEmpty()) {
                final ClassFile classFile = new ClassFile();
                classFile.readJasmin(new StringReader(compiledCode[0]), "",
                    false);
                final Path outputPath = tempDir.resolve(currentFile
                    .getAbsolutePath().substring(0, currentFile
                    .getAbsolutePath().length() - 2) + ".class");
                classFile.write(Files.newOutputStream(outputPath));
                final ClassFile[] extraFiles = new ClassFile[compiledCode
                    .length - 1];
                // Create class files for the declared structs
                for (int i = 0; i < compiledCode.length - 1; ++i) {
                    final ClassFile file = new ClassFile();
```

```java
                    extraFiles[i] = file;
                    file.readJasmin(new StringReader(compiledCode[i + 1]),
                        "", false);
                    final Path newPath = tempDir.resolve(file.getClassName()
                        + ".class");
                    file.write(Files.newOutputStream(newPath));
                }
            }
        }
        // Run the compiled program
        final String result = runClass(tempDir, fileName.replace("/", "_"
            ));
        // Delete the created temporary directory
        deleteTempDir();
    }

    public static String compile(final CharStream sourceCode) {
        // Control characters are ignored
        ELexer lexer = new ELexer(sourceCode);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        EParser parser = new EParser(tokens);
        // Start rule
        ParseTree tree = parser.program();
        // Collect all function definitions (without body)
        FunctionPrototypeList definedFunctionPrototypes =
            FunctionDefinitionVisitor.findFunctionPrototypes(tree);
        // Collect all structs with declared variables and types
        // (including references)
        LinkedHashMap<String, CustomType> declaredStructs =
             CustomTypeVisitor.findTypes(tree);
        // Collect all static variables in the main program
        // (including references to structs)
        Map<String, TypeInformation> declaredStaticVars =
            StaticVariableVisitor.findStaticVariables(tree,
            declaredStructs);
        // Create an assembler program for Jasmin based on instructions
        return new EVisitor(definedFunctionPrototypes, declaredStructs,
            declaredStaticVars, namespace, parentDir).visit(tree);
    }
...
    public static String runClass(final Path dir, final String className)
        throws Exception {
        final Process process =
        Runtime.getRuntime().exec(new String[] {"java", "-cp", dir
            .toString(), className});
        try (InputStream input = process.getInputStream()) {
            Scanner scanner = new Scanner(input);
            if (scanner.useDelimiter("\\A").hasNext()) {
```

```
                String result = scanner.useDelimiter("\\A").next();
                scanner.close();
                return result;
            }
            scanner.close();
            return null;
        }
    }
}
```

Code 13: Implementation of `Main.java`

In general, the main procedure is as follows:

1. The main program is added to the queue.
2. As long as the queue is not empty, do the following:
   a) Get the current file to compile from the queue.
   b) Convert the file to a namespace (for class names, references, etc.).
   c) Compile the file in several steps (see figure 1 on page 8).
   d) If there are structures, they are stored in separate class files.

Since there can also be imports (see `visitIncludes` in code 3 on page 28), these are also processed in the queue. Finally, the compiled code can be executed.

After the described context analysis (see chapter 2.3 on page 27) of the example from the figure 8 on page 27 the following code 14 would be generated:

```
.class public e_test_main
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
    .limit stack 4
    .limit locals 1

    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc 3
    ldc 2
    ldc 4
    imul
    iadd
    invokevirtual java/io/PrintStream/print(I)V

