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

software engineering
IKM Fakultät Hagenberg

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 understand. This allows the application developer to focus only 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. This flexibility saves the developer a lot of work, because they don't need to rewrite their application in the case they also want to target another operating system. Furthermore, there also exist compilers that target virtual machines like the Java Virtual Machine (JVM). Generating code for a virtual machine has the advantage that there don't need to be compilers written for every target operating system. 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. In this step the source code is read, and an abstract syntax tree (AST) is constructed. 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 machine code. The process of constructing the AST is based 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 parser and lexer construct a full parse tree from the input. From this an AST can be constructed using for example the visitor-pattern.

The AST functions then as the input for the backend of the compiler. In this section the machine code for the target system is generated. In the case of the JVM this is the so called Bytecode. The Bytecode could be written by hand, however this is rather difficult. For this a detailed understanding of the instruction set is needed, and the actual generation would have to be performed on a byte array. Therefore, 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. 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++. This cut down version of C++ is used at the university 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. Coco-2 includes both, the lexer and the parser. There is only one grammar-file required for the definition of the lexer and parser. Furthermore, semantic actions can also 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). ANTLR is 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 the options.

1.3 Theoretical Fundamentals

The following section provides an overview over the theoretical fundamentals of compiler construction.

1.3.1 Formal Languages and Compiler

Formal languages make up the fundament on which compilers are built upon. In comparison to natural languages, formal languages have a fixed 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 symbol A can be replaced by a chain α. The chain may contain terminal and non-terminal tokens.

Grammars can be classified according to the Chomsky hierarchy. Chomsky classifies 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 $A \rightarrow a|B$. This means that a non-terminal symbol A can only be replaced by either a terminal symbol a or another non-terminal symbol B. The only exception is the root rule S which can be replaced by the empty chain.

To recognize a sentence of a regular grammar a deterministic finite automaton (DFA) is needed. A DFA 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, $F \subset S$

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 has 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 in 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 chain β containing terminal and non-terminal symbols or also ϵ, the empty chain.

In a context-free grammar central recursion is possible (direct or indirect). This enables the nested structures that are needed for programming languages e.g. for expression hierarchies. Central recursion cannot be detected 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 how many k symbols look ahead while parsing are needed to deterministically decide on the next rule when using the *top-down* strategy.
- **LR(k)-Condition:** Defines how many k symbols look ahead while parsing are needed to deterministically decide on the next rule when using the *bottom-up* strategy.

The higher 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 read an arbitrary amount of symbols at the same time. 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 able of LR(k) grammars, is also able to recognize all 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 machine code for a given operating system and architecture, or also a virtual machine. Compiling code involves the following steps:

- Lexical Analysis
- Syntactic Analysis
- Semantic Evaluation
- Intermediate Language Generation
- Optimization
- Code Generation

The lexical analysis is the first step of the compilation. It reads the source code and classifies it. The goal is to group the individual characters into symbols. The grammar of the source language provides the information about the symbols. This part of the grammar is defined within the constraints of a regular grammar.

The symbols can be divided into terminal symbols and terminal classes. Terminal symbols are 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 as input to construct the syntax tree. The structure of the syntax tree is defined as a context-free grammar. 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.

During the syntactic analysis the semantic evaluation is performed. This includes 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 a semantic action associated with it, that gets 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.

Taking the AST as a basis, the intermediate language code generation is performed. This includes for example generating the symbol table.

Afterward, 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 the methods and tools that are available 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 Grammar

Using parser generators like ANTLR or Coco-2 requires the definition of the grammar in a specific format. These formats also allow for the declaration of semantic actions in the grammar. Semantic actions have access to the symbols (terminal and non-terminal) of a rule. Each symbol then has attributes associated with it. 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 in 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 « » 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 action 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 language. Parser generators like ANTLR provide multiple different target languages to generate a parser for.

Listing 2.1: Attributed Grammar in 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 the history of ANLR will be explained, followed by the introduction of the parsing algorithm currently employed by ANTLR, namely ALL(*). Finally, the general functionality of ANTLR is highlighted.

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 source under the BSD license. Its source code can be viewed on GitHub[1].

Many projects utilize ANTLR in their work. 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). Version2 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 falls back down using backtracking. However, a maximum amount of k lookahead tokens still needed to be specified. Version 3 also introduced ANTLRWorks[2],

[1] https://github.com/antlr/antlr4

[2] https://www.antlr3.org/works

a graphical IDE for the construction of ANTLR grammars.

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 number of lookahead tokens no longer need to be specified manually. ALL(*) will be explained in more detail in section 2.2.2. ANTLR v4 added support for the visitor and listener patterns, enabling easier interaction with the parsed data.

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 the basis 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 GLL (Scott and Johnstone 2010) and GLR (Tomita and Ng 1991) parsers. For ambiguous grammars these parsers return multiple parse trees, which are undesired for parsers of programming languages. This is due to them being 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.

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 context-free grammars. The only exception are grammars that contain indirect or hidden left-recursion[3].

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

[3] Indirect left-recursion is a rule like $A \to B, B \to A$. 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.

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 subparser needs to inspect the entire remaining input.

2.2.3 Functionality

References

Literature

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).

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).

Parr, Terence (1994). *History of PCCTS.* URL: https://github.com/apache/cassandra/blob/trunk/src/antlr/Parser.g/ (visited on 11/25/2024) (cit. on p. 7).