Dobler
März 2025

Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

MiniC++ Compiler with Java Technologies

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Andreas Zauner, BSc

Betreuung: FH-Prof. DI Dr. Dobler Heinz
Begutachtung: FH-Prof. DI Dr. Dobler Heinz

Hagenberg, Juni 2025

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

This printed thesis is identical with the electronic version submitted.

Date	Signature

Contents

Chapter 1

Introduction

1.1 Motivation

are

functionality of the

Compilers ~~function as~~ the backbone for computer programming. A compiler ~~takes care of~~ translating human-readable source code into something a computer can execute. This allows ~~the application~~ developers to focus on ~~writing~~ the application, without having to worry about the technicalities of the concrete computer where the software will run on. For one programming language there may exist multiple compilers targeting different kinds of computers. This allows the same source code to run for example on Linux and Windows computers with Intel or ARM processors. This flexibility saves developers a lot of work, because they don't need to rewrite their application in the case they also want to target another operating system and/or processor. Furthermore, ~~there exist~~ #[...] compilers that target virtual machines like the Java Virtual Machine (JVM). Generating code for a virtual machine has the advantage that there is no need for compilers for every target operating system and/or processor. Instead, for each operating system an implementation of the virtual machine is provided.

The process of compiling source code begins in the frontend of the compiler. The frontend reads the source code and constructs an abstract syntax tree (AST). The AST is a ~~funtime~~ representation of the source code in memory. It contains only the necessary information that is ~~later on~~ needed to generate target code. The process of constructing compiler the AST is based on the grammar of the programming language. Based on this grammar a lexer and parser are either written manually or get generated by a ~~parser~~ generator tool like ANTLR. In the case of ANTLR the generated lexer and parser construct a full parse tree from the input. From the parse tree an AST can be constructed using for example the visitor pattern.

by default

The AST functions then as the input for the backend of the compiler: The backend generates code for the target system. In the case of the JVM this is the so called bytecode. APIs exist that provide an abstraction layer ~~to~~ the code generation. One API for bytecode generation is the open source project ObjectWeb ASM or just ASM (Bruneton 2007). It provides an API that utilizes the visitor pattern to generate bytecode instructions.

for

1.2 Task and Goal

in Hagenberg another

MiniC++ is a subset of the C++ programming language. The scope of MiniC++ is very limited in comparison to C++. It is used at the University of Applied Sciences Upper Austria for teaching software engineering master students about compilers in the formal languages class. In this class, all aspects of a compiler are discussed. First, the the principles of lexers and parsers are explained. Then the concepts of syntax trees and further abstract syntax trees are introduced. Finally, code generation is explained.

In the exercises, students use a MiniC++ compiler to compile MiniC++ source code to .NET Common Intermediate Language (CIL). The frontend of the compiler is generated by using the compiler generator Coco-2 (Dobler and Pirklbauer 1990). Which generates both, the lexer and the parser. There is only one input-file required for the definition of the lexer and the parser. Furthermore, attributes and semantic actions can be included to create an attributed grammar (ATG).

In this master thesis, a compiler for MiniC++ will be created. ~~The~~ this compiler will be built upon Java technologies. Output of the compiler will be Java bytecode that can be executed on the Java Virtual Machine (JVM). The frontend is based on a lexer and parser generated by the parser generator ANTLR[1] (ANother Tool for Language Recognition). They are used to generate a full syntax tree. From this syntax tree an abstract syntax tree (AST) is constructed. The backend utilizes the ObjectWeb ASM[2] library. This library provides an API to generate Java bytecode.

This master thesis will further explore the capabilities of ANTLR. ANTLR provides multiple ways to interact with the generated parser. The master thesis compares the advantages and disadvantages of each of these options.

1.3 Theoretical Fundamentals

This section explains the basic concepts ~~behind~~ of formal languages and how they are used in compilers. Furthermore, the individual components of a compiler are highlighted.

1.3.1 Formal Languages and Compilers

Formal languages make up the fundament for on which compilers are built upon. In comparison to natural languages, formal languages have a strict syntax which can be defined by a grammar. This grammar does not evolve naturally, as it does with natural languages. A formal grammar is defined by replacement rules. A replacement rule defines that a non-terminal symbol A can be replaced by a sequence α. The sequence may contain terminal and non-terminal symbols.

Grammars can be classified according to the Chomsky hierarchy (Chomsky 1959). Chomsky classifies formal languages and their grammars into four categories. Of those, the first two are relevant for compiler construction. Namely, regular grammars and context-free grammars. The four categories are differentiated by the type of rules that can be defined. The types of rules used then define which kind of automaton is needed to recognize sentences of the given language.

[1] https://www.antlr.org/

[2] https://asm.ow2.io/

or $S \to \varepsilon$!

Regular Grammars

(DFA)

Regular grammars make up the simplest group of grammars. For a grammar to be ~~a~~ regular ~~grammar~~ all rules must be in the form of $A \to a|aB$. This means that a non-terminal symbol A can only be replaced by either a terminal symbol a or a terminal symbol a followed by a non-terminal symbol B. The only exception is the root rule S which can be replaced by the empty sequence. OK!

To recognize a sentence of a regular grammar a finite automaton (FA) can be used. A deterministic FA consists of the following elements:

- S finite, non-empty set of states ,
- Σ finite, non-empty set of symbols (alphabet) ,
- s_0 initial state, $s_0 \in S$,
- δ state transition function, $S \times \Sigma \to S$ and
- F set of final states, $F \subseteq S$.

? or ... ?

The DFA proceeds to read the symbols in σ one symbol at a time. The current symbol is then used in combination with the current state in the state transition function to acquire a new state. This process is continued until a final state is reached, meaning that a sentence has successfully been recognized. In case that for the current symbol and state no entry in the state transition function can be found, the recognition failed, and the given input is not a sentence of the language. again!

A DFA can be implemented in a program to efficiently ~~detect~~ recognize sentences of a language. For more complicated regular ~~grammars a~~ nondeterministic finite automaton (NFA) ~~is~~ are easier to construct. A NFA program however is more complicated and slower compared to a DFA one. Every NFA can be transformed in to a DFA to overcome this limitation. After transformation the constructed DFA may have more than the minimal ~~amount~~ number of states needed. A second transformation can be performed that reduces the DFA to a minimal DFA. languages

Context-Free Grammars

Context-free grammars are the second group of grammars according to the Chomsky hierarchy. Context-free grammars ~~also~~ include regular grammars, meaning that every regular grammar is also a context-free grammar. A replacement rule of a context-free grammar is in the form $A \to \beta$. Meaning that a non-terminal symbol A can be replaced by a sequence β containing terminal and/or non-terminal symbols or also ϵ, the empty sequence. ?

(PDA)

In a context-free grammar central recursion is possible (direct or indirect). This allows the nested structures that are needed for programming languages, e.g., for expression hierarchies. Central recursion cannot be handled by a ~~D~~FA, for this a pushdown automaton is needed. With a deterministic pushdown automaton (DPDA) all deterministic context-free grammars can be recognized. To recognize all context-free grammars a nondeterministic pushdown automaton is needed. For programming languages deterministic context-free grammars are used.

There are two strategies for constructing a syntax tree from a sentence of a context-free grammar, namely top-down and bottom-up. Which strategy can be used depends

(NPDA)

on the kind of deterministic context-free grammar that is used. Following are the two most important conditions for context-free grammars:

- **LL(k) Condition:** Defines that a maximum of ~~(k)~~ symbols look ahead ~~while parsing~~ are sufficient to deterministically decide on the next rule when using the *top-down* strategy.
- **LR(k) Condition:** Defines that a maximum of ~~(k)~~ look ahead ~~while parsing~~ are sufficient to deterministically decide on the next ~~rule~~ action (shift or reduce) when using the *bottom-up* strategy.

The higher the value of k, the more complicated parsing becomes. Therefore, LL(1) and LR(1) grammars are preferred. For an LL(1) or LR(1) grammar only one look ahead symbol is needed ~~to deterministically decide on the next rule~~ for a decision.

LL(k) grammars can be recognized with a normal DPDA. For LL(1) grammars it is also feasible to implement an efficient recursive descent parser. In the case of an LR(1) grammar, the DPDA must be extended to be able to use an arbitrary amount of symbols on top of the ~~At~~ stack. Only then is it able to recognize a sentence of an LR(1) grammar with the *bottom-up* strategy. It has to be noted that a DPDA which is able to recognize LR(k) grammars, is also able to recognize LL(k) grammars.

1.3.2 Compiler Construction

The task of a compiler is to translate code ~~of~~ ok! a given source language into code of a target language. The source language being a human-readable programming language like Java and the target language being code for a given operating system and processor architecture, or a virtual machine. Compiling code can be separated into two main stages: frontend and backend. The frontend ~~consists~~ executes of the following steps:

- lexical analysis,
- syntactic analysis,
- semantic evaluation and
- intermediate language generation

???

The backend performs optimization and code generation.

The lexical analysis is the first step of the compilation. It reads the source code and organizes it. The goal is to group individual characters into symbols (tokens) and to skip meaningless characters (e.g., comments). The grammar of the source language provides the information about the symbols. This part of the grammar is defined using a regular grammar.

The symbols can be divided into terminal symbols and terminal classes. Terminal symbols are special symbols like =, (and the keywords of the source language, e.g., `int`, `break`, `function`. Terminal classes are for example all numbers or identifiers. Comments are also handled at this step. Since comments usually have no influence on the generated code, they are removed. All recognized symbols are then passed ~~on~~ to the parser (syntactic analysis and semantic evaluation).

The syntactic analysis takes the terminal symbols and classes recognized in the lexical analysis phase as input to construct the syntax tree. A context-free grammar provides the basis for the syntax tree. During the syntactic analysis the terminal symbols are grouped into syntactic elements according to the grammar. Furthermore, the syntactic integrity is also checked. In case there is no grammar rule available for the current terminal symbol, the syntactic analysis has failed, and a syntax error is reported.

According to the principle of syntax-directed parsing, during the syntactic analysis the semantic evaluation is performed. This may include constructing the abstract syntax tree (AST). In the AST only the relevant information for the code generation is contained. For each rule in the grammar, there may be semantic actions associated with it, that get executed when the rule is visited. The semantic actions has access to the attributes of the rule. This information is used to generate the AST.

Afterwards, the intermediate language code is analyzed and optimized. This may include optimizations such as inlining or loop unrolling. Depending on the use case more aggressive optimizations can also be performed.

Finally, the code generation unit takes the optimized code and generates the appropriate instructions for the target language.

Chapter 2

Methods and Tools for Compiler Frontends

In this chapter methods and tools for the construction of compiler frontends are explained. This explanation ~~is~~ focuse~~d~~ on the parser generator ANTLR. The basis for this chapter is the book "The Definitive ANTLR 4 Reference" by Parr (2013).

2.1 Attributed Grammars

Parser generators like ANTLR or Coco-2 require the definition of the grammar of the source language in a specific format. These formats also allow for the declaration of attributes and semantic actions in the grammar. Semantic actions have access to the attributes of symbols (terminal and non-terminal) of a rule. Some symbols have attributes associated with them. The combination of a grammar, attributes and semantic actions is called an attributed grammar. (ATG)

There are two types of attributes: inherited and synthesized attributes. The former ones are computed based on the attributes of the parent node. Synthesized attributes are based on the attributes of the children nodes. The type of attributes available depends on the parsing strategy. For a top-down strategy the attributes of child-nodes are not available, as they have not been parsed yet. Conversely, when using the bottom-up strategy, the attributes of parent nodes are not available.

Especially relevant are the attributes of terminal classes. Through the attribute of a terminal class like `number`, the actual number that this class node holds can be accessed. These kinds of attributes are provided by the lexical analyzer.

In listing 2.1 a simple attributed grammar for Coco-2 for arithmetic expressions is shown. This grammar uses semantic actions to calculate the result of an arithmetic expression. Semantic actions are encoded inside `SEM<< >>` blocks; in this case C# code. Synthesized attributes provide the results of the calculations from the child nodes. These attributes are available inside the semantic actions where the actual calculation is performed.

While it is convenient to embed semantic actions directly into the grammar, it is not without disadvantages. By embedding code of a specific language, it is no longer possible to use the same grammar to generate a parser in another implementation language. Parser generators like ANTLR provide multiple implementation languages to generate a parser for.

Listing 2.1: Attributed Grammar for Coco-2 for simple arithmetic expressions.

```
Expr<<out int e>> =     LOCAL<<int t = 0; e = 0;>>
  Term<<out e>>
  { '+' Term<<out t>>     SEM<<e = e + t;>>
  | '-' Term<<out t>>     SEM<<e = e - t;>>
  }.

Term<<out int t>> =     LOCAL<<int f = 0; t = 0;>>
  Fact<<out t>>
  { '*' Fact<<out f>>   SEM<<t = t * f;>>
  | '/' Fact<<out f>>   SEM<<t = t / f;>>
  }.

Fact<<out int f>> =     LOCAL <<f = 0;>>
    number<<out f>>
  | '(' Expr<<out f>> ')'.
```

2.2 ANTLR

In this section, the parser generator ANTLR (ANother Tool for Language Recognition) is explained. First, a general overview of the history of ANTLR is given, followed by the introduction of the parsing algorithm currently employed by ANTLR, namely ALL(*). Finally, the general functionality of ANTLR is explained.

2.2.1 History

"ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files". As the acronym of ANTLR states, it is a tool for language recognition. ANTLR was first released in 1992 and has since then been in continuous development. The original creator and maintainer of the project is Terence Parr. ANTLR is written in Java and is open sourced under the BSD license. Its source code can be viewed on GitHub[1].

Many projects utilize ANTLR. Notable examples include the Java Object-Relational Mapping tool Hibernate 2024 and the NoSQL database Apache Cassandra (2024).

ANTLR originally started ~~of~~ as the master thesis of Terrence Parr (Parr 1994). A first alpha release was created in 1990, that only generated LL(1) parsers. Version 1 of ANTLR incorporated the new parsing algorithm developed by Parr that allowed to create parsers for LL(k) grammars (Parr 1993). Version 2 then provided incremental improvements.

Version 3, released in 2006 introduced another a new parsing algorithm called LL(*) (Parr and Fisher 2011b). The LL(*) parsing strategy performs parsing decisions at parse-time with a dynamic lookahead. The number of lookahead tokens increases to an arbitrary amount and decreases again using backtracking. However, the maximum amount of k lookahead tokens still needs to be specified. Version 3 also introduced ANTLRWorks[2], a graphical IDE for the construction of ANTLR grammars.

[1] https://github.com/antlr/antlr4
[2] https://www.antlr3.org/works

The current version 4, released in 2013 again introduced a new parsing algorithm Adaptive-LL(*) or ALL(*). The most significant improvement of ALL(*) over LL(*) is that the maximum number of lookahead tokens no longer needs to be specified. ANTLR v4 added support for the visitor and listener patterns[3], enabling easier interaction with the syntax tree.

