Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Entwicklung eines Compilers in Rust zu Lehrzwecken

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Daniel Bucher
geboren am: 18.10.1996
geboren in: Bonn

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Lars Grunske

eingereicht am: ..... verteidigt am: .....

Inhaltsverzeichnis

Abstract. Im Rahmen dieser Arbeit wurde ein modularer und erweiterbarer Interpreter in der Programmiersprache Rust entwickelt. Der Fokus lag bei der Entwicklung darauf, den Interpreter auch für die Erstellung weiterer Übungsinhalte im universitären Übungsbetrieb nutzen zu können. Zunächst hatten wir uns das Ziel gesetzt, einen vollständigen Compiler zu entwickeln. Im Laufe der Arbeit haben wir aber festgestellt, dass ein Interpreter die von uns gestellten Anforderungen besser erfüllen kann als ein Compiler. Der von uns entwickelte Interpreter wurde für eine eingeschränkte Variante der Sprache C, unter der Bezeichnung C1, entwickelt. Aus dieser Variante konnten wir im weiteren Verlauf der Arbeit fünf Übungsaufgaben ableiten, mithilfe derer Lernende schrittweise die Teilbereiche des Compilerbaus und deren Zusammenhänge verstehen können. Bei der Entwicklung wurde darauf geachtet, einfache Herangehensweisen zu nutzen, um auch Lernenden ohne großes Vorwissen in Rust den Einstieg in den Themenkomplex zu erleichtern. Des Weiteren haben wir im Verlauf der Arbeit die bisherigen Lösungsansätze präsentiert und diese mit dem neu entwickelten Verfahren aus dieser Arbeit verglichen. Dabei konnten wir zeigen, dass unser neuer Ansatz entscheidende Vorteile gegenüber den bisherigen Lösungsvarianten aufweisen kann und somit den Übungsbetrieb bereichern kann.

1. Einleitung

Diese Arbeit beschäftigt sich mit der Entwicklung eines Compilers in der Programmiersprache Rust. Der Compiler hat den Schwerpunkt darauf, als Grundlage für die Entwicklung von Lerninhalten genutzt zu werden. Zudem sollen aus dem Compiler fünf Übungsaufgaben ableitet werden, die als Übungen zur Vorlesung dienen können. Diese sollen den Studierenden die verschiedenen Konzepte des Compilerbaus näher bringen und verständlich vermitteln.
Gut ausgearbeitete Übungsaufgaben spielen eine entscheidende Rolle in der Lehre, da diese den Lernprozess wesentlich unterstützen und vertiefen können. Sie ermöglichen den Studierenden, das erlernte theoretische Wissen in praktische Anwendungen umzusetzen und so ein tieferes Verständnis zu entwickeln, was letztlich zu besseren Leistungen und einem tieferen Verständnis des Fachgebiets führt. Daher wurde bei der Entwicklung darauf geachtet, dass auch Lernende ohne Vorwissen in der Sprache Rust mit nur geringfügigem neu zu erwerbendem Wissen erfolgreich am Übungsbetrieb teilnehmen können.
In Folgenden werden wir zunächst die Gründe erläutert, die zur Neugestaltung des Übungskonzepts der Compilerbau-Vorlesung geführt haben. Zudem werden einige Grundlagen der theoretischen Informatik und des Compilerbaus definiert und beschrieben, um in den folgenden Kapiteln auf diesem Wissen aufbauen zu können.

1.1. Motivation

Der Bereich des Compilerbaus ist ein wichtiger Teilbereich der Softwareentwicklung. Compiler dienen als Schnittstelle, um Programme in einer höheren Programmiersprache wie Java, Rust oder C in Maschinencode (auch als Zwischencode bezeichnet) zu übersetzen. Dabei werden im Compilerbau viele grundlegende Inhalte der theoretischen Informatik praktisch angewandt [LSUA06]. Somit bieten die Prinzipien des Compilerbaus gute Anwendungsbeispiele für diese theoretischen Inhalte. Da mehrere Teilbereiche des Compilerbaus, wie die lexikalische oder syntaktische Analyse, auch häufig außerhalb dieses Themengebiets zum Einsatz kommen, vermittelt praktische Erfahrung im Compilerbau auch gleichzeitig ein besseres Verständnis für diese programmierbaren Konzepte [Hof22].
Aus diesen Gründen ist es uns wichtig, angehenden Informatikern ein umfangreiches und praxisbezogenes Wissen im Themengebiet des Compilerbaus zu vermitteln. Mit dieser Zielsetzung präsentieren wir in dieser Arbeit ein Übungskonzept, das die Konzepte des Compilerbaus sowie deren praktische Anwendung einsteigerfreundlich vermittelt. Für die Umsetzung in dieser Arbeit haben wir uns für die Sprache Rust entschieden.

Die noch recht junge Programmiersprache Rust [Rusg] erfreut sich unter Entwickelnden bereits seit mehreren Jahren großer Beliebtheit. Im Rahmen des *Stack Overflow Developer Survey* gaben im Jahr 2022 ca. 87% der Befragen Rust als ihre beliebteste Programmiersprache an [SODb]. Damit belegt Rust das siebte Jahr in Folge den ersten Platz und konnte seine Beliebtheit zum Jahr 2016 (mit ca. 79%) sogar weiter steigern [SODa]. Rust belegt im Jahre 2022 mit 17,6% ebenfalls den ersten Platz bei der Fragestellung nach dem Interesse eine Sprache zu lernen.
Diese hohe Beliebtheit verdankt Rust vermutlich der Tatsache, dass die Sprache im Vergleich zu etablierten Sprachen wie C und C++ einige Vorteile bietet [Jun20]. Rust legt einen besonders starken Fokus auf die Speichersicherheit. Neben sogenannten Raw-Pointern, die einem Pointer in C entsprechen, bietet Rust auch Referenzen. Diese Referenzen werden vom Rust-Compiler überwacht. Dadurch kann Rust bestimmte Eigenschaften für Referenzen garantieren. Beispielsweise können Referenzen nur auf gültigen Speicher zeigen und niemals den Wert `null` annehmen. Der Rust-Compiler stellt sicher, dass eine Referenz nicht auf ein Objekt zeigt, das außerhalb seiner Gültigkeitsdauer überlebt oder dass ein Objekt nicht verändert wird, solange weitere Referenzen auf das Objekt existieren. Mit dem System der Referenzen bietet Rust eine sichere Zugriffsmöglichkeit auf den Speicher [Rusd]. Dank des Systems von Ownership und Borrowing für Referenzen ermöglicht Rust sichere Nebenläufigkeit ohne die Gefahr von Race-Conditions [Rusa]. Da Rust bereits zur Übersetzungszeit sicherstellt, dass keine Race-Conditions bei Referenzen auftreten können, werden lockfreie, gleichzeitige Zugriffe auf gemeinsam genutzte Ressourcen ermöglicht.
Darüber hinaus bietet Rust mit der standardmäßig mitgelieferten Projektverwaltung `Cargo` eine einfache Verwaltung von Projekten [Ruse]. Abhängigkeiten werden dabei vom Compiler aufgelöst und verwendete Bibliotheken werden automatisch von der

offiziellen Paket-Repository crates.io heruntergeladen [Rusb].
Aufgrund dieses hohen Sicherheitsniveaus und der damit verbundenen veränderten Herangehensweise an herkömmliche Probleme (sichere dynamische Speicherverwaltung) bietet Rust eine interessante Grundlage für dieses Projekt. Lernende können die Konzepte des Compilerbaus erlernen und in einer sicheren Programmierumgebung anwenden, die dennoch viele Freiheiten bei der Nutzung von Ressourcen bietet.
Nach der Erläuterung, aus welchen Gründen wir uns für das Thema des Compilerbaus und die Verwendung der Sprache Rust entschieden haben, klären wir in folgenden Abschnitten grundlegende Definitionen, um die Konzepte des Compilerbaus besser verstehen zu können.

1.2. Definitionen

In diesem Teilabschnitt wollen wir zunächst einige Grundlagen der theoretischen Informatik definieren und erläutern. Diese dienen als Grundlage, um auch den Aufbau und die Funktionsweise von Compilern verstehen zu können. Zu Beginn werden die Begriffe *Alphabet*, *Wort*, *Sprache* und *Grammatik* definiert. Anschließend betrachten wir die Klassifizierung von Sprachen gemäß der Chomsky-Hierarchie, wobei wir insbesondere auf die regulären und kontextfreien Sprachen eingehen. Im Anschluss erläutern wir die Grundlagen des Compilerbaus sowie den Aufbau von Compilern und Interpretern.

1.2.1. Theoretische Grundlagen des Compilerbaus

Ein **Alphabet** bezeichnet in der theoretischen Informatik eine geordnete Menge endlich vieler *Zeichen*. Ein *Alphabet* wird mit $\Sigma = \{a_1, \ldots, a_m\}$ bezeichnet.
Eine Folge $x = x_1 \ldots x_n \in \Sigma^n$ von n *Zeichen* bildet das *Wort* x mit der Länge n. Ein *Wort* kann auch die Länge $n = 0$ haben. In diesem Fall wird es als *leeres Wort* ϵ bezeichnet. Die Menge aller *Wörter* über einem *Alphabet* Σ wird als Σ^* bezeichnet, definiert als

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$$

Eine Teilmenge aller *Wörter* eines *Alphabets* $L \subseteq \Sigma^*$ wird als **Sprache** bezeichnet.
Zur Beschreibung von *Sprachen* führen wir **Grammatiken** ein. Eine *Grammatik* ist ein 4-Tupel $G = (V, \Sigma, P, S)$, wobei

- V eine endliche Menge von Variablen (auch Nichtterminalsymbole genannt),
- Σ das *Alphabet* (auch Terminalsymbole oder Terminale),
- $P \subset (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ eine endliche Menge von Regeln (auch Produktionsregeln) und
- $S \in V$ die Startvariable ist.

Eine Produktionsregel ist eine Relation, bei der eine nicht leere Folge von Variablen oder Terminalen auf eine Folge von Variablen oder Terminalen abgebildet wird.
Für $\alpha, \beta \in (V \cup \Sigma)^*$ sagen wir, dass β in einem Schritt aus α ableitbar ist, falls eine Regel $u \to v$ und Wörter $l, r \in (V \cup \Sigma)^*$ gibt, sodass $\alpha = lur$ und $\beta = lvr$. Wir schreiben diese Ableitung als $l\underline{u}r \to lvr$. Eine *Grammatik* G erzeugt eine *Sprache*, die alle *Wörter* enthält, die durch die Anwendung der Ableitungsregeln aus G, beginnend mit der Startvariable, erzeugt werden können.
Nach der Betrachtung von *Sprachen* und ihrer Konstruktion klassifizieren wir *Grammatiken* anhand von Kriterien bzw. Einschränkungen. Hierfür nutzen wir die Chomsky-Hierarchie, die Grammatiken in 4 Klassen unterteilt (gezählt von 0 bis 3) [Cho56]. Sprachen, welche von Grammatiken der dritten und am stärksten eingeschränkten Klasse erzeugt werden, heißen **reguläre Sprachen**. Eine Grammatik G heißt genau dann regulär (oder vom Typ 3), falls für alle Regeln $u \to v$ gilt $u \in V$ und $v \in \Sigma V \cup \Sigma \cup \{\epsilon\}$. Außerdem gilt, dass eine Sprache dann regulär ist, wenn sie von einem deterministischen endlichen Automaten (kurz: DEA, oder DFA für für *deterministic finite automaton*) akzeptiert wird. Ein DFA ist ein Modell aus der theoretischen Informatik [Cho56].
Ein endlicher Automat besteht aus Zuständen, zwischen denen mittels vorgegebener Transitionen gewechselt werden kann. Formal ist ein **deterministischer endlicher Automat** als 5-Tupel $M = (Z, \Sigma, \delta, q_0, E)$ beschrieben, wobei

- $Z \neq \emptyset$ eine endliche Menge von Zuständen,
- Σ das Eingabealphabet,
- $\delta : Z \times \Sigma \to Z$ die Überführungsfunktion,
- $q_0 \in Z$ der Startzustand,
- $E \subseteq Z$ die Menge der Endzustände ist.

Als Eingabe wird ein Wort mit endlicher Länge erwartet. Mit Beginn im Startzustand wird nun das Maschinenmodell durchlaufen. Durch Konsumieren des aktuellen Zeichens der Eingabe kann durch eine passende Regel vom aktuellen Zustand aus eine in δ vorgegebene Transition in einen anderen Zustand durchgeführt werden. Befindet man sich nach dem Konsumieren der vollständigen Eingabe in einem der definierten Endzustände, so wird das Wort akzeptiert und befindet sich in der vom Automaten beschriebenen Sprache.
Neben einem DFA kann als Modell auch ein **nichtdeterministischer endlicher Automat** (kurz: NFA für *nondeterministic finite automaton*) $N = (Z, \Sigma, \Delta, Q_0, E)$ genutzt werden. Dieser ist ähnlich aufgebaut wie ein DFA, jedoch gibt es mehrere Startzustände (zusammengefasst als $Q_0 \subseteq Z$) und die Überführungsfunktion hat die Form $\Delta : Z \times \Sigma \to \mathcal{P}(Z)$, wobei $\mathcal{P}(Z)$ die Potenzmenge (die Menge aller Teilmengen) von Z ist. Ein NFA kann bei einer Eingabe mehrere Wege parallel durchlaufen. Ein Wort x gehört genau dann zur Sprache $L(N)$, wenn mindestens ein Weg nach vollständigem Konsumieren der Eingabe einen Endzustand erreicht.
Im Gegensatz zu einem DFA kann ein NFA auch “Stecken bleiben“. Dies tritt auf,

wenn er in einen Zustand q gelangt, von dem aus es keine Transition gibt, welche das nächste Zeichen der Eingabe liest [Hof22].
Neben DFAs und NFAs wollen wir nun noch ein weiteres Modell betrachten, mit dem reguläre Sprachen beschrieben werden können. **Reguläre Ausdrücke** sind Zeichenketten beliebiger Länge über dem Alphabet, die mit den Operatoren Vereinigung, Produkt und Sternhülle die vollständige Struktur aller Wörter einer Sprache beschreiben können. Diese werden formal wie folgt definiert:
Ein regulärer Ausdruck γ und die durch γ dargestellte Sprache $L(\gamma)$ ist wie folgt definiert. Die Symbole $\emptyset$, ϵ und a ($a \in \Sigma$) sind reguläre Ausdrücke, die

- die leere Sprache $L(\emptyset) = \emptyset$,
- die Sprache $L(\epsilon) = \{\epsilon\}$ und
- für jedes Zeichen $a \in \Sigma$ die Sprache $L(a) = \{a\}$

beschrieben. Sind α und β reguläre Ausdrücke, die die Sprachen $L(\alpha)$ und $L(\beta)$ beschreiben, so sind auch $\alpha\beta$, $(\alpha|\beta)$ und (α^*) reguläre Ausdrücke, die die Sprachen

- $L(\alpha\beta) = L(\alpha)L(\beta)$,
- $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$ und
- $L(\alpha^*) = L(\alpha)^*$

beschreiben. Da eine reguläre Sprache sowohl von einem endlichen Automaten als auch von einem regulären Ausdruck beschrieben werden kann und umgekehrt, folgt daraus, dass endliche Automaten und reguläre Ausdrücke über eine äquivalente Ausdrucksstärke verfügen. Äquivalent dazu folgt, dass für jede reguläre Sprache ein äquivalenter endlicher Automat und ein ebenfalls äquivalenter regulärer Ausdruck existieren [Hof22].

Als Nächstes wollen wir Grammatiken vom Typ-2 der Chomsky-Hierarchie betrachten. Eine Grammatik G heißt **Typ-2** oder **kontextfrei** (kurz: CFG für *context free grammar*), falls für alle Regeln $u \to v$ gilt $u \in V$. Somit fällt die Einschränkung der rechten Seite der Ableitungsregeln im Vergleich zu den regulären Grammatiken weg. Daraus können wir folgern, dass die Sprachklasse der regulären Sprachen in der Klasse der kontextfreien Sprachen enthalten ist. Auch in dieser Sprachklasse kann ein Automatenmodell genutzt werden, um eine solche kontextfreie Sprache zu beschreiben [Cho56].
Um auch kontextfreie Sprachen abbilden zu können, müssen wir unser Modell des endlichen Automaten erweitern. Das hier genutzte Automatenmodell wird als **Kellerautomat** (ES-PDA für *empty stack push down automaton* oder PDA für *push down automaton*) bezeichnet. Dieses Modell verfügt über einen Speicherstack (auch Keller genannt), in welchem sich Zeichen eines separaten Alphabets gemerkt werden können. Ein Wort wird dann von einem Kellerautomaten akzeptiert, wenn nach vollständigem Konsumieren der Eingabe der Kellerspeicher geleert wurde. Durch diese Erweiterung des Automatenmodells kann zum Beispiel die Sprache $L = \{a^n b^n | n \geq 0\}$ erkannt

werden [Hof22].
Der Vollständigkeit halber wollen wir als Abschluss der Chomsky-Hierarchie noch einen kurzen Blick auf **Typ-1** und **Typ-0** Grammatiken werfen. Diese werden im weiteren Verlauf dieser Arbeit nicht benötigt. Grammatiken vom Typ-1 werden als kontextsensitive Grammatiken bezeichnet. Für Grammatiken dieses Typs gilt die Einschränkung, dass für alle Regeln $u \to v$ gilt: $|u| \geq |v|$, mit Ausnahme einer Sonderregel für ϵ. Für Grammatiken vom Typ-0 hingegen gelten keine Einschränkungen mehr. Sprachen, die von einer Typ-0 Grammatik konstruiert werden, heißen auch rekursiv aufzählbare Sprachen [Cho56].
Zum Abschluss der theoretischen Grundlagen wollen wir noch zwei Varianten zur Notation von Typ-2 Grammatiken besprechen. Zur Beschreibung der Grammatik von Programmiersprachen wurde die **Backus-Naur-Form** (kurz: BNF) entwickelt [Bac80]. Ableitungsregeln werden dabei wie in Listing 1 dargestellt. Zur besseren Verständlichkeit und Lesbarkeit einer Grammatik in BNF-Darstellung wurde diese zur **erweiterten Backus-Naur-Form** (kurz: EBNF) erweitert [Wir77]. Dabei wurden vereinfachte Schreibweisen hinzugefügt. In EBNF kann ein optionaler Teil einer Regel mit `[...]` oder `(...)?` dargestellt werden. Wiederholungen von Teilen einer Regel werden als `{...}` oder als `(...)+` (für mindestens einmal) bzw. als `(...)*` (für beliebig häufig) geschrieben. Beide Darstellungen sind dabei gleich mächtig und unterscheiden sich nur in der Darstellung von Regeln [Wir77].

```
<expression> ::= <term> '+' <expression>
               | <term>
<term>       ::= <factor> '*' <term>
               | <factor>
<factor>     ::= '(' <expression> ')'
               | <number>
<number>     ::= [0-9]+
```

Listing 1.: Beispiel von Grammatikregeln in BNF Darstellung.

Nachdem wir nun die theoretischen Grundlagen besprochen haben, wollen wir uns im Folgenden den Grundlagen des Compilerbaus zuwenden.

1.2.2. Grundlagen eines Compilers

Als Compiler werden im Allgemeinen Programme bezeichnet, welche Quellcode als Eingabe erhalten und diesen in für Computer verständlichen Maschinencode oder eine andere gewählte Zielsprache übersetzen [LSUA06]. Im Rahmen des Compilerbaus sind auch Interpreter angesiedelt, die sich größtenteils im Aufbau mit Compilern decken. Interpreter übersetzen den Quellcode jedoch nicht, sondern interpretieren den Code direkt zur Laufzeit.
Zur Veranschaulichung der Struktur eines Compilers/Interpreters haben wir diese

in Abbildung 1 dargestellt. In den folgenden Abschnitten werden wir auf die fünf Inhaltsblöcke des Front- und Back-Ends eingehen, ihre jeweiligen Aufgaben erläutern und die Funktionsweise beschreiben.

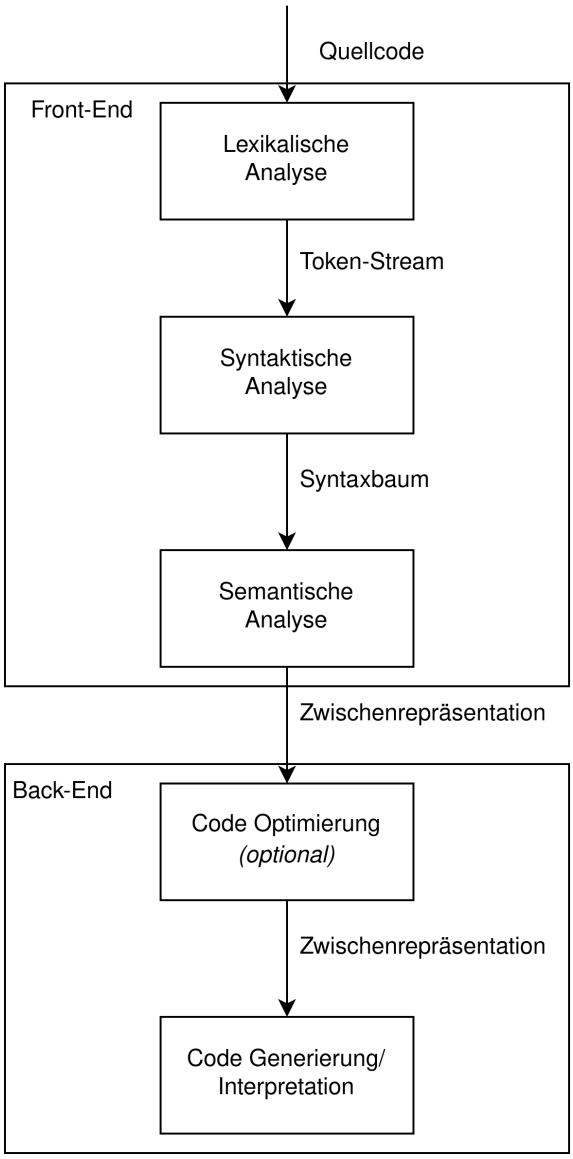

Abbildung 1: Grafische Darstellung eines Compilers/Interpreters. Der Aufbau entspricht hierbei der von Lam et al. Struktur aus [LSUA06].

Die **lexikalische Analyse** bildet den ersten Teil eines Compilers [Wir13]. Dabei wird der Programmcode des eingegebenen Quellprogramms analysiert und in vordefinierte Zeichenketten zerteilt. Diese Zeichenketten werden auch als Tokens bezeichnet. Programme, die diese Aufgabe erfüllen, werden als Lexer, Tokenizer oder Scanner bezeichnet. Im Folgenden betrachten wir den Aufbau und die Funktionsweise eines

solchen Lexers.
Um einen Lexer aufzubauen, werden zu Beginn Tokens benötigt, die vom Lexer erkannt werden sollen. Damit die Beschreibungen der Tokens ausreichend umfangreich sein können, werden diese in Form von regulären Ausdrücken angegeben. Bei der Konstruktion eines Lexers werden zunächst aus den regulären Ausdrücken NFAs erstellt. Nach festgelegten Vorgehensweisen können die verschiedenen Operatoren von regulären Ausdrücken (Alternative, Verkettung und Sternhülle) in Teil-NFAs umgewandelt werden. Nach der Erstellung der NFAs für die einzelnen Token werden diese zu einem neuen NFA kombiniert. Anschließend wird dieser NFA zum Beispiel mit dem Verfahren der Potenzmengenkonstruktion in einen DFA umgewandelt und minimiert [Hof22]. Dieser DFA bildet nun den Lexer, welcher das Quellprogramm als Eingabe verwendet. Sobald im DFA ein Endzustand erreicht wird, erzeugt der Lexer ein Token für die erkannte Zeichenkette, und der DFA kehrt in den Startzustand zurück, um die nächste Zeichenfolge zu analysieren. Falls nach dem Verarbeiten eines Zeichens der Eingabe kein Endzustand erreichbar ist, wird in den meisten Fällen ein Fehler-Token erzeugt. Fehler, die im Lexer auftreten, werden als lexikalische Fehler bezeichnet [JZ08]. Auf diese Weise wandelt der Lexer die Zeichenfolge des eingegebenen Quelltexts in eine Abfolge von Tokens um. Diese Tokenfolge wird abschließend an den zweiten Teil des Compilers weitergegeben.