    return
.end method
```

Code 14: Generated code of *E*

To produce this code, a `StringBuilder` is used (see `visitProgram` in code 3 on page 28). This procedure saves the use of various objects and thus increases performance. The command `getstatic` places the object behind `System/Out` on the stack. Furthermore, the constants 3, 2 and 4 are placed one after the other on the stack. The command `imul` multiplies the two top elements 4 and 2 and puts the result back on the stack. This is then added to the 3 by `iadd`. This procedure is visualized as follows (see figure 9):

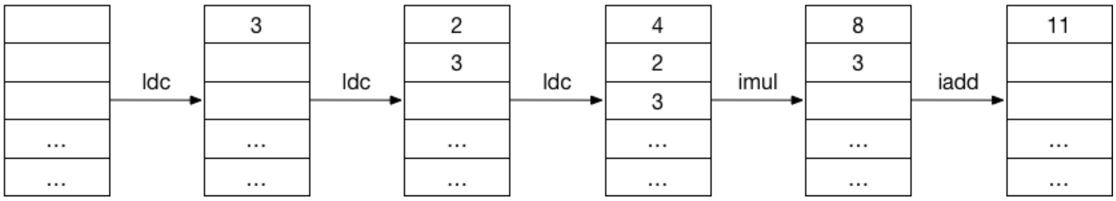

Figure 9: Relation between the operations and stack

Note: If functions or structs are used, the respective parameters respectively attributes are also stored on the stack.

With the command `invokevirtual` the method `print` is called with an integer as argument which outputs the result. If the symbol table is to be extended with regard to variables, there are the commands `istore` and `iload` to store a value in a variable from the stack or to get a value from this position (see code 3 on page 28).

In addition, there are many other constructs such as branches, loops, different operators and so on. The following tables 3, 4 and 5 starting on page 52 show the transformation of various such constructs or examples:

Table 3: Code transformations

Component	Instructions	Transformation

`void mul(int a`
`  ,int b) {`
`  print(a*b);`
`}`
`mul(5, 3);`
	`getstatic`
	`iload`
	`imul`
	`invokevirtual`
	`ldc`
	`invokestatic`
		`.method public static mul(II)V`
		`.limit locals 2`
		`.limit stack 3`
		`getstatic java/lang/System/out Ljava/io/PrintStream;`
		`iload 0`
		`iload 1`
		`imul`
		`invokevirtual java/io/PrintStream/print(I)V`
		`return`
		`.end method`
		`ldc 5`
		`ldc 3`
		`invokestatic e_test_main/mul(II)V`

`int a = 1;`
`if (a) {`
`  print(5);`
`} else {`
`  print(3);`
`}`
	`ldc`
	`putstatic`
	`getstatic`
	`ifne`
	`goto`
		`ldc 1`
		`putstatic main/v0 I`
		`getstatic main/v0 I`
		`ifne onTrue1`
		`getstatic java/lang/System/out Ljava/io/PrintStream;`
		`ldc 3`
		`goto endIf1`
		`onTrue1:`
		`getstatic java/lang/System/out Ljava/io/PrintStream;`
		`ldc 5`
		`endIf1:`

`print(0 && 1);`	`ldc`
	`ifeq`
	`goto`
	`invokevirtual`
		`getstatic java/lang/System/out Ljava/io/PrintStream;`
		`ldc 0`
		`; jump if a is zero`
		`ifeq onAndFalse1`
		`ldc 0`
		`ifeq onAndFalse1`
		`ldc 1`
		`goto endAnd1`
		`onAndFalse1:`
		`ldc 0`
		`endAnd1:`
		`invokevirtual java/io/PrintStream/print(I)V`

With regard to the operations `and` and `or`, the direct use of the instructions `iand` and `ior` is deliberately waived in order to ensure lazy evaluation. With the disjunction the logic is only reversed in comparison to the conjunction. So that the labels for the jumps are unique, they are uniquely identified by a number.

Table 4: Code transformations 2

Component	Instructions	Transformation

struct Point {
  int x;
  int y;
}
Point p = new
  Point(1, 2);
p.x = 5;
	aload
	invokespecial
	return
	new
	dup
	putstatic
	getstatic
	putfield
	ldc
		.class struct_main0
		.super java/lang/Object
		.field public a0 I
		.field public a1 I
		.method public <init>()V
		aload_0
		invokespecial java/lang/Object/<init>()V
		return
		.end method
		
		.class public main
		.super java/lang/Object
		.field public static v0 Lstruct_main0;
		.method public static main([Ljava/lang/String;)V
		.limit stack 4
		.limit locals 1
		new struct_main0
		dup
		invokespecial struct_main0/<init>()V
		putstatic main/v0 Lstruct_main0;
		getstatic main/v0 Lstruct_main0;
		ldc 1
		putfield struct_main0/a0 I
		getstatic main/v0 Lstruct_main0;
		ldc 2
		putfield struct_main0/a1 I
		getstatic main/v0 Lstruct_main0;
		putstatic main/v0 Lstruct_main0;
		getstatic main/v0 Lstruct_main0;
		ldc 5
		putfield struct_main0/a0 I
		return
		.end method

A separate class is created for each structure. Attributes are assigned via `.field` and there is a default constructor that is initially loaded. In the main class there is a reference to this structure which is identified by `L`. A structure is created with `new` or the standard constructor is called with `invokespecial`. With `.putfield` values are assigned to the attributes. The suffixes of each variable name result from the position in the symbol table.

Table 5: Code transformations 3

<table>
<tr><th>Component</th><th>Instructions</th><th>Transformation</th></tr>
<tr><td>int[] a =
new int[3];
a[0] = 5;
print(a[0]);</td><td>ldc
newarray
putstatic
getstatic
iastore
iaload
invokevirtual</td><td>ldc 3
newarray int
putstatic main/v0 [I
getstatic main/v0 [I
ldc 0
ldc 5
iastore
getstatic java/lang/System/out Ljava/io/PrintStream;
getstatic main/v0 [I
ldc 0
iaload
invokevirtual java/io/PrintStream/print(I)V</td></tr>
<tr><td>int i = 0;
while (i < 3)
{
  i = i + 1;
}</td><td>ldc
putstatic
getstatic
if_icmplt
goto
ifeq
iadd</td><td>ldc 0
putstatic main/v0 I
beginLoop1:
getstatic main/v0 I
ldc 3
if_icmplt onCmpTrue1
ldc 0
goto endCmp1
onCmpTrue1:
ldc 1
endCmp1:
ifeq endLoop1
getstatic main/v0 I
ldc 1
iadd
putstatic main/v0 I
goto beginLoop1
endLoop1:</td></tr>
<tr><td>int i = 2 < 2;</td><td>ldc
if_icmplt
goto
putstatic</td><td>ldc 2
ldc 2
if_icmplt onCmpTrue1
ldc 0
goto endCmp1
onCmpTrue1:
ldc 1
endCmp1:
putstatic main/v0 I</td></tr>
</table>

Overall, these results are aggregated. In code 3 on page 28, the method `visitChildren` can be recognized during traversing in this context. This method is executed on several nodes which is why the method `aggregateResult` is called internally there to aggregate the specific instructions. The relevant operands are also placed on the stack.

A list of more relevant commands with descriptions can be found in the appendix on page 78. Finally, a more complex example can be found in the appendix on page 80.

3 Tests

This chapter presents the tests for the compiler. The development process was *Test Driven Development* (TDD). This process is shown in the following figure 10:

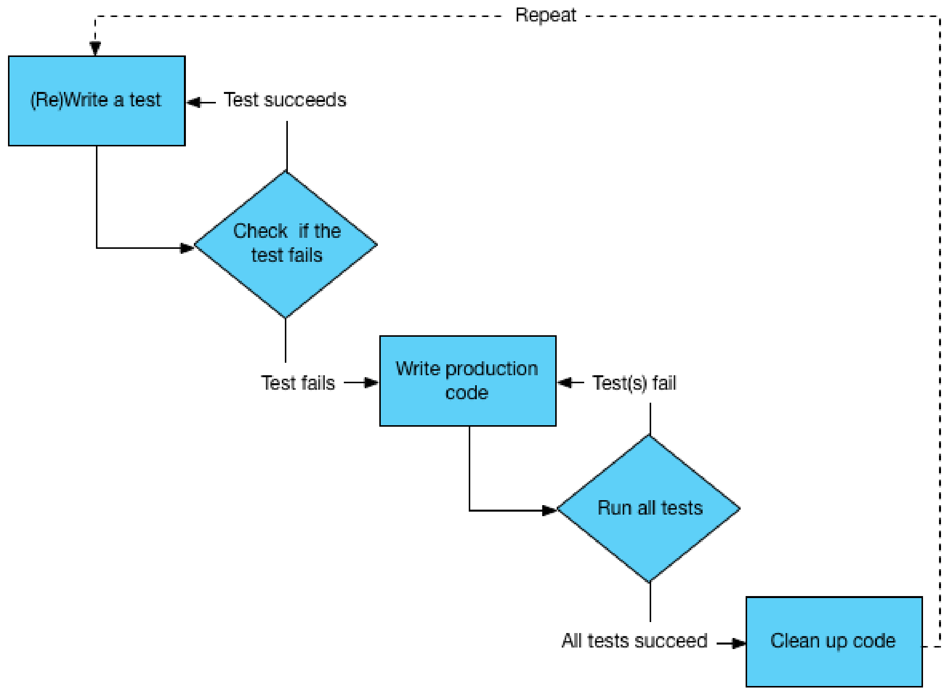

Figure 10: Test Driven Development

Test cases were created as early as possible and derived from desired use cases of the compiler. Furthermore, they specified the preconditions and the behavior of the respective product functions. If the parser did not throw any more mistakes, then the grammar was correct. However, a `NoSuchElementException` indicated that the instruction code was still produced incorrectly. Therefore, the test drivers were developed first whereby methods were partially defined at the beginning.

The unit and integration tests serve to exclude errors of the calculated results. TestNG[13] was used as test framework. It is inspired by JUnit[14] but is easier to use and offers more functionalities like annotations or a flexible test configuration. The test suite is structured as follows (see code 15):

```
public class CompilerTest {
    @BeforeClass
    public void createTempDir() throws Exception {
        Main.tempDir = Main.createTempDir("compilerTest");
    }
```

[13] TestNG, `https://testng.org/doc/index.html`
[14] JUnit, `https://junit.org/junit5/`

```java
@AfterClass
public void deleteTempDir() {
    Main.deleteRecursive(Main.tempDir.toFile());
}

@DataProvider
public Object[][] provideCodeExpectedOutput() {
    return new Object[][] {
        {"Print a number", "print(1);", "1"},
        {"Print a number with a new line", "println(1);", "1" +
            System.lineSeparator()},
        {"Addition", "print(1+2);", "3"},
        {"Chained addition", "print(1.0+2.3+50.8);", "54.1"},
        {"Subtraction", "print(5-3);", "2"},
        {"Multiplication", "print(2*5);", "10"},
        {"Division", "print(9/3);", "3"},
        {"Integer division", "print(10/3);", "3"},
        {"Floating point number division", "print(7.0/2.0);", "3.5"},
        {"Modulo", "print(12%5);", "2"},
        {"Division and multiplication", "print(15/5*3);", "9"},
        {"Subtraction and addition", "print(3-2+5);", "6"},
        {"Addition and subtraction", "print(3+2-5);", "0"},
        {"Order of operations", "print(9-1*3);", "6"},
        {"Order of operations 2", "print(3+5*2);", "13"},
        {"Multiple output", "println(1); println(2);", "1" +
            System.lineSeparator() + "2" + System.lineSeparator()},
        {"Variable declaration", "int a; a = 5; print(a);", "5"},
        {"Variable declaration 2", "int _a; _a = 5; print(_a);", "5"},
        {"Variable declaration and constant", "int a; a = 5;
            print(a+3);", "8"},
        {"Variable declaration and calculation", "int a; a = 5;
            int b; b = 3; print(a+b);", "8"},
        loadTestCode("function/simple", "3"),
        loadTestCode("function/local_parameter", "3"),
        loadTestCode("function/scope", "3" + System.lineSeparator() +
            "5"),
        loadTestCode("function/current_formal_parameter", "15"),
        loadTestCode("function/overloading", "1" +
            System.lineSeparator() + "5"),
        loadTestCode("branch/if-else_zero_false", "1"),
        loadTestCode("branch/if-else_one_true", "1"),
        loadTestCode("branch/if-else_other_true", "1"),
        {"Lower than to true", "print(0 < 1);", "1"},
        {"Lower than to false", "print(2 < 2);", "0"},
        {"Lower than to false 2", "print(3 < 2);", "0"},
        {"Lower than/equal to true", "print(0 <= 1);", "1"},
        {"Lower than/equal to true 2", "print(2 <= 2);", "1"},
        {"Lower than/equal to false", "print(3 <= 2);", "0"},
```