2.2.2 Parsing Algorithm Adaptive-LL(*)

The Adaptive-LL(*) or ALL(*) parsing strategy is introduced in the paper "Adaptive LL(*) parsing: the power of dynamic analysis" by Parr, Harwell, and Fisher (2014) and is the basis for this section. This parsing algorithm is used for ANTLR version 4. As the title suggests, ALL(*) performs the analysis of the grammar at parse time.

Limitations of LL(*) Parsing Algorithm

To understand the need for ALL(*), it is necessary to highlight why the previous strategy LL(*) is insufficient. LL(*), introduced by Parr and Fisher (2011a), was developed as an improvement to the existing general LL (GLL) (Scott and Johnstone 2010) and general LR (GLR) (Tomita and Ng 1991) parsers. For ambiguous grammars these parsers return multiple parse trees, which ~~are~~ undesirable for parsers of programming languages. GLL and GLR are designed for natural languages, which are inherently ambiguous. LL(*) overcomes these limitations by using regular expressions that are stored inside a deterministic finite automaton (DFA) to offer mostly deterministic parsing. Using the DFA allows for regular lookahead even though the grammar itself is context-free.

However, the LL(*) grammar condition cannot be checked statically, leading to the case that sometimes no regular expression is found that distinguishes the possible productions. Such situations are detected by ~~the~~ static analysis and then backtracking is used instead. Backtracking however comes with the disadvantage that for rules in the format $A \to a|ab$, the second alternative will never be matched, since backtracking always chooses the first alternative.

to remedy this

Dynamic Grammar Analysis with ALL(*)

With ANTLR version 4 the parsing strategy Adaptive-LL(*) or ALL(*) was introduced. The main difference to ANTLR version 3 is that the grammar analysis is now performed at parse-time, and is no longer static. This overcomes the limitations of the static analysis LL(*) performs and enables the generation of correct parsers for context-free grammars. The only exception are grammars that contain indirect or hidden left-recursion[4]. From an engineering perspective it was seen to be too much effort, since these grammars are deemed to be not common. Direct left-recursion is possible, because ANTLR rewrites the grammar to be non-direct left-recursive before passing it to the ALL(*) parsing algorithm.

[3] https://github.com/antlr/antlr4/blob/dev/doc/listeners.md

[4] Indirect left-recursion is a rule like $A \to Bx, B \to Ay$. ϵ productions cause hidden left-recursion. Take a rule $B \to \epsilon$ that produces only the empty chain ϵ and another rule $A \to BA$. Since B's only production is to ϵ the second rule causes a left-recursion.

At a decision point (a rule containing multiple alternatives), ALL(*) starts a subparser for each alternative in pseudo-parallel. A subparser tries to match the remaining input to the selected alternative. If the input does not match, the subparser dies off. All subparsers process one symbol at the time in pseudo-parallel. This guarantees that the correct alternative can be found with minimum lookahead. In the case of ambiguity due to multiple subparsers reaching the end of file or coalescing, the first alternative will be chosen.

The performance of ALL(*) is improved by employing a cache. This cache is implemented in the form of a DFA. The DFA stores the same information as the DFA generated by LL(*) from static analysis. After a lookahead, the DFA stores which production resulted from the lookahead phrase. If at a later time the same lookahead phrase is being processed, the correct production can be retrieved from the DFA. Theoretically, a DFA is not able to recognize a context-free grammar, however due to the analysis being performed at parse time, the analysis only needs to be performed on the remaining input. Since the remaining input is a subset of the context making it regular. Another optimization is the usage of a graph-structured stack (GSS). The GSS makes sure that during the prediction, no computation is performed twice, effectively acting as a cache.

The theoretical runtime complexity of ALL(*) is $O(n^4)$. This stems from the fact that in the worst case the ALL(*) parser needs to make a prediction for each symbol and each launched subparser then needs to inspect the entire remaining input. In practice ALL(*) performs linearly for common programming languages like Java or C#.

the syntax of

wille

2.2.3 Functionality

end

ANTLR generates a combined lexer and parser from a single grammar file. The generated parser is a recursive descent parser. ANTLR supports various implementation languages such as Java, C# or C++. The syntax used by the ANTLR grammar supports extended BNF (EBNF) operators such as ?. To interact with the generated parser, ANTLR optionally generates interfaces and implementations for the listener and visitor pattern.

ALL(*) does not use a separate lexical analysis phase. Instead lexical and syntactical analysis are integrated into a unified process. Lexical rules are treated as parser rules, therefore a separate lexical analysis phase is not needed. Since the phases are combined, it is possible for ALL(*) to perform context-sensitive lexing. The lexer can make a decision based on the current parsing context. The parsing is directly performed on the raw input stream and not on a separate token stream.

Was heisst das ???

Semantic Predicates

ANTLR supports the definition of so-called *semantic predicates*. Semantic predicates are boolean expressions, defined in the host language that allow for the dynamic alteration of the language generated by the grammar. If a semantic predicate is present for a production, the production can only be accepted if the semantic predicate evaluates to true. Semantic predicates are expressed inside { } parenthesis followed by ?. Listing 2.2 ~~illustrates~~ an example use case of a semantic predicate. The rule `blockEnd` contains a semantic predicate specifying that the production can only be accepted if there is ~~currently~~ a block on the stack. still

A semantic predicate also has access to the current token. This enables conditions

shows

ELSE???

Listing 2.2: Example grammar using a semantic predicate and a semantic action.

```
grammar Example;
@members {
    java.util.Stack<String> blockStack = new java.util.Stack<>();
}

program: statement* EOF;

statement
    : blockStart
    | blockEnd
    | otherStatement
    ;

blockStart: 'begin' { blockStack.push("block"); };

blockEnd: 'end' { !blockStack.isEmpty() }? { blockStack.pop(); };

otherStatement: 'print' IDENTIFIER;

IDENTIFIER: [a-zA-Z_][a-zA-Z_0-9]*;

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

that directly interact with the input. For example separate productions for even and uneven numbers could be used. The semantic predicate then checks if the number is even or not.

have used

Semantic Actions

In ANTLR grammars, semantic actions can be ~~defined~~. Semantic actions can be inserted in every parser rule, before, in between and after symbols. Similar to semantic predicates, the semantic actions ~~is~~ to be defined in the implementation language. Semantic actions are defined inside { } parenthesis. To access a symbol, the name of the symbol prefixed by $ can be used. In Listing 2.2 semantic actions are used in the blockStart blockEnd rules. For the blockEnd it has to be noted that semantic predicates and actions can be used together.

Alternative Labels for Rule Alternatives

Per default, ANTLR generates one method for each rule. In the case of multiple alternatives for a rule, the handling of the alternatives would need to be done manually. Therefore, ANTLR offers the possibility to attach a label to each of the alternatives. Then, a method for each alternative will be generated separately. One use case is highlighted in listing 2.3. The rule type matches ~~to~~ either one of the four types. Each alternative has a label associated to it by using # as the prefix for the alternative name. With this definition, four methods will be generated corresponding to each of the alternatives as explained above.

Listing 2.3: Example rule using alternative labels for the rule alternatives.

```
type:
    : 'int'      #IntType
    | 'bool'     #BoolType
    | 'long'     #LongType
    | 'string'   #StringType
    ;
```

2.3 Syntax Tree and Abstract Syntax Tree (AST)

A syntax tree is a hierarchical representation of the syntactical structure of a sentence. Also referred to as a parse tree, this representation is usually generated by a parser. A syntax tree contains the information of the entire sentence based on the grammar of that language. Each inner node in the syntax tree corresponds to a rule in the grammar. The leaf nodes represent terminal symbols and inner nodes are non-terminal symbols. Concatenating the leaf nodes from left to right of the syntax tree results in the original sentence from which the syntax tree was constructed from.

Listing 2.4 shows the syntax tree of the arithmetic expression `5 * 3 + 7` based on the grammar in 2.1. This syntax tree contains the non-terminal symbol **`Expression`**, **`Term`**, **`Fact`** and the terminal class **`number`**. The expression is built from two terms and one operator. The left term represents a multiplication consisting of two factors and an operator. All factors in the syntax tree contain the terminal class **`number`** which hold the concrete numbers. This structure further ~~enables the~~ representation of the precedence rules of arithmetic operations directly in the syntax tree.

2.3.1 Abstract Syntax Tree (AST)

constructed and

In an abstract syntax tree (AST) only a subset of the nodes from the original syntax tree are included. The goal is to focus on the semantic aspects of the syntax tree. Syntactical details, e.g., semicolons are omitted.

The generation of an AST from a syntax tree can be done in multiple ways. One method is to generate the AST during the parse, which increases performance since the syntax tree does not need to be traversed ~~twice~~. This can be implemented by using an attributed grammar with semantic actions that embed the AST generation code directly into the parser. Parsers like ANTLR also support the listener pattern to execute code during the parse. Alternatively, the AST can be generated after the parse phase from the syntax tree. To traverse the syntax tree, the visitor pattern can be used for example.

The AST is then used in the subsequent stages of a compiler. This transformation is performed to create a new tree which omits all information that is of no importance to the following stages of the compiler. Subsequent code optimization may further slim down the AST.

Continuing with the previous example, listing 2.5 shows the AST of the arithmetic expression `5 * 3 + 7`. This AST still contains the same semantic meaning as the full syntax tree, however it needs fewer nodes for that. Instead of using the non-terminal symbols, the operator is used, effectively encoding the same information. In this example,

Listing 2.4: Syntax tree of the arithmetic expression `5 * 3 + 7` based on the grammar in listing 2.1.

```
                Expr
             /   |   \
          Term   +   Term
       /   |   \       |
    Fact   *   Fact   Fact
     |          |       |
   number     number  number
     |          |       |
     5          3       7
```

Listing 2.5: Abstract syntax tree of the arithmetic expression `5 * 3 + 7`.

```
        +
       / \
      *   7
     / \
    3   5
```

the node count can be reduced from 14 to 5.

2.4 Visitor-Pattern for Tree Transformation

In the case that a syntax tree ~~is~~ has already ~~present~~ been built, the visitor-pattern can be used to create an AST from the syntax tree. Using the visitor-pattern, the syntax tree gets traversed and then step by step the AST is constructed. The visitor-pattern allows for the separation of algorithms from the objects they operate on. Instead of including the ~~code for~~ the generation of an AST object in the syntax tree object, a separate object, a so-called ~~*visitor*~~ ~~is~~ tak~~ing~~es care of this. Welche zwei? for

To implement visitor-pattern for a syntax tree, the best approach is to use interfaces or abstract classes for the nodes of the syntax tree and the visitors. Listing 2.6 shows a possible implementation for an interface and abstract class in Kotlin. This code is based on the syntax tree shown in listing 2.4. Each class ~~of~~ in the syntax tree implements the abstract class `SyntaxTreeNode`. For the visitor class, the `SyntaxTreeVisitor` interface needs to be implemented. Both classes are generic. This allows the implementation of the visitor to use an arbitrary type as a return value. Multiple visitors can then be implemented using the generics, so that each visitor can return one type of the AST types. for ? In this case it is helpful to create an abstract base visitor that provides an empty implementation for all interface's methods. Then the concrete visitor only needs to override the methods that are relevant for the specific AST type.

A `SyntaxTreeNode` can then be visited by calling its `accept` method. Inside the `accept` method, the appropriate method of the `SyntaxTreeVisitor` will be called. As can be seen in listing 2.7 the `NumberNode` calls the `visitNumberNode` with itself as the parameter. This behavior is analogous for all other nodes of the syntax tree.

1. 2.

Listing 2.6: Interface and abstract class used to implement the visitor-pattern.

2.
```
sealed class SyntaxTreeNode {
    abstract fun <T> accept(visitor: SyntaxTreeVisitor<T>): T
}
```
1.
```
interface SyntaxTreeVisitor<T> {
    fun visitNumberNode(node: NumberNode): T
    fun visitOperatorNode(node: OperatorNode): T
}
```

Listing 2.7: Implementation of the `NumberNode` class inheriting from the `SyntaxTreeNode`.

```
data class NumberNode(val value: Int) : SyntaxTreeNode() {
    override fun <T> accept(visitor: SyntaxTreeVisitor<T>): <T> {
        return visitor.visitNumberNode(this)
    }
}
```

Listing 2.8: Implementation of the `ExpressionListener` interface.

```
interface ExpressionListener {
    fun enterExpr(node: Expression)
    fun exitExpr(node: Expression)