Bei der **syntaktischen Analyse**, dem zweiten Teil eines Compilers, soll jetzt die Struktur einer Eingabe analysiert werden. Programme, die diese Aufgabe übernehmen, werden als Parser bezeichnet. Parser verwenden eine vorwiegend kontextfreie Grammatik, um damit die Struktur einer Eingabe zu überprüfen. Als Ausgabe gibt ein Parser im Falle eines Compilers überwiegend einen Syntaxbaum zurück, welche die Syntax der Sprache widerspiegelt. Im Gegensatz zu einem Lexer gibt es verschiedene Ansätze für einen Parser. Dabei wird bei Parsern unterscheiden, aus welcher Richtung eine Eingabe gelesen wird, in welcher Richtung abgeleitet wird und wie viele Zeichen vorausgeschaut werden können. Die Menge der Zeichen, die vorausgesehen werden können, wird als *Look-Ahead* bezeichnet. Die Beschreibung eines Parsers gibt die Leserichtung, die Ableitungsrichtung und Look-Ahead Länge. Zum Beispiel steht LR(1) für einen Parser, der von links nach rechts liest, rechts ableitet und ein Zeichen Voraussicht hat [NLNL94]. Im Folgenden betrachten wir einige der gängigsten Parservarianten.

LL(1)-Parser Der LL(1)-Parser (Look-Ahead Left-to-right Leftmost Derivation Parser) ist eine Variante eines Top-Down-Parsers mit einem Look-Ahead-Zeichen. Im Gegensatz zu einem Bottom-Up-Parser leitet ein Top-Down-Parser die Eingabe anhand der Grammatik vom Startsymbol zu den Terminalen hin ab. Dabei wird von der Startregel ausgehend versucht, die Eingabe durch Anwendung der Produktionsregeln zu parsen. Der Parser entscheidet bei jeder Regel Schritt für Schritt, basierend auf dem aktuellen Zeichen der Eingabe und den Look-Ahead-Zeichen, welche Regelvariante als nächstes angewendet wird. LL-Parser lassen sich leicht konstruieren,

da keine Entscheidungen für Shift-and-Reduce getroffen werden müssen. Aufgrund dieser Art der Ableitung ist jedoch die Menge der möglichen Grammatiken begrenzt. Die Klasse der Grammatiken, welche durch einen LL-Parser abgeleitet werden kann, unterliegt den Einschränkungen, dass sie links-rekursivitätsfrei sein muss und ebenfalls (abhängig von der Größe der Lookahead-Menge) nur begrenzt mehrdeutig sein darf [NLNL94]. Aufgrund dieser Einschränkungen haben LR-Parser einen Vorteil gegenüber LL-Parsern, da LR-Parser uneingeschränkt kontextfreie Grammatiken ableiten können.

LR(1)-Parser Der LR(1)-Parser (Look-Ahead Rightmost Derivation Parser) ist eine Variante eines Bottom-up-Parsers mit einem Look-Ahead-Zeichen. Bei einem Bottom-up-Parser beginnt der Parsevorgang bei den Terminalen der Grammatik und arbeitet sich von dir dort schrittweise zur Startregel hoch. Der Parser kombiniert dabei Teilausdrücke in der Eingabe und fasst sie zu größeren Konstrukten zusammen, bis schließlich das Startsymbol der Grammatik erreicht wird. Dabei erkennt der Parser Muster (Folgen von Symbolen auf der rechten Seite von Regeln) in der Eingabe und reduziert sie durch Anwendung von Grammatikregeln. Hierbei kommt das Verfahren *Shift-and-Reduce* zum Einsatz. Dieses Verfahren legt ein Zeichen der Eingabe entweder auf einen *Stack* (Shift-Operation) oder führt das aktuelle Zeichen mit einer Menge von Symbolen auf dem *Stack* durch die entsprechende Regel zu einem Nichtterminalsymbol zusammen (Reduce-Operation) [NLNL94]. Durch das Vorhandensein eines Look-Ahead-Zeichens hat der Parser die Möglichkeit, bessere Entscheidungen zu treffen und so möglicherweise fehlerhafte Ableitungen zu vermeiden.

LALR(1)-Parser Der LALR(1)-Parser (Look-Ahead LR(1)) basiert auf der Analysemethode eines LR(1) Parsers und verwendet einen erweiterten Vorausblick. Wie ein LR-Parser ist ein LALR-Parser ein Bottom-Up-Parser, der das Shift-Reduce-Verfahren anwendet. Im Vergleich zu einem herkömmlichen LR-Parser verwendet ein LALR-Parser jedoch eine effizientere Datenstruktur, um die Zustandsmenge zu komprimieren. Dadurch benötigt der Parser weniger Speicherplatz und Rechenzeit. Ein LALR-Parser ist eine gängige Wahl, für viele Compiler und Parser-Generatoren [LSUA06]. Es kann jedoch vorkommen, dass ein LALR-Parser bei bestimmten Grammatiken auf Konflikte oder Mehrdeutigkeiten stößt. In solchen Fällen kann es erforderlich sein, die Grammatik anzupassen oder eine andere Analysemethode zu wählen, um solche Probleme zu lösen.

Der dritte Teil eines Compilers ist die **semantische Analyse**. Hierbei wird die Struktur des Syntaxbaums, den der Parser erzeugt hat, überprüft und in eine zuvor gewählte Zwischenrepräsentation umgewandelt. Der Schwerpunkt liegt darauf, die Semantik des Programms auf Korrektheit zu überprüfen. Die semantische Analyse umfasst unter anderem die folgenden Aufgaben:

1. Die **Typenüberprüfung** stellt sicher, dass Variablen, Ausdrücke und Funktionen die korrekten Typen aufweisen [WSH13]. Dabei werden eventuelle

Typenfehler erkannt. Ebenso werden im Rahmen dieser Aufgabe Typenkonvertierungen auf Verträglichkeit geprüft. Häufig werden dabei auch implizite Typumwandlungen durch explizite Casts präzisiert [LSUA06].

2. Die **Deklarationsüberprüfung** stellt sicher, dass Variablen und Funktionen korrekt deklariert wurden [WSH13]. Dabei wird unter anderem gewährleistet, dass keine doppelten Deklarationen auftreten und dass keine Variablen vor ihrer Deklaration verwendet werden.

3. Die **Überprüfung von Sichtbarkeitsbereichen (Scopes)** stellt sicher, dass Variablen im richtigen Sichtbarkeitsbereich verwendet werden [WSH13]. Je nach Implementierung können Variablen andere Variablen in darüber liegenden Sichtbarkeitsbereichen mit demselben Namen überdecken. Dies wird als *Shadowing* bezeichnet.

4. Bei der **Auswertung von Konstanten** werden konstante Ausdrücke vereinfacht, um die Effizienz des Codes zu verbessern. Dabei werden auch Konstanten in Termen ersetzt [Muc97]. Dieser Teilbereich der semantischen Analyse kann auch in die Optimierung einfließen.

5. Im letzten Schritt der semantischen Analyse wird der bisher verwendete Syntaxbaum erweitert und in die gewählte Zwischenrepräsentation umgewandelt, die in den nachfolgenden Schritten weiterverwendet wird.

Bei der semantischen Analyse dient die sogenannte **Symboltabelle** als Hilfsmittel. In dieser Struktur werden verschiedene Informationen über Variablen, Funktionen und Typen (zusammengefasst als Symbole) gespeichert [CT11]. Allgemeine Informationen zu den Symbolen können wie folgt aussehen:

- Name des Symbols: Dies ermöglicht die Identifikation des Symbols in der Symboltabelle und erkennt Namenskonflikte.

- Typ des Symbols: Diese Information kann sowohl zur Berechnung des zukünftigen Speicherbedarfs verwendet werden als auch zur Durchführung der Typenüberprüfung.

- Sichtbarkeitsbereich des Symbols: Diese Information ermöglicht die Bestimmung des Gültigkeitsbereichs einer Variable und an welcher Stelle auf dieses Symbol zugegriffen werden kann.

Je nach Programmiersprache können auch weitere Informationen, wie die Anzahl der Zeiger auf ein Symbol, in der Symboltabelle hinterlegt werden.
Mit der semantischen Analyse ist die Korrektheitsprüfung der Eingabe abgeschlossen. Gleichzeitig endet mit diesem Schritt der Aufgabenbereich des Front-Ends. Die erzeugte Zwischenrepräsentation wird in den folgenden Schritten weiterverarbeitet. Im nächsten beginnen wir mit der Besprechung der Aufgaben des Back-Ends. Eine

optionale Stufe dabei ist die Codeoptimierung.

Die **Codeoptimierung**, welche den vierten Teilabschnitt eines Compilers darstellt, beinhaltet die Verbesserung der von der semantischen Analyse zurückgegebenen Struktur hinsichtlich Leistung und Effizienz. Dieser Schritt ist optional und nicht erforderlich, um den Code zu interpretieren oder in Code der Zielsprache umzuwandeln. Während der Optimierung werden verschiedene Transformationen und Analysen auf den aktuellen Code angewandt. Die Optimierung kann in verschiedene Phasen unterteilt werden:

- Ähnlich zur semantischen Analyse kann in dieser Phase die Auflösung von konstanten Inhalten durchgeführt werden [Muc97]. Dies umfasst die Auflösung von Ausdrücken, die konstante Terme enthalten, um diese Berechnungen nicht zur Laufzeit durchführen zu müssen.

- In einem weiteren Schritt der Optimierung wird redundanter Code entfernt bzw. optimiert [Muc97]. Zum Beispiel werden ausgeführte Operationen in temporären Variablen gespeichert, um dieselben Berechnungen nicht mehrfach zur Laufzeit durchführen zu müssen.

- Im dritten Schritt erfolgt die Auflösung von konstanten Variablen. Dabei werden Variablen identifiziert, die zur Laufzeit konstante Werte annehmen. Diese Variablen werden durch ihre konstanten Werte ersetzt [Muc97], was die Laufzeit des Programms verbessern kann.

- Neben der Reduzierung von redundantem Code wird auch nicht verwendeter Code optimiert [Muc97]. Nicht erreichbare Codesegmente oder ungenutzte Variablen werden entfernt, was die Ausführungszeit des generierten Codes verkürzen kann.

- Eine weitere Optimierungsphase konzentriert sich auf Schleifen. Ziel ist es, die Anzahl der Schleifendurchläufe zu reduzieren. Techniken wie Schleifenentfaltung (Loop Unrolling), Schleifenverschmelzung (Loop Fusion) oder Verschieben von invariantem Code (Loop-Invariant-Code-Motion) werden angewandt [Muc97]. Die Optimierung von Schleifen kann die Laufzeit eines Programms erheblich verkürzen.

- Im letzten präsentierten Schritt wird, wenn möglich und sinnvoll, Inline-Code eingesetzt [Muc97]. Dadurch können die Kosten für Funktionsaufrufe reduziert werden.

Durch diese und weitere Verfahren kann die Effizienz des Codes gesteigert und die Laufzeit sowie der Speicherverbrauch verbessert werden. Viele dieser Verfahren erfordern eine gründliche Analyse des Programms, um Verbesserungen durchzuführen, ohne die Funktionalität des Ursprungsprogramms zu beeinträchtigen. Im nächsten und letzten Schritt betrachten wir die Interpretation bzw. die Umwandlung des Codes

in die Zielsprache.

Der letzte Teilschritt eines Compilers besteht in der Umwandlung der zuvor in den vorherigen Schritten erstellten Struktur in Maschinencode. In diesem Zusammenhang betrachten wir auch eine Alternative zur **Codegenerierung**, nämlich die direkte **Interpretation**.
Ein Interpreter führt den Code direkt aus, anstatt ihn in eine Zielsprache umzuwandeln [LSUA06]. Ein bekanntes Beispiel für einen Interpreter ist die Java Virtual Machine (JVM). Im Gegensatz zu Compilern können Interpreter oft bessere Fehlerdiagnosen stellen, da sie die Anweisungen des Quellprogramms Schritt für Schritt ausführen. Allerdings kann das einmalig erzeugte Zielprogramm eines Compilers in der Regel deutlich schneller ausgeführt werden als der wiederholte Durchlauf eines Interpreters [LSUA06].
Bei der Codegenerierung wird der zuvor durch das Front-End oder die Codeoptimierung vorbereitete Zwischencode in Maschinencode oder eine andere Zielsprache umgewandelt [LSUA06]. In diesem Prozess werden den Variablen Speicherregister des Prozessors zugewiesen. Dabei ist das Ziel, die Anzahl der benötigten Register zu minimieren, um die Registerverwaltung zu optimieren. In einem weiteren Schritt wird der Zwischencode analysiert, um Anweisungen (Befehle) in der entsprechenden Zielsprache zu generieren [LSUA06]. Ebenso ist die Speicherverwaltung ein integraler Bestandteil der Codegenerierung. Hierbei müssen Speicherplätze für Variablen und Datenstrukturen verwaltet werden. Die Wahl der Speicherverwaltungstechnik kann auch von den Eigenschaften des Zielsystems abhängen. Während des gesamten Kompilierungsprozesses kann der Zwischencode weiter optimiert werden, wobei meist Optimierungsstrategien angewandt werden, die speziell auf die jeweilige Zielsprache zugeschnitten sind [LSUA06]. Im letzten Schritt der Codegenerierung wird die Ausgabedatei erzeugt. Diese Datei kann anschließend weiterverarbeitet oder auf dem Zielsystem ausgeführt werden.
Mit Abschluss dieses Schritts haben wir das Konzept und die Vorgehensweisen eines Compilers erläutert. Im nächsten Teilabschnitt dieser Arbeit werden wir die Problemstellung erläutern, der sich diese Arbeit im weiteren Verlauf widmen wird.

1.3. Problemstellung

Ursprünglich wurde für die praktischen Übungen zur Compilerbau-Vorlesung ein Übungskonzept in der Programmiersprache C entwickelt. Zum Sommersemester 2022 wurde dieses bestehende Übungskonzept nahezu vollständig in die Programmiersprache Rust überführt, wie im Abschnitt 2.2 dargestellt wird. Eine grundlegende Überarbeitung und Neustrukturierung wurde dabei nicht durchgeführt.
Wie im folgenden Kapitel 2 näher erläutert wird, weisen sowohl das ursprüngliche Übungskonzept in C als auch das daraus abgeleitete Konzept in Rust einige Probleme auf. Der Rust-Ansatz nutzt die Vorzüge der Sprache nur eingeschränkt aus. Durch eine umfassende Neuentwicklung des Übungskonzepts können die Stärken von Rust besser genutzt werden, um den Lernfokus bei der Bearbeitung der Übungsaufgaben

gezielt auf die Kernaspekte des Compilerbaus zu lenken. An dieser Stelle setzt unser im Rahmen dieser Arbeit entwickeltes Projekt an. Unser Ziel ist es, durch eine Bedarfsanalyse basierend auf den bisherigen Übungskonzepten und der Formulierung weiterer Entwurfskriterien ein strukturiertes und durchdachtes neues Übungskonzept für die Compilerbau-Vorlesung zu erstellen.

1.4. Aufbau der Arbeit

In diesem Abschnitt der Arbeit haben wir zu beginn die zunächst unsere Motivation erläutern, aus welchen Gründen wir im Rahmen dieser Arbeit das Übungskonzept der Compilerbau-Vorlesung neu gestalten wollen. Zudem Haben wir einige Grundlagen der theoretischen Informatik und des Compilerbaus definiert und beschrieben.
Im Abschnitt 2 wird der aktuelle Zustand des Übungsbetriebs, sowie die bisher genutzten Ansätze in den Sprachen C und Rust präsentiert und detailliert erläutert. Unter Berücksichtigung der bisherigen Verfahren werden Kriterien erarbeitet, die Grundlage für die Entwicklung unseres neuen Ansatzes dienen sollen. Im Kapitel 3 wird die Entwicklung unseres neu entwickelten Interpreters und der daraus abgeleiteten Übungsaufgaben erläutert. Dabei werden auch unsere Entscheidungen während der Entwicklung und aufgetretene Probleme besprochen. Diese Erklärungen werden durch Beispiele veranschaulicht. Anschließend werden in Abschnitt 4 die bisherigen Lösungsverfahren, die zuvor in Kapitel 2 präsentiert wurden, mit unserem neuen Ansatz aus Kapitel 3 verglichen. Dabei werden wir sowohl die erarbeiteten Bewertungskriterien als auch unsere Entwurfskriterien berücksichtigt. Die Unterschiede sowie gemeinsame Ansätze werden anhand von Beispielen verdeutlicht. Abschließend wird in Kapitel 5 zusammengefasst, was wir im Rahmen dieser Arbeit erreichen werden. Zudem wird ein Ausblick auf zukünftige Verbesserungsmöglichkeiten und Erweiterungen gegeben.
Im folgenden zweiten Teil der Arbeit werden wir die bisherigen Ansätze aus der Compilerbau-Übung sowie weitere Einflüsse genauer betrachten und analysieren. Dadurch sollen der benötigte Umfang sowie wichtige Eckpunkte für eine neu strukturierte Compilerbau-Übung erarbeitet werden.

2. Aktuelle Praxis der Compierbau-Vorlesung

In diesem Abschnitt der Arbeit werden wir näher auf den Umfang der bisherigen Programmieraufgaben eingehen. Unser Ziel ist es, durch diese Analyse herauszufinden, welche Aspekte für die Umsetzung eines neuen Übungskonzepts in der Compilerbau-Vorlesung besonders wichtig sind und welche weniger wichtig erscheinen. Wir beginnen dieses Kapitel mit einer Betrachtung der Zielgruppe der Compilerbau-Übung. Hierzu werden wir den empfohlenen Studienverlaufsplan des Studiengangs sowie die Modulbeschreibungen relevanter Veranstaltungen heranziehen. Im nächsten Schritt konzentrieren wir uns genauer auf die beiden bisherigen Lösungsansätze in C und Rust. Wir werden die Inhalte dieser Ansätze analysieren und anhand von Beispielen beschreiben. Nach dieser Analyse führen wir eine Bewertung der bisherigen Lösungen durch, in der wir spezifische Probleme der jeweiligen Ansätze herausarbeiten. Abschließend werden wir basierend auf dem Vorwissen der Studierenden und unserer Analyse der Übungsaufgaben Kriterien erarbeiten, die uns im weiteren Verlauf der Arbeit bei der Ausarbeitung unseres eigenen Ansatzes als Leitfaden dienen sollen.

2.1. Zielgruppe

In diesem Teilabschnitt des Kapitels gehen wir auf die Zielgruppe der Compilerbauveranstaltung ein. Dazu besprechen wir zunächst die Modulbeschreibung aus der Studienordnung. Das Modul W*1: Compilerbau wird zum Zeitpunkt der Arbeit im Monobachelor Informatik in unterschiedlichen Fassungen in den Studienordnungen von 2015 und 2022 angeboten [SPOb][SPOc]. Ebenfalls kann das Modul im Kombibachelor mit Lehramtsbezug belegt werden. In diesem Fall findet die Modulbeschreibung des Monobachelors Anwendung [SPOa]. In der Studienordnung von 2015 sind dem Modul Compilerbau 5 Leistungspunkte (kurz LP, oder CP für Credit-Points, oder ECTS-Punkte für European Credit Transfer System Punkte) zugewiesen [SPOb]. In der neueren Auflage von 2022 hingegen werden dem Modul 8 Leistungspunkte zugewiesen [SPOc]. Die Studienordnung in ihrer Fassung von 2022 findet mit ihrer Veröffentlichung für alle neu immatrikulierten Studierenden Anwendung. Wir wollen im Folgenden primär Angaben aus dieser Studienordnung betrachten. Abweichungen zur älteren Ordnung von 2015 werden entsprechend kenntlich gemacht.
Als Lernziel des Moduls Compilerbau wird angegeben, dass die Studierenden die Grundlagen der Analyse und der Übersetzung von Programmiersprachen erlernen und diese beim Bau eines einfachen Compilers selbst anwenden können.
Zum Belegen des Moduls sind keine strikten Vorbedingungen vorgeben. Es wird empfohlen, Grundkenntnisse in Programmierung sowie in den theoretischen Aspekten formaler Sprachen vorweisen zu können. Diese Inhalte können von den Studierenden in den Modulen *Grundlagen der Programmierung* (Modul B1) und *Einführung in die theoretische Informatik* (Modul A1) erworben werden.
Die Veranstaltung wird in drei Veranstaltungstypen unterteilt. Angeboten wird eine Vorlesung im Umfang von 4 Semesterwochenstunden (kurz SWS, eine SWS entspricht

einer Unterrichtsstunde pro Woche über ein Semester hinweg) (Ordnung 2015: 3 SWS). Dort werden im Laufe des Semesters die folgenden Inhalte behandelt:

- Architektur und Aufgaben eines Compilers
- Anwendung der Theorie der Automaten auf Probleme des Übersetzerbaus
- Konzepte und Techniken der lexikalischen Analyse
- Konzepte und Techniken des Parsings
- Semantische Analyse
- Grundlagen der Codegenerierung, Codeoptimierung, und Verlinkung im Überblick
- Praktische Konstruktion eines Compilers aus den einzelnen Phasen (nur 2022)
- Moderne Techniken wie JIT Compilation und neuere Forschungsthemen (nur 2022)

Dem Umfang der Vorlesung sind 5 der 8 Leistungspunkte (2015: 3 von 5 LP) des Moduls zugeordnet.
Als weiterer Teil der Veranstaltung wird eine Übung im Umfang von 2 SWS (2015: 1 SWS) angeboten. Im Rahmen der Übung werden begleitend zur Vorlesung theoretische Übungsaufgaben im Umfang von 2 LP gestellt (in der Regel bis zu sechs Aufgabenblätter pro Semester), die zur Vertiefung der in der Vorlesung vermittelten Inhalte dienen.
Als dritter Teil der Veranstaltung wird eine Modulabschlussprüfung im Umfang von einem LP abgehalten. Diese Prüfung kann mündlich (30 Minuten) oder schriftlich (120 Minuten, 2015: 150 Minuten) angeboten werden.
Nachdem wir nun die Modulbeschreibung des Moduls beschrieben haben, wollen wir im Folgenden auf weitere Punkte der Studienordnung eingehen. Compilerbau ist in beiden Mono-Studienordnungen Teil einer "zwei aus drei" Regelung. Studierende müssen dabei im Rahmen des Wahlpflichtbereichs (2015: 47 von 180 LP, 2022: 32 von 180 LP) zwei der drei folgenden Module belegen.

