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

Kurzfassung

Peer-to-Peer-Netzwerke bieten eine Alternative zum klassischen Client-Server-Modell, um Daten auszutauschen. In Peer-to-Peer-Netzwerken kommunizieren alle Clients miteinander. Dadurch kann auf den Server als zentrale Schnittstelle verzichtet werden. Diese Charakteristik ermöglicht es Peer-to-Peer-Netzwerken zu funktionieren, obwohl einzelne Teilnehmer im Netzwerk ausfallen. Zudem nutzen Peer-to-Peer-Netzwerke die (meist bei traditionellem Filesharing ungenutzte) Upload-Bandbreite der einzelnen Clients.

Diese Bachelorarbeit setzt sich detailliert mit Peer-to-Peer-Netzwerken auseinander. Dabei werden zuerst bekannte Peer-to-Peer-Netzwerke vorgestellt und deren Charakteristiken erläutert. Weiters wird gezeigt, wo Unternehmen und Organisationen Peer-to-Peer-Netzwerke einsetzen. Unterschieden wird hierbei zwischen frei verfügbaren Netzwerken und von Unternehmen eigens entwickelten Netzwerken. Abschließend wird ein Client für das Netzwerk BitTorrent entwickelt. Dieser Client ist in der Lage, unter Verwendung des BitTorrent-Protokolls eine Datei von anderen Peers herunter- und hochzuladen. Dadurch wird gezeigt wie der Datenaustausch in einem Peer-to-Peer-Netzwerk auf technischer Ebene funktioniert und welche Technologien dazu benötigt werden.

Abstract

Peer-to-peer networks offer an alternative to the classic client-server model for exchanging data. In peer-to-peer networks, all clients communicate with each other. This means that the server, as the central element, can be omitted. This characteristic enables peer-to-peer networks to function even if individual participants in the network fail. In addition, peer-to-peer networks use the upload bandwidth of the individual clients, which is usually unused in traditional file sharing.

This bachelor thesis deals in detail with peer-to-peer networks. First, known peer-to-peer networks are introduced and their characteristics are explained. Then it is shown where companies and organisations utilize peer-to-peer networks. A distinction is made between freely available networks and networks developed by companies themselves. Finally, a client for the BitTorrent network is developed. This client is able to exchange a file with other peers using the BitTorrent protocol. This shows how data exchange in a peer-to-peer network works on a technical level and which technologies are required for this.

Chapter 1

Introduction

1.1 Motivation

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 runtime 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 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.

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.

1.2 Task and Goal

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 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 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 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 on which compilers are built upon. In comparison to natural languages, formal languages have a 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/

Regular Grammars

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 $A \rightarrow a|aB$. 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.

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, $S \times \Sigma \rightarrow S$
- F set of final states, $F \subseteq S$

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.

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

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 $A \rightarrow \beta$. Meaning that a non-terminal symbol A can be replaced by a sequence β containing terminal and non-terminal symbols or also ϵ, the empty sequence.

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 DFA, 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

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 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.

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 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 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 of the following steps:

- lexical analysis
- syntactic analysis
- semantic evaluation
- 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 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 action 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 focused 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.

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 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 $A \rightarrow a|ab$, the second alternative will never be matched, since backtracking always chooses the first alternative.

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 $A \rightarrow Bx, B \rightarrow Ay$. ϵ productions cause hidden left-recursion. Take a rule $B \rightarrow \epsilon$ that produces only the empty chain ϵ and another rule $A \rightarrow BA$. 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#.

2.2.3 Functionality

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.

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.

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

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.

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 action 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.

Abstract Syntax Tree (AST)

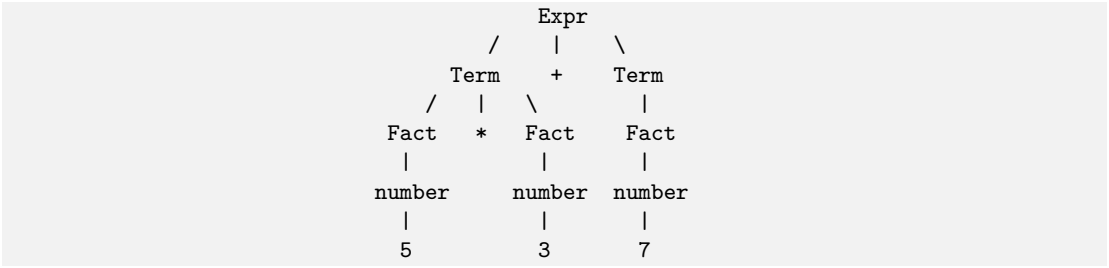
In an abstract syntax tree (AST) only a subset of the nodes from the original syntax tree are included. The goal is to focus on the semantic aspects of the syntax tree. Syntactical details, e.g., semicolons are omitted.