```java
        {"Greater than to true", "print(1 > 0);", "1"},
        {"Greater than to false", "print(2 > 2);", "0"},
        {"Greater than to false 2", "print(1 > 2);", "0"},
        {"Greater than/equal to true", "print(1 >= 0);", "1"},
        {"Greater than/equal to true 2", "print(2 >= 2);", "1"},
        {"Greater than/equal to false", "print(0 >= 1);", "0"},
        {"Negation", "print(3 * -5);", "-15"},
        {"Logical conjunction to true", "print(1 && 1);", "1"},
        {"Logical conjunction to false", "print(0 && 1);", "0"},
        {"Logical conjunction to false 2", "print(1 && 0);", "0"},
        {"Logical conjunction to false 3", "print(0 && 0);", "0"},
        {"Logical disjunction to true", "print(1 || 1);", "1"},
        {"Logical disjunction to true 2", "print(0 || 1);", "1"},
        {"Logical disjunction to true 3", "print(1 || 0);", "1"},
        {"Logical disjunction to false", "print(0 || 0);", "0"},
        loadTestCode("operators/lazy_eval_and", "0" +
          System.lineSeparator() + "0"),
        loadTestCode("operators/lazy_eval_or", "1" +
          System.lineSeparator() + "1"),
        {"Logical contravalence to true", "print(0 ^ 1);", "1"},
        {"Logical contravalence to true 2", "print(1 ^ 0);", "1"},
        {"Logical contravalence to false", "print(0 ^ 0);", "0"},
        {"Logical contravalence to false 2", "print(0 ^ 0);", "0"},
        {"Print string literal", "print(\"Hello world\");",
          "Hello world"},
        {"Print string literal 2", "String a = \"Hello world\";
          print(a);", "Hello world"},
        loadTestCode("comments/line_comment", "5"),
        loadTestCode("comments/multiline_comment", "5"),
        loadTestCode("comments/special_comment", "5"),
        loadTestCode("loop/while", "4"),
        {"Casting to integer", "print(toInt(5.3));", "5"},
        {"Casting to float", "print(toFloat(\"3\"));", "3.0"},
        {"Casting to string", "String a = toString(5.0); print(a);",
          "5.0"},
        {"Append characters", "String a = append(\"a\", \"b\");
          print(a);", "ab"},
        loadTestCode("assembly/inline_asm", "5.5"),
        {"Access to an array", "int[] a = new int[3]; a[0] = 5;
          print(a[0]);", "5"},
        {"Print array length", "int[] a = new int[3]; print(length(a)
          );", "3"}
    };
  }

  private static String[] loadTestCode(final String filePath,
    final String expectedResult) throws Exception {
    try (InputStream input = CompilerTest.class.getResourceAsStream(
```

```java
                "/" + filePath + ".e")) {
            if (input == null) {
                throw new IllegalArgumentException("The file " + filePath +
                    ".e does not exist");
            }
            String code = new Scanner(input).useDelimiter("\\A").next();
            return new String[]{filePath, code, expectedResult};
        }
    }

    @Test(dataProvider = "provideCodeExpectedOutput")
    public void testOutputs(final String description, final String
        sourceCode, final String expectedOutput) throws Exception {
        final String currentOutput = compileAndRun(sourceCode);
        Assert.assertEquals(currentOutput, expectedOutput);
    }

    @Test(expectedExceptions = UndeclaredVariableException.class,
        expectedExceptionsMessageRegExp = "1:6 Undeclared variable: <a>;")
    public void testReadingUndeclaredVariable() throws Exception {
        compileAndRun("print(a);");
    }

    @Test(expectedExceptions = UndeclaredVariableException.class,
        expectedExceptionsMessageRegExp = "1:0 Undeclared variable: <a>;")
    public void testWritingUndeclaredVariable() throws Exception {
        compileAndRun("a = 9;");
    }

    @Test(expectedExceptions = AlreadyDefinedVariableException.class,
        expectedExceptionsMessageRegExp = "2:4 Already defined variable:
        <a>;")
    public void testWritingAlreadyDefinedVariable() throws Exception {
        compileAndRun("int a;" + System.lineSeparator() + "int a;");
    }

    @Test(expectedExceptions = UndefinedFunctionException.class,
        expectedExceptionsMessageRegExp = "1:6 Undefined function:
        <foo>;")
    public void testReadingUndefinedFunction() throws Exception {
        compileAndRun("print(foo());");
    }

    @Test(expectedExceptions = AlreadyDefinedFunctionException.class,
        expectedExceptionsMessageRegExp =
    "2:4 Already defined function: <get_val>;")
    public void testWritingAlreadyDefinedFunction() throws Exception {
        compileAndRun("int get_val() { return 1; }" + '\n' +
```

```
            "int get_val()
            { return 2; }");
    }

    @Test(expectedExceptions = AlreadyDeclaredStructException.class,
        expectedExceptionsMessageRegExp = "2:7 Already declared struct:
        <a>;" )
    public void testWritingAlreadyDeclaredStruct() throws Exception {
    compileAndRun("struct a { int a; }" + System.lineSeparator() +
        "struct a { int b; }");
    }

    @Test(expectedExceptions = UndeclaredStructException.class,
        expectedExceptionsMessageRegExp = "1:11 Undeclared struct:
        <Point>;")
    public void testReadingUndeclaredStruct() throws Exception {
        compileAndRun("struct a { Point p; }");
    }

    @Test(expectedExceptions = WrongDataTypeException.class,
        expectedExceptionsMessageRegExp = "1:8 Wrong data type: <3>;")
    public void testUsingWrongDataType() throws Exception {
        compileAndRun("int a = 3 + 5.0;");
    }

    @Test(
    expectedExceptions = UnknownModuleException.class,
        expectedExceptionsMessageRegExp = "1:6 The module could not be
        found: <math>;")
    public void testUsingFunctionOfUnknownModule() throws Exception {
        compileAndRun("print(math.square(5): int);");
    }