- W*1: Compilerbau
- W*2: Betriebssysteme 1
- W*3: Grundlagen von Datenbanksystemen

Der idealtypische Studienverlaufsplan empfiehlt die Belegung von Wahlpflichtmodulen ab dem vierten Semester. Eine strikte Vorgabe existiert nicht.
Darüber hinaus wollen wir im Folgenden noch die beiden empfohlenen Voraussetzungen *Grundlagen der Programmierung* (Modul B1) und *Einführung in die theoretische Informatik* (Modul A1) betrachten. Es wird empfohlen, beide Module im ersten Semester zu belegen. Das Modul *Einführung in die theoretische Informatik* im Umfang von 9 Leistungspunkten umfasst eine Einführung in grundlegende Konzepte der

theoretischen Informatik. Dabei liegt der Fokus auf Automatenmodellen, formalen Sprachen, Berechenbarkeit sowie Komplexität [SPOc]. Das Modul *Grundlagen der Programmierung* im Umfang von 12 Leistungspunkten hat das Ziel, dem Studierenden die Grundlagen von verschiedenen Programmierparadigmen, Konzepten imperativer Programmiersprachen, Konzepte der Objektstrukturierung, eine Einführung in die Programmiersprache Java, ein Verständnis von einfachen Datenstrukturen und Algorithmen, Softwareentwicklung und auch alternative Programmierkonzepte zu vermitteln [SPOc].

Wir haben nun die Vorgaben und Empfehlungen der Studienordnung besprochen. Dabei konnten wir erfahren, dass das Modul Compilerbau optional, als Teil der "zwei aus drei" Regelung, im Wahlpflichtbereich angeboten wird. Es wird empfohlen, vor Belegen der Veranstaltung Wissen in Programmiergrundlagen sowie in Grundlagen der theoretischen Informatik zu erwerben. Eine Verpflichtung zum Erwerben dieser Grundlagen ist jedoch nicht gegeben. Inhaltlich deckt die Vorlesung neben den Grundlagen des Compilerbaus wie Architektur auch die fünf Teilbereiche eines Compilers, nämlich der lexikalischen Analyse, des Parsings, der semantischen Analyse, Optimierung und der Codegenerierung ab. Ebenfalls wurden mit der Studienordnung von 2022 die Themen der praktischen Konstruktion eines Compilers aus den einzelnen Phasen sowie moderne Techniken und neuere Forschungsthemen mit in die Veranstaltung aufgenommen. Da für das Modul Compilerbau keine verpflichtenden Abhängigkeiten bezüglich der Voraussetzungen bestehen, können wir keine allgemeingültigen Aussagen über das Vorwissen der Studierenden treffen, welche diese Veranstaltung belegen. Im weiteren Verlauf der Arbeit wollen wir aufgrund des empfohlenen Vorwissens und der gegebenen Möglichkeiten, dieses Wissen zu erwerben, annehmen, dass die teilnehmenden Studierenden zu Beginn des Moduls die entsprechenden Grundlagen in Programmierung und theoretischer Informatik erworben haben.
In diesem Teilabschnitt haben wir die Zielgruppe des Moduls Compilerbau betrachtet. Dabei haben wir Annahmen darüber getroffen, welches Vorwissen die Studierenden im Laufe ihres Studiums vor Belegen der Compilerbau-Vorlesung erworben haben sollten. Ebenfalls konnten wir einen Blick auf den angebotenen Umfang der Vorlesung werfen und auch den Übungsbetrieb betrachten. Im nächsten Abschnitt wollen wir die bisher genutzten Ansätze in den Sprachen C und Rust besprechen.

2.2. Bisherige Lösungen

In diesem Kapitel der Arbeit wollen wir nun die bisherigen praktischen Übungskonzepte analysieren. Dazu werden wir sowohl den Lösungsansatz in der Programmiersprache C als auch den Ansatz in Rust besprechen. Beide Ansätze wurden zum Zeitpunkt dieser Arbeit bereits im Übungsbetrieb der Compilerbau-Vorlesung eingesetzt. Der Ansatz, welcher in der Sprache C entwickelt wurde (im Folgenden als C-Ansatz bezeichnet), wurde bereits seit über 15 Jahren im Übungseinsatz genutzt. Dabei wurde dieser im Laufe der Zeit mehrfach angepasst und optimiert. Zum Sommersemester

2022 wurde zum ersten Mal auch ein Ansatz in der Programmiersprache Rust (im Folgenden als Rust-Ansatz bezeichnet) angeboten. Dabei können die Studierenden zum Zeitpunkt der Arbeit selbst wählen, ob sie die Aufgaben in Rust oder in C bearbeiten. Die Bearbeitung erfolgt zum Zeitpunkt der Arbeit in Gruppen von bis zu drei Studierenden.
Wir werden im Folgenden sowohl den C-Ansatz als auch den Rust-Ansatz analysieren. Dabei werden wir aufgabenweise beide Ansätze vergleichen, um so die Unterschiede beziehungsweise die Entwicklung des Rust-Ansatzes aus dem C-Ansatz verstehen zu können. Im weiteren Verlauf des Kapitels werden wir Probleme herausarbeiten und bewerten, wo die Schwachstellen der jeweiligen Ansätze liegen. Abschließend wollen wir aus den beiden Ansätzen Kriterien ableiten, welche Faktoren für ein Übungskonzept relevant sind.

Einführungsaufgabe

Im Folgenden werden wir die erste Aufgabe des Übungskonzeptes der Compilerbau-Vorlesung betrachten. Dieses Übungsblatt dient in beiden Ansätzen dazu, einen thematischen Einstieg zu schaffen. Einer der wichtigsten Gründe dafür ist dabei die Abstimmung mit den Vorlesungsinhalten. Zum Beginn eines Semesters werden die ersten Vorlesungstermine unter anderem für organisatorische Inhalte und grundlegende Einführungen genutzt. Somit ergeben sich zum Semesterstart nur wenige Inhalte, die direkt von einer praktischen Übung aufgegriffen werden können. Der zweite Grund ist das Vorwissen der Veranstaltungsteilnehmer. Wie wir in Abschnitt 2.1 festgestellt haben, können die Studierenden im Rahmen des Moduls "Grundlagen der Programmierung" nur den Umgang mit Java erlernen. Da ebenfalls kein Grundwissen in Rust oder C durch die Modulbeschreibung vorausgesetzt wird, wurde sich in beiden Übungsansätzen dazu entschieden, die erste praktische Übungsaufgabe unter anderem für einen Einstieg in die jeweilige Programmiersprache zu nutzen. Wir betrachten nun die beiden Ansätze gesondert, da sie sich im Umfang der Aufgabenstellungen unterscheiden.

Im **C-Ansatz** werden auf Übungsblatt 1 zwei Teilaufgaben gestellt. Beide der Aufgaben behandeln jeweils eine Datenstruktur. In der ersten Teilaufgabe geht es dabei um einen Stack, der Zahlenwerte (Integer) speichern können soll. Den Studierenden eine C-Headerdatei vorgegeben, in der Methodenköpfe enthalten sind. Im Rahmen der Aufgabe müssen die Studierenden eine passende C-Datei erstellen und dort die in der Headerdatei vorgegebenen Funktionen implementieren. Folgende Methoden und die jeweiligen Beschreibungen sind dabei vorgegeben:

- `int stackInit(IntStack *self)` – Initialisiert einen neuen Stack und liefert den Rückgabewert 0, falls keine Fehler bei der Initialisierung aufgetreten sind und andernfalls einen Fehlercode.
- `void stackRelease(IntStack *self)` – Gibt den Stack und alle assoziierten Strukturen frei.

- `void stackPush(IntStack *self, int i)` – Legt den Wert von i auf den Stack.
- `int stackTop(const IntStack *self)` – Gibt das oberste Element des Stacks zurück.
- `int stackPop(IntStack *self)` – Entfernt und liefert das oberste Element des Stacks.
- `int stackIsEmpty(const IntStack *self)` – Gibt zurück, ob ein Stack leer ist, d.h. kein Element enthält (Rückgabewert != 0, wenn leer; == 0, wenn nicht).

Darüber hinaus wird in der Aufgabenstellung vorgegeben, dass die im Stack gespeicherten Werte nur durch die Größe des Arbeitsspeichers begrenzt werden sollen. Durch diese Vorgabe zielt die Aufgabe darauf ab, dass bei der Implementierung eine dynamische Speicherverwaltung genutzt wird.
Bei der zweiten Aufgabe des ersten Übungsblattes steht die Datenstruktur eines Syntaxbaums im Fokus. Auch hier werden Methodenköpfe in einer Headerdatei vorgegeben, die im Rahmen der Aufgabe implementiert werden müssen. Folgende Methodenköpfe und Beschreibungen sind vorgegeben:

- `int syntreeInit(Syntree *self)` – Initialisiert einen neuen Syntaxbaum und liefert den Rückgabewert 0, falls keine Fehler bei der Initialisierung aufgetreten sind und andernfalls einen Fehlercode.
- `void syntreeRelease(Syntree *self)` – Gibt den Syntaxbaum und alle damit assoziierten Strukturen frei.
- `SyntreeNodeID syntreeNodeNumber(Syntree *self, int number)` – Erstellt einen neuen Knoten mit einem Zahlenwert als Inhalt und gibt dessen ID zurück; der ID-Typ ist durch Sie Ihrer Implementation entsprechend festzulegen.
- `SyntreeNodeID syntreeNodeTag(Syntree *self, SyntreeNodeID id)` – Kapselt einen Knoten innerhalb eines anderen Knotens und gibt dessen ID zurück.
- `SyntreeNodeID syntreeNodePair(Syntree *self, SyntreeNodeID id1, SyntreeNodeID id2)` – Kapselt ein Knotenpaar innerhalb eines Knotens und gibt dessen ID zurück.
- `SyntreeNodeID syntreeNodeAppend(Syntree *self, SyntreeNodeID list, SyntreeNodeID elem)` – Hängt einen Knoten an das Ende eines Listenknotens und gibt dessen ID zurück.
- `SyntreeNodeID syntreeNodePrepend(Syntree *self, SyntreeNodeID elem, SyntreeNodeID list)` – Hängt einen Knoten an den Anfang eines Listenknotens und gibt dessen ID zurück.

- `void syntreePrint(const Syntree *self, SyntreeNodeID root)` – Gibt alle Zahlenknoten zwischen runden und alle zusammengesetzten Knoten zwischen geschweiften Klammern rekursiv (depth-first) auf der Standardausgabe aus. Hinweis: es ist erlaubt bei der Ausgabe beliebig viele Leerzeichen und Zeilenenden zu verwenden; wir werden alle Zeichen aus der Ausgabe entfernen, für die isspace() wahr ist.

Sowohl die Datenstruktur des Stacks als auch die des Syntaxbaums wurden für dieses Aufgabenblatt gewählt, da sie im späteren Verlauf des Projektes wieder aufgegriffen werden. Dadurch können die Studierenden mit diesen beiden Teilaufgaben sich nicht nur grundlegend mit der Sprache C auseinandersetzen, sondern ebenfalls bereits eine Vorarbeit für das Projekt leisten.
Für beide Aufgaben wird jeweils eine C-Datei mitgeliefert, in der eine Folge von Operationen auf den implementierten Datenstrukturen durchgeführt wird. Diese sollen den Studierenden als Tests für ihre Implementierung dienen und ihnen dabei helfen, Fehler in ihrer Lösung zu entdecken.

Im **Rust-Ansatz** werden ebenso wie im C-Ansatz ebenfalls zwei Aufgaben gestellt. Dabei beschäftigt sich auch hier die erste der beiden Aufgaben mit der Datenstruktur eines Stacks. Als Teil der Aufgabenstellung wird ein *Trait* [1] vorgegeben, welches in Listing 2 dargestellt ist. Ein *Trait* definiert die Funktionalität, die ein bestimmter Typ hat und mit anderen Typen teilen kann. Ein *Trait* ähnelt einer Funktion, die in anderen Sprachen oft als Interface bezeichnet wird, allerdings mit einigen Unterschieden.

[1] `https://doc.rust-lang.org/book/ch10-02-traits.html`

```rust
pub trait Stack {
    /// Initialisiere einen leeren Stack
    fn init() -> Self;

    /// Füge einen neuen Wert zum Stack hinzu
    fn push_val(&mut self, i: i32);

    /// Lese den obersten Wert, ohne diesen zu entfernen
    fn top_val(&self) -> Option<&i32>;

    /// Entferne den obersten Wert und gib diesen zurück
    fn pop_val(&mut self) -> Option<i32>;

    /// Prüfe ob der Stack leer ist
    fn is_empty(&self) -> bool;
}
```

Listing 2.: Vorgegebenes Trait, welches als Teil von Aufgabe 1 zu vervollständigen ist. Entnommen aus Übungsblatt 1 des Rust-Ansatzes, *lib.rs*

Diese, durch das *Trait* vorgegebenen Funktionen, sollen nun in zwei Teilaufgaben ausgefüllt werden. In Teilaufgabe *a* soll das *Trait* für einen Vektor (`Vec<T>`) implementiert werden. In der zweiten Teilaufgabe *b* soll das teilweise bereits implementierte *Trait* für die mitgelieferte Enumeration `ListStack` vervollständigt werden. Dafür sind die auszufüllenden Abschnitte in der Implementation markiert. Zum Testen ihrer Lösung stehen den Studierenden für jede Teilaufgabe mehrere Rust-Unittests zur Verfügung.
Die zweite Aufgabe dieses Übungsblattes beschäftigt sich, ähnlich wie die C-Implementierung, mit der Datenstruktur eines abstrakten Syntaxbaums. Dabei wird das `struct` der Baumstruktur vorgegeben, welches in Listing 3 dargestellt ist.

```rust
pub type ID = usize;

[...]

#[derive(Clone, Debug, PartialEq)]
pub struct SyntaxTree<T> {
    id: ID,
    value: T,
    children: Vec<SyntaxTree<T>>,
}
```

Listing 3.: Vorgegebenes `struct` des Syntaxbaums. Entnommen aus Übungsaufgabe 1 des Rust-Ansatzes, *syntax_tree.rs*

Darüber hinaus sind mehrere Funktionen vorhanden, welche bereits für diese Datenstruktur vorgegeben sind. Diese haben wir im Folgenden mit einer Beschreibung des Funktionsumfangs dargestellt:

- `fn new(value: T) -> SyntaxTree<T>;` – Erstellt einen neuen Syntaxbaum, welcher `value` enthält.
- `fn push_node(&mut self, new_node: SyntaxTree<T>);` – Fügt einen Knoten als hinterstes Kind eines anderen ein.
- `fn prepend_node(&mut self, new_node: SyntaxTree<T>);` – Fügt einen Knoten als erstes Kind eines anderen ein.
- `fn insert_node(&mut self, index: usize, new_node: SyntaxTree<T>);` – Fügt einen Syntaxbaum als Kind in den Baum mit dem gegeben `index` ein.
- `fn find_node(&self, predicate: fn(&SyntaxTree<T>) -> bool) -> Option<&SyntaxTree<T>>;` – Gibt die Referenz auf einen gesuchten Knoten zurück.
- `fn find_node_mut(&mut self, predicate: fn(&SyntaxTree<T>) -> bool,) -> Option<&SyntaxTree<T>>;` – Gibt die mutable Referenz auf einen gesuchten Knoten zurück.

Von den Studierenden sind die entsprechend markierten Abschnitte in den Funktionen zu vervollständigen. Damit auch hier die Studierenden ihren Lösungsansatz testen können, sind mehrere Unittests mitgeliefert.

Scanner

Bei dieser Übungsaufgabe steht in beiden Übungskonzepten die lexikalische Analyse im Vordergrund. Dieses Aufgabenblatt bildet den ersten Schritt für die Entwicklung

des Praktikumsinterpreters im Laufe der weiteren Übungsblätter. Dabei nutzen sowohl der C-Ansatz als auch der Rust-Ansatz nahezu identische Aufgabenstellungen, obwohl diese bedingt durch die verwendete Sprache und die unterschiedlich genutzten Bibliotheken zu unterschiedlichen Lösungsansätzen führen können. Beide Ansätze teilen dabei den zu erfüllenden Umfang auf zwei Aufgaben auf, wobei nur die erste der beiden Teilaufgaben als Grundstein für die Übungsinterpreter gewertet werden kann und die zweite Aufgabe das Konzept der lexikalischen Analyse sowie den Umgang mit regulären Ausdrücken weiter vertiefen soll.

Im **C-Ansatz** wird für die lexikalische Analyse das Metawerkzeug `flex` [2] verwendet. Flex nutzt als Grundlage für den Scanner eine `Lex Language Format`-Datei (Dateiendung .l), in der eine Mischung aus C-Syntax und Flex eigener Syntax verwendet wird, um die benötigten Token anzugeben. Als Tokens können sowohl Strings als auch reguläre Ausdrücke angegeben werden. Ebenfalls ist es möglich, Makros als Abkürzung für vorher definierte Tokenfolgen zu nutzen. Im Standardumfang stehen bereits Makros wie `:digit:` als Abkürzung für die Menge der Ziffern zur Verfügung. Nach vollständigem Erkennen eines Tokens kann vorher entsprechend hinterlegter C-Code ausgeführt werden, um die erkannte Zeichenkette weiterzuverarbeiten. Dabei arbeitet Flex nach dem Verfahren des *Longest Prefix Matching*. Dabei wird immer die längste mögliche Zeichenfolge erkannt.
In Aufgabe 1 soll mithilfe von Flex ein Scanner implementiert werden, der die Tokens der Sprache C1 erkennt. Bei C1 handelt es sich dabei um eine reduzierte Version der Sprache C. Im Funktionsumfang sind lediglich Schleifen, Verzweigungen, Funktionen sowie die Datentypen `void`, `int`, `float` und `bool` enthalten. Die Liste der möglichen Tokens in Sprache C1 ist in Listing 15 dargestellt. Die Studierenden haben müssen bei dieser Aufgabe die mitgelieferte Tokenliste in das Eingabeformat von Flex überführen. Neben den Tokens sollen auch ein- und mehrzeilige Kommentare sowie *Whitespaces* erkannt, aber ignoriert werden. Zum Überprüfen ihrer Lösung steht den Studierenden eine Beispieleingabe sowie eine Datei mit einer erwarteten Ausgabe zum manuellen Vergleich zur Verfügung.
In der zweiten der beiden Aufgaben auf dem diesem Übungsblatt soll ebenfalls ein Scanner mithilfe von Flex konstruiert werden. Dabei ist das Ziel in diesem Fall, URLs und die damit verbundenen Linkbeschreibungen in xhtml-Dokumenten zu finden, zu extrahieren und zurückzugeben. Ein zu erkennender HTML-Tag folgt dabei der Form `<a ... href=ÜRL"... >LINKTEXT</a>`. Die Studierenden sollen hierfür eigenständig Tokens entwickeln, um so die benötigten Informationen zuverlässig erkennen zu können. Auch für diese Übungsaufgabe steht den Studierenden eine Eingabedatei sowie eine Datei mit der erwarteten Aufgabe zur Verfügung, die manuell verglichen werden muss.

Auch im Falle des **Rust-Ansatz** wird für die lexikalische Analyse ebenfalls ei-

[2] `https://github.com/westes/flex`

ne externe Bibliothek herangezogen. So wird hier die Crate `Logos` [3] in der Version v0.12.0 verwendet. Wie auch bei Flex handelt es sich bei Logos ein Scanner. Für Logos wird als Grundlage eine Rust-Datei angelegt, in der die Tokens in Rust-Syntax in einer Enumeration hinterlegt werden. Per Annotation an den Enumerationsvarianten kann einem Token ein String, oder ein regulärer Ausdruck angefügt werden, welcher beim Scannen das jeweilige Token erzeugen soll. Per Parameter in der Annotation können auch Rust-Funktionen hinterlegt werden, die nach Erkennen des Tokens ausgeführt werden sollen. Wie auch Flex arbeitet Logos nach dem Prinzip des *Longest Prefix Matching*.
In Aufgabe 1 sollen, wie auch schon im C-Ansatz, die Tokens der Sprache C1 erkannt werden. Hierfür wird dieselbe Tokenliste (Listing 15), wie auch schon im C-Ansatz, in der Aufgabenstellung vorgegeben. Zum Testen des Lösungsansatzes der Studierenden wird ein Unittest mitgeliefert, welche das Ergebnis des Scanners für eine mitgelieferte Eingabe gegen eine vordefinierte Ausgabe testet.
In der zweiten Aufgabe muss ein weiterer Scanner implementiert werden, der analog zum C-Ansatz URLs und den zugehörigen Linktext aus einem HTML-Dokument extrahieren können soll. Mit der Aufgabenstellung wird auch eine bereits vorbereitete Grundstruktur für den Scanner mitgeliefert. In dieser müssen die Annotationen an zwei Tokens ergänzt werden, sowie eine Funktion zur Auswertung der Tokens und Rückgabe der URLs und Linktexte als Tuple implementiert werden. Auch hier steht den Studierenden ein Unittest zur Verfügung, der mit einer vorgegebenen Ein- und Ausgabe das Verhalten des implementierten Scanners prüft.

Handparser

In diesem Abschnitt wollen wir uns dem ersten Aufgabenblatt widmen, welche das Thema der semantischen Analyse behandelt. An dieser Stelle zweigt die Reihenfolge der Aufgaben beider Ansätze voneinander ab. Im C-Ansatz wurde diese Aufgabe als Übungsaufgabe 6 eingereiht, beziehungsweise als Zusatzaufgabe geführt. Im regulären Verlauf des Übungsbetriebs wurde diese Aufgabe nicht gestellt. Sie wurde vorgehalten, um Übungsgruppen von Studierenden nach Abschluss der übrigen Aufgaben noch die Möglichkeit zu geben, fehlende Punkte für die Zulassung zur Modulabschlussprüfung zu erhalten. Im Rust-Ansatz wurde diese Aufgabe regulär als dritte Übungsaufgabe gestellt.
Im Rahmen dieses Übungsblattes sollen die Studierenden per Hand einen rekursiv absteigenden Parser implementieren. Für diese Aufgabe wurde die Grammatik der C1 Sprache reduziert. Die Sprache dieser reduzierten Grammatik wird als C(-1) bezeichnet. Dabei wurden unter anderem Funktionalitäten wie Schleifen oder globale Variablen entfernt. Die auch in der Aufgabenstellung vorgegebenen Produktionsregeln der C(-1) Grammatik sind in Listing 17 dargestellt.

Im **C-Ansatz** wird in der Aufgabenstellung ein passender Scanner in Flex mitgelie-

[3] `https://crates.io/crates/logos`

fert. Die Studierenden können jedoch auch wahlweise den von ihnen implementierten Scanner aus der vorherigen Übungsaufgabe verwenden. Im Rahmen des Handparsers muss ebenfalls der Umgang mit dem Tokenstream des Scanners implementiert werden. Für diesen Fall sind in der Aufgabenstellung Namenskonventionen für Variablen und Funktionen vorgegeben. Die Auswertung über das korrekte Ableiten der Eingabe erfolgt per Rückgabewert. Als Mindestvoraussetzung für die Abgabe wird den Studierenden ein valides C(-1) Programm mitgeliefert, das fehlerfrei abgeleitet werden können soll.