    fun enterNumer(number: Number)
    fun exitNumber(number: Number)
}
```

b

Term, Fact ?

2 NODE ?

2.5 Listener-Pattern for Tree Transformation

The listener-pattern is used for *listening* to events or notifications from another object. In the context of parsing, the listener-pattern is used to handle parse events coming from the parser. This includes events such as entering and exiting a node during the parse. When using the listener-pattern the parse tree is only traversed once. This is because the events get pushed to the listeners during the parse.

To implement the listener-pattern for the construction of an AST, a listener interface is needed. The listener interface contains method declarations for entering and exiting each node type. The methods take the syntax tree node as the input parameter. A possible implementation for the listener interface is shown in Listing 2.8. In case of the *enter* methods, the symbols for the syntax tree node have not been parsed yet, so no data from them is available yet.

A concrete listener will then implement the interface and register/subscribe itself ~~to~~ für the events of the parser. When the parser enters or exits a node during parse, it will call the respective method with the parsed syntax tree node for all listeners.

Kurze Zusammenfassung und Überleitung in das nächste Kapitel

Chapter 3

Java Virtual Machine (JVM)

used bytecode

This chapter focuses on the Java Virtual Machine (JVM). First the foundation and history of the JVM will be explained. Further focus is put on JVM itself and its functionality. In the ~~following~~ section the ~~language~~ of the JVM ~~*bytecode*~~ is introduced. Finally, the bytecode manipulation tool ObjectWeb ASM is highlighted. This chapter is based on the specification of the JVM provided by Oracle (2024).

(static Web pages)

3.1 History

JVM

As the name suggests, the ~~Java Virtual Machine~~ is the virtual machine used to execute Java programs. In 1994 Sun Microsystems Inc. developed the JVM because of ~~their~~ the requirement for Java to be platform and operating system independent. By using a virtual machine as an intermediary, Sun was able to move the multiplatform aspect away from the compiler.

One of the original use cases for Java and therefore the JVM was embedding of so-called applets in browsers. Applets were used in addition to the HTML document format, which at that time only provided limited functionality. Similar to HTML, the applets were platform independent, which eased the development for the website creators. The first browser incorporating applets was HotJava.

Java was originally closed source, however in 2006 Sun ~~Microsystems Inc~~. began work on open sourcing the Java compiler and the JVM under the OpenJDK project (Sun 2006). On November 13, 2006 the JVM implementation developed by Sun, called HotSpot was open sourced under the GPL license. the

The version of the JVM specification is tied to the Java version, but for the `class` files, a separate version number so-called *class file format version* is used. For the initial JDK release 1.0, the class file format version 45 was used. they were

Various companies and organizations provide implementations of the JVM. For example, GraalVM is an implementation of the JVM with the ability to perform ahead-of-time (AOT) compilation for ~~a~~ Java programs. While this increases the performance of the applications, ~~it~~ they can only be executed on the platform ~~it was~~ compiled for. Another example is picoJava, which is a processor specification with the goal of enabling native execution of bytecode for embedded systems (McGhan and O'Connor 1998). Puffitsch and Schoeberl (2007) presented an implementation of picoJava on an FPGA.

#[...]

#[...] ok

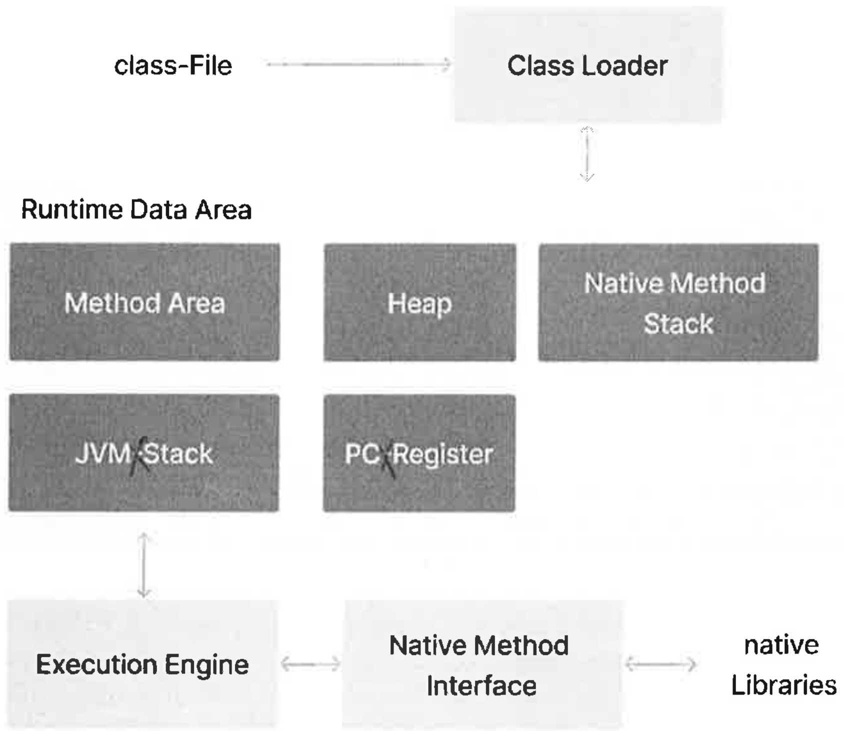

Figure 3.1: Architecture of the JVM.

3.2 Architecture

The basic task of the JVM is to read a `class` file and execute the bytecode instructions contained in it. The specification defines only the abstract machine. How the bytecode is executed on the actual physical processor, or what optimizations are to be performed, is up to the implementer of the JVM specification. An official reference implementation of the JVM called OpenJDK[1] is provided by Oracle. The JVM architecture can be seen in figure 3.1. It consists of the following elements:

- **Class Loader**: Loads `class` files into memory.
- **Runtime Data Area**: Manages all runtime memory used in the JVM.
- **Execution Engine**: Executes bytecode instructions.
- **Native Method Interface**: Interfaces with the native host system.

[1] https://openjdk.org/

3.2.1 class File Format

A `class` file contains the necessary information that is needed to execute a program on the JVM. One `class` file contains the definition of either a single class, interface or module. A `class` file is structured as follows:

- Magic Number,
- Version Info,
- Constant Pool,
- Access Flags,
- This Class,
- Super Class,
- Interfaces,
- Fields,
- Methods and
- Attributes.

Wieso alles in Großbuchstaben?

At the beginning of the `class` file is the magic number. It is responsible for identifying a `class` file. The magic number is the same for every `class` file. Next ~~is~~ comes the version of the `class` file format. The JVM uses the version to determine if the `class` file is compatible with it.

The *constant pool* acts as a storage for all constants and symbolic references contained in the file. For example the reference to a method or a string literal. An entry in the constant pool consists of a tag which specifies one of 17 constant types, followed by information describing the constant. Depending on the type, the length of the constant may change. A string literal for example ~~would~~ requires more memory than an integer.

The *access flag* entry is a flag mask that defines the permissions and properties of the class or interface. Possible flags include for example, whether the class is `public`, `final`, `abstract` or not.

The *this class* entry contains the name of the current class in the form of an index to an entry in the constant pool. Analogous the *super class* entry defines the name of the superclass of this class. In case that the class does not inherit from a superclass, the index is zero. The following section lists all implemented interfaces, again as a list of indexes in the constant pool.

In the *fields* section all member fields of the class are listed. Each field description consists of four elements: Access flags, similar to the class level access flags, e.g., `public` or `private`; an index to the name of the field in the constant pool; an index to the descriptor (type specification) of the field in the constant pool; and finally, the entry can have optional attributes associated with it, e.g., the constant value (for static fields). An entry in the *methods* section contains the same values, only the descriptor is used to describe the method signature (parameter and return type).

Finally, in the *attributes* section, additional metadata of the class is stored. Most importantly, this section contains the bytecode for each method of the class. Other information includes for example, a list of exceptions thrown by each method, the name of the source file or a mapping from bytecode instructions to source code line numbers.

3.2.2 Class Loader

The class loader takes care of loading bytecode into the JVM memory. There are three tiers of class loaders:

- **Bootstrap**: Loads JDK internal classes and core libraries. Implemented in native code and not accessible by an application.
- **Extension**: Loads extensions of the standard Java classes from the JDK extensions directory.
- **Application**: Loads all a application level classes. These are located via the classpath CLASSPATH. Classes can be put on ~~the~~ classpath by using an environment variable or command line option. (-cp for short)

then

The class loaders are organized in a parent-child hierarchy. The Bootstrap class loader is the parent of the Extension class loader, which itself is the parent of the Application class loader. When a request is made to load a `class` file, the class loader first delegates the request to its parent class loader. Only if the parent class loader cannot locate the class the current class loader will attempt to load it. This process is performed so that no two class loaders attempt to load the same class. The loading process is separated into three stages. Loading, linking and initialization.

In the loading stage the bytecode is loaded into the JVM. The bytecode can be loaded from a file, the network or another source. The bytecode is loaded into the JVM as a `Class` object.

expression's type these , e.g.

In the second stage linking is performed. This stage is separated into the three substages verification, preparation and resolution. Verification is performed to ensure that the loaded bytecode adheres to the JVM's rules. Rules include ~~such as requiring~~ that a return instruction must match its method's return type, or that a `throw` instruction must only throw values that are instances or subclasses of `Throwable`. Verification is performed because the JVM must guarantee that only correct `class` files are executed and no exploitation through malicious bytecode is taking place. The second substage preparation creates the static fields of a class or interface and allocates the memory needed for them. The static fields further are assigned their respective default values. Explicit initializers are executed during initialization, during preparation no bytecode is executed. Resolution then resolves all symbolic references inside the class. Symbolic references are used for example when referencing another class or interface. For each symbolic reference resolution determines a concrete value.

~2a objects ~2b ? (~2c

in the linking stage.

If linking has been successful the class or interface is initialized. Explicit initializer s of static fields are executed as are static initializer blocks of the class or interface.

3.2.3 Runtime Data Areas

purpose

The runtime data areas of the JVM are regions of memory used during the execution of a program. Each memory area serves a specific ~~use case~~. Some memory areas are specific to a thread. They get created when a thread is ~~created~~ and cleaned up on thread termination. Others are alive for the entire duration of the JVM's runtime.

? instantiated

PC Register

The `program counter` or PC register is the memory area which contains the bytecode instruction that is currently being executed. Each thread inside the JVM has its own PC register. In the case that a native method is executed, the PC register's value is undefined.

JVM Stacks

A JVM Stack is a thread-specific memory area that is created in tandem with the thread. A stack in the JVM is similar to a stack in languages such as C. An instance of a stack stores *frames* for method-calls. On method invocation, a new frame is created. Conversely, when the method invocation is finished, the frame is destroyed.

A frame contains information related to a single method invocation. This includes the following:

- Local variables and method parameters,
- Operand stack,
- Reference to constant pool of the method's class and
- Return address

The operand stack is used for intermediate calculations and storing results from other method invocations. The reference to the constant pool is needed to resolve the targets for method calls and field accesses. The return address stores the address of the calling method. Once the method invocation has completed control will be returned to this address.

Heap

for

In the *heap*, all object instances and arrays are stored. The heap is shared across all threads and is created on JVM startup. Contrary to programming languages like C, it is not possible in the JVM to manually reclaim/free the memory allocated ~~by~~ an object or array. Instead, the JVM utilizes an automatic storage management system known as a *garbage collector*. The garbage collector automatically reclaims memory from objects and arrays that are no longer referenced by any other object or variable ~~in the program~~. The JVM specification does not require a specific garbage collector algorithm, rather the implementer can choose which algorithm to use or also allow the user to select the algorithm. While the garbage collector automatically reclaims memory, it is also possible to manually request a cleanup through an API. There is however no requirement for the garbage collector to honor this request, so it may be ignored.

Method Area

The method area is a section of the memory that is available to all threads inside the JVM and is created on JVM startup. It stores metadata of the classes loaded into the JVM. This includes the runtime constant pool, field and method data and the bytecode for methods and constructors.

Native Method Stacks

originally

Similar to JVM stacks *native method* stacks are associated with a method invocation and store information relevant to that invocation. However, in this case the invoked method is executed natively on the host system. Instead of bytecode, native code, written in e.g. C, is executed. The native method stack serves as an interface between the native code and the bytecode inside the JVM.

3.2.4 Execution Engine

The JVM's execution engine is responsible for executing the bytecode contained in the loaded `class` files. It takes bytecode instructions and transforms them into something the host system can execute. This may be through interpretation or just-in-time (JIT) compilation. The JVM specification does not specify how the bytecode is executed on the host system. Therefore, in this section the execution engine *HotSpot* of the JVM reference implementation OpenJDK (2025) is explained.

The HotSpot execution engine consists of two main parts: The interpreter and the JIT Compiler. For memory management the execution engine is supported by the garbage collector, that automatically reclaims memory from unused objects and arrays. The java native interface (JNI) enables the JVM to call and execute code and libraries written in other languages like C or C++.

Interpreter

The interpreter reads bytecode instructions sequentially and translates them to target code the host system can execute. This allows the JVM to start executing bytecode right away, without having to wait for any JIT compilation to be performed. In comparison, .NETs' Common Language Runtime (CLR) performs a JIT compilation of a methods' code as soon as it is first invoked(Microsoft 2025). ???

HotSpot uses a template-based interpreter. On JVM startup HotSpot creates an interpreter based on the data in the so called `TemplateTable`. The `TemplateTable` contains information on the assembly code corresponding to each bytecode instruction. A template in this case is a description of a bytecode. The generated templates are specific to the host operating system and architecture. The interpreter fetches the template corresponding to the current bytecode instruction and executes it. The template is fetched by using an accessor function provided by the `TemplateTable`. This approach leads to higher performance than using a switch-statement, which may have to compare the current instruction with all cases to find the correct code to execute. A downside of this approach is the need for extra platform and operating system specific code needed for the dynamic code generation. Some operations, like a lookup in the constant pool, are still performed via the JVM runtime, since they are too complicated to be implemented in assembly code directly.

Initially, all code on the JVM is interpreted. The runtime performs adaptive optimization by monitoring the code execution for methods that are executed often, so-called *hotspots*. For those hotspots the runtime performs optimization. Specifically a method detected as a hotspot will be just-in-time compiled, so that it can be natively executed on the host system.

Listing 3.1: Declaration of a native method in Java.