    private String compileAndRun(final String sourceCode) throws
        Exception {
        final String compiledCode = Main.compile(CharStreams
            .fromString(sourceCode));
        // System.out.println(sourceCode);
        final ClassFile classFile = new ClassFile();
        classFile.readJasmin(new StringReader(compiledCode), "", false);
        Path outputPath = Main.tempDir.resolve(classFile.getClassName() +
        ".class");
        try ( OutputStream output = Files.newOutputStream(outputPath) ) {
            classFile.write(output);
        }
        return runClass(Main.tempDir, classFile.getClassName());
    }
}
```

Code 15: Implementation of `CompilerTest.java`

Before execution a temporary folder "compilerTest" is created by the method `createTempDir`. This folder is deleted via `deleteTempDir` or `deleteRecursive`. The test is annotated by `@test` where a data provider `provideCodeExpectedOutput` is registered. This holds the individual code snippets or components of the compiler (see code 2 on page 19) which are tested individually and continuously. The first string represents the description, the second the source code and the third the expected result. Since there are also longer constructs, these are loaded separately from files using the `loadTestCode` method. The individual files are listed in the appendix on page 90. The error messages are also tested which is annotated by the expected exceptions. The current result is determined by the `compileAndRun` method. The corresponding compilation is created in the class `Main` (see code 13 on page 47). An excerpt of the tests can be seen in the figure 11 below:

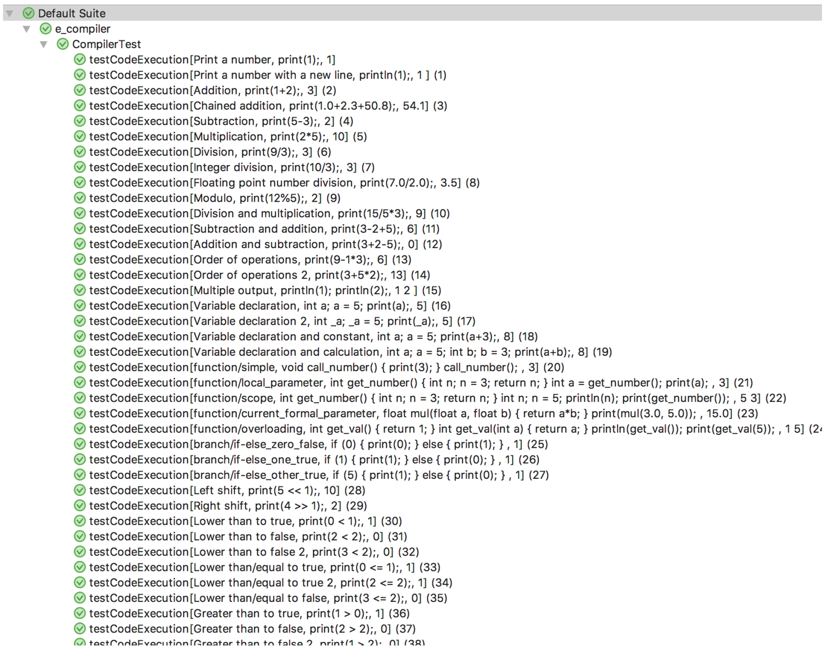

Figure 11: Unit and integrations tests

Accordingly, it can be recognized that all 83 tests regarding arithmetic operators, outputs, functions, variable declarations, loops, branches, structures etc. have been successful and that the compiler works correctly regarding language *E*. To ensure the code quality, Google Java Format[15] was still used which formats the code. Checkstyle[16] was also used. It can find class design problems, method design problems. It also has the ability to check code layout and formatting issues.

[15] Google Java Format, `https://github.com/google/google-java-format`
[16] Checkstyle, `https://maven.apache.org/plugins/maven-checkstyle-plugin/`

4 Summary and Conclusion

In this paper the development of an own language E or the corresponding platform-independent compiler was described. This concerns both analysis and synthesis (see figure 1 on page 8). The focus was on traversing the AST (see chapter 2.3 on page 27) and code generation (see chapter 2.4 on page 47). Accordingly, the source code is translated into byte code after traversing (to exclude type errors, etc.) and can be executed in the JVM. The aim was to be able to describe a lot with as few expressions as possible and to demonstrate different constructs. In addition, conventional language properties should be removed until a Turing machine-like language remains. Various unit and integration tests should ensure the quality of the compiler.

The language E supports different data types (see chapter 2.1 on page 12) like booleans, integers, floating point numbers, strings, arrays and objects as well as different commands (see chapter 2.2 on page 18). This includes functions, structures and different statements like branches, loops, variables and inline assembler. These are separated so that global visibility is also possible. Thus e.g. function definitions or structure declarations can stand below their use. Through the use of while loops, the language is even Turing complete, thus providing universal programmability. The system and a universal Turing machine can thus emulate each other or each program part can be converted into a different language and vice versa. Since the libraries provided are still restrictive, it is possible to program different routines with inline assembly, e.g. in dealing with file systems (see appendix on page 92).

TDD was used as the development process (see figure 10 on page 55). Various tests (see chapter 3 on page 55) check the correctness of the provided constructs. This refers to the direct application as well as the interaction between different components. For example, branches can also be used in loops. Furthermore, a static code analysis was performed to find possible errors such as unreachable code. This procedure was repeated until all tests were successful (see figure 11 on page 60).

Finally, it should be said that optimized code is produced in relation to the realized language constructs (see appendix from page 80) and all known possible programming errors are found by the lexer, parser and context analysis. Also a possible conversion to interpretation would be possible. It would be conceivable, for example, that a generic data type instead of a string would be used as the return value of the visitor methods (see code 3 on page 28). Thus, individual data types can be set and checked. Output then takes place, for example, directly in Java. Also a flexible further development of the language constructs is given by the ease of maintenance.

Bibliography

[BDR06] Bouchez, F. ; Darte, A. ; Rastello, F.: *Register Allocation: What does Chaitin's NP-completeness Proof Really Prove?* `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2006/RR2006-13.pdf`. Version: 2006. – Last accessed on: 2019-02-03

[Chu09] Chu, I.: *BNF and EBNF.* `http://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf`. Version: 2009. – Last accessed on: 2019-02-03

[HM11] Hopcroft, J. E. ; Motwani, J. D. R. und Ullman U. R. und Ullman: *Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit.* Addison-Wesley, 2011

[Vis13] Viswanathan, M.: *Introduction to Theory of Computation.* `https://courses.engr.illinois.edu/cs373/sp2013/Lectures/lec07.pdf`. Version: 2013. – Last accessed on: 2019-02-03

[Zei18] Zeil, S.: *Turing Completeness.* `https://www.cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html`. Version: 2018. – Last accessed on: 2019-02-03

Maven dependencies

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.runekrauss.e_compiler</groupId>
    <artifactId>e_compiler</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>attached</goal>
                        </goals>
                        <phase>package</phase>
                        <configuration>
                            <descriptorRefs>
                                <descriptorRef>
                                    jar-with-dependencies
                                </descriptorRef>
                            </descriptorRefs>
                            <archive>
                                <manifest>
                                    <mainClass>
                                        com.runekrauss.compiler.Main
                                    </mainClass>
                                </manifest>
                            </archive>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.0.2</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>com.runekrauss.compiler.Main</mainClass>
```

```
                            <addDefaultImplementationEntries>
                                true
                            </addDefaultImplementationEntries>
                            <addDefaultSpecificationEntries>
                                true
                            </addDefaultSpecificationEntries>
                            <addClasspath>true</addClasspath>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.5.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>com.coveo</groupId>
                <artifactId>fmt-maven-plugin</artifactId>
                <version>2.1.0</version>
                <configuration>
                    <sourceDirectory>
                        src/main/java/com/runekrauss/compiler
                    </sourceDirectory>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>format</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <configLocation>google_checks.xml</configLocation>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
```

```
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>6.14.3</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.antlr</groupId>
            <artifactId>antlr4</artifactId>
            <version>4.7.1</version>
        </dependency>
        <dependency>
            <groupId>net.sourceforge</groupId>
            <artifactId>jasmin</artifactId>
            <version>2.4.0</version>
        </dependency>
    </dependencies>
</project>
```

Code 16: Implementation of `pom.xml`

ASCII table

Table 6: Control characters and digits in ASCII

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
0	0x00	000	NUL	32	0x20	040	SP
1	0x01	001	SOH	33	0x21	041	!
2	0x02	002	STX	34	0x22	042	"’
3	0x03	003	ETX	35	0x23	043	#
4	0x04	004	EOT	36	0x24	044	$
5	0x05	005	ENQ	37	0x25	045	%
6	0x06	006	ACK	38	0x26	046	&
7	0x07	007	BEL	39	0x27	047	’
8	0x08	010	BS	40	0x28	050	(
9	0x09	011	TAB	41	0x29	051	)
10	0x0A	012	LF	42	0x2A	052	*
11	0x0B	013	VT	43	0x2B	053	+
12	0x0C	014	FF	44	0x2C	054	,
13	0x0D	015	CR	45	0x2D	055	-
14	0x0E	016	SO	46	0x2E	056	.
15	0x0F	017	SI	47	0x2F	057	/
16	0x10	020	DLE	48	0x30	060	0
17	0x11	021	DC1	49	0x31	061	1
18	0x12	022	DC2	50	0x32	062	2
19	0x13	023	DC3	51	0x33	063	3
20	0x14	024	DC4	52	0x34	064	4
21	0x15	025	NAK	53	0x35	065	5
22	0x16	026	SYN	54	0x36	066	6
23	0x17	027	ETB	55	0x37	067	7
24	0x18	030	CAN	56	0x38	070	8
25	0x19	031	EM	57	0x39	071	9
26	0x1A	032	SUB	58	0x3A	072	:
27	0x1B	033	ESC	59	0x3B	073	;
28	0x1C	034	FS	60	0x3C	074	"<
29	0x1D	035	GS	61	0x3D	075	=
30	0x1E	036	RS	62	0x3E	076	">
31	0x1F	037	US	63	0x3F	077	?