Die Aufgabenstellung des **Rust-Ansatz** folgt der des C-Ansatzes in Teilen wörtlich. Der in diesem Ansatz mitgelieferte Lexer verwendet, wie auch in der vorherigen Aufgabe, die Crate Logos. Wie im C-Ansatz kann hier der von den Studierenden implementierte Scanner aus der vorherigen Aufgabe ebenfalls weiter verwendet werden. Der mitgelieferte Scanner wurde um eine Schnittstelle für den zu implementierenden Handparser erweitert, welche im Scanner aus Aufgabe 2 nicht zur Verfügung steht. Ebenfalls sind die Definitionen einzelner Tokens unterschiedlich. Dadurch stehen den Studierenden im mitgelieferte Scanner die folgenden Funktionen für ihren Ansatz zur Verfügung:

- `pub fn new(text: &'a str) -> C1Lexer` – Initialisiert den Lexer
- `pub fn current_token(&self) -> Option<C1Token>` – Gibt den Typ des aktuellen Tokens zurück, falls vorhanden.
- `pub fn peek_token(&self) -> Option<C1Token>` – Gibt den Tokentyp des nächsten Tokens zurück, falls vorhanden.
- `pub fn current_text(&self) -> Option<&str>` – Gibt den Text des aktuellen Tokens zurück, falls vorhanden.
- `pub fn peek_text(&self) -> Option<&str>` – Gibt den Text des nächsten Tokens zurück, falls vorhanden.
- `pub fn current_line_number(&self) -> Option<usize>` – Gibt die Zeilennummer an, in welcher sich der Token befindet, falls vorhanden.
- `pub fn peek_line_number(&self) -> Option<usize>` – Gibt die Zeilennummer an, in welcher sich das nächste Token befindet, falls vorhanden.
- `pub fn eat(&mut self)` – Konsumiert das aktuelle Token.

Zusätzlich zur Schnittstelle des Lexers werden in der Aufgabenstellung weitere optionale Funktionen angegeben, die die Studierenden als Hilfestellung selbst implementieren können. Zum Testen des implementierten Handparsers wird in diesem Rahmen ein Unittest mitgeliefert. Ebenfalls werden mit dem Lexer mehrere Unittests mitgeliefert, die jedoch nur diesen Prüfen.

Parser

Nach den Aufgabenstellungen zum Scanner und zum Handparser wurde eine weitere Aufgabe konzipiert, die sich mit dem Themengebiet des Parsers auseinandersetzt. Im Rahmen dieser Aufgabe soll ein Parsergenerator verwendet werden, der automatisch aus einer eingegebenen Grammatik einen Parser generiert. Für diese Aufgabe wurde sich in beiden Ansätzen für das Meta-Werkzeug Bison entschieden. Dabei handelt es sich um einen universellen Parsergenerator aus dem GNU-Projekt[4]. Bison wird seit 1985 entwickelt und ging zu diesem Zeitpunkt aus dem Tool yacc[5] hervor.
Bison erwartet als Eingabe sowohl eine Grammatik in BNF als auch die Bekanntgabe der Tokens bzw. Terminalsymbole, die in der Eingabe bzw. Grammatik vorkommen. Grammatikregeln können dabei an jeder Stelle unterbrochen werden, um dort eingebetteten Quellcode anzugeben. Ebenfalls ist es in Bison möglich, Präferenzen für mehrdeutige Ableitungsregeln zu definieren. Dadurch ist es nicht notwendig, eventuelle auftretende Mehrdeutigkeiten durch Umformen der Grammatik aufzulösen.
Als Grammatik wird, im Gegensatz zum Handparser, wieder die vollständige Grammatik der Sprache C1 verwendet. Diese ist im Anhang in Listing 16 dargestellt.

Im **C-Ansatz** wird die entsprechende C-Implementation von Bison verwendet. Aufgabe der Studierenden ist es, die angegebene Grammatik der C1-Sprache, die in EBNF angegeben ist, in die passende Form umzuwandeln und in die mit der Aufgabenstellung mitgelieferten Bison-Datei (`minako-syntax.y`) zu ergänzen. Wie auch schon in der Aufgabe zum Handparser wird den Studierenden mit der Aufgabenstellung ein vorbereiteter Lexer mitgeliefert. Ebenfalls wird den Studierenden zum Testen ihres Lösungsansatzes wieder eine Testeingabe mitgeliefert, die ein syntaktisch korrektes C1-Programm enthält.
In der Aufgabenreihenfolge wird diese Aufgabe als im C-Ansatz als Übungsblatt 3 geführt.

Im **Rust-Ansatz** wird auf ein Bison-Skeleton zurückgegriffen, um so einen in Rust geschriebenen Bison-Parser generieren zu können. Die Crate des Bison-Skeleton steht seit September 2020 auf Crates.io[6] zur Verfügung und wird für die Übungsaufgabe in Version "0.41.0"verwendet. Diese Implementation von Bison für Rust orientiert sich an der ursprünglichen C-Implementation und bietet ein Rust-Frontend für die Verwendung von Bison. Dabei entspricht das Eingabeformat ebenfalls der C-Vorlage und weicht nur zugunsten der Rust-Syntax von diesem Format ab. Regelunterbrechende Codeblöcke sind ebenfalls mit Rust-Code möglich. Eine beabsichtigte Kompatibilität zum verwendeten Parser Logos ist nicht gegeben. Zur Umwandlung der von Logos verwendeten Tokens in das passende Format für das Bison-Skeleton wird eine entsprechende Funktion mit der Aufgabenstellung mitgeliefert.
Wie auch im C-Ansatz ist das Ziel der Übungsaufgabe, die Umwandlung der Regeln

[4] `https://www.gnu.org/software/bison/`
[5] `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/yacc.html`
[6] `https://crates.io/crates/rust-bison-skeleton`

der C1-Grammatik von EBNF zu BNF und die anschließende Vervollständigung der mitgelieferten Bison-Datei. Im Gegensatz zum C-Ansatz werden den Studierenden hier die zu verwendenden Präferenzen für die Behandlung von Mehrdeutigkeiten mit angegeben. Im Gegensatz zu den anderen Aufgaben aus dem Rust-Ansatz werden in dieser Aufgabe keine expliziten Testfälle für den Parser mitgeliefert. Die einzigen enthaltenen Unittests dienen dem Testen des vorgegebenen Lexers. Für die Studierenden steht zum Testen ihrer Lösungen nur eine syntaktisch korrektes C1-Programm zur Verfügung.

Semantische Analyse

Als fünften Schritt wollen wir nun die Übungsaufgabe zur semantischen Analyse betrachten. In dieser Aufgabe wird in beiden Ansätzen sowohl die semantische Analyse durchgeführt als auch ein Syntaxbaum für das eingegebene Programm konstruiert. Die Struktur, aus der der Syntaxbaum erstellt wird, entspricht in beiden Ansätzen dem Syntaxbaum, der auf dem ersten Aufgabenblatt vervollständigt werden sollte. Als Hilfsmittel wird den Studierenden darüber hinaus für diese Aufgabe eine bereits vollständig implementierte Symboltabelle zur Verfügung gestellt, welche die Möglichkeit bietet, darin Symbole (Variablen, Parameter und Funktionen) sowie die Sichtbarkeitsbereiche zu verwalten. Der Funktionsumfang der Symboltabelle ist in beiden Ansätzen gleich, nur die Implementation unterscheidet sich bedingt durch die Möglichkeiten der jeweiligen Sprache.
Mit der Aufgabenstellung wird ebenfalls in beiden Ansätzen die Semantik der Sprache C1 vorgegeben. Dabei liegt der Fokus auf den folgenden beiden Punkten.

- Überprüfung der Typenkorrektheit. Dazu wird angegeben, dass keine impliziten Typenumwandlungen außer der von `int` zu `float` zulässig sind. Die Typenkorrektheit soll für Zuweisungen, bei Rückgabewerten und Ausdrücke überprüft werden.
- Prüfung von Sichtbarkeitsbereichen. Es wird vorgegeben, dass die Codesegmente, welche zu einem der folgenden Metasymbole `block`, `forstatement` und `functiondefinition` reduziert werden, jeweils neue Sichtbarkeitsbereiche darstellen.

Darüber hinaus sind die folgenden weiteren Regeln angegeben, welche ebenfalls für die Sprache C1 gelten:

1. Es muss eine parameterlose `main()`-Funktion mit dem Rückgabetyp `void` geben.
2. Bezeichner müssen vor ihrer Benutzung deklariert werden.
3. Ein Sichtbarkeitsbereich darf keinen Bezeichner doppelt enthalten.
4. Es gibt keine Variablen vom Typ `void`.
5. Ausdrücke vom Typ `void` können nicht mit `printf()` ausgegeben werden.

6. Bedingungen von `while`, `do while`, `for` und `if` sind boolische Ausdrücke.
7. Funktionen müssen parameterkonform sein.
8. Der Rückgabewert einer Funktion muss zum Typen der Funktion passen.
9. Zuweisungen erfolgen nur in zuweisungsfähige Strukturen und der Typ der rechten Seite ist kompatibel zum Typ der linken Seite.
10. Alle hier nicht betrachteten Fälle werden entsprechend dem C-Standard[7] behandelt.

Sowohl im C- als auch im Rust-Ansatz muss der Code der semantischen Analyse in die Bison-Datei ergänzt werden, welche in der vorherigen Aufgabe dazu genutzt wurde, um daraus den Parser zu generieren. Hierfür wird die Funktion von Bison genutzt, dass Produktionsregeln durch Quellcodeblöcke unterbrochen werden können.

Im **C-Ansatz** wird den Studierenden eine Bison-Datei zur Verfügung gestellt, welche bereits die Konstruktion des Syntaxbaums sowie einige Schritte der semantischen Analyse enthält. Darüber hinaus werden keine weiteren Angaben gemacht, an welcher Stelle noch Schritte der semantischen Analyse ergänzt werden müssen. Erstmals steht den Studierenden eine umfangreiche Testsuite mit 81 Testfällen für die Übungsaufgaben zur Verfügung.

Im **Rust-Ansatz** hingegen sind in der Bison-Datei neben der Konstruktion des Syntaxbaums auch Kommentare eingefügt, aus welchen bereits ersichtlich wird, an welcher Stelle welche semantische Überprüfung durchgeführt werden muss. Auch im Rust-Ansatz steht den Studierenden eine umfangreiche Testsuite zur Verfügung. Es werden insgesamt 91 Testeingaben, aufgeteilt in semantische und syntaktische Tests, mitgeliefert.

Interpreter

In diesem Abschnitt wollen wir die letzte Übungsaufgabe über das Thema eines Interpreters betrachten. Diese Aufgabe ist nur im C-Ansatz vorhanden. Der Rust-Ansatz endet mit der Aufgabe zur semantischen Analyse. Somit werden wir im Folgenden nur die Übungsaufgabe im C-Ansatz betrachten.
Die Aufgabenstellung im C-Ansatz ist dabei kurz gehalten und weist nur aus, dass in einer mitgelieferten C-Datei (`minako.c`) Codepassagen vervollständigt werden sollen. Der Aufbau der mitgelieferten Strukturen ist in Listing 4 dargestellt. Die Studierenden sollen, orientiert am C-Callstack, Variablenzugriffe, Funktionsaufrufe sowie Rückgaben mittels dieser Struktur umsetzen. Dabei entspricht der `Variablenstack` der Stack-Struktur, welche auf dem ersten Übungsblatt erstellt werden sollte. Eine explizite Nutzung der mitgelieferten Struktur ist nicht vorgegeben. Es dürfen darüber hinaus

[7] `https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf`

beliebige Änderungen vorgenommen werden. Zum Testen der Lösungsansätze der Studierenden stehen drei nach Komplexität geordnete C1-Programme zur Verfügung.

```
22 typedef struct {
23     /**@brief Typ des Variablenwertes.
24     */
25     SyntreeNodeType type;
26 
27     /**@brief Variablenwert.
28     */
29     union {
30         int boolean;  /**<@brief Boolescher Wert. */
31         int integer;  /**<@brief Ganzzahliger Wert. */
32         float real;   /**<@brief Fließkommawert. */
33         char *string; /**<@brief Zeiger auf Zeichenkette. */
34     } value;
35 } MinakoValue;
36 
37 /**@brief Struktur des Laufzeitzustands des Interpreters.
38 */
39 typedef struct {
40     MinakoValue stack[MINAKO_STACK_SIZE]; /**<@brief Variablenstack. */
41     MinakoValue eax;  /**<@brief Ausgaberegister. */
42     MinakoValue *ebp; /**<@brief Base pointer. */
43     MinakoValue *esp; /**<@brief Stack pointer. */
44     int returnFlag; /**<@brief Signalisiert das Verlassen einer Funktion. */
45 } MinakoVM;
```

Listing 4.: Vorgegebene Datenstrukturen im Interpreter des C-Ansatzes. Entnommen aus der 6. Übungsaufgabe des C-Ansatzes, *minako.c*

Nachdem wir in diesem Teilabschnitt die beiden bisherigen Übungsansätze betrachtet und anhand kleiner Beispiele erläutert haben, möchten wir diese im nächsten Abschnitt bewerten. Anschließend können wir ermitteln, welche Aspekte durch eine Überarbeitung des Übungskonzepts verbessert werden könnten.

2.3. Bewertung

In diesem Teilabschnitt der Arbeit wollen wir nun die bisherigen Lösungsansätze in C und Rust, welche wir im vorherigen Teilabschnitt erläutert haben, bewerten, um im Anschluss dazu Bewertungskriterien daraus ableiten zu können. Das Ziel dieses Abschnitts ist es, Schwachpunkte der bisherigen Lösungen aufzuzeigen und zu

ermitteln, wie diese verbessert werden könnten. Als Grundlage der Bewertung wollen wir die folgenden Metakriterien betrachten.

- Verwendung der Sprache – Dabei wollen wir die Nutzung der jeweils zu nutzenden Programmiersprache betrachten. Einfluss haben unter anderem, welche Funktionen der Sprache vorgegeben, bzw. im Rahmen der Bearbeitung erwartet werden. Wie wir im Vorhinein festgestellt haben, kann von den Studierenden kein Vorwissen in C oder Rust erwartet werden. Ein einfacher Einstieg in die Funktionen der Sprache bietet somit auch den Studierenden die Möglichkeit, sich nachvollziehbar mit den Möglichkeiten und Funktionsweisen der jeweiligen Sprachen auseinander zu setzen.

- Umfang und Design der Aufgaben – Bei diesem Metakriterium betrachten wir die Aufgabenstellung. Dies beinhaltet unter anderem, wie viele und welche Informationen über die zu bearbeitende Aufgaben in der Aufgabenstellung vermittelt werden können. Ebenfalls wollen wir den zu erfüllenden Umfang einer Übungsaufgabe untersuchen. Während aus einer nicht sehr umfangreiche Aufgabenstellung bzw. ein großer Umfang der Aufgabe ein zu hoher Arbeitsaufwand resultieren kann, besteht ebenso die Möglichkeit, dass durch zu viele Vorgaben zwar der Arbeitsaufwand verringert wird, dafür aber auch der Lerneffekt reduziert wird.

- Feedback – Hierbei wollen wir die Rückmeldung für die Studierenden betrachten. Im besten Fall sollen Studierende die Möglichkeit erhalten, selbstständig zu prüfen, ob sie eine Aufgabe vollständig gelöst haben. Besseres Feedback seitens der, in der Aufgabenstellung mitgelieferten, Testmöglichkeiten ermöglicht ein besseres Verständnis für Fehler und erleichtert das Beheben dieser Fehler.

Dazu wollen wir im Folgenden die Übungsblätter erneut nacheinander besprechen.

Vorbereitung

In der ersten Aufgabe des ersten Übungsblatts im C-Ansatz wird durch die Implementierung eines Stacks ein guter Einstieg geboten, um die Grundlagen der Sprache C zu erlernen. Hierbei werden auch komplexere Teilbereiche wie Zeigerarithmetik oder der Umgang mit dem Speichermanagement vermittelt. Im Gegensatz dazu trifft dies im Rust-Ansatz nur eingeschränkt zu. Da Rust im Gegensatz zu C bereits passende Datenstrukturen im Standardumfang der Sprache bietet, sinkt hier die Komplexität bei der Auseinandersetzung mit dem Funktionsumfang der Sprache. Jede der fünf zu implementierenden Funktionen für den `Vec<i32>` lässt sich durch einen einzeiligen Aufruf einer Standardfunktion lösen, da die `Vec`-Struktur bereits umfangreiche Funktionen mitliefert. Auch bei der zweiten Teilaufgabe des Rust-Ansatzes werden nur wenige spezifische Funktionen wie `match`, `enum` oder `Option` verwendet. Da auch nur wenige Zeilen von den Studierenden selbst ausgefüllt werden müssen, ist keine starke Auseinandersetzung mit diesen Funktionalitäten nötig.

Im C-Ansatz erachten wir es als eine gute Synergie, dass der hier implementierte Stack bereits die Grundstruktur für den Callstack des Interpreters in der letzten Übungsaufgabe liefert. Durch diese Verknüpfung kann im späteren Verlauf auf bereits erworbene Kenntnisse aufgebaut werden. Im Gegensatz dazu kann im Rust-Ansatz, da kein Übungsblatt zur Interpretation vorhanden ist, diese Synergie nicht genutzt werden.
Die zweite Übungsaufgabe befasst sich in beiden Ansätzen mit der Konstruktion eines Syntaxbaums. Wie im Abschnitt 2.2 dargelegt, müssen im C-Ansatz vorgegebene Funktionen implementiert werden, die aufgrund ihrer Formulierung für die Konstruktion des Syntaxbaums im Rahmen der Aufgabe zur semantischen Analyse benötigt werden. Hier wird bereits durch die spätere Wiederverwendung der Datenstruktur in der semantischen Analyseaufgabe eine Verbindung zum späteren Interpreter geschaffen. Die Studierenden setzen sich außerdem in dieser Aufgabe erneut und verstärkt mit den Themen der Zeigerarithmetik und dem Speichermanagement auseinander. Beides sind wichtige Teilbereiche der Sprache C. Wir erachten es als sinnvoll, dass diese beiden Aspekte der Sprache im ersten Übungsblatt mehrfach behandelt werden, da sie in der Vorlesung "Grundlagen der Programmierung" bei der Auseinandersetzung mit Java nicht relevant sind.
Im Rust-Ansatz muss ebenfalls ein Syntaxbaum vervollständigt werden. Dabei wird ein ID-basierter Syntaxbaum eingeführt. Die Struktur dieses Baums haben wir bereits in der Listing 3 betrachtet. Wie dort zu sehen ist, wird neben einem Datenwert eines generischen Typs `<T>` und einer Liste von Kinderknoten auch eine ID gespeichert. Für die Zählung der IDs wird hierfür ein globaler Wert vom Typ `usize` verwendet. Die Funktion zur Zählung der IDs ist in Listing 5 dargestellt.

```
15 fn next_id() -> ID {
16     unsafe {
17         let id = LAST_ID;
18         LAST_ID += 1;
19         id
20     }
21 }
```

Listing 5.: Funktion zur Zählung der IDs im Syntaxbaum des Rust-Ansatzes. Entnommen aus Übungsaufgabe 1 dem bisherigen Rust-Ansatz, *syntax_tree.rs*

Durch die für diese Aufgabe gewählte Implementation des Syntaxbaums wird für diese globale Zählvariable eine `unsafe`-Block benötigt.
In Rust wird `unsafe` verwendet, um Code zu kennzeichnen, in dem der Compiler die strikte Speichersicherheitsgarantie aufhebt. Dadurch können Operationen durchgeführt werden, welche außerhalb von `unsafe`-Blöcken möglicherweise vom Compiler aufgrund mangelnden Wissens abgelehnt werden würden. Durch die Aufhebung der Überwachung der Speichergarantie in `unsafe`-Blöcken muss der Entwickler selbst-

ständig die Korrektheit sicherstellen. Durch die falsche Verwendung von `unsafe`-Code kann es in Programmen zu Speicherfehlern führen, zum Beispiel durch die Dereferenzierung eines Nullpointers [Rusf].
Da die verwendete `static mut`-Variable `LAST_ID` nicht durch zum Beispiel ein Mutex oder ein Lock abgesichert ist, könnte dies besonders in nebenläufigen Programmen zu Problemen führen. Unter anderem kann der gleichzeitige Zugriff verschiedener Threads zu Data Races führen. Wie wir in der Untersuchung der Zielgruppe des Übungskonzeptes festgestellt haben, wird zu Beginn der Veranstaltung kein Vorwissen über Rust vorausgesetzt. Wir erachten es als ungeeignet, `unsafe`-Blöcke, die erst im vorletzten Kapitel des Rust-Buchs eingeführt werden, in Übungsaufgaben zu verwenden, die für Anfänger im Umgang mit Rust geeignet sein sollen. Darüber hinaus bietet Rust unabhängig von dieser Übungsaufgabe andere Möglichkeiten für die Implementierung von Syntaxbäumen, wie `rustc_ast`[8] (ein aktuell noch unstable-Feature) oder auch die Verwendung von Enumerationen und Strukturen, um so eigene dynamische Syntaxbäume zu implementieren. Diese nativen Varianten bieten neben dem intakt bleiben der sicheren Nebenläufigkeit auch den Vorteil, dass andere native Funktionen von Rust, wie die Nutzung von Design-Patterns, so einfach verwendet werden können. Aus diesen Gründen sehen wir diese Implementation des Syntaxbaums als ungeeignet an und finden, dass diese Herangehensweise so nicht für Anfänger im Umgang mit Rust geeignet ist.

Scanner

Nun möchten wir die Übungsaufgabe zum Umgang mit Scannern besprechen. Diese Aufgabe bildet auch anteilig die erste Aufgabe aus der Reihe des Übungsinterpreters, der im Laufe der Aufgaben erstellt werden soll. Beide Ansätze nutzen auf diesem Übungsblatt, wie auch schon auf Blatt 1, dieselben Aufgabenkonstellationen. Dabei ist in Aufgabe 1 ein Scanner für die verwendete Sprache C1 zu vervollständigen. Der C-Ansatz verwendet für diese Aufgabe den Scanner `flex`, während im Rust-Ansatz die Crate `Logos` genutzt wird. Beide Ansätze nutzen dieselbe Tokenliste für diese Aufgabe. Trotz Unterschieden in den verwendeten Bibliotheken ist hier das Vorgehen in beiden Ansätzen recht naheliegend.
Als zweite Teilaufgabe muss ebenfalls in beiden Ansätzen dieselbe Aufgabenstellung bearbeitet werden. Dabei sollen URLs und ihre zugehörigen Linktexte aus einem HTML-Dokument extrahiert werden. Aufgrund des Funktionsumfangs der beiden Bibliotheken unterscheidet sich hier die Herangehensweise stark.
Im C-Ansatz wird die Funktion von flex ausgenutzt, dass eigene Zustände definiert werden können. So kann die Position in einem erkannten HTML-Tag gespeichert werden, um so zustandsabhängig andere Ableitungsregeln zu nutzen.
Da die Funktionalität von eigenen Zuständen im Rust-Ansatz, der die Crate `Logos` nutzt, nicht gegeben ist, wird hier ein anderes Verfahren angestrebt. In der Musterlösung werden neben einem Error-Token nur zwei Tokens verwendet. Die restliche

[8] `https://doc.rust-lang.org/beta/nightly-rustc/rustc_ast/`

Auswertung des HTML-Tags sowie die Extraktion der URLs und Linktexte erfolgt dabei in einer separaten Rust-Funktion.
Wir finden, dass die erste Aufgabe gut geeignet ist, um so einen Grundstein für die Entwicklung des Übungsinterpreters zu legen. Ebenfalls lernen die Studierenden hier den Umgang mit dem jeweiligen Scanner und vertiefen ihr Wissen in der Verwendung von regulären Ausdrücken. Die zweite Übungsaufgabe hingegen stellt jedoch nur eine Wiederholung dar. Im C-Ansatz werden weitere Funktionen von Flex wie `Startbedingungen` eingeführt, die im weiteren Verlauf des Übungskonzeptes jedoch nicht relevant sind. Dadurch, dass im Rust-Ansatz ein Großteil der Erkennung von URLs und Linktexten nicht mehr durch den Scanner selbst, sondern durch eine Rust-Funktion übernommen wird, erweitert sich der Lernfokus. Dieser richtet sich so eher auf die Bearbeitung von Texte unterstützt durch den Scanner und weniger auf die Konstruktion von regulären Ausdrücken. Der Effekt der zweiten Aufgabe zur Vertiefung des Umgangs mit dem Scanner wird durch die geänderte Herangehensweise im Rust-Ansatz stark reduziert.
Ein weiterer Kritikpunkt, den wir hier anführen möchten, ist der Umfang der mitgelieferten Testfälle. Im Falle der ersten Aufgabe wird zu Testzwecken in beiden Ansätzen dasselbe Testprogramm mitgeliefert. Dieses deckt jedoch nicht die vollständige Menge der Tokens ab. Anhand dieses Testprogramms können die Studierenden somit nicht sicherstellen, ob ihre Lösung in allen Fällen den Erwartungen entspricht. Ein umfangreicheres Testprogramm, das jeden Token mindestens einmal enthält, wäre hier sinnvoller.