```
public class Example {
    public native void nativeMethod();
}
```

Just-In-Time Compilation (JIT) (compilation)

To increase performance, the JVM runtime employs just-in-time (JIT) compilation. Contrary to ahead-of-time compilation, which translates the code before the execution, JIT compilation translates the code during the execution of the program. Because the compilation is performed while the program is executing, considerations need to be made about the performance implication of the compilation. Therefore, the JVM uses a two stage tiered compilation: The C1 or *client* compiler and the C2 or *server* compiler.

Through profiling the JVM runtime identifies hotspots, also referred to as *hot methods*. These are methods that are executed often. Methods that are only called rarely are referred to as *cold methods*. The JIT compiler focuses only on hot methods for multiple reasons: Compiling bytecode to native code takes up processor time that cannot be used for the actual execution of the program. Furthermore, the compiled code needs to be stored in memory and thus completely compiling bigger programs to native code make y take up a significant amount of memory. Only compiling hot methods strikes a balance between performance and memory consumption. Also, empirically programs spend most of their execution time on a small amount of the entire codebase.

Once a method has been identified for compilation, the first JIT compiler C1 compiles the method to native code. The C1 compiler prioritizes compilation speed and therefore only performs basic optimizations. After compilation the methods' body is replaced by the compiled code, leading to the method being executed natively and no longer interpreted. During compilation code used for profiling is also added. The profiling information is used for the second stage of the JIT compilation.

When a method that was compiled with C1 passes an execution threshold, the C2 compiler will compile the method again. This time the focus is on performing aggressive optimizations for maximum performance, which consumes more times than the first compilation. The C2 compiler uses the information gained through profiling to perform optimizations that lead to the best performance. This may include optimization techniques such as loop unrolling or inlining. The C2 compiler does not add any code for profiling which further improves performance. In some cases the assumptions that the C2 compiler made based on the profiling data can turn out to be wrong, which in turn can lead to the method being returned to the C1 compilation level.

Java Native Interface (JNI) (native)

The Java Native Interface (JNI) is an API that enables the code executed inside the JVM to interoperate with applications and libraries that are written in other languages. This API is necessary because there are cases when the entirety of the application cannot be implemented inside the JVM. For example there might be libraries only available in C/C++, but not for the JVM. The JNI then allows calling those libraries from within the JVM. Listing 3.1 shows the declaration of a native method in Java. The `native` keyword signalizes to the JVM that the implementation of the method will be provided in native code.

The JNI makes it possible to create, inspect and update JVM objects. For that, the JNI provides a type mapping between the JVM types and native equivalents. Further, methods located inside the JVM can be called or exceptions thrown from within native code.

Using the JNI inside an application however limits the number of systems it can be executed on. The native part of the application needs to be compiled for every architecture and operating system the application is intended to run on. Native methods manually manage the memory they have allocated and therefore programming errors can lead to memory leaks within the application.

3.3 Bytecode

the

Bytecode is the instruction set of the JVM. It serves as an intermediate language between high level languages such as Java or Kotlin, and low level languages such as assembly which can be natively executed on a CPU. High level languages only need to target bytecode to be cross-platform. As long as a JVM implementation is present for a given architecture and operating system, the bytecode can be executed without needing to be compiled again.

consists of

3.3.1 Structure

In terms of code execution, JVM is organized as a stack machine with registers. Each method being executed is structured as a frame containing an operand stack and local variables, which can be seen as registers. The operand stack and number of variables inside a frame are each able to contain up to 65,535 entries. code

A bytecode instruction ~~is structured as~~ a one byte long opcode followed by zero or more one byte long operands. The maximum possible number of opcodes is therefore 256. Most of them are in use, while some are reserved for internal and future use. Each instruction has a mnemonic associated with it. Instructions that can operate on multiple types are prefixed by the concrete type they are operating on. For example the instruction for adding together two integers is known by the mnemonic `iadd`. The following types are supported in bytecode:

- `boolean`,
- `byte`,
- `char`,
- `short`,
- `int`,
- `float`,
- `reference`,
- `returnAddress`,
- `long` and
- `double`

Most instructions for the types **byte**, **char** and **short** and all for **boolean** are internally converted to **int**, therefore in these cases the **int** based instructions are used instead.

The **reference** type is analogous to pointer types in languages like C. It is type-safe and managed by the JVM. The JVM also keeps track of the references for garbage collection purposes. If there are no references anymore pointing to an object, the garbage collector can reclaim the memory it occupied. The **returnAddress** type represents pointers to opcodes of JVM instructions. This type is only used internally and is not accessible otherwise.

3.3.2 Categories of Instructions

The instructions in the bytecode instruction set can be categorized depending on their functionality. Instructions from each category work together to perform more complex actions.

and loads this

Load and Store Instructions

reference

Load and store instructions allow the loading of values onto the operand stack and storing values from it into variables. These instructions function within the frame of a method. Load instructions like **iload** or **aload** load an integer or array respectively onto the operand stack. To load a constant onto the operand stack instructions like **bipush** and **ldc** can be used. **ldc** loads a constant from the constant pool of the class while **bipush** takes one operand (the constant value), ~~that is loaded~~ onto the operand stack. When a value from the operand stack is to be stored into a variable instructions like **istore** or **astore** are available.

Arithmetic Instructions

&&

Arithmetic instructions perform calculations using the values on the operand stack. The result of the calculation is then put on the operand stack. There are separate instructions for integer and floating point calculations, e.g. **iadd** and **fadd** for integer and floating point additions respectively. In the case of an over or underflow no exception is thrown. The bytecode instruction set further includes instructions for bitwise logical operations like AND (**iand**).

Type Conversion Instructions

E.g.,

Type conversion instructions make it possible to change the type of a numeric value. They can be used to perform ~~an~~ explicit conversion. The JVM supports widening conversions (e.g. from **int** to **long**; **i2l**) and narrowing conversions (e.g. from **float** to **int**; **f2i**). For some conversions there may be a loss of information. Widening conversions like from **int** to **float** can lose some of the least significant bits of the source value.

Object related Instructions

The bytecode instruction set contains separate instructions for class instances and arrays, even though they are both considered as objects by the JVM. To create a class instance element

the instruction **new** is used, while for an array **newarray** is used. To access class fields the **getfield** instruction can be used for instance variables and **getstatic** for class variables. When loading an ~~entry~~ from an array, there is a separate instruction for each type, e.g. **iaload** for loading an integer from an array. For storing a value inside a class instance or an array, analogous instructions are available.

Operand Stack Management Instructions

In some cases it is beneficial to perform manipulations on the operand stack directly. For example, to implement peephole optimization, it is necessary to duplicate a value on the operand stack as an alternative to loading a variable two times (McKeeman 1965). The instruction for that is **dup**. The instruction set further provides instructions like **pop** or **swap**.

Control Transfer Instructions

By using control transfer instructions, the execution path can be changed to continue with an instruction other than the one after the control transfer instruction. Labels are used to specify where the execution should continue. There are three kinds of control transfer instructions ~~Available~~:

- **Conditional Branch**: If a condition is met, the execution jumps to the instruction specified by a label. The label is provided as an operand to the conditional instruction. If the condition is not met, the following instruction is executed. For example, the instruction `ifnull` checks whether a value on the operand stack equals null or not.
- **Compound Conditional Branch**: The instructions `tableswitch` and `lookupswitch` allow for multi-way branching, which are available as `switch` statements in Java. Depending on the case values one of those two instructions is used. `tableswitch` is optimized for dense case values, while `lookupswitch` is preferred for sparse ones.
- **Unconditional Branch**: Instructions like `goto` are used to jump to a label that may be before or after the instruction itself. Welche anderen? (mit Label)

Method Invocation and Return Instructions

For method invocation, the instruction set offers different instructions based on the type of method that should be invoked. For example, for regular instance methods the `invokevirtual` instruction can be used. Return instructions of methods are distinguished by their type. Each instruction is prefixed by its type, e.g., for `int` it is `ireturn`. In the case of a `void` return type the instruction is `return`.

Exception Throwing

The JVM supports programmatically raising exceptions and thus returning control to the caller of the method to deal with the exception. In the bytecode instruction set this is realized by the `athrow` instruction.

initialisation !

Listing 3.2: Instantiation of two variables and a third one with the sum of the first two variables in Java.

```
public static void main(String[] args) {
    int a = 6;
    int b = 10;
    int c = a + b;
}
```

Listing 3.3: Bytecode of the Java program shown in listing 3.2.

```
 0: bipush       6
 2: istore_1
 3: bipush       10
 5: istore_2
 6: iload_1
 7: iload_2
 8: iadd
 9: istore_3
10: return
```

3.3.3 Sample program

a simple used

To illustrate the functionality of bytecode and how computations on the operand stack work, ~~the following~~ example is ~~shown~~: Listing 3.2 shows the Java code for the ~~instantiation~~ initialisation of two variables a and b, followed by the ~~instantiation~~ of a third variable c which is initialized with the sum of a and b.

Compiling this small Java program, leads to the bytecode shown in listing 3.3. To init. ~~instantiate~~ the integer variable a with the value 6, the value must first be pushed onto the operand stack. This is done via the `bipush` instruction with the value as the operand. Bytecode indexes local variables starting with zero. The first index is used for the args parameter. Therefore, variable a uses index 1. The instruction `istore_1` pops the top value from the operand stack and stores it in the local variable with index 1. For variable b the process is analogous, but using index 2. To store the sum of the first two variable into variable c the addition must first be performed. Using the `iload` instruction, both values are put onto the operand stack. The `iadd` instruction then pops the top two entries from the operand stack and pushes the sum of the addition onto it. The sum is then stored in variable c with index 3. Finally, the `return` instruction completes the execution of this method.

3.4 ObjectWeb ASM

The basis for the following section is the official Website of the ObjectWeb ASM project (ASM 2024). ASM is a Java-based framework for interacting with Java Bytecode. ASM can be used to inspect, manipulate and generate bytecode. ASM was first released in 2000 by Eric Bruneton. At that time, ASM included just a code generator. Since 2002, the project is open source and the source code can be viewed on the project's gitlab[2]

[2] https://gitlab.ow2.org/asm/asm

Welcher Baum ist hier gemeint? (Arten von Knoten)

instance.

ASM is used in various projects to take care of the bytecode interaction. For example, ~~the~~ OpenJDK uses ASM to generate lambda call sites (Oracle 2014). ASM is also used in in compilers that produce Java bytecode like the Kotlin compiler (Kotlin 2024).

ASM offers a visitor-pattern based API for generating bytecode. There is also the possibility to construct an explicit tree via the Tree API. The Tree API is however just a wrapper around the already existing visitor-pattern. The visitor-pattern offers the advantage of not having to construct an explicit tree, which makes writing code to generate the bytecode more flexible. ASM operates directly on the bytecode which enables its efficiency.

3.4.1 Functionality

ASM's visitor-pattern based API can be used to read, manipulate and create bytecode. The API offers four main classes that work together to enable this functionality: `ClassReader`, `ClassWriter`, `ClassVisitor` and `MethodVisitor`.

ClassReader

The task of the ClassReader is to read and parse a `class` file, either from an input stream or directly from a byte array. The ClassReader extracts relevant information like the class name, implemented interfaces and methods. To interface with the parsed data, the ClassReader uses the visitor-pattern and accepts a `ClassVisitor`.

ClassVisitor

which

is an

The ClassVisitor makes it possible to inspect and manipulate a class. The ClassVisitor itself is an abstract class that provides abstract methods for visiting all structural elements of a class like fields or methods. For more complex structures like fields, an auxiliary visitor class is used. For example, when visiting an annotation the ClassVisitor's `visitAnnotation` method returns an `AnnotationVisitor`. This again-abstract visitor class ~~then~~ provides methods for visiting all structures contained in an annotation like it's name.

MethodVisitor

The MethodVisitor enables the inspection and manipulation of ~~class~~ methods. This is the visitor responsible for ~~actually~~ interfacing and modifying the bytecode of a method. The MethodVisitor is not an abstract class, in contrast to the ClassVisitor. Depending on the type of bytecode instruction, different visitor methods are available. For instructions like `istore` the visitor method `visitVarInsn` is used. When programmatically adding bytecode instructions to a method, the appropriate method of the visitor needs to be invoked. Listing 3.4 shows the ASM code for the bytecode generation representing the Java code shown in listing 3.2. The `mv` field represents the MethodVisitor instance used for invoking the visitor methods.

Listing 3.4: ASM code for bytecode generation of the Java program shown in listing 3.2.

```
mv.visitIntInsn(Opcodes.BIPUSH, 6);
mv.visitVarInsn(Opcodes.ISTORE, 1);

mv.visitIntInsn(Opcodes.BIPUSH, 10);
mv.visitVarInsn(Opcodes.ISTORE, 2);


mv.visitVarInsn(Opcodes.ILOAD, 1);
mv.visitVarInsn(Opcodes.ILOAD, 2);
mv.visitInsn(Opcodes.IADD);
mv.visitVarInsn(Opcodes.ISTORE, 3);

mv.visitInsn(Opcodes.RETURN);
```

kurze Zusammenfassung
und Überleitung in
das nächste Kapitel

Chapter 4

of the

Implementation Frontend

via the for mini C++

This chapter explains the implementation of the frontend of the compiler. First the additional technologies that are used ~~in~~ the development of the frontend are listed. Then the AST and symboltable are explained. In the following section the ANTLR grammar is shown. Based on this grammar three implementations for the AST transformation are explained: Visitor-pattern, listener-pattern and ~~via~~ an attributed grammar (ATG).

the

4.1 Used Technologies

In this section the additional technologies used for the development of the frontend are explained. This includes chosen the programming language and the additional libraries used.

4.1.1 Kotlin

Java has been

For the implementation the programming language Kotlin ~~is~~ chosen. Kotlin can be compiled to bytecode, which makes it possible to use Java libraries in Kotlin code. Kotlin has advantages over Java in some aspects. For example, null safety[1] is implemented into the language via explicit nullability within it's type system. This requires the caller of a field to explicitly handle nullable fields and thus reduces the risk of a null reference exception. ???

4.1.2 AspectJ

(AOP)

The compiler frontend utilizes AspectJ in it's handling of semantic errors. AspectJ is a library that enables aspect-oriented programming in Java. ~~Aspect oriented programming~~ AOP makes it possible to handle cross-cutting concerns in a central place without having to modify code in other areas. It can be used for compile time and runtime weaving of cross-cutting concerns. In the compiler frontend, runtime weaving using annotations is used.

[1]https://kotlinlang.org/docs/null-safety.html

4.1.3 ANTLR Preview Plugin

provides

During development with the JetBrains IntelliJ IDE, the ANTLR preview plugin[2] is used. The plugin developed by the ANTLR creator Terrance Parr, offers various features that enhance the process of creating and working with ANTLR. When developing an ANTLR grammar, syntax highlighting and checking for syntactic and semantic errors is provided. The included navigation window inside the IDE further enables testing of the grammar, without having to generate the combined lexer and parser manually first. To generate the combined lexer and parser the plugin ~~includes~~ a tool window which includes common configuration settings.

4.2 ANTLR Grammar

The ANTLR grammar is based on the MiniC++ grammar in EBNF-form. This grammar is transformed into the ANTLR grammar syntax. ANTLR grammars are stored in~~side~~ `.g4` files. Each rule in~~side~~ the grammar is delimited by a semicolon.

4.2.1 Header Section

At the top of the grammar file, the name of the grammar is specified. In this case `minicpp`. In this section options can also be specified, like the implementation language of the lexer and parser or the package/namespace of the generated code. These and other options can also be specified via command line options during the generation. In this case the necessary options are specified in the tool window of the ANTLR preview plugin.

decimal

4.2.2 Terminal Classes and Comments

for

The grammar contains three terminal classes shown in listing 4.1. The `IDENT` terminal class is used for all identifiers and requires them to start with a letter followed by an arbitrary number of letters and digits. For integer numbers the `INT` terminal class specifies a sequence of one or more digits. Signs are handled in the parser rules. Strings are defined as a sequence of characters starting and ending with double quotes. All characters except the special characters for line ~~end~~ are allowed. The comment and whitespace handling is performed by the `WS`, `LINE_COMMENT` and `BLOCK_COMMENT` lexical rules. These are special rules that when matched tell the parser to skip them and therefore not include them in the syntax tree.

end of Element

4.2.3 Root

The top rules of the grammar are shown in listing 4.2. The root rule `miniCpp` contains zero or more elements of the rule `miniCppEntry` followed by the lexical rule `EOF`. `EOF` is a default lexical rule provided by ANTLR signaling the end of the file. `miniCppEntry` defines the elements that can be used at the top level of the miniC++ source file. These are variable and constant definitions ~~and~~ function declarations and definitions. `SEM` is the lexical rule defining a semicolon.

[2] https://plugins.jetbrains.com/plugin/7358-antlr-v4

white space and comments

Listing 4.1: Terminal classes of the MiniC++ ANTLR grammar.

```
IDENT:          [a-zA-Z_][a-zA-Z_0-9]*;
INT:            [0-9]+;
STRING:         '"' (~[\r\n"] | '""')* '"';
WS:             [ \t\n\r]+ -> skip;
LINE_COMMENT:   '//' ~[\r\n]* -> skip;
BLOCK_COMMENT:  '/*' .*? '*/' -> skip;
```

Element

Listing 4.2: Top rules of the MiniC++ ANTLR grammar.

```
miniCpp:        (miniCppEntry)* EOF;
miniCppEntry:     constDef
                | varDef
                | funcDecl
                | funcDef
                | SEM
                ;
```

Listing 4.3: Variable and constant defintions of the MiniC++ ANTLR grammar.