Table 7: Letters in ASCII

Dez	Hex	Okt	Zeichen	Dez	Hex	Okt	Zeichen
64	0x40	100	@	96	0x60	140	‘
65	0x41	101	A	97	0x61	141	a
66	0x42	102	B	98	0x62	142	b
67	0x43	103	C	99	0x63	143	c
68	0x44	104	D	100	0x64	144	d
69	0x45	105	E	101	0x65	145	e
70	0x46	106	F	102	0x66	146	f
71	0x47	107	G	103	0x67	147	g
72	0x48	110	H	104	0x68	150	h
73	0x49	111	I	105	0x69	151	i
74	0x4A	112	J	106	0x6A	152	j
75	0x4B	113	K	107	0x6B	153	k
76	0x4C	114	L	108	0x6C	154	l
77	0x4D	115	M	109	0x6D	155	m
78	0x4E	116	N	110	0x6E	156	n
79	0x4F	117	O	111	0x6F	157	o
80	0x50	120	P	112	0x70	160	p
81	0x51	121	Q	113	0x71	161	q
82	0x52	122	R	114	0x72	162	r
83	0x53	123	S	115	0x73	163	s
84	0x54	124	T	116	0x74	164	t
85	0x55	125	U	117	0x75	165	u
86	0x56	126	V	118	0x76	166	v
87	0x57	127	W	119	0x77	167	w
88	0x58	130	X	120	0x78	170	x
89	0x59	131	Y	121	0x79	171	y
90	0x5A	132	Z	122	0x7A	172	z
91	0x5B	133	[	123	0x7B	173	{
92	0x5C	134	\	124	0x7C	174	|
93	0x5D	135	]	125	0x7D	175	}
94	0x5E	136	ˆ	126	0x7E	176	"
95	0x5F	137	_	127	0x7F	177	DEL

Rules for graph transformation of lexical analysis

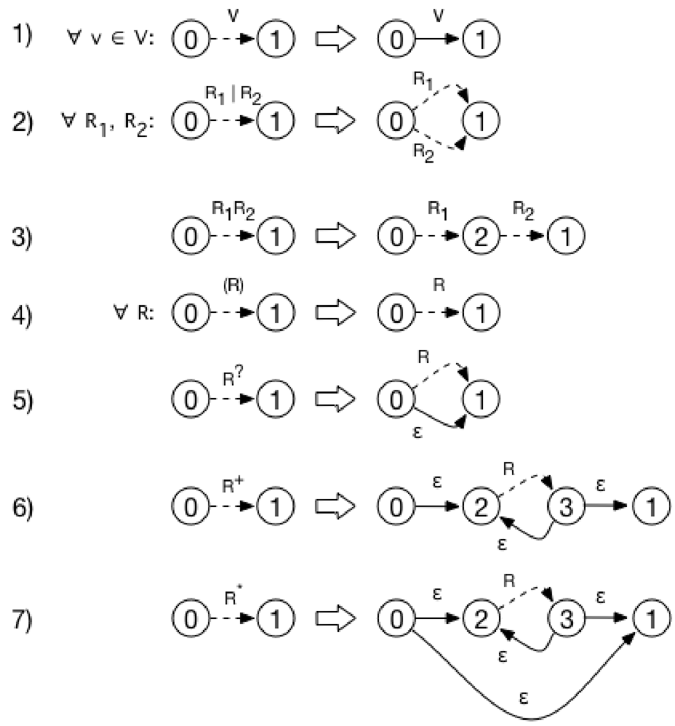

Figure 12: Rules for graph transformation of lexical analysis

Generation of the lexer

```java
public class ELexer extends Lexer {
    static { RuntimeMetaData.checkVersion("4.7.2", RuntimeMetaData
        .VERSION); }

    protected static final DFA[] _decisionToDFA;
    protected static final PredictionContextCache _sharedContextCache =
        new PredictionContextCache();
    public static final int
        T__0=1, T__1=2, T__2=3, T__3=4, T__4=5, T__5=6, T__6=7, T__7=8,
        ...
        ASSIGN=51, QMARK=52, OPAREN=53, CPAREN=54, OBRACE=55, CBRACE=56,
        OBRACKET=57, CBRACKET=58, OCBRACKET=59;
    public static String[] channelNames = {
        "DEFAULT_TOKEN_CHANNEL", "HIDDEN"
    };

    public static String[] modeNames = {
        "DEFAULT_MODE"
    };

    private static String[] makeRuleNames() {
        return new String[] {
            "T__0", "T__1", "T__2", "T__3", "T__4", "T__5", "T__6", "T__7",
            ...
            "CBRACE", "OBRACKET", "CBRACKET", "OCBRACKET", "LETTER", "DIGIT"
        };
    }
    public static final String[] ruleNames = makeRuleNames();

    private static String[] makeLiteralNames() {
        return new String[] {
            null, "'use'", "'#define'", "'noMain'", "'print'", "'println'",
            ...
            "'='", "'\"'", "'('", "')'", "'{'", "'}'", "'['", "']'", "'[]'"
        };
    }
    private static final String[] _LITERAL_NAMES = makeLiteralNames();
    private static String[] makeSymbolicNames() {
        return new String[] {
            "BUILTINFUNCTION", "BOOL", "INTEGER", "FLOAT", "STRING",
            ...
            "ASSIGN", "QMARK", "OPAREN", "CPAREN", "OBRACE", "OCBRACKET"
        };
    }
    private static final String[] _SYMBOLIC_NAMES = makeSymbolicNames();
    public static final Vocabulary VOCABULARY = new VocabularyImpl
```

```
        (_LITERAL_NAMES, _SYMBOLIC_NAMES);

    public static final String[] tokenNames;
    static {
        tokenNames = new String[_SYMBOLIC_NAMES.length];
        for (int i = 0; i < tokenNames.length; i++) {
            tokenNames[i] = VOCABULARY.getLiteralName(i);
            if (tokenNames[i] == null) {
                tokenNames[i] = VOCABULARY.getSymbolicName(i);
            }

            if (tokenNames[i] == null) {
                tokenNames[i] = "<INVALID>";
            }
        }
    }
...
    public ELexer(CharStream input) {
        super(input);
        _interp = new LexerATNSimulator(this,_ATN,_decisionToDFA,
            _sharedContextCache);
    }

    public String getGrammarFileName() { return "E.g4"; }

    public String[] getRuleNames() { return ruleNames; }
...
    public static final ATN _ATN =
        new ATNDeserializer().deserialize(_serializedATN.toCharArray());
    static {
        _decisionToDFA = new DFA[_ATN.getNumberOfDecisions()];
        for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
            _decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);
        }
    }
}
```

Code 17: Generation of `ELexer.java`

Deterministic context-free language classes

Recursive descent parser (top-down)		Shift reduce parser (bottom-up)	
$LL(k)$	Predicts the next production using k tokens (taking the stack contents into account). The left context must therefore be noted.	$LR(k)$	Detects syntax errors as soon as possible and can parse all deterministic context-free languages. $LR(1)$ is canonical but relatively large parser tables are created.
$SLL(k)$	Indicates a subset of LL(k) where the left context does not have to be kept.	$SLR(k)$	Creates simple parser tables and is relatively easy to implement. However, not all deterministic grammars can be handled with it. It does not have to scan all lookahead reductions.
		$LALR(k)$	The complexity lies between *LR* and *SLR*. It therefore has a similar number of states to *SLR* but recognizes more languages. It tries to combine states if goto tables and lookaheads are compatible. Conflicts are recognized by rules.
		$GLR(k)$	Detects ambiguous grammars through rules and can be combined with *LALR*.