Handparser

Wie bereits in Abschnitt 2.2 besprochen wurde, dient diese Teilaufgabe der Implementierung eines Handparsers. Gemeinsam mit der anschließenden Aufgabe deckt sie das Themenfeld der syntaktischen Analyse ab. Im Rahmen der Aufgabenstellung ist vorgegeben, dass der handgeschriebene Parser nach dem Prinzip des rekursiven Abstiegs implementiert werden soll. Dabei wird schriftlich nur aus den vorgegebenen Konventionen zur Benennung von Variablen und Funktionen deutlich, dass das Ziel die Implementation eines LL(2)-Parsers ist. Fragen, die sich hierbei ergeben könnten, wie die Erfüllung der LL(k)-Eigenschaften oder die notwendige Umformung der Grammatik, werden in der Aufgabenstellung nicht direkt behandelt. Eine Besprechung dieser Grundlagen findet in der jeweiligen Übungsstunde statt. Dort werden wie auch bei den anderen Übungsblättern die jeweilige Aufgabe präsentiert und besprochen.
Ein weiterer Kritikpunkt, welchen wir anmerken wollen, betrifft den Umfang des zu erstellenden Codes. In beidem Übungsansätzen wird keine Datei mit angegeben, welche vervollständigt werden soll. Die Studierenden müssen den Handparser vollständig selbst implementieren. Unserer Ansicht nach löst dieses Vorgehen den Fokus von den compilerbaubezogenen Inhalte, da auch unter anderem die Auswertung des Tokenstreams, der vom Lexer erzeugt wird, selbst implementiert werden muss.
Da die Aufgabenstellung nicht explizit auf die Besonderheiten von LL-Parser eingeht, muss von den Studierenden die jeweilige Übungsstunde besucht werden, um so

einen vollständigen Überblick über den Schwerpunkt der Übungsaufgabe zu erlagen. Wir sind der Ansicht, dass Inhalte, welche den Schwerpunkt der Aufgabe betreffen, ebenfalls in der Aufgabenstellung schriftlich festgehalten werden sollten, um diese auch den Studierenden zu vermitteln, welche die Übungsstunde nicht besuchen.

Parser

In diesem Abschnitt wollen wir die Übungsaufgabe zum Thema des Parsergenerators besprechen. Wie auch schon bei den vorherigen Aufgaben nutzen beide Ansätze die grundlegend gleiche Aufgabenstellung. Das Besondere an dieser Aufgabe ist, dass auch beide Ansätze dasselbe Metawerkzeug verwenden, jedoch mit verschiedenen Zielsprachen. Im C-Ansatz wird GNU Bison verwendet, welches zum Zeitpunkt der Arbeit in Version 3.8.2 auf dem Referenzsystem für die Korrektur der Übungsaufgaben installiert ist.
Im Rust-Ansatz wird ebenfalls Bison verwendet, allerdings mithilfe des Rust-Bison-Skeletons. Diese Implementierung steht seit September 2020 zur Verfügung, und für die Aufgaben im Rust-Ansatz wird die Version "0.41.0"genutzt. Aufgrund seiner Zugehörigkeit zum GNU-Projekt ist GNU Bison eine gängige Wahl für die Sprachen C & C++. In Rust hingegen stehen im Vergleich zur Bison-Implementierung auch andere Optionen zur Verfügung. Aufgrund der Tatsache, dass das Rust Übungskonzept direkt aus dem C-Ansatz abgeleitet wurde, wird hier leider keine besser geeignetere Bibliothek verwendet.
Wir möchten hier allgemein eine Kritik am Eingabeformat von Bison anbringen. Durch die unintuitive Kombination aus C- bzw. Rust-Syntax und der Bison-Syntax zur Notation der Grammatik wirkt das Eingabeformat sehr unübersichtlich.
Darüber hinaus wird, wie auch schon beim zweiten Übungsblatt, nur eine unzureichende Menge an Testfällen bereitgestellt. Im C-Ansatz steht lediglich eine Testeingabe zur Verfügung, die nicht den vollständigen Regelumfang abdeckt. Im Rust-Ansatz sind neben diesem unzureichenden Testprogramm drei weitere Testfälle enthalten, die lediglich das vorhandensein bzw. nichtvorhandensein von Parsingfehlern überprüft.
Im Allgemeinen stellt diese Übungsaufgabe jedoch eine sinnvolle Behandlung von Parsergeneratoren dar und bildet somit einen weiteren Schritt für die Entwicklung des geplanten Übungsinterpreters.

Semantische Analyse

Nun wollen wir die Übungsaufgabe zum Thema der semantischen Analyse in beiden Ansätzen betrachten. Als Grundlage nutzen beide Ansätze für diese Aufgabe den Bison-Parser der vorherigen Übungsaufgabe. Die semantische Analyse wird sowohl im C- als auch im Rust-Ansatz in den Parser integriert. Dadurch werden die syntaktische Analyse und die semantische Analyse stark vermischt. Dies kann zu mehreren Nachteilen führen. Für die Studierenden sinkt die Übersichtlichkeit über die einzelnen Komponenten des Compilerbaus und damit verbunden auch das Verständnis für die Aufgabenbereiche der einzelnen Teilabschnitte. Ebenfalls nimmt durch diese

Herangehensweise die Übersicht über die zu bearbeitenden Übungsinhalte ab. Es ist außerdem zu erwähnen, dass durch diese Vermischung der Komponenten auch die Modularität des gesamten Übungskonstrukts abnimmt. Ein Austausch von zum Beispiel dem Parsergenerator wäre durch diese Verknüpfung nur schwer zu realisieren. Wir möchten im Folgenden auf einen weiteren Punkt eingehen, der besonders im C-Ansatz hervorsticht. In beiden Ansätzen werden die zu erfüllenden semantischen Regeln in der Übungsaufgabe angegeben. Im C-Ansatz gibt es jedoch keine Indikation in der zu vervollständigenden Parser-Datei, an welcher Stelle Änderungen oder Ergänzungen vorgenommen werden sollen. Dies erschwert den Studierenden die Orientierung in der zu erweiternden Parser-Datei.

Interpretation

Zum Abschluss der Bewertung der bisherigen Übungsaufgaben wollen wir die Übungsaufgabe zur Interpretation besprechen. Der Rust-Ansatz verfügt nur über 5 Aufgabenblätter. Eine Übung zur Interpretation als Folge der semantischen Analyse ist nicht enthalten. Aus diesem Grund können wir somit nur die Übungsaufgabe im C-Ansatz besprechen.
Unter Betrachtung der Terminplanung der Vorlesung (exemplarisch am Sommersemester 2023)[9] gehen wir davon aus, dass das Fehlen einer Aufgabe zu Interpretation oder Codegenerierung dem geänderten Tempo der Vorlesung geschuldet ist. Durch den angepassten Umfang der Vorlesungsinhalte war bei der Ableitung des Rust-Ansatzes aus dem C-Ansatz keine Übungsaufgabe mehr notwendig. Im Sommersemester 2023 wurde erstmals neben den praktischen Übungsaufgaben ein Projekt zur Entwicklung eines Interpreters in ANTLR[10] eingeführt. Dieses Projekt übernimmt für die Studierenden die praktische Auseinandersetzung mit dem Thema der Interpretation. Wir sehen es trotzdem als negativ an, dass durch das Angebot eines neuen Projekts das klassische Übungskonzept eingekürzt wurde, da so das aufeinander aufbauende Lernziel zur Entwicklung eines Übungsinterpreters nicht mehr erreicht wird.
Im **C-Ansatz** hingegen wird mit dieser Übungsaufgabe ein Abschluss zur Serie der praktischen Aufgaben gefunden. Die Aufgabenstellung gibt lediglich vor, dass mit `TODO` markierte Stellen im Quellcode zu vervollständigen sind. Ebenfalls dürfen explizit beliebige Änderungen an den vorhandenen Strukturen vorgenommen werden. Dadurch ergibt sich ein sehr großer Spielraum für die Bearbeitung. Die drei als Hilfestellung mitgelieferten Testprogramme bilden den einzigen Anhaltspunkt für die Korrektheit der Bearbeitung durch die Studierenden. Wir sehen darin ein Problem, da einige der zu implementierenden Abschnitte im Quellcode durch keines der drei Testprogramme abgedeckt werden. Dazu zählen unter anderem globale Variablen oder `while`-Schleifen. Ebenfalls werden keine korrekten Ausgaben für die Testprogramme mitgeliefert. Somit müssen die Studierenden zum Vergleich mit ihrer Lösung eigenständig ein Schreibtischtest für die Testprogramme durchführen. Bei Fehlern im Schreibtischtest kann somit das gesamte Testprogramm falsch evaluiert werden.

[9] `https://www.informatik.hu-berlin.de/de/forschung/gebiete/se/teaching/ss2023/cb`
[10] `https://www.antlr.org/`

Bei einem Umfang von nur drei Testeingaben erhöht dies das Fehlerrisiko bei der Selbstkontrolle des Interpreters. Wir denken, dass diese Fehlerquelle am besten durch die Bekanntgabe von vergleichbaren Testausgaben behoben werden könnte.
Darüber hinaus sehen wir die zu vervollständigenden Codeabschnitte als nur wenig geeignet an. Insgesamt sind 24 Teilabschnitte im Code zu vervollständigen. Davon ist ein Großteil nur das erneute Implementieren von bereits vorgegebenen Fällen mit geringen Änderungen, wie im Falle der Rechenoperatoren oder der Vergleichsoperatoren. Wir denken, es würde für die Studierenden ein besserer Lerneffekt erzielen, wenn in solchen Teilaufgaben keine Fälle vorgegeben wären und diese vollständig von den Studierenden erarbeitet werden müssen.
Im nächsten Abschnitt wollen wir aus den Aufgabenstellungen unter Beachtung unserer Bewertungskriterien und wichtiger Punkte ableiten, die wir als Grundlage für die Formulierung unserer Übungsaufgaben nutzen wollen.

2.4. Kriterien

In diesem Teilabschnitt wollen wir aufgrund der Bewertung der beiden Übungsansätze im vorherigen Abschnitt nun einige Kriterien und Punkte ableiten, welche für die Erstellung eines Übungskonzepts wichtig sind. Diese wichtigen Punkte und Kriterien wollen wir im späteren Verlauf nutzen und sie für die Erstellung unseres neuen Übungskonzepts berücksichtigen.
Die Struktur beider Ansätze beginnt jeweils mit einem einführenden Übungsblatt. Die Aufgabe auf diesem Blatt dient dabei als Einstieg in die genutzte Sprache und schafft mit den dort behandelten Strukturen bereits eine Grundlage für die späteren Übungsaufgaben. Wie wir in Abschnitt 2.1 erfahren haben, werden den Studierenden im Laufe ihres Studiums lediglich Programmiergrundlagen in der Sprache Java vermittelt. Somit kann zu Beginn der Übung nicht davon ausgegangen werden, dass die Studierenden entsprechendes Vorwissen in C oder Rust haben. Wir sehen dieses Vorgehen als sinnvoll an und nehmen daher den Punkt auf, dass eine Einstiegsaufgabe eine sinnvolle Idee darstellt, um den Studierenden den Einstieg in eine neue Sprache zu erleichtern. Als optionales Ziel wollen wir dabei aufnehmen, dass ein Zusammenhang zum Thema Compilerbau geeignet ist, um auch einen inhaltlichen Einstieg für die weiteren Aufgaben zu bieten.
Ein ebenfalls wichtiges Kriterium stellt die Evaluierungsmöglichkeit der Übungsaufgaben sowohl durch die Studierenden als auch für die Korrektur dar. So konnten wir beobachten, dass in jeder der Übungsaufgaben in beiden Ansätzen Testeingaben oder direkte Testfälle mitgeliefert werden. Diese Testbarkeit der Übungsaufgaben erfüllt dabei mehrere Aufgaben. Als wichtigsten Anwendungsfall gilt dabei die Selbstkontrolle der Studierenden. Durch Testfälle können noch bestehende Fehler identifiziert werden. Ebenfalls schafft eine ausreichend große Testumgebung zusammen mit der Aufgabenstellung eine klare Zielsetzung für die Studierenden. Ein weiterer wichtiger Anwendungsfall für eine Testumgebung ist die spätere Korrektur der Übungsaufgaben. Wir können somit eine umfangreiche Testumgebung zu den wichtigen Kriterien für unseren neuen Ansatz zählen.

Wir können ebenfalls sowohl im C- als auch im Rust-Ansatz eine aufeinander aufbauende Struktur erkennen. Dies wird in beiden Ansätzen besonders daran deutlich, dass in einigen Aufgabenstellungen darauf hingewiesen wird, dass auch die Lösungen der vorherigen Übungsaufgaben im weiteren Verlauf der Aufgaben verwendet werden können. Durch dieses Konzept wird der Zusammenhang der einzelnen Teilbereiche des Compilerbaus deutlich. Wir erachten eine solche aufbauende Vorgehensweise als sinnvoll, um so den Studierenden die Möglichkeit zu geben, den Übungsinterpreter so vollständig wie möglich selbst zu implementieren. Ein Aufbau des Übungskonzepts, in dem bis auf einige Hilfsstrukturen keine weiteren Vorgaben gemacht werden, könnte ein positives Erfolgserlebnis erzeugen. Wir wollen diesen Punkt somit als optionales Kriterium aufnehmen.
Als letzten Punkt wollen wir das Programmierziel der Übungsaufgaben betrachten. Im C-Ansatz wird als vollständiges Ziel der Aufgaben die Entwicklung eines Interpreters angestrebt. Es wurde sich dabei für die Lösung entschieden, einen Interpreter anstelle eines Compilers zu entwickeln. Der Grund dafür war, den Arbeitsaufwand der Programmieraufgaben an den ursprünglichen veranschlagten Umfang von 5 Leistungspunkten (SPO 2015) und dem damit verbundenen vorgegebenen Zeitaufwand anzupassen. Durch die Umstrukturierung der Vorlesungsinhalte wurde damit einhergehend auch der Umfang praktischen Übungsaufgaben erneut reduziert. Seit dieser Änderung des bisherigen Konzepts wurde dauerhaft die Übungsaufgabe zur Interpretation gestrichen. Diese Aufgabe wurde somit im Rust-Ansatz gar nicht erst konzipiert. Dadurch entfällt einerseits die Möglichkeit, einen vollständigen Interpreter zu implementieren, als auch die praktische Vermittlung eines Interpreters, als Teil dieser praktischen Aufgaben (seit SoSe 2023 durch das ANTLR-Projekt abgedeckt). Wie wir aus dieser Entwicklung ableiten können, ermöglicht eine Streichung von optionalen und eine Reduzierung von aufwendigen Aufgaben, wie der Codeoptimierung und der Codegenerierung die mögliche Umsetzung eines vollständigen Interpreters. Aus diesem Grund wollen wir unter Streichung von Codeoptimierung und Codegenerierung die Entwicklung eines Interpreters anstreben. Wie wir bei der Besprechung der semantischen Analyse festgestellt haben, kann die Vermischung von Teilbereichen des Compilerbaus in den Übungsaufgaben zu Problemen führen. Wir wollen aus diesem Grund eine Trennung der einzelnen Komponenten anstreben, um so eine flexible Gestaltung des Übungsbetriebs zu ermöglichen.
Nachdem wir nun als Folge der Betrachtung der bisherigen Übungskonzepte einige Kriterien und wichtige Eckpunkte ermittelt haben, wollen wir im nächsten Abschnitt der Arbeit die Entwicklung unseres neuen Übungskonzepts besprechen.

3. Ausarbeitung

In diesem Abschnitt der Arbeit möchten wir nun unser neu entwickeltes Übungskonzept vorstellen. Wir haben uns dafür entschieden, zunächst einen vollständigen Interpreter zu entwickeln, welcher im späteren Verlauf aus Grundlage für die Ausarbeitung der Übungsaufgaben dienen soll. Zunächst werden wir Entwurfskriterien festlegen, die als Grundlage für die Entwicklung des Interpreters und der späteren Aufgaben dienen sollen. Dabei werden auch die in Abschnitt 2.4 erarbeiteten Kriterien einfließen. Im weiteren Verlauf werden wir dann die Entwicklung des Interpreters besprechen. Dabei werden wir unter anderem auf unsere Designentscheidungen eingehen, sowie Probleme in der Entwicklung ansprechen und diese anhand von Beispielen vertiefen. Der letzte Teil dieses Kapitels wird sich mit der Erstellung und Ausarbeitung der Übungsaufgaben befassen. Wir werden ebenfalls erläutern, welche Zielsetzung jede der ausgearbeiteten Aufgaben erfüllt und wie sie indas Grundkonzept des Übungsplans passen. Anschließend werden wir im nächsten Kapitel das bisherige Übungskonzept mit unserem neu entwickelten Konzept vergleichen.

3.1. Entwurfskriterien

Im Laufe dieses Kapitels wollen wir Kriterien ausarbeiten, welche als qualitative Grundlage unseres neuen Übungskonzepts dienen sollen. Dabei werden wir sowohl neue Kriterien mit einbringen als auch die Ergebnisse der Auswertung der bisherigen Übungskonzepte beachten.
Das Ziel unseres neuen Ansatzes soll die Ausarbeitung eines neuen Übungskonzepts sein, welches sowohl positive Ansätze und Vorgehensweisen aus den bisherigen Ansätzen mit aufgreifen, als auch neue Vorgehensweisen mit einbringt. Ebenfalls setzen wir uns als Ziel, eine gute Nutzbarkeit zu bieten, um so den Fokus beim Bearbeiten der Übungsaufgaben voll auf die Inhalte zu lenken.
Diese im Folgenden erarbeiteten Entwurfskriterien wollen wir zur besseren Übersicht in vier Kategorien aufteilen, sodass jede der Kategorien ein gesondertes Teilgebiet der Kriterien behandelt.

Interaktionskriterien

Als Interaktionskriterien wollen wir Kriterien zusammenfassen, welche sich auf die Interaktion unserer Zielgruppe mit dem Interpreter und besonders mit den Übungsaufgaben beziehen. Dabei sollen sowohl die Nutzergruppe der Studierenden, die die Aufgaben bearbeiten, als auch die Personengruppe, welche die Einreichungen der Studierenden bewerten, berücksichtigt werden. Wir wollen zunächst den Fokus auf die größte Schnittstelle legen und uns mit dem Design der Aufgabenstellungen beschäftigen.
Wie auch in den bereits besprochenen Ansätzen wollen wir uns zum Ziel setzen, für jede der späteren Übungsaufgaben eine umfassende Aufgabenstellung mitzuliefern. Wir wollen eine einheitliche Struktur der Übungsaufgaben entwerfen, in welcher die

wichtigen Punkte vermittelt werden. Unter Betrachtung der Aufgabenstellungen aus dem C- und Rust-Ansatz sollen die folgenden Informationen in allen unseren neuen Aufgabenstellungen enthalten sein:

- Um später einen reibungslosen Übungsbetrieb gewährleisten zu können, sollen in einem Abschnitt der Aufgabenstellung **Allgemeine Hinweise** zum Ablauf des Übungsbetriebs mitgeteilt werden. Darin sollen insbesondere minimale Ausschlusskriterien bestimmt werden, unter welchen keine abgegebene Lösung akzeptiert werden kann.
- Unter Beachtung einer späteren Kontrolle sollen auch Informationen zum **Abgabemodus** mitgeteilt werden. Dieser Abschnitt soll einen Überblick über die zu bearbeitenden Strukturen geben sowie Informationen zum Testen und Einreichen der Übungsaufgaben.
- Um den Studierenden eine klare Übersicht über die zu bearbeitenden Dateien zu geben, sollen diese klar in einem Abschnitt hervorgehoben werden.
- Neben den Formalitäten zur Bearbeitung einer Aufgabe sollen auch die Aufgaben beschrieben werden. Wir erachten es als sinnvoll, den Studierenden zu Beginn der Aufgabenstellung zu vermitteln, welche Themengebiete in den jeweiligen Übungsaufgaben vermittelt werden. Dafür soll auf jedem Übungsblatt das **Lernziel** klar erkennbar definiert werden.
- Ebenfalls erachten wir es als sinnvoll, den Studierenden zu Beginn einer neuen Übungsaufgabe eine Übersicht in Form einer **Kurzbeschreibung** zu geben, was im Rahmen der jeweiligen Übungsaufgabe zu erledigen ist.
- Abschließend soll für jede Teilaufgabe eines Übungsblattes eine detaillierte Beschreibung der **Aufgabenstellung** veröffentlicht werden. Dadurch sollen die Studierenden genau verstehen können, was von ihnen im Verlauf der jeweiligen Aufgabe erwartet wird und welche Implementationen vorzunehmen sind.
- Sollten die Studierenden zur Bearbeitung der Aufgaben weitere Informationen benötigen oder sollte es inhaltliche Vorgaben geben, so sollen auch diese möglichst direkt auf dem Aufgabenblatt hinterlegt werden. So ist es möglich, das gesamte Übungskonzept möglichst gekapselt von äußeren Inhalten zu betreiben.

Durch diese Vorgaben für die Aufgabenstellungen erhoffen wir uns, dass die Studierenden eine klare Vorstellung der zu bearbeitenden Inhalte bekommen. Ebenfalls soll so, durch eine eindeutige Klärung der Formalitäten, die Abgabe und spätere Bewertung der Übungsaufgaben erleichtert werden.
Durch die Wahl von Rust als Sprache unseres neuen Ansatzes ergeben sich weitere Möglichkeiten zur Aufbereitung der Übungsaufgaben. Durch die Verwendung von `Cargo` zur Verwaltung von Projekten kann jede einzelne Übungsaufgabe als Projekt angelegt werden. Cargo bietet ebenfalls den Vorteil, dass jedes angelegte Projekt als Git-Repository ausgeführt wird. So kann eine Übungsaufgabe als ein Repository

angelegt und verwaltet werden. Wie auch im bisherigen Rust-Ansatz soll die Aufgabenstellung in Form einer Readme-Datei im gekapselten Projekt vorhanden sein. Durch die Verwendung von Cargo-Projekten können so auch die Studierenden bei der Bearbeitung von den Vorteilen, wie der Versionierung durch Git, profitieren.