```
constDef:       CONST type constDefEntry (COMMA constDefEntry)* SEM ;
constDefEntry:  IDENT init ;

varDef:         type varDefEntry (COMMA varDefEntry)* SEM ;
varDefEntry:    STAR? IDENT (init)? ;

init:           EQUAL  initOption ;
initOption:     BOOLEAN      #BooleanInit
              | NULLPTR      #NullptrInit
              | (SIGN)? INT  #IntInit
              ;
```

4.2.4 Variables and Constants

for

Listing 4.3 shows the parser rules variable and constant definitions. Both definitions can have multiple entries, which are separated by a comma. In the case of a constant definition ~~entry~~, the initialization value is required. For a variable, this is optional. The `STAR` optional lexical rule classifies a field as an array if present. The `initOption` parser rule consists of three production alternatives specifying the possible initialization values.

4.2.5 Function Declaration and Definition

The rules for a function declaration and definition are shown in listing 4.4. In MiniC++, to call a function there must be at least a declaration of the function further ahead in the source code. Function declarations and definitions both start with the function head that consists of the return type, identifier and an optional parameter list. In the case of a function definition, the function head is followed the `block` rule, which contains the method's body.

Listing 4.4: Function declaration and defintion of the MiniC++ ANTLR grammar.

```
funcDecl:          funcHead SEM;
funcDef:           funcHead block;
funcHead:          type STAR? IDENT LPAREN formParList? RPAREN;
formParList:       (     VOID
                   |     formParListEntry (COMMA formParListEntry)*
                   );
formParListEntry: type STAR? IDENT (BRACKETS)?;
```

Listing 4.5: Statement rule and it's production alternatives of the MiniC++ ANTLR grammar.

```
stat:  ( emptyStat  | breakStat
       | blockStat  | exprStat
       | ifStat     | whileStat
       | inputStat  | outputStat
       | deleteStat | returnStat
       );
```

from

Listing 4.6: If Statement rule ~~of~~ the MiniC++ ANTLR grammar.

```
ifStat:      'if' LPAREN expr RPAREN stat elseStat?;
elseStat:    'else' stat;
```

The parameter list can consist either of a single entry, the `void` type, or of one or more actual input parameters. Each parameter specified by the **`formParListEntry`** rule, consists of a type, an optional star and brackets indicating an array followed by the identifier of the parameter.

4.2.6 Statements

true

The parser rule defining a statement is shown in listing 4.5. The statement rule~~s~~ serves as a container for all concrete statement types. For example, the **`ifStat`** rule can be seen in listing 4.6. It consists of the **`if`** keyword followed by the condition in parentheses and a statement which should be executed if the condition is ~~met~~. Optionally, an else statement can be specified. This rule does not suffer from the *dangling else* problem. In such cases ANTLR resolves the ambiguity by always choosing the first successful production.

4.2.7 Expressions

A Part of the grammar for expressions is shown in listing 4.7. At the top of every expression is an **`orExpr`** followed by zero or more **`exprEntry`** elements. In case an **`exprEntry`** is present, the expression performs one or more assignments. Each **`exprEntry`** consists of an assignment operator signalizing the type of assignment, and an **`orExpr`** that provides the value to be assigned. The **`orExpr`** and **`andExpr`** rules implement their respective boolean operators. The **`relExpr`** rule consists of a **`simpleExpr`** and zero or

ENTRY?

Listing 4.7: Expression rules for assignment and boolean operations of the MiniC++ ANTLR grammar.

```
expr:                  orExpr (exprEntry)*;
exprEntry:             exprAssign orExpr;
exprAssign:            EQUAL       #EqualAssign
                     | ADD_ASSIGN #AddAssign
                     | SUB_ASSIGN #SubAssign
                     | MUL_ASSIGN #MulAssign
                     | DIV_ASSIGN #DivAssign
                     | MOD_ASSIGN #ModAssign
                     ;
orExpr:               andExpr ( '||' andExpr )*;
andExpr:              relExpr ( '&&' relExpr )*;
relExpr:              simpleExpr
                      ( relExprEntry )*;
relExprEntry:         relOperator simpleExpr;
simpleExpr:           (SIGN)?
                      term ( simpleExprEntry )*;
simpleExprEntry:      SIGN term;
```

relational

more **relExprEntry** elements. The **relExprEntry** rule handles relative expressions with the **relExprOperator** rule, which contains the relative operators like greater or less than. The **simpleExpr** rule begins with an optional sign that is relevant for integer values, followed by a term and zero or more **simpleExprEntry** elements. The **simpleExprEntry** rule consists of a sign followed by a term. The precedence rules for arithmetic operations are realized inside the grammar.

allocation a

The rules for terms and factors are shown in listing 4.8. A term consists of a fact, which contains an optional negation making it a **notFact**, followed by zero or more **termEntry** elements. The **termEntry** rule realizes multiplication, division and modulo operations via the **termOperator** rule. The **fact** rule contains six possible productions. Three types of value literals can be used as a factor: integer, boolean or null-pointer. Another option is the array ~~initialization~~, defined by the **#NewArrayFact** alternative. The **#ExprFact** alternative allows for precedence using an expression contained in parentheses. To read the value of a variable or array, or call a function the **#CallFact** alternative using the **callFactEntry** rule is used. The **callFactEntry** rule contains an optional increment/decrement at the beginning and end of the rule. Each **INC_DEC** element has a named alias so that in the syntax tree it can be easily checked which element is null. Via the **IDENT** terminal class the name of the variable or array to read can be specified. If **callFactEntryOperation** is not null, then either a function call or array access is performed. Depending on the type of function that is called, parameters may be necessary. The **#ActParListFactOperation** alternative allows for parameters via the optional **actParList** rule. This rule consists of one or more expressions, that make up the parameters.

Listing 4.8: Expression rules for terms and factors of the MiniC++ ANTLR grammar.

```
term:               notFact (termEntry)*;
termEntry:          termOperator notFact;
termOperator:       STAR #StarOperator
                  | DIV  #DivOperator
                  | MOD  #ModOperator;
notFact:            NOT? fact;
fact:               BOOLEAN #BooleanFact
                  | NULLPTR #NullptrFact
                  | INT     #IntFact
                  | callFactEntry          #CallFact
                  | NEW type LBRACK expr RBRACK #NewArrayFact
                  | LPAREN expr RPAREN           #ExprFact
                  ;
callFactEntry:      preIncDec=INC_DEC?
                    IDENT
                    callFactEntryOperation?
                    postIncDec=INC_DEC?
                    ;
callFactEntryOperation:
          ( LBRACK expr    RBRACK)         #ExprFactOperation
        | ( LPAREN (actParList)?   RPAREN) #ActParListFactOperation
        ;
actParList:         expr (COMMA expr)*;
```

4.3 Abstract Syntax Tree (AST)

The syntax tree created by ANTLR contains information that is not necessary for the later stages of the compiler. For this reason an abstract syntax tree (AST) is generated from the syntax tree.

For the implementation of the AST nodes Kotlin *data classes* are used primarily. A Kotlin `data class` provides implementations for common methods like `equals` and `hashCode`. The method implementations are generated from the primary constructor of the data class. For leaf nodes which always contain the same information, e.g. the `VOID` of the `formParList` rule, Kotlin's `data object` construct is used. A `data object` is a singleton built into the language itself. In case a rule has more than one possible production, interfaces are used. Specifically Kotlin provides *sealed* interfaces. With sealed interfaces, all classes that implement the interface are known at compile time. This allows for exhaustive pattern matching using the `when` expression.

The root class of the AST is `MiniCpp`. It serves as a container for all classes that implement the `MiniCppEntry` interface. The implementation classes are shown in figure 4.1. `Sem` is a `data object` since it encodes the same information every time it is used. `ConstDef` and `VarDef` both contain a list of entries, which store the information of the variables. For type management the AST uses the `ExprType` enum shown in figure 4.9. This enum contains all possible data types that can be used in MiniC++. It further includes a descriptor which is used in the backend of the compiler during the bytecode generation. The `FuncDef` and `FuncDecl` classes store the information about the function's signature

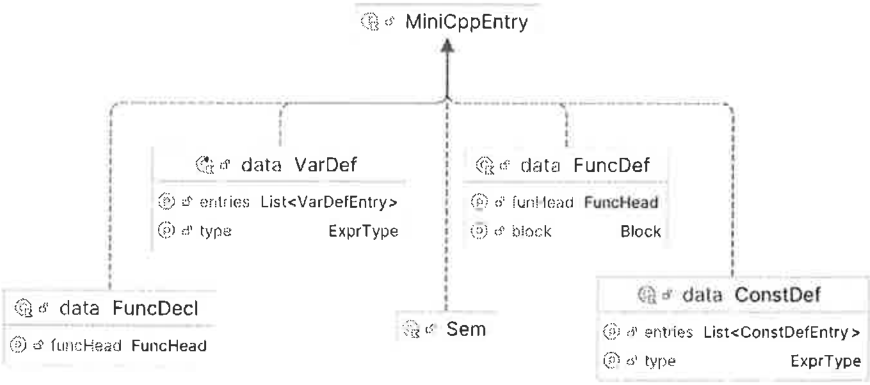

Figure 4.1: Implementation classes of the `MiniCppEntry` sealed interface.

in the `FuncHead` class. The `FuncDef` class further includes a field which contains the actual body of the function in the form of the `Block` class. A block consists of many block entries. Each block entry is a class that implements the `BlockEntry` sealed interface. These are `VarDef`, `ConstDef` and the sealed interface `Stat`. All concrete statement types like a while statement inherit from the `Stat` interface.

The AST nodes for expressions follow a similar pattern. The hierarchy is similar to the structure defined in the grammar. For the representation of a factor the `Fact` interface is used. The classes that implement the `Fact` interface are shown in figure 4.2. The `DataType` interface is responsible for handling literals. Each of its implementation classes is responsible for handling one type of literal, e.g. the `IntType` for integer literals. For the instantiation of an array the `NewArrayTypeFact` class provides the required information. The `type` field stores the data type of the array. The expression provides an integer value which serves as the length of the array. For nesting of expressions the `ExprFact` class is used. For reading a variable or array value and calling a function the `ActionFact` class is used. The `prefix` and `suffix` fields are responsible for the optional increment/decrement operator. In case the `actionOp` field is null, the identifier is the name of a variable whose value should be read. `ActionOperation` is again an interface which is implemented by the `ArrayAccessOperation` and `CallOperation`. The `ArrayAccessOperation` contains an expression which provides the index for the array access. The `CallOperation` consists of the actual parameter list for the function call which is a list of expressions.

according to block structure

4.4 Symboltable

Besides the AST a symboltable is needed for the management of the variables and functions of the program. For variables, it is necessary to keep track of their lifetime and shadowing, in case the same variable name is used twice. Since MiniC++ supports function overloading, each overload of a function needs to be accounted for, so that during the bytecode generation the correct function is called. The `Scope` class is responsible

SEIT WANN ???

Listing 4.9: Implementation of the `ExprType` enum.

```
enum class ExprType(val descriptor: String = "") {
        VOID("V"),
        BOOL("Z"),
        INT("I"),
        NULLPTR,
        INT_ARR("[I"),
        BOOL_ARR("[Z"),
}
```

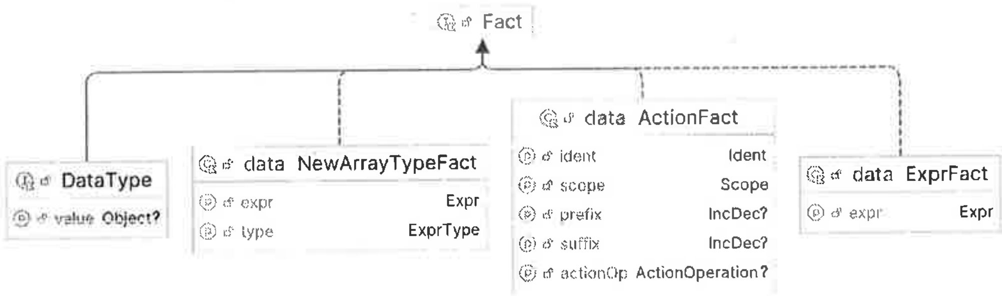

Figure 4.2: Implementation classes of the `Fact` sealed interface.

when definition

for both variables and functions. It stores information about all variables and functions declared in the current scope. Every time the scope changes, e.g. a function is entered, a new `Scope` instance is created. Each `Scope` instance stores a reference to its parent. This reference is used when a variable or function defined in the parent scope needs to be accessed. Only the root instance does not have a parent scope. In this scope functions and global variables are defined.

4.4.1 Variables

The `Variable` class shown in listing 4.10 stores information about a variable. The backend uses this information to generate the code for the variable. The identifier is only relevant during the AST generation, as it serves as the link connecting the variable instance to the concrete variable definition node in the AST. The `Variable` class is used for both variable and constant definitions, indicated by the boolean flag `const`. Additionally for constant values the value itself is ~~also~~ stored in the `constValue` field. In case the variable is defined in the global scope the `static` flag is active.

Bytecode does not rely on identifiers for stack variables and instead uses indexes. The index of a variable is also stored as a field. The index of a variable is determined during

Listing 4.10: Implementation of the `Variable` class.

```
class Variable(
    val ident: Ident,
    val type: ExprType,
    val const: Boolean,
    val static: Boolean,
    var index: Int,
    val constValue: Any? = null
)
```

Listing 4.11: Implementation of the `getNextAvailableIndex` method.

```
private fun getNextAvailableIndex(static: Boolean): Int {
        if (static) return -1

        val nonStaticVars = variables.filterNot { it.static }
        return if (nonStaticVars.isEmpty()) {
            parent?.getNextAvailableIndex(static) ?: 0
        } else {
            nonStaticVars.maxOf { it.index } + 1
        }
}
```

Listing 4.12: Implementation of the `Function` class.