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.

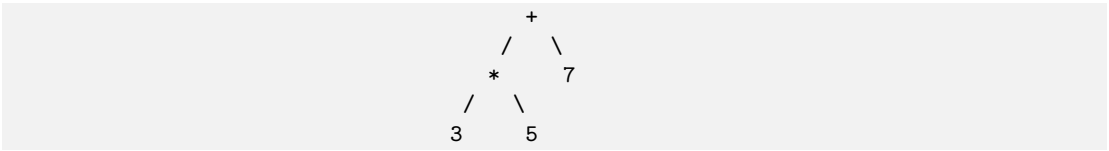
The AST is then used in the subsequent stages of a compiler. This transformation is performed to create a new tree which omits all information that is of no importance to the following stages of the compiler. Subsequent code optimization may further slim down the AST.

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.



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



the node count can be reduced from 14 to 5.

2.3.1 Visitor-Pattern for Tree Transformation

In the case that a syntax tree is already present, 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 taking care of this.

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 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. 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.

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

```
sealed class SyntaxTreeNode {
    abstract fun <T> accept(visitor: SyntaxTreeVisitor<T>): T
}

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)
}
```

2.3.2 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. 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 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.

Chapter 3

Java Virtual Machine (JVM)

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).

3.1 History

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 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 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.

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 program. While this increases the performance of the application, it 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.

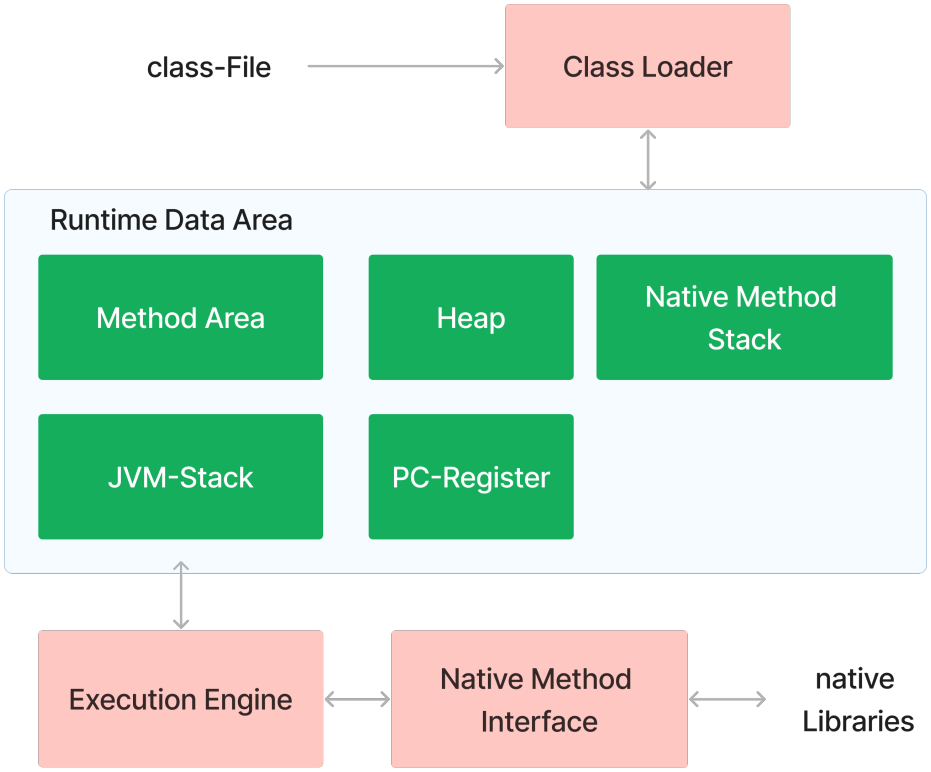


Figure 3.1: Architecture of the JVM.

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
- Attributes

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 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 require 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, 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. 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 application level classes. These are located via the classpath. Classes can be put on the classpath by using an environment variable or command line option.

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.

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.

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

3.2.3 Runtime Data Areas

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.

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
- 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

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

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)

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 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)

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

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.

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 65535 entries.

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`
- `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.

Load and Store Instructions

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

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

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 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 on 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.

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`.

3.3.3 Sample program

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

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).

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 (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).