Inhaltskriterien

Inhaltliche Kriterien sollen sich auf die vermittelten Inhalte und die genutzte Sprache beziehen. Wie wir bereits zu Beginn dieser Arbeit in Abschnitt 1.1 festgestellt haben, bietet Rust im direkten Vergleich zu C einige Vorteile. Ebenfalls benötigt Rust, unter anderem aufgrund seiner strikten Reglementierung von Referenzen, andere Herangehensweisen an Probleme als in C. Somit wollen wir als eines der Kriterien festlegen, dass die durch die Nutzung von Rust gegebenen Vorteile bestmöglich ausgenutzt werden sollen. In diesem Rahmen soll ebenfalls versucht werden, durch die Konzeption der Aufgaben einen möglichst einfachen Einstieg in die Sprache Rust zu bieten. Diese Kriterien sollen jedoch nicht als harte Kriterien gedacht sein. Wir wollen uns bei den Inhalten, die durch die Aufgaben vermittelt werden, auf das Themengebiet des Compilerbaus fokussieren und die Sprache Rust nicht in den Fokus der Aufgaben stellen. Dementsprechend sollen Funktionen von Rust nicht auf Kosten von Inhalten des Compilerbaus genutzt werden.
Nun wollen wir die zu vermittelnden Inhalte des Compilerbaus festlegen. Dazu wollen wir besonders die Arbeit der vorherigen Ansätze mit einbeziehen. Wie wir aus den bisherigen Übungskonzepten in Abschnitt 2.4 festgestellt haben, wollen wir uns anstelle eines Compilers auf einen Interpreter beschränken. Darüber hinaus wollen wir uns für die Struktur, wie auch die Übungsblätter der bisherigen Übungskonzepte, an den in Abschnitt 1.2.2 beschrieben Teilbereichen eines Interpreters orientieren. Wir wollen jeweils in einer Übungsaufgabe die Themengebiete lexikalische Analyse, syntaktische Analyse, semantische Analyse und Interpretation behandeln. Den Abschnitt der Codeoptimierung wollen wir jedoch nicht behandeln. Dies hat die folgenden Gründe: Zum einen muss bei der Gestaltung des Übungskonzepts die zugewiesene Anzahl der Leistungspunkte und die damit verbundene Zahl an Arbeitsstunden berücksichtigt werden. Da neben den praktischen Übungsaufgaben auch theoretische Aufgaben bearbeitet werden sollen, steht nicht die volle Zeit der Arbeitsstunden für die Bearbeitung der praktischen Aufgaben zur Verfügung. Deswegen orientieren wir uns zu diesem Zweck an den bisherigen Übungskonzepten.
Zum anderen ist der Inhalt der Codeoptimierung zu beachten. Wie wir in Abschnitt 1.2.2 beschrieben haben, müssen für die Optimierung teils umfangreiche Analysen des Quellcodes vorgenommen werden. Ebenfalls kommen dabei komplexe Algorithmen zur Anwendung, die die angedachte Komplexität der Übungsaufgaben übersteigen. Wir wollen aus diesen Gründen ausnutzen, dass die Codeoptimierung als optionaler Schritt eines Compilers gilt, und diese ohne Einschränkung des gesamten Übungsinterpreters nicht durch eine Übungsaufgabe abdecken.

Entwicklungskriterien

Die hier präsentierten Entwicklungskriterien beziehen sich auf die grundlegende Gestaltung der Software. Wir wollen dabei durch geschickte Wahl verschiedener Kriterien die Entwicklungsansätze des Interpreters lenken. Wie bereits in Abschnitt 3 angedeutet, wollen wir die Struktur und den Umfang der Übungsaufgaben beeinflussen, indem wir diese aus dem zuvor vollständig implementierten Interpreter ableiten. Unser erstes Zwischenziel soll somit die Erstellung eines Interpreters sein. Durch diese Vorgehensweise soll der Umfang und die Reihenfolge der Übungsaufgaben nachvollziehbar auf das Ziel hinarbeiten, im Laufe der Übungen einen vollständigen Übungsinterpreter zu konstruieren. So kann während der Entwicklung des Interpreters bereits eine Struktur für die daraus abzuleitenden Übungsaufgaben ermittelt werden. Wir wollen uns in diesem Abschnitt ebenfalls mit der Strukturierung des Interpreters beschäftigen. Wie auch in den alten Übungskonzepten wollen wir für bestimmte Teilaufgaben, explizit den Scanner und den Parsergenerator auf externe Bibliotheken zurückgreifen. Für diese Crates sollen die folgenden Kriterien erfüllt werden:

- Der Funktionsumfang eines gewählten Crates soll die Anforderungen des Projektes bestmöglich erfüllen.
- Die gesetzte Lizenz des jeweiligen Crates muss die Nutzung in unserem Übungsinterpreter erlauben.
- Es soll eine nachvollziehbare Dokumentation des Crates vorhanden sein, um so auch den Studierenden das Verständnis und den leichten Umgang zu ermöglichen.
- Da wir Crates für Lexer und Parser verwenden wollen, sollen diese nach Möglichkeit kompatibel zueinander sein. Durch Setzen dieses weichen Kriteriums hoffen wir, die Modularität der einzelnen Komponenten zu erhöhen.

Wir hoffen, dass unter Erfüllung möglichst vieler dieser Kriterien die Qualität des Interpreters gesteigert werden kann. Ebenso können die Studierenden davon profitieren, indem sie sich nur mit wenigen qualitativ guten Crates auseinandersetzen müssen, die durch die hier festgelegten Kriterien ausgewählt wurden.

Designkriterien

Bei den Designkriterien wollen wir abschließend Kriterien betrachten, die sich unabhängig von der eigentlichen Programmierung auf die Gestaltung des gesamten Projektes und der Übungsaufgaben auswirken.
Wie bereits in Abschnitt 3.1 besprochen, soll der Umfang des Übungsinterpreters vier Teilbereiche des Compilerbaus abdecken. Wir setzen uns das Ziel, diese einzelnen Komponenten des späteren Übungsinterpreters möglichst modular zu gestalten. Dies bietet mehrere Vorteile. So könnten einzelne Komponenten leicht gegen andere ausgetauscht werden, sollte sich der inhaltliche Fokus der Übungsaufgaben ändern.

Ebenfalls bietet ein modularer Aufbau den Vorteil, dass wir so eine einfache Erweiterbarkeit des Projektes erreichen können. Bei Bedarf können wir weitere Komponenten und somit auch weitere Übungsinhalte hinzufügen. Ein weiteres Kriterium, das wir bereits in Abschnitt 2.4 besprochen haben, ist die Testbarkeit der Software und der einzelnen Übungsaufgaben. Hierzu soll das von Rust direkt mitgelieferte Feature der Unittests genutzt werden. Wie bereits erläutert, erleichtert dies sowohl den Studierenden das Bearbeiten der Übungsaufgaben als auch den Korrektoren die Korrektur und Bewertung derselben. Unser Ziel ist es, für jede der Komponenten des Übungsinterpreters und somit auch für jede der Aufgaben eine ausreichend umfassende Testsuite breit zustellen.

Wir haben nun Kriterien für verschiedene Teilbereiche unseres Projektes besprochen und erläutert, welche Vorteile die Nutzung der jeweiligen Kriterien für unser Projekt mit sich bringt. Im folgenden Teilabschnitt wollen wir unser Vorgehen bei der Entwicklung des Übungsinterpreters, der als Grundlage für die Übungsaufgaben dient, beschreiben und unsere Designentscheidungen anhand der erarbeiteten Kriterien sowie Probleme bei der Entwicklung erläutern.

3.2. Umsetzung

In diesem Abschnitt der Arbeit wollen wir die Entwicklung des Übungsinterpreters dokumentieren. Dabei werden wir in logischer Reihenfolge eines Interpreters die einzelnen Teilbereiche der Programmierung betrachten. Wir wollen besonders unsere Entscheidungen bei der Entwicklung erklären und auch die aufgetretenen Probleme besprechen und deren Lösungen erläutern.
Vorangehend wollen wir als ersten Punkt bestimmen, für welche Zielsprache wir unseren Übungsinterpreter entwickeln wollen. In beiden der bisherigen Ansätze wurde eine reduzierte Version der Sprache C, unter der Bezeichnung C1, verwendet. Diese Sprache C1 wurde von Dr. Kunert für den Einsatz in der Compilerbau-Übung an der Humboldt-Universität zu Berlin entwickelt. Die dort verwendeten Tokens sowie die verwendete Grammatik haben wir im Anhang der Arbeit in den Abschnitten 15 und 16 beigefügt. Da unser neuer Ansatz jedoch nun vollständig in Rust implementiert wird, haben wir in Betracht gezogen, eine eigene Übungssprache auf Rust-Basis zu erstellen. Wir haben uns jedoch aus den folgenden Gründen dagegen entschieden. Zum einen ist dabei der Umfang der Sprache ausschlaggebend.
Würden wir für Rust charakteristische Funktionen wie Referenzen oder eine Variante des *Ownership*-Systems in den Funktionsumfang aufnehmen, würde damit auch die Komplexität des Übungsinterpreters und der späteren Übungsaufgaben stark ansteigen. Ebenfalls würde sich durch den daraus resultierenden Programmieraufwand der Fokus von den Konzepten des Compilerbaus zu sehr auf die spezifischen Funktionalitäten von Rust verschieben. Würden wir jedoch diese charakteristischen Funktionen nicht hinzufügen, würde sich der Funktionsumfang der reduzierten Sprache nicht signifikant von der bisherigen Sprache C1 unterscheiden.
Der andere Punkt, der gegen eine reduzierte Rust-Sprache spricht, ist der Zusam-

menhang zur Vorlesung und den anderen Lehrinhalten. Besonders das Themengebiet der Speicherverwaltung zur Laufzeit wird in der Vorlesung anhand des C-Callstacks vermittelt. Die Einführung eines weiteren Systems für den Übungsinterpreter auf Basis der Rust-Speicherverwaltung zur Laufzeit erachten wir als nicht sinnvoll.
Aus diesen Gründen haben wir uns dazu entschieden, auch für unseren im Laufe dieser Arbeit entwickelten Interpreter die Sprache C1 zu nutzen.
Nun folgend wollen wir die Implementation unseres Übungsinterpreters Schritt für Schritt besprechen.

Lexer

Der erste Teilbereich des Interpreters, den wir entwickelt haben, war der Lexer. Ein Lexer erfüllt die Aufgabe, das eingegebene Programm Zeichen für Zeichen einzulesen und daraus vorgegebene Tokens zu erstellen. Für diese Aufgaben standen uns zwei verschiedene Vorgehensweisen zur Verfügung. Einerseits hätten wir, wie im C- und auch im Rust-Ansatz, eine externe Bibliothek dafür verwenden können. Anderseits hätten wir einen Lexer teilweise oder auch vollständig selbst implementieren können. Wir haben uns aus mehreren Gründen dazu entschieden, für diese Tätigkeit eine externe Bibliothek zu verwenden. Einer der wichtigsten Gründe dafür war, dass die Verwendung einer Bibliothek bessere Möglichkeiten bietet, um daraus eine Übungsaufgabe abzuleiten. Ebenfalls war die Berücksichtigung des Vorwissens der Studierenden, die die Aufgaben später bearbeiten sollten, wichtig für unsere Entscheidung. Der Umgang mit einer geeigneten Bibliothek ist für Rust-Anfänger besser geeignet als die teilweise oder vollständige manuelle Implementation eines Lexers. Darüber hinaus gehen wir davon aus, dass eine vollständige Implementation eines Lexers den zeitlichen und inhaltlichen Rahmen einer Übungsaufgabe weit überschritten hätte. Nun folgend wollen wir die Wahl einer für unseren Einsatzzweck geeigneten Bibliothek diskutieren.
Als ein Kandidat für eine passende Crate bietet sich unter anderem die im Rust-Ansatz verwendete Crate `Logos` an. Wir wollen zunächst anhand unserer aufgestellten Kriterien prüfen, ob Logos für unseren Ansatz geeignet ist. Dazu haben wir im Folgenden Logos anhand unserer vier in Abschnitt 3.1 aufgestellten Kriterien für die Nutzung von externen Bibliotheken überprüft.
Unser erstes Kriterium besagt, dass eine verwendete Crate die Anforderungen des Projektes erfüllen soll. Die Entwickler geben an, mit Logos schnelle und einfache Lexer erstellen zu können. Dabei verwendet Logos für die internen Prozesse, wie auch im theoretischen Vorgehen beschrieben, eine *deterministic state machine* [Rusc]. Damit sehen wir das erste Kriterium als erfüllt an.
Das Zweite von uns aufgestellte Kriterium besagt, dass eine verwendete Crate aufgrund seiner Lizenzbestimmungen von unserem Projekt genutzt werden können soll. Da Logos unter der MIT-Lizenz[11] und der Apache-2.0 Lizenz[12] veröffentlicht wurde, sehen wir auch dieses Kriterium als erfüllt an.

[11] `https://choosealicense.com/licenses/mit/`
[12] `https://choosealicense.com/licenses/apache-2.0/`

Als drittes Kriterium haben wir die Bedingung gestellt, dass die Bibliothek gut dokumentiert sein muss. Zu diesem Zweck haben wir die mitgelieferte Dokumentation[13] untersucht. Diese umfasst neben einem Nutzungsbeispiel auch eine Dokumentation des Funktionsumfangs. Um einen guten Überblick über den Funktionsumfang des Crates zu bekommen, haben wir uns mit der Dokumentation des Projekts beschäftigt. Dadurch konnten wir zum einen guten Überblick über die Funktionen bekommen und zum anderen uns davon überzeugen, dass diese Crate über eine einsteigerfreundliche Dokumentation verfügt. Nach unseren dadurch erworbenen Erfahrungen sehen wir auch dieses Kriterium als erfüllt an.
Das letzte der von uns gesetzten Kriterien besagt, dass nach Möglichkeit eine Kompatibilität zwischen dem Lexer und dem Parser ermöglicht werden soll. Wie wir an dieser Stelle aus dem nachfolgenden Abschnitt vorgreifen, besteht eine Kompatibilität zu dem von uns dort gewählten Parsergenerator. Dadurch wird auch dieses Kriterium erfüllt.

Nachdem wir nun geprüft haben, dass Logos für unsere Zwecke geeignet ist und all unsere Kriterien erfüllt, wollen wir diese Crate nutzen. Für die Nutzung von Logos muss eine Enumeration angelegt werden, in der die Tokens, die vom Lexer generiert werden sollen, beschrieben werden. An jedem Token wird zusätzlich annotiert, durch welchen Text bzw. durch welchen regulären Ausdruck der jeweilige Token erzeugt wird. Als Anschauungsbeispiel haben wir einen Auszug aus unserem Scanner in Listing 6 dargestellt. Wie dort zu sehen ist, werden die Tokens sehr übersichtlich angegeben, was zu einer guten Lesbarkeit des Programms führt. Als Rückgabe liefert Logos einen Tokenstream. Zu jedem Token können über Funktionen die Position in der Eingabe und der genau erkannte String ausgegeben werden. Diese Ausgabe des Lexers wollen wir nun im nächsten Abschnitt als Eingabe für den Parsergenerator nutzen.

```
96  #[token("{")]
97  LBrace,
98
99  #[token("}")]
100 RBrace,
101
102 #[regex("[0-9]+", |lex| lex.slice().parse())]
103 ConstInt(i64),
```

Listing 6.: Auszug aus der für Logos vorbereiteten Enumeration. Entnommen aus unserem Interpreter, *c1_lex.rs*

[13] https://docs.rs/logos/0.12.1/logos/index.html

Parser

In diesem Teilabschnitt wollen wir uns dem Parser widmen. Dazu werden wir die Ausgabe nutzen, die von unserem Interpreter produziert wurde. Ein Parser prüft anhand einer vorhandenen Grammatik, ob eine Tokenfolge durch die Ableitung mit der Grammatik akzeptiert werden kann. Programme, die solche Parser generieren, werden als Parsergeneratoren bezeichnet. Wir haben im Abschnitt 1.2.2 verschiedene Varianten von Parsern besprochen. Als gängiges Vorgehen in modernen Compilern werden meist tabellenbasierte Bottom-up-Parsers verwendet. Für unseren Übungsinterpreter wollen wir auf einen LALR-Parser zurückgreifen, der durch einen Parsergenerator generiert wird. Wir haben uns für einen LALR-Parser entschieden, da diese Art von Parser mächtigere Grammatiken als LL-Parser akzeptieren kann und aufgrund einer tabellenbasierten Architektur effizienter ist als handgeschriebene Parser. Ebenfalls haben wir uns dazu entschieden, den Parser nicht manuell zu implementieren, da die Komplexität einer vollständigen manuellen Implementierung den Arbeits- und Zeitaufwand einer späteren Übungsaufgabe überschritten hätte.
Im Folgenden wollen wir unsere Entscheidung für die Wahl des Parsergenerators diskutieren. Als Zielvorgabe möchten wir eine externe Bibliothek wählen, die einen Parser auf Basis einer kontextfreien Grammatik erstellt. Um den Parser inhaltlich von den anderen Teilbereichen des Interpreters zu trennen, soll dieser bereits einen Syntaxbaum ausgeben können. So entstehen ein gekapseltes Modul, das als Eingabe einen Tokenstream annimmt und einen Syntaxbaum zurückgibt. Zur Auswahl einer geeigneten Bibliothek haben wir im Crate-Verzeichnis auf Crates.io nach passenden Bibliotheken gesucht. Als Kriterium, um eine Crate in die engere Auswahl aufzunehmen, haben wir nach den Keywords *#parser* und *#grammar* gesucht. Dabei haben wir die folgenden drei Crates ausgewählt, die wir im Folgenden genauer betrachten wollen.

- pest[14] beschreibt sich selbst als Allzweck-Parser, der sich auf Zugänglichkeit, Korrektheit und Leistung fokussiert. Dabei verwendet pest PEG (Parsing Expression Grammar) als Eingabe. Diese PEGs ähneln regulären Ausdrücken, wurden jedoch erweitert, um komplexe Sprachen darstellen zu können.
- lalrpop[15] ist ein Parsergenerator-Framework, das hohe Benutzerfreundlichkeit anstrebt. Dabei sollen kompakte und leicht lesbare Grammatiken geschrieben werden können. Wie der Name schon sagt, generiert **lalrpop** tabellenbasierte LR(1)-Parser.
- rust-bison-skeleton[16] stellt technisch ein Frontend für die herkömmliche Bison-Implementation dar. Dabei wird ebenfalls eine Installation von Bison benötigt. Wir betrachten diesen Ansatz als Referenz, da er auch im bisherigen Rust-Ansatz verwendet wird.

[14] `https://crates.io/crates/pest`
[15] `https://crates.io/crates/lalrpop`
[16] `https://crates.io/crates/rust-bison-skeleton`

Nun wollen wir prüfen, ob die jeweiligen Crates unsere Kriterien für die Wahl externer Bibliotheken erfüllen. Dabei gehen wir schrittweise die Kriterien durch und betrachten jeweils die Erfüllbarkeit, für die drei infrage kommenden Crates.
Das erste Kriterium besagt, dass der Funktionsumfang einer Crate den Anforderungen des Projekts entsprechen soll. Wie wir bereits festgestellt haben, soll der Parsergenerator einen Parser aus einer kontextfreien Grammatik generieren können, der einen Tokenstream annimmt und einen Syntaxbaum ausgeben kann. Im Folgenden haben wir den Funktionsumfang hinsichtlich dieser Punkte für jede der drei Crates in der Tabelle 1 aufgeführt.

	Eingabeformat	Grammatik	Ausgabeformat
pest	String	*PEGs*	pest Tokens oder *Pairs*
LALRPOP	Logos Tokens	CFG in EBNF	Frei wählbare Rückgabe
rust-bison-skeleton	Eigenes Tokenformat	CFG in BNF	Frei wählbare Rückgabe

Tabelle 1: Gegenüberstellung der Crates pest, LALRPOP und rust-bison-skeleton bezüglich des ersten Kriteriums der zuvor definierten Entwicklungskriterien.

Wie wir anhand des Funktionsumfangs erkennen können, erfüllen alle drei Crates die Bedingung, dass eine Grammatik als Eingabe verwendet wird. Pest erfüllt als einzige der Crates nicht die Bedingung, dass die Tokens des Lexers als Eingabe verwendet werden. Ebenfalls bietet Pest keine direkte Möglichkeit, einen Syntaxbaum zu generieren. Die Ausgaben des Parsers können nur durch eine eigens implementierte Funktion in einen Syntaxbaum überführt werden. Hingegen bieten LALRPOP und das rust-bison-skeleton ähnliche Funktionalitäten. Die von LALRPOP angebotenen Funktionen sind jedoch offener gestaltet und bieten mehr Möglichkeiten. So kann eine Grammatik direkt in EBNF und nicht nur in BNF Format eingelesen werden. Auch bieten die dynamisch aufgebauten Regeln mehr Möglichkeiten, um daraus direkt einen individuellen Syntaxbaum aufzubauen.
Da Pest bereits mehrere Anforderungen unseres Projekts an den Parsergenerator nicht oder nur unzureichend erfüllt, wollen wir diese Crate im weiteren Verlauf nicht mehr betrachten.
Im nächsten Schritt wollen wir die restlichen Kriterien für *LALRPOP* und *rust-bison-skeleton* in Form der Tabelle 2 prüfen.
Abschließend zu diesem Vergleich möchten wir unsere Entscheidung für LALRPOP als Parsergenerator weiter begründen. Im direkten Vergleich der einzugebenden Grammatiken hat uns das Format von LALRPOP überzeugt. Durch die starke

[17] `https://docs.rs/lalrpop/0.20.0/lalrpop/index.html`
[18] `https://lalrpop.github.io/lalrpop/`
[19] `https://docs.rs/rust-bison-skeleton/0.41.0/rust_bison_skeleton/`

	LALRPOP	rust-bison-skeleton
Nutzungslizenz	Apache-2.0 oder MIT	MIT
Dokumentation	Docs.rs[17] Dokumentation & LALRPOP Buch[18], inklusive einem Tutorial & Beispielprogrammen.	Docs.rs[19] Dokumentation, inklusive Abschnitt zum Setup & der grundlegenden Nutzung.
Kompatibilität zu Logos	Möglich, wird explizit im Buch behandelt.	Keine native Kompatibilität, händische Umwandlung der Tokens ins passende Format ist möglich.

Tabelle 2: Gegenüberstellung der Crates LALRPOP und rust-bison-skeleton bezüglich des zweiten bis vierten Kriteriums der zuvor definierten Entwicklungskriterien.

Ähnlichkeit zur Rust-Syntax ist die Grammatik sehr übersichtlich und erleichtert so den Umgang. Ebenfalls hat uns die umfassende Dokumentation überzeugt. Durch das mitgelieferte Buch und die dortigen Tutorials war es sehr einfach, sich in LALRPOP einzuarbeiten. Des Weiteren möchten wir die einfachere Konfiguration gegenüber dem rust-bison-skeleton hervorheben. So reicht ein Hinzufügen zu den benötigten Abhängigkeiten, während beim rust-bison-skeleton eine komplexere Konfiguration benötigt wird. Allgemein bietet LALRPOP weitere komfortable Features wie das Akzeptieren von EBNF-Grammatiken, oder die Kompatibilität mit Logos, die ebenfalls einen Einfluss auf unsere Entscheidung hatten.
Nachdem wir nun anhand unserer Kriterien einen Parsergenerator gewählt haben, wollen wir im Folgenden unseren Umgang mit LALRPOP, sowie den Aufbau des Syntaxbaums besprechen. LALRPOP bietet zwar die Möglichkeit, Grammatiken direkt in EBNF einzutragen, hat dabei jedoch auch eine gravierende Einschränkung. So dürfen Ableitungsregeln nicht zu mehrdeutigen Ableitungsbäumen führen. Zum besseren Verständnis haben wir in Listing 7 ein einfaches Beispiel für mehrdeutige Ableitungsmöglichkeiten dargestellt.