```
data class Function(
        val ident: Ident,
        val returnType: ExprType,
        val formParTypes: List<ExprType>,
        var isDefined: Boolean = false
)
```

this

the AST generation. The `getNextAvailableIndex` method returns the next free index for a variable. Listing 4.11 shows the implementation of ~~the~~ method. Static variables in bytecode use identifiers and therefore are excluded from the index determination process. Indexes start by zero ind are incremented by one. The method determines the current highest index then returns it incremented by one.

4.4.2 Function

Information about a function is managed in the `Function` class shown in listing 4.12. A function is identified by its signature: a combination of the identifier and the formal parameter list. MiniC++ allows the declaration and definition of a function to be specified separately. The `isDefined` flag keeps track of the definition status of a function. This field is further used in a later stage when semantic checks are performed.

4.5 Visitor Implementation these

The visitor implementation works by traversing the generated syntax tree. When generating the ANTLR combined lexer and parser, a visitor interface and base class are generated. The interface `minicppVisitor` is implemented by the `minicppBaseVisitor` base class. This base class contains empty implementations for all `visit` methods of the interface. This allows the developer to only override ~~the~~ methods that are actually needed. The interface is also generic, enabling different return types for the visitor methods based on the specific requirements.

This implementation creates a concrete visitor class for every type of the AST. The class extends the `mincCppBaseVisitor` and overrides only ~~the~~ methods relevant for that AST type. The visitor implementation for the `MiniCpp` AST type is shown in listing 4.13. The AST type serves as the generic type of the base visitor. The `visitMiniCpp` method then returns this type. The input parameter of the method is an instance of the `MiniCppContext` class, which is the syntax tree node representing the `MiniCpp` rule. From this object the relevant information for the AST node is extracted. In this case it contains a list of `miniCppEntry`. Each entry in the list is processed by the `MiniCppEntryVisitor` visitor. The `MiniCppEntryVisitor` overrides all methods for the `varDef`, `funcDecl`, `funcDef` and `SEM` rules. The return type of all methods is the interface `MiniCppEntry`. The root node of the AST is made up of these entries plus the current scope called `globalScope` since it is the parent of all other scopes. The scope instance is passed on to the `MiniCppEntryVisitor` so that for example variable definitions can be added to it.

Named alternatives in the grammar as shown in 4.7 for the `termOperator` rule generate the following code seen in listing 4.14. The `TermOperatorVisitor` class contains shown a method for each alternative, eliminating the need to perform not null checks on each symbol to figure out which alternative was produced. Each method returns the enum value corresponding to the node. For more complex rule alternatives, as ~~seen~~ in the `callFactEntryOperation` rule, all symbols belonging to an alternative are grouped inside a context object with the same name as the alternative.

The code shown in listing 4.15 produces an AST using the visitor pattern. The instantiation of the character stream followed by the lexer, token stream and parser is the same for all implementations. The syntax tree is generated by ~~calling the~~ `parser.miniCpp()` ~~method~~. The syntax tree is then passed into the `visit` method of the `MiniCppVisitor` which ~~who~~ initiates the generation of the AST.

4.5.1 Listener Implementation ?

The listener implementation generates the AST during parse process of the syntax tree. Every time, a node is entered and exited, listener methods are called which build up the AST. The listener implementation follows a similar pattern to the visitor implementation in that it creates a separate listener for each type of the AST. Similar to the `minicppBaseVisitor`, ANTLR also generates a `minicppBaseListener` class. This class contains empty implementations for all listener methods. Only the methods relevant for a specific AST node then need to be overridden. For each node in the syntax tree, there is an `enter` and `exit` method, suffixed by the node's name. The `enter` methods

Listing 4.13: Implementation of the `MiniCppVisitor` class.

```
class MiniCppVisitor(private val className: String) : minicppBaseVisitor<MiniCpp>()
{
    override fun visitMiniCpp(ctx: minicppParser.MiniCppContext): MiniCpp {
        val scope = Scope(null)
        val entries = ctx.miniCppEntry().map { it.accept(MiniCppEntryVisitor(scope))
}
        return MiniCpp(
            globalScope = scope,
            entries = entries,
            className = className
        )
    }
}
```

Listing 4.14: Implementation of the `TermOperatorVisitor` class.

```
class TermOperatorVisitor : minicppBaseVisitor<TermOperator>() {
    override fun visitStarOperator(ctx: minicppParser.StarOperatorContext):
    TermOperator {
        return MUL
    }

    override fun visitDivOperator(ctx: minicppParser.DivOperatorContext):
    TermOperator {
        return DIV
    }

    override fun visitModOperator(ctx: minicppParser.ModOperatorContext):
    TermOperator {
        return MOD
    }
}
```

Listing 4.15: Code for the generation of the AST using the visitor-pattern.

```
fun generateASTForFileVisitor(inputStream: InputStream, className: String): MiniCpp
{
    val charStream = CharStreams.fromStream(inputStream)
    val lexer = minicppLexer(charStream)
    val tokenStream = BufferedTokenStream(lexer)
    val parser = minicppParser(tokenStream)
    return MiniCppVisitor(className).visit(parser.miniCpp())
}
```

also include the context object as an input parameter, however since the node ha ~~is~~ not been fully parsed yet, no data can be extracted.

The listener for the `MiniCpp` AST node is shown in listing 4.16. The constructor of the `MiniCppListener` requires an instance of the `MiniCppEntryListener`. Via this reference, the parsed `miniCppEntry` elements are retrieved. The `exitMiniCpp` method sets the variable `result` which stores the root node of the AST. This variable is used to

Listing 4.16: Implementation of the `MiniCppListener` class.

```
class MiniCppListener(
    private val entryListener: MiniCppEntryListener,
    private val scopeHandler: ScopeHandler,
    private val className: String
) : minicppBaseListener() {

    lateinit var result: MiniCpp

    override fun exitMiniCpp(ctx: minicppParser.MiniCppContext) {
        result = MiniCpp(className,
                         scopeHandler.getScope(),
                         entryListener.getAllEntries()
                         )
    }
}
```

pass the AST to the rest of program once it has been parsed.

The listeners internally use stacks to store the parsed AST nodes. Listeners that are higher up in the AST hierarchy then fetch the parsed nodes from the respective listeners. The implementation of the `BlockListener` shown in 4.17 uses the `BlockEntryListener` to get the entries every time a block is exited. The `BlockContext` element knows how many entries it contains and thus the correct number of entries can be retrieved from the block entry stack. This is done via the `getBlockEntry` method, which returns the last parsed block entry and removes it from the stack. The `BlockListener` itself provides the `getBlock` method, which performs the same function for `Block` nodes. Once all entries have been retrieved, their order needs to be reversed. This must be done due to the fact that the entries were retrieved starting with the last parsed entry. By reversing the entries the original program order is preserved.

The code for adding a block entry to the stack is shown in 4.18. The `BlockEntry` interface is implemented by `Stat`, `VarDef` and `ConstDef`. Even though there are separate listener methods for these types available, the individual nodes need to be retrieved in the `exitBlockEntry` method. There may be statements or variable definitions that are not part of a block and thus, only when a block entry is exited can these nodes be safely retrieved. To retrieve the correct node a null check needs to be performed on all possible productions.

For the management of variables and functions the `Scope` class is used, same as in the visitor based implementation. For the communication between listeners the `ScopeHandler` class is used. The code of this class is shown in listing 4.23. Fundamentally, the `ScopeHandler` manages a stack of `Scope` instances. The `init` block instantiates the root scope. The `pushChildScope` method instantiates a new child scope and puts it onto the stack. Whenever a variable or function needs to be added, the current scope can be retrieved via the `getScope` method. The `ScopeHandler` instance is the same over the entire AST generation. All listeners that need to interact with the scope have access to the same instance.

The execution process is similar to the visitor implementation shown in listing 4.15. First all the listener instances are created. Depending on the listener it's constructor

Listing 4.17: Implementation of the `BlockListener` class.

```
class BlockListener(private val blockEntryListener: BlockEntryListener,
    private val scopeHandler: ScopeHandler
) : minicppBaseListener() {

    private val blocks = mutableListOf<Block>()

    override fun exitBlock(ctx: minicppParser.BlockContext) {
        val scope = scopeHandler.getScope()
        val entries = mutableListOf<BlockEntry>()
        repeat(ctx.blockEntry().size) {
            entries.add(blockEntryListener.getBlockEntry())
        }
        blocks.add(Block(entries = entries.reversed(), scope = scope))
    }

    fun getBlock(): Block {
        return blocks.removeLast()
}
}
```

Listing 4.18: Implementation of the `exitBlockEntry` method.

```
    override fun exitBlockEntry(ctx: minicppParser.BlockEntryContext) {
        val entry = when {
            ctx.stat() != null -> statListener.getStat()
            ctx.varDef() != null -> varDefListener.getVarDef()
            ctx.constDef() != null -> constDefListener.getConstDef()
            else -> throw IllegalStateException("Unknown block entry")
        }
        blockEntries.add(entry)
    }
```

may contain one or more references to other listeners from where AST nodes can be retrieved. In some cases the relation between the listeners is too complex to be able to pass all the required listeners as constructor parameters. For such cases a separate `initialize` method is used which supplies the required listeners. The `minicppParser` then provides a method to register listeners. Every listener is registered to the parser using this method. The `miniCpp` method of the parser then starts the parsing process. Once this method has finished executing, the generated AST can be retrieved from the `result` variable of the `MiniCppListener`.

4.6 ATG Implementation

The implementation of the attributed grammar (ATG) is an extension of the grammar shown in 4.2. As ANTLR does not support Kotlin as an implementation language, the ATG is implemented in Java. The code written in the grammar file is directly embedded into the generated parser and executed during the parse process. Because Java and

Listing 4.19: Implementation of the `ScopeHandler` class.

```
class ScopeHandler {

    private val scopes = mutableListOf<Scope>()

    init {
        scopes.add(Scope(null))
    }

    fun getScope(): Scope {
        return scopes.last()
    }

    fun popScope() {
        scopes.removeLast()
    }

    fun pushChildScope() {
        scopes.add(Scope(getScope()))
    }
}
```

Kotlin are compatible on a bytecode level, the AST which is written in Kotlin, can be used in the Java based ATG implementation.

Since the AST and other needed classes like `ArrayList` are in different packages, they need to be imported. For this the `@header` section is used. The code written in this section is added to the top of the generated parser before the class declaration and thus can be used for the required import statements.

Similar to the listener implementation, the ATG implementation relies on stacks for the management of AST nodes. The stacks are defined in the `members` section. This section is pasted into the class body of the parser, and thus member variables can be defined here. Besides the stacks, ~~Also~~ the result of the AST generation, the `MiniCpp` object is stored as an instance variable. The scope handling is performed in the same way as in the listener implementation via the `ScopeHandler` class. Some utility methods are also defined in the member section. them

Listing 4.20 shows the ATG for the `miniCpp` and `miniCppEntry` rules. The semantic action for the `miniCpp` rule instantiates the `MiniCpp` AST class with the current scope and all `miniCppEntries`. The positioning of the semantic action after the `EOF` causes it to be executed at the end of the parsing process after all `miniCppEntry` nodes have been parsed. The semantic actions for the `miniCppEntry` rule retrieve nodes from their respective stack and put ~~it~~ onto the `miniCppEntries` stack.

In the case of rules like `constDef` where there are one or more `constDefEntry` nodes stored inside the `ConstDef` AST node, a list is used to store the entries during the parse of the rule. This process is shown in listing 4.21. After the first `constDefEntry` is parsed, a list is created which stores all following `constDefEntry` nodes. At the end of the rule, a new `ConstDef` node containing all `constDefEntry` nodes, is instantiated and put onto the `constDefs` stack.

The ATG code ~~visible~~ in listing 4.22, shows the rules `init` and `initOption`. For
shown

Listing 4.20: ATG for the `miniCpp` and `miniCppEntry` rules.

```
miniCpp: (miniCppEntry)* EOF
  { result = new MiniCpp(className, scopeHandler.getScope(),
  miniCppEntries.stream().toList()); }
  ;
miniCppEntry:
    constDef               { miniCppEntries.push(constDefs.pop()); }
  | varDef                 { miniCppEntries.push(varDefs.pop()); }
  | funcDecl               { miniCppEntries.push(funcDecls.pop()); }
  | funcDef                { miniCppEntries.push(funcDefs.pop()); }
  | SEM                    { miniCppEntries.push(Sem.INSTANCE); }
;
```

UFF!

Listing 4.21: ATG for the `constDef` rule.

```
constDef:
  CONST type constDefEntry
    { var entries = new ArrayList(List.of(constDefEntries.pop())); }
  (
    ',' constDefEntry
    { entries.add(constDefEntries.pop());}
  )
  * SEM
    { constDefs.push(new ConstDef(types.pop(), entries)); }
```

the **BooleanInit** and **IntInit** alternatives, values of their respective terminal classes must be processed. For the **BooleanInit** alternative this means reading the text of the terminal class and parsing it to a boolean. The **NullptrInit** alternative just puts the singleton instance of the **NullPtrType** on the stack and thus does not need to read any value. For the **IntInit** alternative, the integer value is retrieved by parsing the text of the **INT** terminal class. In case a sign is present it is checked if it is negative. In case of a negative sign the integer value is inverted.

The execution of the ATG functions in the same way as for the visitor implementation in listing 4.15. The parser is executed by calling the **miniCpp** method. Once this method has finished executing, the generated AST can be retrieved via the **result** variable of the parser. ???

4.7 Detection of semantic errors

The MiniC++ compiler frontend implements a number of semantic checks. During the parsing process if a semantic error is detected a **SemanticException** is raised. This exception is then caught by the **SemanticErrorAspect** aspect. The **performErrorHandling** method shown in listing 4.23 processes the exception. First the **ParseRuleContext** object is extracted from the arguments of the original method. This object contains the information about the current line and column number in the parsing process. The exception contains a message about the cause of the error. The information is then

Listing 4.22: ATG for the `initOption` rule.

```
init:        '='  initOption;
initOption:
   BOOLEAN
       { inits.push(new Init(newBoolType(
           Boolean.parseBoolean($BOOLEAN.text)
         )));
       }              #BooleanInit
 | NULLPTR
       { inits.push(new Init(NullPtrType.INSTANCE)); }
                    #NullptrInit
 | (SIGN)? INT
       {
         var value = Integer.parseInt($INT.text);
         if ($SIGN != null && $SIGN.text.equals("-")) {
             value = -value;
         }
         inits.push(new Init(new IntType(value)));
       }              #IntInit
;
```

Listing 4.23: Implementation of the `performErrorHandling` method.