Generation of the parser

```
public class EParser extends Parser {
    static { RuntimeMetaData.checkVersion("4.7.2", RuntimeMetaData
        .VERSION); }

    protected static final DFA[] _decisionToDFA;
    protected static final PredictionContextCache _sharedContextCache =
        new PredictionContextCache();
    public static final int T__0=1, T__1=2, T__2=3, T__3=4, T__4=5,
        T__5=6, T__6=7, T__7=8, T__8=9,
        ...
        RULE_block = 24, RULE_statements = 25, RULE_formalParameters = 26,
        RULE_dataType = 27, RULE_primitive = 28;
    private static String[] makeRuleNames() {
        return new String[] {
            "program", "includes", "module", "command", "statement",
            "print", ..., "formalParameters", "dataType", "primitive"
        };
    }
    public static final String[] ruleNames = makeRuleNames();

    private static String[] makeLiteralNames() {
        return new String[] {
            "'use'", "'#define'", "'noMain'", "'print'",
            ...
            "'else'", "'while'", "'asm'", "'invoke'", "'new'"
        };
    }
    private static final String[] _LITERAL_NAMES = makeLiteralNames();
    public static final Vocabulary VOCABULARY = new VocabularyImpl(
        _LITERAL_NAMES, _SYMBOLIC_NAMES);

    public static final String[] tokenNames;
    static {
        tokenNames = new String[_SYMBOLIC_NAMES.length];
        for (int i = 0; i < tokenNames.length; i++) {
            tokenNames[i] = VOCABULARY.getLiteralName(i);
            if (tokenNames[i] == null) {
                tokenNames[i] = VOCABULARY.getSymbolicName(i);
            }

            if (tokenNames[i] == null) {
                tokenNames[i] = "<INVALID>";
            }
        }
    }
...
```

```java
    public static class ProgramContext extends ParserRuleContext {
        public IncludesContext includes;
        public List<IncludesContext> incls =
            new ArrayList<IncludesContext>();
        public NoMainContext noMain;
        public List<NoMainContext> noMains =
            new ArrayList<NoMainContext>();
        public TerminalNode EOF() { return getToken(EParser.EOF, 0); }
        public List<CommandContext> command() {
            return getRuleContexts(CommandContext.class);
        }
        public CommandContext command(int i) {
            return getRuleContext(CommandContext.class,i);
        }
        public List<IncludesContext> includes() {
            return getRuleContexts(IncludesContext.class);
        }
        public IncludesContext includes(int i) {
            return getRuleContext(IncludesContext.class,i);
        }
        public List<NoMainContext> noMain() {
            return getRuleContexts(NoMainContext.class);
        }
        public NoMainContext noMain(int i) {
            return getRuleContext(NoMainContext.class,i);
        }
        public ProgramContext(ParserRuleContext parent,
            int invokingState) {
            super(parent, invokingState);
        }
        public int getRuleIndex() { return RULE_program; }
        @Override
        public <T> T accept(ParseTreeVisitor<? extends T> visitor) {
            if ( visitor instanceof EVisitor ) return ((EVisitor<? extends
                T>)visitor).visitProgram(this);
            else return visitor.visitChildren(this);
        }
    }

    public final ProgramContext program() throws RecognitionException {
        ProgramContext _localctx = new ProgramContext(_ctx, getState());
        enterRule(_localctx, 0, RULE_program);
        int _la;
        try {
            enterOuterAlt(_localctx, 1);
            {
            setState(61);
            _errHandler.sync(this);
```

```
            _la = _input.LA(1);
            while (_la==T__0) {
                {
                {
                setState(58);
                ((ProgramContext)_localctx).includes = includes();
                ((ProgramContext)_localctx).incls.add(
                    ((ProgramContext)_localctx).includes);
                }
                }
                setState(63);
                _errHandler.sync(this);
                _la = _input.LA(1);
            }
            setState(67);
            _errHandler.sync(this);
            _la = _input.LA(1);
            while (_la==T__1) {
                {
                {
                setState(64);
                ((ProgramContext)_localctx).noMain = noMain();
                ((ProgramContext)_localctx).noMains.add(
                    ((ProgramContext)_localctx).noMain);
                }
                }
                setState(69);
                _errHandler.sync(this);
                _la = _input.LA(1);
            }
            ...
        return _localctx;
    }
...
    public final IncludesContext includes() throws RecognitionException
        { ... }
...
    public static final ATN _ATN =
        new ATNDeserializer().deserialize(_serializedATN.toCharArray());
    static {
        _decisionToDFA = new DFA[_ATN.getNumberOfDecisions()];
        for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
            _decisionToDFA[i] = new DFA(_ATN.getDecisionState(i), i);
        }
    }
}
```

Code 18: Generation of `EParser.java`

Implementation of the exceptions

```java
public class AlreadyDefinedVariableException extends CompilerException {
    private String id;

    public AlreadyDefinedVariableException(Token var) {
        super(var);
        id = var.getText();
    }

    @Override
    public String getMessage() {
        return line + ":" + column +
        " Already defined variable: <" + id + ">;";
    }
}
```

Code 19: Implementation of `AlreadyDefinedVariableException.java`

```java
public class UndeclaredVariableException extends CompilerException {
    private String id;

    public UndeclaredVariableException(Token var) {
        super(var);
        id = var.getText();
    }

    @Override
    public String getMessage() {
        return line + ":" + column +
        " Undeclared variable: <" + id + ">;";
    }
}
```

Code 20: Implementation of `UndeclaredVariableException.java`

```java
public class UndefinedFunctionException extends CompilerException {
    private String id;

    public UndefinedFunctionException(Token func) {
        super(func);
        id = func.getText();
    }

    @Override
    public String getMessage() {
        return line + ":" + column + " Undefined function: <" + id + ">;";
    }
}
```

Code 21: Implementation of `UndefinedFunctionException.java`

```java
public class AlreadyDefinedFunctionException extends CompilerException {
   private String id;

   public AlreadyDefinedFunctionException(Token func) {
      super(func);
      id = func.getText();
   }

   @Override
   public String getMessage() {
      return line + ":" + column + " Already defined function: <"+id+">;";
   }
}
```

Code 22: Implementation of `AlreadyDefinedFunctionException.java`

```java
public class UndeclaredStructException extends CompilerException {
   private final String id;

   public UndeclaredStructException(final Token var) {
      super(var);
      id = var.getText();
   }

   @Override
   public String getMessage() {
      return line + ":" + column + " Undeclared struct: <" + id + ">;";
   }
}
```

Code 23: Implementation of `UndeclaredStructException.java`

```java
public class AlreadyDeclaredStructException extends CompilerException {
   private final String id;

   public AlreadyDeclaredStructException(final Token var) {
      super(var);
      id = var.getText();
   }

   @Override
   public String getMessage() {
      return line + ":" + column + " Already declared struct: <" + id +
         ">;";
   }
}
```

Code 24: Implementation of `AlreadyDeclaredStructException.java`

```java
public class WrongDataTypeException extends CompilerException {
    /** Tracks all of the nodes in the AST traversed by the parser. */
    private final String track;

    public WrongDataTypeException(final Token track) {
        super(track);
        this.track = track.getText();
    }

    @Override
    public String getMessage() {
        return line + ":" + column + " Wrong data type: <" + track + ">;";
    }
}
```

Code 25: Implementation of `WrongDataTypeException.java`

```java
public class UnknownBuiltinFunctionException extends CompilerException {
    private final String id;
    public UnknownBuiltinFunctionException(final Token func) {
        super(func);
        id = func.getText();
    }
    @Override
    public String getMessage() {
        return line + ":" + column + " The built-in function could not be
            found: <" + id + ">;";
    }
}
```