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | Term
Term ::= Num
       | "(" Expr ")"
Num  ::= [0-9]+
```

Listing 7.: Beispiel einer mehrdeutigen Grammatik. Eine Ableitung kann sowohl rechts als auch linksseitig erfolgen.

Wenn wir nun versuchen, die Folge `2+3*4` anhand dieser Grammatik abzuleiten, können dadurch je nach Vorgehen zwei verschiedene Ableitungen und unterschiedlichen Ergebnissen entstehen. Diese sind:

$$2 + 3 - 4 \rightarrow 2 + \ Expr(3, 4) \rightarrow Expr(2, Expr(3, 4))$$

oder

$$2 + 3 - 4 \rightarrow Expr(2, 3) - 4 \rightarrow Expr(Expr(2, 3), 4)$$

Wie in diesem kleinen Beispiel gezeigt, können mehrdeutige Grammatiken zu verschiedenen Ableitungen führen, weswegen LALRPOP diese Mehrdeutigkeiten nicht erlaubt. Daher müssen wir somit unsere Grammatik für die Sprache C1 auf Mehrdeutigkeiten überprüfen und diese durch Umformen der Grammatik auflösen. Diese Problemstellung ergab sich bei der Verwendung von Bison in den alten Übungskonzepten nicht, da dort, wie in Abschnitt 2.2 erläutert, Mehrdeutigkeiten durch Präferenzen entschieden wurden.
Bei der Analyse der C1-Grammatik aus Listing 16 haben wir festgestellt, dass an einer Stelle eine Mehrdeutigkeit besteht. Im Folgenden erläutern wir, wie wir diese Mehrdeutigkeit umgeformt haben, um das Problem zu beheben.

Dangling Else

Die Problemstellung, mit der wir uns beschäftigen, wird allgemein als *dangling else* bezeichnet. Bei diesem Problem geht es um die Zuordnung von `else`-Statements zu `if`-Statements. Da in unserer Grammatik auch ein einzelnes Statement als `statblock` eines `if`-Statements zulässig ist, kann es zu uneindeutigen Konstellationen kommen. Ein Beispiel dafür haben wir in Listing 8 dargestellt. Dabei ist ohne weitere Einschränkung nicht klar, ob das `else` (Zeile 5) dem inneren `if` (Zeile 3) oder dem äußeren `if` (Zeile 2) zuzuordnen ist.

```
1 void main() {
2     if (true)
3         if (true)
4             printf("If");
5         else
6             printf("Else");
7 }
```

Listing 8.: Beispiel eines Dangling Else. Eine Zuordnung des `else` zu einem der beiden `if` ist nicht ohne Bindungsvorschriften möglich.

Nun folgend wollen wir erläutern, wie wir diese Uneindeutigkeit in unserer Grammatik aufgelöst haben. Zu diesem Zweck konnten wir uns für eine der beiden Bindungsmöglichkeiten entscheiden. Dabei steht zur Wahl, ob ein `else`-Statement immer an das innerste oder an das äußerste `if`-Statement bindet. Wir haben uns dazu entschieden, dieselbe Bindungsvorschrift wie in der Sprache C zu implementieren und `else`-Statements an das innerste `if`-Statement zu binden. Im Folgenden wollen wir beschreiben, wie wir die Grammatik umgeformt haben, um diese Bindungsvorschrift umzusetzen. Zur Vereinfachung betrachten wir im folgenden Prozess nur die Produktionsregeln, in denen der Statement-Block im `if`-Statement zu einem nicht separat geklammerten einzelnen Statement abgeleitet wird. Die Bindung eines `else`-Statements an das innerste `if`-Statement können wir allgemeingültig mit der folgenden Aussage beschreiben:

> Der Block eines `if`-Statements mit Else-Block darf rekursiv nicht zu einem `if`-Statement ohne Else-Block abgeleitet werden.

Daraus können wir ableiten, dass wir den Block eines `if`-Statements mit Else-Block einschränken müssen. In diesem Fall führen wir eine Fallunterscheidung ein. Wir formen zunächst die Regel:

$$ifstatement ::= < KW\_IF > \text{"("} assignment \text{")"} statblock (< KW\_ELSE > statblock)?$$

in die folgende Form um:

$$\begin{aligned} ifstatement ::= & < KW\_IF > \text{"("} assignment \text{")"} statblock \\ & | < KW\_IF > \text{"("} assignment \text{")"} stateblock\_no\_if < KW\_ELSE > \\ & statblock \end{aligned}$$

Bei dieser Umformung haben wir eine weitere Regel eingefügt. Die Ableitungsregel `statblock_no_if` entspricht dabei der Regel `statblock`, mit der Einschränkung, dass keine der Regeln, welche rekursiv aus dieser Regel abgeleitet werden können, zu einem `if`-Statement ohne Else-Block führen kann. Darauf folgend haben wir noch weitere, hier nicht im Detail dargestellte, Regeln eingeführt. Die Ableitungsregeln

- `block_no_if`
- `statement_no_if`
- `forstatement_no_if`
- `whilestatement_no_if`
- `ifelsestatement`

unterliegen derselben Einschränkung wie die `statblock_no_if` Regel. Um im Prozess der Ableitung wieder zu einem `if`-Statement ohne Else-Block ableitet werde zu können, kann die Menge dieser eingeschränkten Regeln durch eine der folgenden Ableitungen verlassen werden:

1. `"{"statementlist "}"`-Block, oder
2. einem `dowhilestatement`-Statement

Sowohl durch die Ableitung zu einem geklammerten Block als auch durch ein `do while`-Statement wird der Anfang und das Ende eines Segments begrenzt, was die Struktur des Codes in diesem Teilabschnitt wieder eindeutig macht.
Durch diese Umformungen konnten wir sicherstellen, dass Else-Blöcke in der von uns vorausgesetzten Weise an If-Blöcke gebunden werden und die Mehrdeutigkeit der Regeln aufgelöst wurde. Eine vollständig umgeformte Grammatik, in der alle Regeln enthalten sind, ist in Listing 18 dargestellt. Nachdem wir nun unsere Grammatik angepasst haben, kann LALRPOP daraus einen Parser generieren, der bereits den eingegebenen Tokenstream von Logos anhand der Grammatik ableiten kann. Im nächsten Schritt wollen wir den Parser so erweitern, dass beim Durchlaufen der Ableitungsregeln ein Syntaxbaum konstruiert werden kann. Zu diesem Zweck haben wir zunächst die Enumerationen und Strukturen für den Syntaxbaum angelegt. Da Rust die Möglichkeit bietet, einen Datenteil an eine Enumerationsvariante anzufügen, können wir diese Funktionalität nutzen, um unseren Syntaxbaum strukturieren. Wir haben uns in den meisten Fällen dazu entschieden, Blattknoten als Strukturen und innere Knoten als Enumeration darzustellen. Auf diese Weise schaffen wir es, den Syntaxbaum so einzuschränken, dass ausschließlich eine syntaktisch korrekte Grammatik abgebildet werden kann. Dieses Verfahren haben wir an dem in Listing 9 dargestellten Auszug der Struktur des Syntaxbaums veranschaulicht. Wie wir aus der Grammatik wissen, besteht ein Programmelement entweder aus der Deklaration einer Variable oder aus einer Funktionsdefinition. In der Enumeration `Program` sind diese beiden Möglichkeiten gekapselt. Aufgrund der Einschränkung des Datentyps im Datenteil der Enumerationsvarianten können wir sicherstellen, dass nur zulässige Knoten des Syntaxbaums auftreten können. Im Fall einer Funktionsdefinition oder der Deklaration einer Variable werden mehrere Parameter benötigt. Hierfür greifen wir auf eine Struktur zurück, in der wir die jeweiligen Parameter ablegen. Auch hier wird durch die Wahl der Datentypen die möglichen Zustände der Variablen auf die zulässigen Wertebereiche beschränkt.

```rust
#[derive(Debug, PartialEq)]
pub struct SyntaxTree {
    pub program: Vec<Program>,
}

#[derive(Debug, PartialEq)]
pub enum Program {
    DeclassAssign(DeclassAssignment),
    FunctionDefinition(FunctionDefinition),
}

#[derive(Debug, PartialEq)]
pub struct FunctionDefinition {
    pub func_type: SymbolType,
    pub func_name: String,
    pub func_param: Vec<Parameter>,
    pub func_body: FunctionBody,
}

[...]

#[derive(Debug, PartialEq)]
pub struct DeclassAssignment {
    pub typ: SymbolType,
    pub name: String,
    pub assign: Option<Assignment>,
}
```

Listing 9.: Auszug aus dem grundlegenden Syntaxbaum, welcher vom Parser generiert wird. Entnommen aus unserem Interpreter, *syntax_tree.rs*

LALRPOP bietet den Vorteil, dass der Aufbau einer Regel dem Aufbau einer Funktion in Rust stark ähnelt. Der Rückgabewert jeder Regel kann individuell festgelegt werden. Ebenfalls kann in einem Rust-Codesegment, das zu jeder Regel hinzugefügt werden kann, die Rückgabe der tieferen Ableitungsschritte ausgewertet werden, falls nötig. Zusammen mit dem von uns entwickelten Syntaxbaum ist es nun komfortabel möglich, den Syntaxbaum zu konstruieren. Dies möchten wir an einem kleinen Beispiel zeigen. In Listing 10 ist eine Ableitungsregel dargestellt, die noch keinen Knoten des Syntaxbaums konstruiert. An dieser Regel müssen wir nun einige kleinere Änderungen vornehmen.
Im ersten Schritt weisen wir den Rückgabewerten der tieferen Ableitungsschritte jeweils eigene Bezeichner zu. Unter diesen Bezeichnern kann im Rust-Codesegment auf die jeweiligen Werte zugegriffen werden.

Im zweiten Schritt fügen wir das Rust-Codesegment hinzu und konstruieren dort den passenden Knoten des Syntaxbaums. Durch die gewählte Struktur unseres Syntaxbaums können wir so sicherstellen, dass die entsprechenden Ableitungsregeln Werte vom jeweils passenden Datentyp zurückgeben.
Im letzten Schritt tragen wir den zum Knoten des Syntaxbaums passenden Rückgabedatentypen in den Regelkopf ein.
Die nach diesen drei Schritten modifizierte Regel haben wir in Listing 11 dargestellt. Damit konnten wir anschaulich zeigen, wie einfach die Konstruktion eines Knotens des Syntaxbaums in LALRPOP ist.

```
196 ifelsestatement: () = {
197     "if" "(" assignment ")" block_no_if "else" block_no_if,
198 };
```

Listing 10.: Beispielhafte Darstellung einer Regel im Eingabeformat von LALRPOP. Entnommen aus unserem Interpreter, *c1_pars.LALRPOP*

```
196 ifelsestatement: Statement = {
197     "if" "(" <assign:assignment> ")" <if_block:block_no_if> "else"
    ↪ <else_block:block_no_if> =>
198     Statement::IfStatement{
199         assign, if_block, else_block: Some(else_block),
200     },
201 };
```

Listing 11.: Beispielhafte Darstellung einer Regel im Eingabeformat von LALRPOP, welche einen Knoten des Syntaxbaums generiert und zurückliefert. Entnommen aus unserem Interpreter, *c1_pars.LALRPOP*

In diesem Abschnitt haben wir zunächst besprochen, aus welchen Gründen wir uns für LALRPOP als Parsergenerator entschieden haben. Anschließend konnten wir in der Grammatik die Uneindeutigkeit des "dangling Elseäuflösen und haben die Konstruktion des Syntaxbaums besprochen. Abschließend haben wir der für LALRPOP aufbereiteten Grammatik die Konstruktion des Syntaxbaums hinzugefügt. Somit verfügt unser Übungsinterpreter nun bereits über einen Parser, der einen Syntaxbaum konstruiert, auf den wir anschließend im nächsten Schritt der Arbeit die semantische Analyse anwenden können.

Semantische Analyse

In diesem Teilabschnitt wollen wir die semantische Analyse betrachten. Dabei soll beim Besuchen der Knoten des Syntaxbaums die Einhaltung semantischer Vorschriften der Sprache C1 geprüft werden. Diese orientieren sich an der Semantik der Sprache C. Ebenfalls soll in diesem Schritt im Abschluss des Front-Ends des Interpreters eine Zwischenrepräsentation des Codes erzeugt werden, die ans Backend weitergeleitet wird. Für unseren Übungsinterpreter haben wir uns dazu entschieden, für diese Zwischenrepräsentation eine modifizierte Darstellung unseres Syntaxbaums zu verwenden. Um diese Schritte durchzuführen, müssen wir alle Knoten unseres Syntaxbaums besuchen und abhängig vom Knotentyp bestimmte Operationen ausführen. Zur Erleichterung dieser Aufgabe haben wir uns dazu entschieden, das Visitor Design Pattern zu implementieren.
Beim Visitor Design Pattern handelt es sich um eines der Entwurfsmuster der objektorientierten Programmierung. Das Visitor Pattern wird verwendet, um die Erweiterbarkeit von Operationen auf einer Menge von Objekten zu ermöglichen, ohne dabei die ursprüngliche Klasse für jede neue Operation verändern zu müssen. Dabei wird für jede neue Operation ein externer "Visitor" implementiert [GHJV95]. Das Visitor Pattern wird in die folgenden zwei Teilkomponenten unterteilt:

- Bei den **Elementen** handelt es sich um die Klasse, die die verschiedenen Objekte repräsentiert, auf denen die Operationen ausgeführt werden sollen. Dabei implementiert jedes Element der Klasse eine Schnittstelle, die vom Visitor besucht werden kann [GHJV95].
- Der Visitor ist dabei eine abstrakte Klasse oder ein Interface, das eine Besuchsmethode (Visit), für jeden Elementtyp definiert. Jeder Visitor implementiert diese Schnittstelle und enthält verschiedene Operationen für jeden Elementtyp [GHJV95].

Die Verwendung des Visitor Patterns bietet besonders für unseren Anwendungsfall große Vorteile. Die Trennung der Verantwortlichkeiten zwischen *Element* und *Visitor* trägt entscheidend zu unserem Ziel bei, die einzelnen Komponenten des Interpreters bestmöglich modular zu gestalten. Ebenso tragen wir durch diese Vorgehensweise zur besseren Lesbarkeit und somit zu einem besseren Verständnis des Codes bei. Ein weiterer Vorteil, den wir nutzen, ist die einfache Erweiterbarkeit. Diese gilt sowohl für die Elementklasse als auch für den Visitor. Mit geringem Aufwand können sowohl weitere Operationen auf unserem Syntaxbaum implementiert werden als auch Erweiterungen bzw. Veränderungen des Syntaxbaums verarbeitet werden.
Aus diesen Gründen haben wir uns für die Implementierung des Visitor Patterns entschieden. Dazu haben wir im ersten Schritt unseren Syntaxbaum um ein entsprechendes *Trait* erweitert. Ebenso konnten wir eine Klasse erstellen, die als *Visitor* des Syntaxbaums implementiert wird und die semantische Analyse übernimmt. Als Hilfsmittel für die Prüfungen der semantischen Regeln der Sprache haben wir ebenfalls eine Symboltabelle implementiert. Im Folgenden wollen wir die Funktionalitäten vorstellen, die die Symboltabelle zur Verfügung stellt:

- Verwaltung von Sichtbarkeitsbereichen: Beim Anlegen eines neuen Sichtbarkeitsbereichs wird eine weitere Speicherebene angelegt. Dies haben wir intern als Stack von HashMaps realisiert, die Symbole mit ihrem jeweiligen Namen assoziiert speichern. Die oberste HashMap im Stack bildet dabei den aktuellen Sichtbarkeitsbereich. Beim Verlassen eines Sichtbarkeitsbereichs wird das oberste Element aus dem Stack entfernt, und der Symbolzähler, eine Markierung für die Speicherposition, wird wieder in seinen vorherigen Zustand zurückversetzt.
- Speichern von Symbolen: Bei dieser Funktion werden Symbole (Funktionen, Variablen und Parameter) in der Symboltabelle angelegt. Funktionen werden dabei gesondert von Parametern und Variablen nicht in der HashMap des aktuellen Sichtbarkeitsbereichs gespeichert, sondern in einer separaten HashMap abgelegt. Durch die Verwendung des Symbolzählers für Parameter und Variablen kann so bereits beim Ablegen eines Symbols die spätere Position im *Callstack* bestimmt werden.
- Anfrage von Symbolen: Diese Funktionalität der Symboltabelle realisiert den Zugriff auf Symbole. Diese werden anhand ihres Namens identifiziert. Wenn eine Funktion mit dem jeweiligen Namen vorhanden ist, wird das Symbol der Funktion zurückgegeben. Ansonsten wird der Stapel der Sichtbarkeitsbereiche von oben nach unten durchsucht, um das entsprechende Symbol zurückzugeben. Durch dieses Vorgehen können wir bereits auf der Ebene der Symboltabelle die Gültigkeit von Variablen im jeweiligen Sichtbarkeitsbereich prüfen.

Durch die Symboltabelle können wir nun ohne die Notwendigkeit weiterer Hilfsfunktionen die semantische Analyse durchführen. Im Folgenden wollen wir nun die einzelnen Komponenten und deren Implementierung der semantischen Analyse erklären.
Als erste Teil haben wir uns der Typenüberprüfung gewidmet. Wie wir in Abschnitt 1.2.2 gelernt haben, ist hierbei die Hauptaufgabe, sicherzustellen, dass Operationen nur auf zulässigen Typen ausgeführt werden. Um die Typen von Kinderknoten eines Knotens des Syntaxbaums zu erfahren, haben wir eine weitere Enumeration eingeführt, die alle möglichen Datentypen der Sprache C1 repräsentiert. Auf diese Weise können wir den Typ eines Knotens, sofern vorhanden und relevant, beim rekursiven Besuchen der Knoten des Syntaxbaums an seinen Elternknoten zurückgeben. So kann im entsprechenden Knoten eine Behandlung der Typen der Kinderknoten durchgeführt werden. Dadurch können wir die Typengleichheit bei `Assignments` und `Expressions` sicherstellen. Ebenso prüfen wir so die Typenkorrektheit auf `Boolean` für Bedingungen in Schleifen oder Verzweigungen.
Ein weiterer wichtiger Punkt ist die Prüfung der Parameter bei Funktionsaufrufen. Dabei werden für die Überprüfung eines Funktionsaufrufs sowohl die Typen der Funktionsparameter als auch die Typen der übergebenen Werte benötigt. Wir müssen im Rahmen der semantischen Analyse sowohl auf die paarweise Übereinstimmung der Typen als auch auf die korrekte Anzahl der übergebenen Werte achten.
Der nächste Schritt, dem wir uns gewidmet haben, ist die Deklarationsüberprüfung. Dabei stellen wir sicher, dass Variablen vor ihrer Nutzung auch deklariert wurden.

Diese Funktion ließ sich durch die Vorarbeit mit unserer Symboltabelle leicht umsetzen. Beim Anlegen einer Variable wird diese der Symboltabelle hinzugefügt, und während jeder Nutzung wird geprüft, ob die Variable im aktuellen Sichtbarkeitsbereich verfügbar ist. Damit haben wir ebenfalls bereits den Aufgabenbereich der *Überprüfung von Sichtbarkeitsbereichen* realisiert.
Als letzten großen Teilbereich der semantischen Analyse haben wir uns der Erstellung der Zwischenrepräsentation gewidmet. Zu diesem Zweck haben wir zunächst, basierend auf dem bisher verwendeten Syntaxbaum, eine neue Struktur angelegt. Unser Ziel für diesen neuen Syntaxbaum (im Folgenden als Zwischenbaum bezeichnet) ist es vor allem, namensbezogene Referenzen aufzulösen, um so im Backend des Interpreters auf das Nachschlagen von Informationen verzichten zu können. Im Rahmen der Konstruktion des neuen Zwischenbaums haben wir die folgenden Änderungen an unserem bisherigen Syntaxbaum vorgenommen.
Bei der Auflösung der Referenzen haben wir Variablen, Parameter und Funktionen betrachtet. Bisher wurden Variablen und Parameter anhand ihres Namens identifiziert. Da diese bei der Interpretation in einer Stack-Struktur gespeichert werden, sollen diese im Zwischenbaum nur noch anhand ihrer Position im Speicher identifiziert werden. Diese Umwandlung haben wir in Abbildung 3 exemplarisch dargestellt. Im Fall von Funktionen werden auch diese bei einem Funktionsaufruf über ihren Namen identifiziert. Bei der in Abbildung 2 dargestellten Umwandlung wird der Funktionsname durch eine direkte Verknüpfung zum Knoten des Funktionskörpers ersetzt.

Ebenfalls wollen wir im Zwischenbaum implizite Typenumwandlungen spezifizieren.

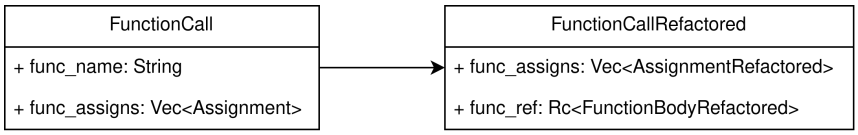

Abbildung 2: Darstellung der Umwandlung des *FunctionCall*-Knoten. Dabei wird der Funktionsname durch eine direkte Referenz auf den Knoten des *FunktionBody* ersetzt.

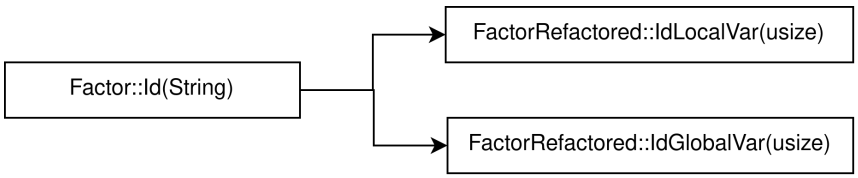

Abbildung 3: Exemplarische Darstellung der Ersetzung eines Variablen-Knoten, welcher über den Namen der Variable identifiziert wird, in einen Knoten, welcher direkt auf die Position im Speicher verweist. Dabei wird zwischen globalen und lokalen Variablen unterschieden.

In der C1-Sprache ist nur die Umwandlung von `Integer` in `Float` zulässig. Zu diesem

Zweck haben wir einen neuen Knotentyp hinzugefügt. Eine Umwandlung des Syntaxbaums durch Hinzufügen des neuen `IntToFloat`-Knoten haben wir in Abbildung 4 für den *Factor* dargestellt. Die Umwandlung läuft im Falle von *Assignment* und *Expr* analog ab.
Abschließend haben wir uns dazu entschlossen, die Form des Wurzelknotens im neuen

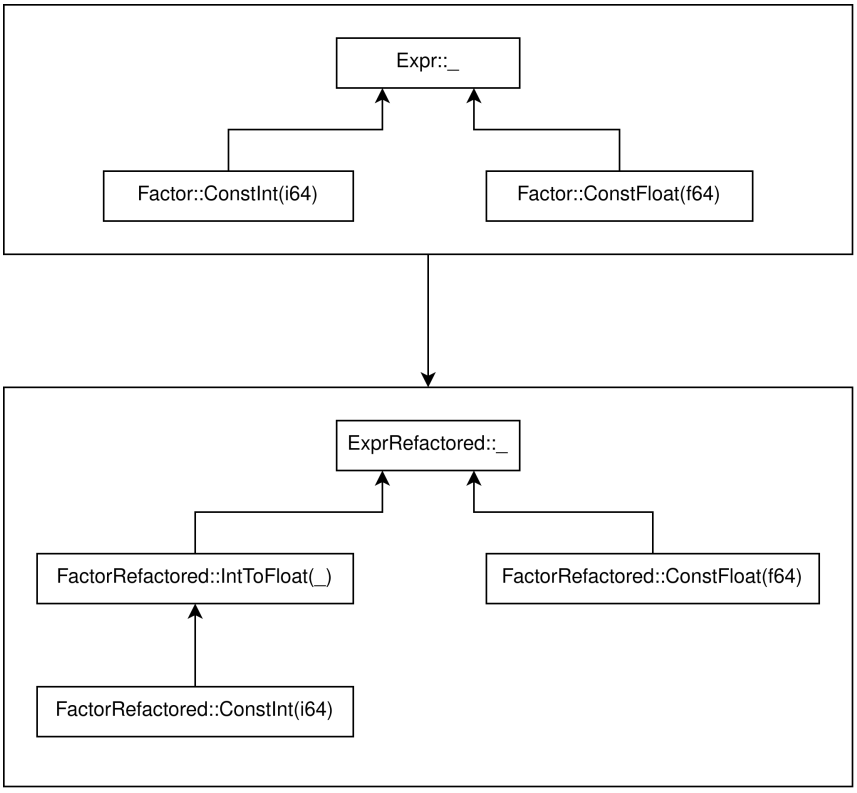

Abbildung 4: Darstellung des Einfügens eines Knotens, welcher eine explizite Typenumwandlung von Integer zu Float realisiert.

Zwischenbaum zu verändern, um das spätere Interpretieren des Codes zu erleichtern. Im bisherigen Syntaxbaum enthält der Wurzelknoten eine Liste von Knoten, die entweder eine Funktionsdefinition oder die Deklaration einer globalen Variable beschreiben können. Da wir Funktionen im ersten Schritt der Umwandlung direkt mit dem Funktionsaufruf verknüpft haben, ist eine Liste aller Funktionen nicht mehr vonnöten. Als Startknoten des Programms benötigen wir nur eine Referenz auf die Hauptfunktion des Programms (die `main`-Funktion). Ebenso müssen zu Beginn der Ausführung eines Programms die globalen Variablen bekannt sein. Daher speichern wir diese auch als Liste im neuen Wurzelknoten des Zwischenbaums.
Für die Konstruktion des Zwischenbaums gehen wir analog zur Erstellung des Syntaxbaums im Parser vor. Beim rekursiven Besuchen aller Knoten des Syntaxbaums wird in jeder Funktion des *Visiors*, wenn nötig, ein passender Knoten des neuen Zwischenbaums generiert.
Nachdem wir nun die semantische Analyse durchgeführt haben und als Abschluss den

Zwischenbaum konstruiert haben, ist damit das Frontend des Übungsinterpreters abgeschlossen. Mithilfe des Visitor Patterns haben wir eine einfache und übersichtliche Methode implementiert, um so die Einhaltung der semantischen Regeln zu prüfen und als Ausgabe einen Zwischenbaum generieren zu können. Im folgenden Abschnitt wollen wir nun als letzten Schritt des Interpreters das im Zwischenbaum gespeicherte Programm ausführen.

Interpretation

In diesem letzten Schritt des Übungsinterpreters wollen wir als einzigen Teilbereich des Backends die Interpretation des Codes durchführen. Als Eingabe für diesen Schritt erwarten wir den Zwischenbaum, der vom Frontend des Interpreters generiert wurde. Bei der Interpretation des eingegebenen Programms durchlaufen wir den Zwischenbaum und führen den von ihm repräsentierten Code aus. Wir haben uns im Vorhinein für die von uns genutzte Sprache C1 dazu entschieden, als Struktur zur Speicherung von Variablen und Parametern - ähnlich wie auch in der Ursprungssprache Sprache C- eine Speicherstack zu verwenden. Ebenso orientieren wir uns funktional am C-Callstack. Diese Speicherstruktur entspricht außerdem der Repräsentation, die im Rahmen der Vorlesung verwendet wird. Somit können wir durch die Wahl einer solchen Speicherinfrastruktur die Verbindung und Abstimmung mit der Vorlesung verbessern.
Da wir im Verlauf dieser Komponente den Zwischenbaum durchlaufen möchten, haben wir uns - ähnlich wie bei der semantischen Analyse - für die Nutzung des Visitor Patterns entschieden. Aufgrund dieser Entscheidung haben wir mit der Implementierung des entsprechenden *Traits* im Zwischenbaum begonnen. Dadurch ergab sich ebenfalls unsere Struktur des *Visitors*. Da wir für den Interpreter eine stackbasierte Struktur nutzen möchten, haben wir als Nächstes die Implementierung des Stacks sowie der weiteren Variablen und Funktionen vorgenommen. Die Struktur unseres Interpreters haben wir in Listing 12 dargestellt. Zur Speicherung von Variablenwerten und Rücksprungadressen im Stack haben wir zusätzlich eine entsprechende Enumeration angelegt. Im Interpreter repräsentiert der `stack` dabei die Speicherstruktur. Dort sollen Variablenwerte und die Rücksprungadressen beim Aufrufen von Funktionen abgelegt werden. Das Register `eax` enthält den Rückgabewert der aktuellen Funktion. Dieser ist beim Betreten einer Funktion anfangs vom Typ `void`. Der Pointer `ebp` ist der Base-Pointer. Dieser zeigt beim Aufrufen einer Funktion immer auf die Position im Stack, an welcher der vorherige Base-Pointer abgelegt wurde. Ebenfalls dient der aktuelle Base-Pointer als Referenzpunkt für lokale Variablen, von dem aus ihr jeweiliger Offset gezählt wird. Die `return_flag` gibt an, ob eine Funktion mittels eines `return`-Statements vorzeitig beendet wurde. Wenn diese Flag aktiv ist, werden die restlichen Zeilen Code in einer Funktion nicht mehr ausgeführt und die Funktion wird direkt beendet. Die letzte Variable des Interpreters gehört nicht zur Speicherverwaltung. In diesem `ausgabe`-Vektor werden die von `printf` geschriebenen Zeilen gespeichert. Dies hat den Grund, dass so Programmausgaben leichter getestet werden

können, wenn der Interpreter die produzierte Ausgabe zurückliefert und nicht direkt in der Kommandozeile ausgibt.