```
fun performErrorHandling(joinPoint: ProceedingJoinPoint): Any {
    return try {
        joinPoint.proceed()
    }catch (e: SemanticException) {
        val ctx = joinPoint.args.first() as ParserRuleContext
        if(previousException != e) {
            System.err.println("Error on line ${ctx.start.line}:${ctx.start.
    charPositionInLine} : ${e.message}")
        }

        previousException = e

        if(doExit) {
            exitProcess(1)
        }else {
            throw e
        }
    }
}
```

uFF!

???

printed on the error output stream. A `doExit` flag allows setting whether the program should exit upon detection of a semantic error. If the program should continue executing, the error is thrown again and saved in the `previousException` field to avoid raising the same exception twice.

Chapter 5

of the

Implementation Backend

MiniC++

The focus of this chapter is on the implementation of the backend of the compiler. First, the differences between Java and C++ are presented. This is followed by the explanation of the source code generation. Finally, the generation of the bytecode is shown.

5.1 Differences between Java and C++

When compiling MiniC++ source code to Java bytecode, there are multiple differences in the functionality of both languages that need to be considered. The goal is to consider these differences and preserve the functionality of ~~the~~ MiniC++ ~~code~~ in bytecode.

5.1.1 Array Deletion

MiniC++ includes the `delete` keyword which reclaims the memory used for an array and invalidates the reference to it. Java on the other hand does not provide such a mechanism. In Java, the memory is managed by the JVM and the program can only request for the memory to be reclaimed by the garbage collector. To mimic MiniC++'s behavior as best as possible, the delete statement is transformed into a null assignment. ? a = null?
Thus, if the array is only used inside one function, its memory can be reclaimed by the garbage collector. This solution however does not work if an array is passed as an input parameter into a function. This is because then a reference to the array will also exist in another function making it impossible to reclaim the memory.

5.1.2 cout and cin

In MiniC++, input and ~~output~~ output to and from the console can be performed via the `cout` and `cin` streams. ~~For~~ For output, Java uses the `System.out` stream with separate methods for normal print ~~and~~ and print, with new line. All output statements therefore need to be transformed ~~to~~ to either `System.out.print` or `System.out.println`. The latter one is used when ~~a~~ `endl` is detected.

For input, the `java.util.Scanner` class can be used. The constructor of this class takes an input stream as a parameter. For the console in Java this is `System.in`. The scanner then provides methods to conveniently read the types supported in MiniC++, namely integer and boolean.

values of

5.1.3 Expression Evaluation

valid to

MiniC++ allows for more complex expression evaluations than Java. An expression like $4 < 5 < 3$ is ~~possible~~ in MiniC++ but not in Java. The way this expression is evaluated is as follows: First the left side $4 < 5$ is evaluated resulting in either a 1 or a 0. Then this is compared ~~against~~ the 3, e.g., $0 < 3$, if the previous expression resulted in a 0.

In Java this needs to be implemented as nested `if` statements in the scheme of `if (expr) ? 1 : 0`. If the expression is true then the result is a 1 and otherwise 0. The following expression then uses this result for its comparison.

5.1.4 Function Declarations and Classes

called used / called

MiniC++ requires at least a declaration (or a full definition) of a function earlier in the source code file before it can be ~~referenced~~. In Java however, methods can be ~~referenced~~ even if they are ~~only~~ defined later in the source code file. Therefore, there would be no need to enforce this rule, besides making sure that the function is actually defined at some point in the source code file. However, to honor this functionality of MiniC++, a semantic exception is raised during the parse process if a reference of a function that is not yet declared is detected.

MiniC++ does not include the concept of classes. Multiple functions are defined in a file but are not related to each other on a class level. In Java there can only be methods ~~defined~~. Standalone functions outside of classes are not possible. To translate the MiniC++ source code to Java bytecode, all functions are put inside the same class. To mimic the behavior of MiniC++ as close as possible, all methods are defined as static methods.

5.2 Source Code Generation

For the development of a compiler it is beneficial to implement a module for source code generation. The source code generation module takes the AST as input and generates MiniC++ source code. By generating source code from the AST, the correctness of the compiler frontend can be tested. If the code generated from the source code generator matches the original source code, it can be assumed that the AST has been correctly generated. When comparing there are some potential problems like formatting and comments. Therefore, it is best to take the generated source code and repeat the generation process one more time. The then generated source code can be used for comparison without any formatting or comments interfering. String Builder

The implementation of the source code generator uses a ~~string builder~~ combined with Kotlin extension functions. For each type of the AST, there is a **generateSourceCode** extension function which takes a ~~string builder~~ instance as the sole parameter. The extension functions are grouped according to their type into the following files:

- `BlockGenerator`,
- `ConstVarDefGenerator`,
- `ExprGenerator`,
- `FuncGenerator`,
- `MiniCppGenerator`)

Listing 5.1: Implementation of the `generateSourceCode` method for the `MiniCpp` class.

```
fun MiniCpp.generateSourceCode(): String {
    val sb = StringBuilder()
    entries.forEach {
        when (it) {
            is ConstDef -> it.generateSourceCode(sb)
            is FuncDecl -> it.generateSourceCode(sb)
            is FuncDef -> it.generateSourceCode(sb)
            Sem -> sb.appendLine(";")
            is VarDef -> it.generateSourceCode(sb)
        }
    }
    return sb.toString()
}
```

Listing 5.2: Implementation of the `generateSourceCode` method for the `ConstDef` class.

```
fun ConstDef.generateSourceCode(sb: StringBuilder) {
    sb.append("const ")
    type.generateSourceCode(sb)
    sb.append(" ")
    entries.forEachIndexed { index, entry ->
        sb.append("${entry.ident.name} = ")
        entry.value.generateSourceCode(sb)

        if (index != entries.lastIndex) {
            sb.append(", ")
        }
    }
    sb.appendLine(";")
}
```

- `StatGenerator` and
- `TypeGenerator`

The code for the `MiniCpp` AST node is shown in listing 5.1. In this function the string builder is instantiated and eventually returned as a normal string. For each `miniCppEntry` the respective `generateSourceCode` function is called. The code to generate the `ConstDef` node is shown in listing 5.2. First the `const` keyword is added to the string builder, followed by the source code for the type. Then all identifiers with their respective value are appended to the string builder. On the last entry the delimiter is omitted.

5.3 Classes

The first step when generating bytecode is to handle everything that is relevant on a class level. Every piece of code in Java is organized inside a class and stored inside a `.class` file. For this task the ASM framework provides the `ClassWriter` class. This class provides visitor pattern based methods for generating a class file. The code for generating the class definition is shown in listing 5.3. The constructor for the `ClassWriter` takes an integer

Listing 5.3: Code for the definition of a class.

```
val classWriter = ClassWriter(ClassWriter.COMPUTE_FRAMES + ClassWriter.COMPUTE_MAXS)
className = miniCpp.className
classWriter.visit(
    CLASS_FILE_VERSION,
    ACC_PUBLIC,
    miniCpp.className,
    null,
    "java/lang/Object",
    null
)
```

parameter that functions as a flag which modifies the behavior of the class writer. In this case, the COMPUTE_FRAMES and COMPUTE_MAXS flags are used. COMPUTE_FRAMES enables computation of stack map frames of methods from the bytecode. Further, COMPUTE_MAXS calculates the maximum stack size from the bytecode. Those two flags combined ease the development of the code generation since those two aspects are now computed automatically. Otherwise, it would be necessary to keep track of those values manually for every method generation, increasing complexity. the

The `visit` method defines a class. The first parameter is the CLASS_FILE_VERSION constant which has the value 65. This corresponds to Java Version 21. The second parameter defines the access flags of the class. ACC_PUBLIC means that the class is public. The third parameter is the class name. The fourth parameter defines the signature of the class, which is only relevant for generic classes and therefore left as `null`. The superclass is described by the fifth parameter. As the concept of classes does not exist in MiniC++ Java's default superclass `java.lang.Object` is used as the superclass. Via the last parameter, implemented interfaces can be defined. This parameter is also set to `null` as MiniC++ does not support interfaces.

Once the class has been initialized, the bytecode generation based on the AST can begin. On the top level, this process is shown in listing 5.4. First, a StaticVarDefGenerator is instantiated. The same instance is used across the entire generation process, since the generation of static variable definitions requires the modification of the static class initializer block. Then, for each miniCppEntry the appropriate bytecode is generated. For Sem and FuncDecl no code needs to be generated, since they don't encode any semantic information relevant for bytecode. The addStaticScannerField adds a scanner to the static variables. This is needed for the generation of cin statements. To make the class executable, a main method is needed. This is done via the addMainMethod method. Calling the visitEnd method of the classWriter finishes the code generation for the class. Finally, calling the toByteArray method returns the bytecode of the generated class as a byte array.

5.4 Functions

A function in MiniC++ is translated into a class method in bytecode. The FuncGenerator accepts a FuncDef AST node and generates the bytecode for it. The code for the generation is shown in listing 5.5. To generate code for a method a MethodVisitor

Listing 5.4: Top-level code for the bytecode generation.

```
val staticVarDefGenerator = StaticVarDefGenerator(classWriter)
miniCpp.entries.forEach {
    when (it) {
        is VarDef -> staticVarDefGenerator.generateStatic(it)
        is ConstDef -> StaticConstDefGenerator(classWriter).generateStatic(it)
        is FuncDef -> FuncDefGenerator(classWriter, miniCpp.className).generate(it)
        is Sem, is FuncDecl -> ""
    }
}
staticVarDefGenerator.generateStaticInitBlock(miniCpp)
addStaticScannerField(classWriter)
addMainMethod(classWriter)
classWriter.visitEnd()
return classWriter.toByteArray()
```

Listing 5.5: Code for the bytecode generation of the `FuncDef` node.

```
fun generate(funcDef: FuncDef) {
    val methodVisitor = classWriter.visitMethod(
        Opcodes.ACC_PUBLIC + Opcodes.ACC_STATIC,
        funcDef.funHead.ident.name,
        funcDef.funHead.getDescriptor(),
        null,
        null
    )
    methodVisitor.run {
        visitCode()
        BlockGenerator(methodVisitor, className).generate(funcDef.block, null)
        visitInsn(RETURN)
        visitMaxs(0, 0)
        visitEnd()
    }
}
```

define

instance is needed. The visitor can be acquired by calling the `visitMethod` method of the class writer. The parameter of the `visitMethod` defines the signature of the method to be generated. The first parameter defines the access of the method. `ACC_PUBLIC` and `ACC_STATIC` ~~make it so~~ that the method has the modifiers `public` and `static`. The second parameter is the name of the method. The third parameter defines the method's descriptor. The descriptor is a string representation of the input parameter types and the return type of the method. The fourth parameter describes the method's signature. This parameter is only needed for generics and thus can be set to `null`. The final parameter is a string array containing all exceptions that the method may throw. Since it is not possible in MiniC++ to write code that would cause a checked exception, this parameter can also be set to `null`.

Listing 5.6 shows the generation of the descriptor. The descriptor is generated using a string builder. The input parameter types are grouped inside parenthesis. `void` or empty input parameters are represented as `()`. For each input parameter it's type descriptor is added to the method descriptor. The following descriptors are relevant for the code

Listing 5.6: Generation of the descriptor of a method.

```
fun FuncHead.getDescriptor(): String {
    val descriptor = StringBuilder("(")
    if (formParList != null && formParList is FormParListEntries) {
        (formParList as FormParListEntries).entries.forEach {
            descriptor.append(it.type.descriptor)
        }
    }
    descriptor.append(")")
    descriptor.append(type.descriptor)
    return descriptor.toString()
}
```

generation from MiniC++:

- Void: `V`
- Boolean: `Z`
- Integer: `I`
- Boolean Array: `[Z` and
- Integer Array: `[I`

Once the input parameters are added, the descriptor of the return type is appended after the closing parenthesis. For a method with an integer and boolean input parameter and a boolean return type the descriptor is `(IZ)Z`.

Listing 5.5 further shows the code generation for the method's body. The `run` extension function changes the `this` of the functions body to the `methodVisitor`. With this the methods of it can be called without having to explicitly type `methodVisitor`. To start the code generation the `visitCode` method is called. All following visitor calls are then added to the method's body. The method's body is generated by the `BlockGenerator`. The `BlockGenerator` calls the respective code generators for each `blockEntry`. Namely, the `LocalVarDefGenerator` and the `StatGenerator`. For constant definitions no code needs to be generated at the definition stage. Since every method needs a return instruction, even if the method's return type is `void`, a return instruction is added after the `BlockGenerator`. For normal instructions like `RETURN` the `visitInsn` method is used. The `visitMaxs` method call sets the maximum stack size and maximum size of the local variables. Both are set to zero as the ASM framework computes the correct values based on the generated bytecode. To finish the code generation for the method the `visidEnd` method is called.

5.5 Static Fields

Variable definitions of MiniC++ are converted into static variables inside the class. Constant definitions are added into the constant pool. When a constant variable is referenced inside the bytecode, the value from the constant pool is loaded.

Listing 5.7: Code of the `StaticConstDefGenerator` class.

```
class StaticConstDefGenerator(private val cw: ClassWriter) {

fun generateStatic(constDef: ConstDef) {
    constDef.entries.forEach { entry ->
        val index = cw.newConst(entry.value.value.getValue())
        entry.variable.index = index
    }
}
}
```

5.5.1 Constant Definitions

Constant definitions are handled by the `StaticConstDefGenerator` class. Its source code is visible in listing 5.7. The generator takes the `ClassWriter` instance as a constructor argument, so that the constant pool can be accessed. In the `generateStatic` method, a `ConstDef` node is accepted and processed. ASM provides the `newConst` method that creates an entry in the constant pool of the class and returns the index of the value in it. If an entry with the same value already exists, its index is returned instead. For all `constDefEntry` elements this method is called, and the index is stored in the variable associated with it. When the constant is later referenced, the index can be used to put the value on the operand stack.

5.5.2 Variable Definitions

The `StaticVarDefGenerator` class generates the code for static variable definitions. In bytecode the declaration and initialization are split up. First the field is defined and then later in the static initializer block of the class, a value is assigned to it.

Listing 5.8 shows the code for the declaration of a static variable. The `ClassWriter` method `visitField` declares the variable. The first parameter defines the access flags, in this case all variables are `public` and `static`. The second parameter defines the name of the variable. Contrary to local variables which use indexes, static variables are referenced by their name. The type of the variable is defined by the third parameter. For this the type descriptor is used. The fourth parameter handles the variable's signature. It can be set to `null`, because no generics are used. The final parameter sets the value of the field. This field is only relevant for `final` variables, whose value cannot change. Each `VarDef` node is also added to the `generatedVarDefs` list, which is used for the initialization.

The initialization is performed by the code shown in listing 5.9. First the method visitor for the static initializer block is acquired by calling the `visitMethod` method. The name of the static initializer block is predefined with `<clinit>` and the method has no parameters and the `void` return type. For each `VarDef` entries with no default value are filtered. For each entry its default value is pushed onto the operand stack as a constant using the `visitLdcInsn` method. The `visitFieldInsn` method then pops the value from the operand stack and assigns it to the static variable. The first parameter of the method is the operand code. `PUTSTATIC` is the operand code for assigning a new value to a static variable. The second parameter describes the owner of the variable, in

Listing 5.8: Code for the declaration of static variables.

```
fun generateStatic(varDef: VarDef) {
    varDef.entries.forEach { entry ->
        cw.visitField(
            ACC_PUBLIC
                    or ACC_STATIC,
            entry.ident.name,
            varDef.type.toPointerTypeOptional(entry.pointer).descriptor,
            null,
            null
        )
    }
    generatedVarDefs.add(varDef)
}
```

Listing 5.9: Code for the initialization of static variables.