Code 26: Implementation of `UnknownBuiltinFunctionException.java`

```java
public class UnknownModuleException extends CompilerException {
    private final String id;
    public UnknownModuleException(final Token mod) {
        super(mod);
        id = mod.getText();
    }
    @Override
    public String getMessage() {
        return line + ":" + column + " The module could not be found: <" +
            id + ">;";
    }
}
```

Code 27: Implementation of `UnknownModuleException.java`

Jasmin instructions

Table 8: Relevant instructions of Jasmin

Instruction	Opcode	Description
`dup`	0x59	This pops the top single-word value off the operand stack and then pushes that value twice.
`f2i`	0x8B	Pops a single precision float off of the stack, casts it to a 32-bit integer and pushes the integer value back onto the stack.
`getstatic`	0xB2	Pops a reference to an object from the stack, retrieves the value of the static field (also known as a class field) identified by a field specification from the reference and pushes the one-word or two-word value onto the operand stack.
`goto`	0xA7	Causes execution to branch to the instruction at the address (pc + branchoffset), where pc is the address of the goto opcode in the bytecode and branchoffset is a 16-bit signed integer parameter that immediately follows the goto opcode in the bytecode.
`iadd`	0x60	Pops two integers from the stack, adds them and pushes the integer result back onto the stack.
`idiv`	0x6C	Pops the top two integers from the operand stack and divides the second-from top integer by the top integer. The quotient result is truncated to the nearest integer and placed on the stack.
`ifeq`	0x99	Pops the top integer off the operand stack. If the integer equals zero, execution branches to the address (pc + branchoffset).
`ifne`	0x9A	Pops the top integer off the operand stack. If the integer does not equal zero, execution branches to the address (pc + branchoffset).
`if_icmplt`	0xA1	Pops the top two ints off the stack and compares them.
`iload`	0x15	Loads a value from the table regarding variables at a desired position onto the stack.
`imul`	0x68	Pops the top two integers from the operand stack, multiplies them and pushes the result back onto the stack.
`invokespecial`	0xB7	Is used in certain special cases to invoke a method.
`invokestatic`	0xB8	Calls a static method (also known as a class method).

Table 9: Relevant instructions of Jasmin 2

Instruction	Opcode	Description
`invokevirtual`	0xB6	Dispatches a Java method. It is used in Java to invoke all methods except interface methods.
`irem`	0x70	Pops two integers from the operand stack, divides value2 by value1, computes the remainder and pushes the integer remainder back onto the stack.
`ireturn`	0xAC	Pops an integer from the top of the stack and pushes it onto the operand stack of the invoker.
`ishl`	0x78	Pops two integers off the stack. Shifts *value2* left by the amount indicated in the five low bits of *value1*. The integer result is then pushed back onto the stack.
`ishr`	0x7A	Pops two integers off the stack. Shifts *value1* right by the amount indicated in the five low bits of *value2*. The integer result is then pushed back onto the stack. *value1* is shifted arithmetically (preserving the sign extension).
`istore`	0x36	Gets the topmost integer from the stack and places it in the table regarding the variables.
`isub`	0x64	Pops two integers from the stack, subtracts the top one from the second one and pushes the result back onto the stack.
`ixor`	0x82	Pops two integers off the operand stack. Computes the bitwise exclusive or of *value1* and *value2*. The integer result replaces *value1* and *value2* on the stack.
`i2f`	0x86	Pops an integer off the operand stack, casts it into a single-precision float and pushes the float back onto the stack.
`ldc`	0x12	Loads a constant value onto the stack.
`new`	0xBB	Determines the size in bytes of instances of the given class and allocates memory for the new instance from the garbage collected heap.
`newarray`	0xBC	Pops a positive integer off the stack and constructs an array for holding n elements of the given type.
`putstatic`	0xB3	Sets the value of the static field identified by a field specification to the single or double word value on the operand stack.

Note: The *value2* is always below *value1*. There are prefixes for several data types for different operators. For example, there is `fadd` for the addition of floating point numbers. There are also special Jasmin commands like `.field` to add an attribute to a class.

Example of an E program

```
// Different library imports
use(e.std.math)

/**
 * A point in a two-dimensional space
 *
 * @author Rune Krauss
 */
struct Point1 {
    int x;
    int y;
}

/**
 * One more point in a two-dimensional space
 *
 * @author Rune Krauss
 */
struct Point2 {
    float x;
    float y;
}

/**
 * Finds the square root of an integer.
 *
 * @param x Integer
 * @return Square root
 */
int sqrt(int x) {
    int result;
    // Illegal
    if (x < 0) {
        result = -1;
    }
    // Base cases
    if (x == 0 || x == 1) {
        result = x;
    } else {
        int temp = 1;
        int i = 1;
        /*
         * Starting from 1, try all numbers until
         * i*i is greater than or equal to x
         */
        while (temp <= x) {
```

```
            i = i + 1;
            temp = i * i;
        }
        result = i - 1;
    }
    return result;
}

/**
 * Calculates the distance between two points.
 *
 * @param x1 X value of the first point
 * @param y1 Y value of the first point
 * @param x2 X value of the second point
 * @param y2 Y value of the second point
 * @return Distance
 */
int dist(int x1, int y1, int x2, int y2) {
    int part1 = math.square(x2-x1): int;
    int part2 = math.square(y2-y1): int;
    int result = sqrt(part1+part2);
    return result;
}

// Create different points
Point1 p1 = new Point1(5, 2);
Point2 p2 = new Point2(3.7, 4.1);
p1.x = 1;

// Calculate the distance between these points
int dist = dist(p1.x, p1.y, toInt(p2.x), toInt(p2.y));

// Save the result in a list
int[] list = new int[3];
list[0] = dist;

// Output some information about the list
String txt = "Result: ";
String output = append(txt, toString(dist));

print(output); // Result: 2
```

Code 28: Implementation of `main.e`

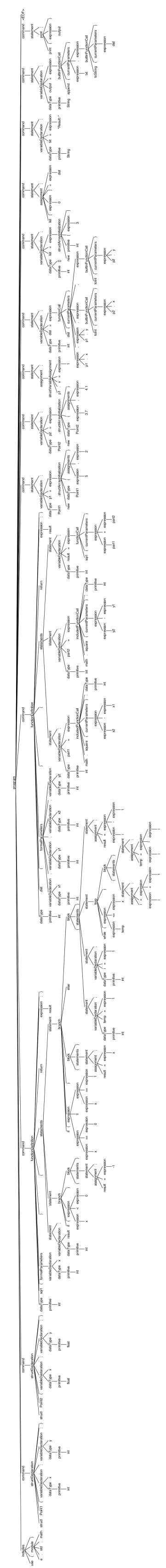

Figure 13: AST of `main.e`

```
.class public e_test_main
.super java/lang/Object

.field public static v0 Lstruct_e_test_main0;
.field public static v1 Lstruct_e_test_main1;
.field public static v2 I
.field public static v3 [I
.field public static v4 Ljava/lang/String;
.field public static v5 Ljava/lang/String;

.method public static sqrt(I)I
.limit locals 4
.limit stack 4

iload 0
ldc 0
if_icmplt onCmpTrue1
ldc 0
goto endCmp1
onCmpTrue1:
ldc 1
endCmp1:
ifne onTrue1

goto endIf1
onTrue1:
ldc -1
istore 1

endIf1:

iload 0
ldc 0
if_icmpeq onCmpTrue2
ldc 0
goto endCmp2
onCmpTrue2:
ldc 1
endCmp2:
ifne onOrTrue1
iload 0
ldc 1
if_icmpeq onCmpTrue3
ldc 0
goto endCmp3
onCmpTrue3:
ldc 1
```

```
endCmp3:
ifne onOrTrue1
ldc 0
goto endOr1
onOrTrue1:
ldc 1
endOr1:

ifne onTrue2
ldc 1
istore 2
ldc 1
istore 3
beginLoop1:
iload 2
iload 0
if_icmple onCmpTrue4
ldc 0
goto endCmp4
onCmpTrue4:
ldc 1
endCmp4:
ifeq endLoop1
iload 3
ldc 1
iadd
istore 3

iload 3
iload 3
imul
istore 2

goto beginLoop1
endLoop1:

iload 3
ldc 1
isub
istore 1

goto endIf2
onTrue2:
iload 0
istore 1

endIf2:

```

```
iload 1
ireturn
.end method

.method public static dist(IIII)I
.limit locals 7
.limit stack 5
iload 2
iload 0
isub

invokestatic e_std_math/square(I)I
istore 4
iload 3
iload 1
isub

invokestatic e_std_math/square(I)I
istore 5
iload 4
iload 5
iadd

invokestatic e_test_main/sqrt(I)I
istore 6
iload 6
ireturn
.end method


.method public static main([Ljava/lang/String;)V
.limit stack 9
.limit locals 1

new struct_e_test_main0
dup
invokespecial struct_e_test_main0/<init>()V
putstatic e_test_main/v0 Lstruct_e_test_main0;
getstatic e_test_main/v0 Lstruct_e_test_main0;
ldc 5
putfield struct_e_test_main0/a0 I
getstatic e_test_main/v0 Lstruct_e_test_main0;
ldc 2
putfield struct_e_test_main0/a1 I
getstatic e_test_main/v0 Lstruct_e_test_main0;

putstatic e_test_main/v0 Lstruct_e_test_main0;
new struct_e_test_main1
```

```
dup
invokespecial struct_e_test_main1/<init>()V
putstatic e_test_main/v1 Lstruct_e_test_main1;
getstatic e_test_main/v1 Lstruct_e_test_main1;
ldc 3.7
putfield struct_e_test_main1/a0 F
getstatic e_test_main/v1 Lstruct_e_test_main1;
ldc 4.1
putfield struct_e_test_main1/a1 F
getstatic e_test_main/v1 Lstruct_e_test_main1;

putstatic e_test_main/v1 Lstruct_e_test_main1;
getstatic e_test_main/v0 Lstruct_e_test_main0;
ldc 1
putfield struct_e_test_main0/a0 I

getstatic e_test_main/v0 Lstruct_e_test_main0;
getfield struct_e_test_main0/a0 I

getstatic e_test_main/v0 Lstruct_e_test_main0;
getfield struct_e_test_main0/a1 I

getstatic e_test_main/v1 Lstruct_e_test_main1;
getfield struct_e_test_main1/a0 F

f2i
getstatic e_test_main/v1 Lstruct_e_test_main1;
getfield struct_e_test_main1/a1 F

f2i

invokestatic e_test_main/dist(IIII)I
putstatic e_test_main/v2 I
ldc 3
newarray int

putstatic e_test_main/v3 [I
getstatic e_test_main/v3 [I
ldc 0
getstatic e_test_main/v2 I
iastore

ldc "Result: "
putstatic e_test_main/v4 Ljava/lang/String;
new java/lang/StringBuffer
dup
invokespecial java/lang/StringBuffer/<init>()V
getstatic e_test_main/v4 Ljava/lang/String;
```

```
invokevirtual java/lang/StringBuffer/append(Ljava/lang/String;)Ljava/
   lang/StringBuffer;
getstatic e_test_main/v2 I
invokestatic java/lang/Integer.toString(I)Ljava/lang/String;

invokevirtual java/lang/StringBuffer/append(Ljava/lang/String;)Ljava/
   lang/StringBuffer;
invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
putstatic e_test_main/v5 Ljava/lang/String;
getstatic java/lang/System/out Ljava/io/PrintStream;
getstatic e_test_main/v5 Ljava/lang/String;
invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V



return

.end method
```

Code 29: Generated code of `main.e`

```
.class struct_e_test_main0
.super java/lang/Object

.field public a0 I
.field public a1 I
.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
```

Code 30: Generated code of struct `Point1`

```
.class struct_e_test_main1
.super java/lang/Object

.field public a0 F
.field public a1 F
.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
```

Code 31: Generated code of struct `Point2`

```
.class public e_std_math
.super java/lang/Object
```

```


.method public static square(I)I
.limit locals 1
.limit stack 2
iload 0
iload 0
imul
ireturn
.end method

.method public static max(II)I
.limit locals 3
.limit stack 3
iload 0
istore 2
iload 1
iload 0
if_icmpgt onCmpTrue1
ldc 0
goto endCmp1
onCmpTrue1:
ldc 1
endCmp1:
ifne onTrue1

goto endIf1
onTrue1:
iload 1
istore 2

endIf1:

iload 2
ireturn
.end method

.method public static min(II)I
.limit locals 3
.limit stack 3
iload 0
istore 2
iload 1
iload 0
if_icmplt onCmpTrue2
ldc 0
goto endCmp2
```

```
onCmpTrue2:
ldc 1
endCmp2:
ifne onTrue2

goto endIf2
onTrue2:
iload 1
istore 2

endIf2:

iload 2
ireturn
.end method

.method public static abs(I)I
.limit locals 1
.limit stack 3
iload 0
ldc 0
if_icmplt onCmpTrue3
ldc 0
goto endCmp3
onCmpTrue3:
ldc 1
endCmp3:
ifne onTrue3

goto endIf3
onTrue3:
iload 0
ineg
istore 0

endIf3:

iload 0
ireturn
.end method
```

Code 32: Generated code of `math.e`

Test files

```
if (1) {
    print(1);
} else {
    print(0);
}
```

Code 33: Implementation of `if-else_one_true.e`

```
if (5) {
    print(1);
} else {
    print(0);
}
```

Code 34: Implementation of `if-else_other_true.e`

```
if (0) {
    print(0);
} else {
    print(1);
}
```

Code 35: Implementation of `if-else_zero_false.e`

```
int i = 0;
while (i < 3) {
    i = i + 1;
}
i = i + 1;
print(i);
```

Code 36: Implementation of `while.e`

```
int mul(int a, int b) {
    return a*b;
}

print(mul(3, 5));
```

Code 37: Implementation of `current_formal_parameter.e`

```
int get_number() {
    int n;
    n = 3;
    return n;
}

print(get_number());
```

Code 38: Implementation of `local_parameter.e`

```
int get_val() {
    return 1;
}

int get_val(int a) {
    return a;
}

print(get_val());
print(get_val(5));
```

Code 39: Implementation of `overloading.e`

```
int get_number() {
    int n;
    n = 3;
    return n;
}

int n;
n = 5;
print(get_number());
print(n);
```

Code 40: Implementation of `scope.e`

```
int get_number() {
    return 3;
}

print(get_number());
```

Code 41: Implementation of `simple.e`

```
int id(int a) {
    println(a);
    return a;
}

print(id(0) && id(1));
```

Code 42: Implementation of `lazy_eval_and.e`

```
int id(int a) {
    println(a);
    return a;
```

```
}
print(id(1) || id(0));
```

Code 43: Implementation of `lazy_eval_or.e`

```
float number() {
    asm {
        "
        ldc 5.1
        ldc 0.4
        fadd
        "
    }
    setTopOfStack "float";
    return topOfStack;
}
print(number());
```

Code 44: Implementation of `inline_asm.e`

```
// Prints 5
print(5);
```

Code 45: Implementation of `line_comment.e`

```
/*
 * Returns an id.
 */
int id(int x) {
    return x;
}
print(5);
```

Code 46: Implementation of `multiline_comment.e`

```
/**
 * Returns an id.
 *
 * @param x Number
 * @return Id
 */
int id(int x) {
    return x;
}
print(5);
```

Code 47: Implementation of `special_comment.e`

Declaration of Academic Integrity

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

$Signature:$ $Place, Date:$