```
fun generateStaticInitBlock(miniCpp: MiniCpp) {
    cw.visitMethod(
        ACC_STATIC,
        "<clinit>",
        "()V",
        null,
        null
    ).apply {
        visitCode()
        generatedVarDefs.forEach { varDef ->
            varDef.entries
                .filter{ it.value != null }
                .forEach { entry ->
                visitLdcInsn(entry.value?.value?.getValue())
                visitFieldInsn(
                    PUTSTATIC,
                    miniCpp.className,
                    entry.ident.name,
                    varDef.type.toPointerTypeOptional(entry.pointer).descriptor
                )
            }
        }
        visitScannerInit(miniCpp)
        visitInsn(RETURN)
        visitMaxs(0, 0)
        visitEnd()
    }
}
```

this case it is the current class name. The third parameter is the identifier and the final one the descriptor of the variable type.

The `visitScannerInit` method initializes the scanner which is used to read input from the console. First, the method ensures that the name for the scanner variable is not taken by looking through all scopes. Then, an instance of the scanner is initialized. To invoke the constructor the op code `INVOKESPECIAL` is used.

Listing 5.10: Code for the definition of local variables.

```
fun generate(varDef: VarDef) {
    varDef.entries.forEach { entry ->
        val type = varDef.type.toPointerTypeOptional(entry.pointer)
        pushInitValue(entry.value, type)
        storeVariable(type, entry.variable.index)
    }
}

private fun storeVariable(type: ExprType, index: Int) {
    val opCode = when (type) {
        ExprType.INT -> ISTORE
        ExprType.BOOL -> ISTORE
        else -> ASTORE
    }
    mv.visitVarInsn(opCode, index)
}
```

5.6 Local Variables

MiniC++ supports local variables and constant definitions. The latter one are handled in the same way as the global constant definitions by the `ConstDefGenerator`. For each constant definition an entry is created in the constant pool of the class. When the constant definition is referenced, the constant's value is pushed onto the operand stack from the constant pool.

The `LocalVarDefGenerator` generates the code for local variable definitions. The relevant bytecode is generated by the code shown in listing 5.10. First, for each entry the type is optionally converted to a pointer type, if the flag is set. Then the `pushInitValue` method takes the entry's value and pushes it onto the stack. In case the value is null, the respective default value of the type is pushed onto the stack. The `storeVariable` method then stores the value from the stack in the variable. For this the variable type and index is needed. Depending on the type of the variable a different opcode needs to be generated. The bytecode instruction is generated by the `visitVarInsn` method, which takes the opcode and the index of the variable as a parameter.

5.7 Statments

Statements are generated by the `StatGenerator` class. When generating a statement first the appropriate generation method for the specific type of statement is chosen. This is done by the `generate` method of the `StatGenerator` shown in listing 5.11. Depending on the complexity of the statement, the code is generated by another generator, e.g. the `OutputStatGenerator`. The `BreakStat` requires just one instruction and is thus generated directly in the method. The `EmptyStat` does not lead to the generation of any bytecode and is ignored.

Listing 5.11: Implementation of the `generate` method of the `StatGenerator`.

```
fun generate(stat: Stat, breakLabel: Label?) {
    when (stat) {
        is InputStat -> generateInputStat(stat)
        is BlockStat -> BlockGenerator(mv, className).generate(stat.block,
    breakLabel)
        is DeleteStat -> generateDeleteStat(stat)
        is ReturnStat -> generateReturnStat(stat)
        is OutputStat -> OutputStatGenerator(mv).generate(stat)
        is ExprStat -> ExprGenerator(mv).generate(stat.expr, false)
        is WhileStat -> generateWhileStat(stat)
        is IfStat -> generateIfStat(stat, breakLabel)
        is BreakStat -> mv.visitJumpInsn(Opcodes.GOTO, breakLabel!!)
        is EmptyStat -> ""
    }
}
```

Listing 5.12: Implementation of the `generateIfStat` method of the `StatGenerator`.

```
private fun generateIfStat(stat: IfStat, breakLabel: Label?) {
    val elseLabel = Label()
    val endLabel = Label()

    ExprGenerator(mv).generate(stat.condition)
    mv.visitJumpInsn(Opcodes.IFEQ, elseLabel)
    generate(stat.thenStat, breakLabel)
    mv.visitJumpInsn(Opcodes.GOTO, endLabel)
    mv.visitLabel(elseLabel)
    stat.elseStat?.let { generate(it, breakLabel) }
    mv.visitLabel(endLabel)
}
```

5.7.1 If Statement

Bitte ein Bild.

An if statement is generated by the `generateIfStat` method shown in listing 5.12. First, two labels are created. Bytecode uses labels as the target for its `GOTO` instructions. For if statements, two labels are needed, one for the else branch and one to mark the end of the statement. Then the bytecode for the if statement's condition is generated by the `ExprGenerator`. The jump instruction `IFEQ` then checks if top value on the stack is equal to zero. If this is the case, a jump to the `elseLabel` is performed. Otherwise, the execution proceeds to the next instruction. The code for the `thenStat` is generated by the `generate` method of the `StatGenerator`. After that, a jump to the end of the statement is ~~performed~~. This is done, because the else branch of the if statement is generated after the then branch and thus needs to be skipped. If an else branch is present, the bytecode for it is generated in the same way.

generated

the

5.7.2 Delete Statement

The delete statement causes the reference to the array to be set to null. Reclaiming memory in the same way as C++ is not possible in the JVM. The bytecode generation

in

Listing 5.13: Implementation of the `generateDeleteStat` method of the `StatGenerator`.

```
private fun generateDeleteStat(stat: DeleteStat) {
    val variable = stat.scope.getVariable(stat.ident)
    mv.visitInsn(Opcodes.ACONST_NULL)
    mv.visitVarInsn(Opcodes.ASTORE, variable.index)
}
```

for the delete statement is shown in figure 5.13. First, the variable associated with the identifier specified in the delete statement is retrieved. The `ACONST_NULL` opcode pushes a `null` value onto the stack. With the `ASTORE` opcode the value is then popped from the stack and assigned to the array variable.

code generation for

5.7.3 Input Statement

The input statement reads a value from the console and then assigns that value to a variable. To read a value from the console the static `Scanner` instance is used. The code implementing the input statement is shown in listing 5.14. The scanner is loaded with the `GETSTATIC` opcode, and it's owner, name and descriptor. The method of the scanner that should be called depends on the type of the variable. To invoke a method of an object the `INVOKEVIRTUAL` opcode is needed. The method visitor generates method invocations using the `visitMethodInsn` method. In addition to the opcode, the qualified name of the scanner, method name, descriptor and a boolean value indicating if an interface is invoked, are needed. Finally, the value put onto the stack by the scanner is stored. In case the variable is static, the `visitFieldInsn` method is used to store the value with the `PUTSTATIC` opcode. The `visitVarInsn` method in combination with the `ISTORE` opcode, stores the value for a local variable. ?

5.7.4 While Statement

→ Bitte ein Bild!

the

Listing 5.15 shows the code for the generation of a while statement. A while statement requires two labels. One for the start and one for the end of the statement. The start label is marked with the `visitLabel` method of the method visitor. Then condition's bytecode is generated by the `ExprGenerator`. The condition is then checked with the `IFEQ` opcode that jumps to the specified label in the case the condition matches the value zero. If the condition is true, the body of the while statement is executed. It is generated by the `generate` method of the `StatGenerator`. The `endLabel` is passed as a parameter. In the case that somewhere in the statement's body a `break` statement is defined, it will jump to this label. At the end of the statement's body the loop is repeated by jumping to the `startLabel` with the `GOTO` opcode. The `endLabel` is visited after the jump instruction.

5.7.5 Output Statement

Output statements are handled by the `OutputStatGenerator` which generates the bytecode for `System.out.print` and `System.out.println` calls. An output statement consists of one or more `outputStatEntry` elements, which can be an expression, a

Listing 5.14: Implementation of the `generateInputStat` method of the `StatGenerator`.

```
private fun generateInputStat(stat: InputStat) {
    mv.visitFieldInsn(GETSTATIC, className, scannerVarName, SCANNER_DESC)
    val variable = stat.scope.getVariable(stat.ident)
    val methodName = if (variable.type == ExprType.INT) "nextInt" else "nextBoolean"
    val methodDesc = if (variable.type == ExprType.INT) "()I" else "()Z"
    mv.visitMethodInsn(
        Opcodes.INVOKEVIRTUAL,
        SCANNER_QUAL_NAME,
        methodName,
        methodDesc,
        false
    )
    if(variable.static) {
        mv.visitFieldInsn(
            Opcodes.PUTSTATIC,
            className,
            variable.ident.name,
            variable.type.descriptor
        )
    } else {
        mv.visitVarInsn(Opcodes.ISTORE, variable.index)
    }
}
```

Listing 5.15: Implementation of the `generateInputStat` method of the `StatGenerator`.

```
private fun generateWhileStat(stat: WhileStat) {
    val startLabel = Label()
    val endLabel = Label()
    mv.visitLabel(startLabel)
    ExprGenerator(mv).generate(stat.condition)
    mv.visitJumpInsn(Opcodes.IFEQ, endLabel)
    generate(stat.whileStat, endLabel)
    mv.visitJumpInsn(Opcodes.GOTO, startLabel)
    mv.visitLabel(endLabel)
}
```

of ? (selected)

string literal or ~~And~~ end line token. Before the code generation for each type, the `System.out` static field is loaded. For an expression, it's bytecode is generated by the `ExprGenerator`. The descriptor for the `print` method is decided based on the type of the expression. The descriptor is then passed on to the `generatePrint` method, which generated bytecode for the `print` method invocation. The qualified name of the `out` print stream is `java/io/PrintStream`. To print a stream literal, its value is put onto the stack using the `visitLdcInsn` method. To print a string the `generatePrint` method must be passed the descriptor of the `String` type, which is `(Ljava/lang/String;)V`.

5.8 Expressions

Listing 5.16: Code for the print generation methods of the `OutputStatGenerator`.

```
private fun generatePrintExpr(expr: Expr) {
    ExprGenerator(mv).generate(expr)
    val descriptor = if (expr.getType() == ExprType.INT) {
        PRINT_INT_DESC
    }else {
        PRINT_BOOL_DESC
    }
    generatePrint(descriptor)
}

private fun generatePrintText(text: String) {
    mv.visitLdcInsn(text)
    generatePrint(PRINT_STRING_DESC)
}

private fun generatePrint(descriptor: String) {
    mv.visitMethodInsn(
        INVOKEVIRTUAL,
        PRINT_STREAM_QUALIFIED_NAME,
        PRINT_METHOD_NAME,
        descriptor,
        false
    )
}
```

References

Literature

Bruneton, Eric (2007). "ASM 3.0 A Java bytecode engineering library". *URL: http://download. forge. objectweb. org/asm/asmguide. pdf* (cit. on p. 1).

Chomsky, Noam (1959). "On certain formal properties of grammars". *Information and Control* 2.2, pp. 137–167. URL: https://www.sciencedirect.com/science/article/pii/S0019995859903626 (cit. on p. 2).

Dobler, H. and K. Pirklbauer (May 1990). "Coco-2: a new compiler compiler". *SIGPLAN Not.* 25.5, pp. 82–90. URL: https://doi.org/10.1145/382080.382635 (cit. on p. 2).

McGhan, H. and M. O'Connor (1998). "PicoJava: a direct execution engine for Java bytecode". *Computer* 31.10, pp. 22–30 (cit. on p. 14).

McKeeman, William M (1965). "Peephole optimization". *Communications of the ACM* 8.7, pp. 443–444 (cit. on p. 23).

Parr, Terence (1993). "Obtaining Practical Variants of LL(k) and LR(k) for k". PhD thesis. Purdue University (cit. on p. 7).

— (2013). *The Definitive ANTLR 4 Reference.* 2nd. Pragmatic Bookshelf (cit. on p. 6).

Parr, Terence and Kathleen Fisher (2011a). "LL (*) the foundation of the ANTLR parser generator". *ACM Sigplan Notices* 46.6, pp. 425–436 (cit. on p. 8).

— (2011b). "LL(*): the foundation of the ANTLR parser generator". 46.6. URL: https://doi.org/10.1145/1993316.1993548 (cit. on p. 7).

Parr, Terence, Sam Harwell, and Kathleen Fisher (2014). "Adaptive LL(*) parsing: the power of dynamic analysis". 49.10. URL: https://doi.org/10.1145/2714064.2660202 (cit. on p. 8).

Puffitsch, Wolfgang and Martin Schoeberl (2007). "picoJava-II in an FPGA". In: *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems.* JTRES '07. Vienna, Austria: Association for Computing Machinery, pp. 213–221. URL: https://doi.org/10.1145/1288940.1288972 (cit. on p. 14).

Scott, Elizabeth and Adrian Johnstone (2010). "GLL Parsing". *Electronic Notes in Theoretical Computer Science* 253.7. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 177–189. URL: https://www.sciencedirect.com/science/article/pii/S1571066110001209 (cit. on p. 8).

Tomita, Masaru and See-Kiong Ng (1991). "The generalized LR parsing algorithm". *Generalized LR parsing*, pp. 1–16 (cit. on p. 8).

Online sources

ASM (2024). *https://asm.ow2.io*. URL: https://asm.ow2.io/index.html (visited on 11/26/2024) (cit. on p. 24).

Cassandra, Apache (2024). *Cassandra Parser*. URL: https://github.com/apache/cassandra/blob/trunk/src/antlr/Parser.g/ (visited on 11/25/2024) (cit. on p. 7).

Hibernate (2024). *Hibernate ORM 6.0.0.Alpha1 released*. URL: https://in.relation.to/2018/12/06/hibernate-orm-600-alpha1-out/ (visited on 11/25/2024) (cit. on p. 7).

Kotlin (2024). *Kotlin ClassBuilder*. URL: https://github.com/JetBrains/kotlin/blob/v1.2.30/compiler/backend/src/org/jetbrains/kotlin/codegen/ClassBuilder.java (visited on 11/26/2024) (cit. on p. 25).

Microsoft (2025). *Compile CIL to Native Code*. URL: https://learn.microsoft.com/en-us/dotnet/standard/managed-execution-process#compile-cil-to-native-code (visited on 01/21/2025) (cit. on p. 19).

OpenJDK (2025). *HotSpot Runtime Overview*. URL: https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html#Bytecode%20Verifier%20and%20Format%20Checker%7Coutline (visited on 01/21/2025) (cit. on p. 19).

Oracle (2014). *JDK LambdaMetafactory*. URL: https://hg.openjdk.org/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/invoke/InnerClassLambdaMetafactory.java (visited on 11/26/2024) (cit. on p. 25).

— (2024). *Oracle*. URL: https://docs.oracle.com/javase/specs/jvms/se23/html/index.html (visited on 12/29/2024) (cit. on p. 14).

Parr, Terence (1994). *History of PCCTS*. URL: http://aggregate.org/PCCTS/history.ps.Z (visited on 11/25/2024) (cit. on p. 7).

Sun (2006). *Sun Opens Java*. URL: https://web.archive.org/web/20070517164922/http://www.sun.com/2006-1113/feature/story.jsp (visited on 01/06/2025) (cit. on p. 14).