```
5  #[derive(Debug, Clone, PartialEq)]
6  pub enum VariableTypes {
7      Void,
8      Boolean(bool),
9      Integer(i64),
10     Float(f64),
11     Size(usize),
12 }
13
14 pub struct Interpreter {
15     stack: Vec<VariableTypes>,
16     eax: VariableTypes,
17     ebp: usize,
18     return_flag: bool,
19
20     ausgabe: Vec<String>,
21 }
```

Listing 12.: Hier ist der Aufbau der Interpreter Struktur dargestellt. Der Ausgabevektor dient dabei nur der Vereinfachung von Unittest. Entnommen aus unserem Interpreter, *interpreter.rs*

Bei der Implementierung des Interpreters haben wir nun der Visit-Methode jedes Knotentyps des Zwischenbaums seine zugewiesene Funktion ergänzt. Die Vorgehensweise ist dabei bei den meisten Knoten analog zu ihrer Funktion. Als einen der komplexen Knoten wollen wir im Folgenden noch den Funktionskörper `refactored_func_call` betrachten. Dieser Knoten stellt einen vollständigen Funktionsaufruf dar. Wir haben die einzelnen Aufgaben dieses Knotens exemplarisch im Folgenden dargestellt:

- EAX-Register zurücksetzen
- Den aktuellen EBP auf den Stack legen
- Argumente für die Funktion auf den Stack legen
- Den Funktionskörper ausführen (In unserer Implementation übernimmt der Knoten des Funktionskörpers auch das anschließende Abräumen des Stacks am Ende der Funktion)
- Wert aus dem EAX-Ausgaberegister nehmen und zurückgeben

Durch unsere Vorbereitung der Speicherverwaltung und den Aufbau durch das Visitor Pattern konnten wir einfach und übersichtlich einen Interpreter auf unserem Zwischenbaum implementieren.
Mit diesem Schritt haben wir alle einzelnen Teilabschnitte unseres Interpreters abgeschlossen. Durch die vorherige Definition unserer Entwurfskriterien konnten wir bei der Ausarbeitung des Interpreters sinnvolle Designentscheidungen treffen und eine gute Grundlage schaffen. Im nächsten Abschnitt der Arbeit wollen wir nun aus dem hier implementierten Interpreter mehrere Übungsaufgaben ableiten, welche zusammen ein vollständiges und praktisches Übungskonzept bieten sollen.

3.3. Aufbau der Aufgaben

Nachdem wir im letzten Abschnitt die Implementierung unseres Interpreters besprochen haben, möchten wir als Teil dieses Abschnitts die Übungsaufgaben besprechen, die wir aus dem Interpreter abgeleitet haben. Dabei werden wir den geplanten Arbeitsumfang, die Inhalte der Aufgabenstellung und auch die Lernziele jeder einzelnen Übungsaufgabe erläutern. Bevor wir uns jedoch den Übungsaufgaben widmen, wollen wir eine kurze Bestandsaufnahme der Struktur des Interpreters vornehmen, da dieser als Grundlage für die Mehrheit der Übungsaufgaben dient. Wir möchten zunächst den finalen Aufbau des Interpreters betrachten. Zu diesem Zweck haben wir die Struktur der einzelnen Komponenten des Interpreters in Abbildung 5 dargestellt. Hieraus wird die Modularisierung der einzelnen Komponenten deutlich. Jede der vier Teilbereiche des Interpreters ist auf eine Klasse konzentriert. Diese Struktur wird uns im Folgenden, bei der Konzeption unserer Übungsaufgaben, einige Vorteile bringen. Nachdem wir nun noch einmal kurz die Struktur des Interpreters betrachtet haben, wollen wir uns den Übungsaufgaben zuwenden. Wie wir in unseren Entwurfskriterien festgelegt haben, möchten wir als Einstieg eine Aufgabe vorbereiten, die noch nicht im direkten Zusammenhang mit dem Interpreter steht und als inhaltliche Einführung dient.

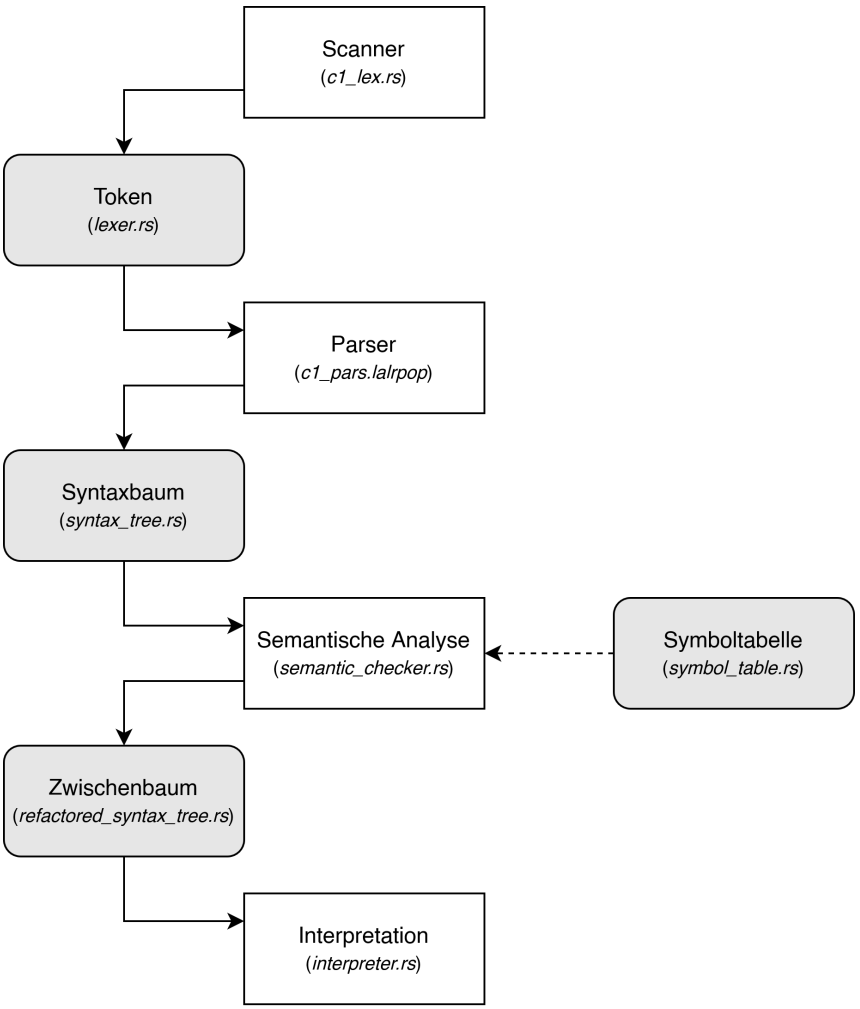

Abbildung 5: Grafische Darstellung des Interpreters. Es ist zu erkennen, dass die vier Komponenten des Interpreters inhaltlich getrennt sind.

Aufgabe 1 - Einstieg

Unsere erste Übungsaufgabe bildet einen Einstieg in die Programmiersprache Rust und bietet bereits eine gewisse Vorbereitung auf die weiteren Übungsaufgaben, die direkt mit dem Interpreter im Zusammenhang stehen. Wie wir jedoch in Kapitel 2.2 festgestellt haben, besitzen die Implementierung eines Stacks oder eines Syntaxbaums nur einen geringen Lerneffekt, da beide Strukturen leicht durch den Standardumfang von Rust erzeugt werden können. Unsere Herausforderung bei der Konzeption einer geeigneten Übungsaufgabe lag somit darin, einen neuen Aufgabentyp zu entwickeln, der sowohl für Einsteiger in Rust geeignet ist als auch bereits eine Vorbereitung auf die späteren Übungsinhalte liefert.
Wir haben uns im Rahmen dieser Anforderungen für einen Rechner entschieden, der auf einem Syntaxbaum basiert. Dabei liegt der Fokus jedoch nicht auf der Struktur des Syntaxbaums, sondern auf dem Umgang mit eben dieser Struktur. Da wir wäh-

rend der Entwicklung des Interpreters mehrfach das Visitor Pattern genutzt haben, wollten wir dies als Grundlage für die erste Übungsaufgabe verwenden.
Im Rahmen des Übungsblattes wird den Studierenden ein einfacher Syntaxbaum, bestehend aus Enumerationen und Strukturen, zur Verfügung gestellt. Dieser ermöglicht die Abbildung von Rechenaufgaben mit den vier Grundrechenarten ('+', '-', '*', '/') sowie die Verwendung von Knotentypen, um Werte in Variablen abzulegen und wieder zu laden. Mit diesen Möglichkeiten können mehrere Rechenaufgaben und Operationen in einer Baumstruktur abgebildet werden. Entsprechende Syntaxbäume können mithilfe einer mitgelieferten Funktion aus einem String in umgekehrter polnischer Notation (Postfix-Notation) generiert werden. Den Studierenden werden zwei Übungsaufgaben gestellt. In der ersten Übungsaufgabe soll ein Visitor erstellt werden, der die Struktur des eingegebenen Syntaxbaums in Infix-Notation und unter Verwendung von Klammern ausgibt. In der zweiten Teilaufgabe soll ebenfalls ein Visitor implementiert werden, der die Rechenaufgaben im eingegebenen Syntaxbaum berechnet und das Ergebnis ausgibt.
Diese Gestaltung der Übungsaufgabe bietet mehrere Vorteile. Die Studierenden benötigen für die Lösung der beiden Teilaufgaben nur grundlegendes Vorwissen in Rust, da die etwas komplexeren Strukturen bereits durch die Aufgabenstellung vorgegeben sind. Ebenfalls von Vorteil ist, dass die Studierenden sich in diesem Kontext bereits mit dem Visitor Pattern auseinandersetzen, welches im späteren Verlauf für die semantische Analyse sowie bei der Interpretation der jeweiligen Baumstrukturen Anwendung findet. Auch auf inhaltlicher Ebene bietet diese Aufgabe einen Einstieg in die späteren Aufgaben. Die Teilaufgaben bilden dabei bereits Teilbereiche des Compilerbaus ab. Die Implementierung des Rechners ist somit bereits eine kleine Variante eines Interpreters. Wir sind der Meinung, dass wir mit diesem Übungsblatt nicht nur einen Einstieg für die späteren Aufgaben schaffen können, sondern aufgrund unseres gewählten Aufbaus sowie des Umfangs der Aufgaben auch das Interesse an den weiteren Teilbereichen des Compilerbaus sowie an der Programmiersprache Rust wecken können.
Um den Studierenden die Überprüfung ihrer Lösungsansätze zu ermöglichen, haben wir der Aufgabe sieben verschiedene Unittests beigefügt. Diese Testfälle decken dabei sowohl alle Rechenoperatoren als auch den Umgang mit Variablen ab. Ebenfalls werden in den Testfällen Randfälle der Rechnung wie eine Nulldivision abgedeckt.
Als Nächstes präsentieren wir die zweite Übungsaufgabe, die ebenfalls als inhaltlicher Einstieg in den Übungsinterpreter dient.

Aufgabe 2 - Scanner

In diesem Abschnitt wollen wir uns der zweiten Übungsaufgabe widmen. Nachdem wir mit der ersten Aufgabe einen Einstieg in den Compilerbau und die Programmiersprache Rust geschaffen haben, wollen wir uns mit dieser Übungsaufgabe nun dem ersten Teil unseres Übungsinterpreters widmen. In dieser Aufgabe soll der Teilbereich des Lexers behandelt werden. Die Studierenden sollen in dieser Aufgabe mit der Funktionsweise von Scannern vertraut gemacht werden. Zu diesem Zweck und als

Grundlage für den weiteren Aufbau des Interpreters stellen wir die Aufgabe, dass die Studierenden einen Scanner für die Sprache C1 erstellen sollen. Dazu liefern wir in der Aufgabenstellung das Grundgerüst eines Logos-Lexers mit und fügen ebenfalls eine Beschreibung der benötigten Tokens zur Aufgabenstellung hinzu. Die Beschreibung der Tokens haben wir in Listing 15 dargestellt. Bei der Besprechung der Aufgaben zum Thema des Scanners in den bisherigen Ansätzen in Abschnitt 2.2 haben wir festgestellt, dass die Komplexität und der Umfang dieser Aufgabe nicht sehr hoch ist. Aus diesem Grund haben wir in den bisherigen Ansätzen jeweils eine zweite Teilaufgabe dem Übungsblatt hinzugefügt. Für die Sprache C1 müssen insgesamt 37 Tokens implementiert werden. Dabei bestehen 30 der 37 Tokens aus festen Zeichenketten, wie den Schlüsselwörtern der Sprache. Nur 7 der benötigten Tokens sind etwas anspruchsvoller, da hier eine Auseinandersetzung mit regulären Ausdrücken erforderlich ist. Wir haben uns dazu entschlossen, die Komplexität der Übungsaufgabe so anzupassen, dass wir keine regulären Ausdrücke für die Tokens angeben, sondern die 7 komplexeren Tokens nur textuell beschreiben. Auf diese Weise erhalten die Studierenden die Möglichkeit, sich auch in dieser Aufgabe mit dem Erstellen von regulären Ausdrücken auseinanderzusetzen. Aufgrund der Verschmelzung der Lernziele der beiden bisherigen Teilaufgaben haben wir beschlossen, auf diesem Übungsblatt keine weitere Teilaufgabe zu formulieren.
Damit die Studierenden ihre Lösungsansätze testen können, liefern wir zu Testzwecken zwei Unittests mit. In einem dieser Tests werden alle Tokens jeweils einmal exemplarisch abgeprüft. Der zweite Test umfasst bereits ein kleines Testprogramm in C1. Wir möchten dieses Programm auch in den späteren Übungsaufgaben in die Testfälle aufnehmen. Dadurch möchten wir den Studierenden an einem konkreten Beispiel die Entwicklung des Interpreters von einer Tokenfolge zur vollständigen Interpretation zeigen. Um für die Korrektur der Übungsaufgaben auch Randfälle der regulären Ausdrücke prüfen zu können, haben wir unter der Verwendung des Crates PropTest[20] weitere Testfälle zur Korrektur hinzugefügt. In Proptest können Eingaben für Funktionen unter anderem durch reguläre Ausdrücke vorgegeben werden. Bei der Ausführung dieser Testfälle werden automatisch Testeingaben generiert, um den möglichen Wertebereich der Eingabe abzudecken.
Im nächsten Schritt wollen wir uns dem Parser und der Erstellung des Syntaxbaums zuwenden.

Aufgabe 3 - Parser

Im Folgenden wollen wir die dritte Übungsaufgabe besprechen. Diese beschäftigt sich mit dem Thema des Parsers. Wie wir in Abschnitt 1.2.2 gelernt haben, umfasst der Bereich der syntaktischen Analyse verschiedene Varianten und Herangehensweisen bei der Implementierung von Parsertechniken. Für diese Übungsaufgabe haben wir uns entschieden, die Grammatik für einen LALR(1)-Parser zu vervollständigen. Im Abschnitt über die syntaktische Analyse 3.2 haben wir uns bei der Entwicklung des

[20] `https://crates.io/crates/proptest`

Interpreters für den Einsatz des Parsergenerators LALRPOP entschieden. Diesen möchten wir auch für diese Übungsaufgabe verwenden. Da LALRPOP keine mehrdeutigen Grammatiken akzeptiert, muss die ursprüngliche Grammatik der C1-Sprache umgestaltet werden. Wir haben beschlossen, dass diese Umgestaltung ebenfalls Teil der Übungsaufgabe sein soll. Auf diese Weise können wir neben der einfachen Umwandlung einer Grammatik auch eine weitere Komponente und somit einen weiteren Aspekt zu den Lernzielen der Aufgabe hinzufügen. Die Studierenden erhalten die mehrdeutige Grammatik der Sprache C1 im Rahmen der Übungsaufgabe und sollen diese zunächst umgestalten. Während des Umgestaltungsprozesses liefert LALRPOP detaillierte Fehlermeldungen, die konkrete Problemstellen in der Grammatik anhand von Beispielen aufzeigen. Als Hilfe haben wir bereits einige Umwandlungen in der bereitgestellten Grammatik angedeutet.
Ein weiterer Teil der Übungsaufgabe befasst sich mit der Erstellung des Syntaxbaums. Zu diesem Zweck stellen wir auch die Struktur der verfügbaren Knotentypen für den Syntaxbaum zur Verfügung. Wir haben aus mehreren Gründen beschlossen, von der Struktur der bisherigen Übungskonzepte abzuweichen und die Konstruktion in dieser Aufgabe zu übernehmen. Der wichtigste Punkt dabei ist, dass, wie wir in Abschnitt 1.2.2 gelernt haben, die Konstruktion des Syntaxbaums als Teil der syntaktischen Analyse betrachtet wird. Ein weiterer Grund ist, dass durch diese Konstruktion ein sichtbarer Fortschritt erzielt wird, der einerseits von den Studierenden anerkannt werden kann und andererseits eine Grundlage für die Bewertung der Parser-Aufgabe bildet.
Wie wir bei der Entwicklung des Übungsinterpreters gezeigt haben, ist die Erweiterung einer Ableitungsregel um die Rückgabe eines Syntaxbaumknotens in nur wenigen Schritten möglich. Daher möchten wir den Schwerpunkt der Übungsaufgabe auf die Bearbeitung der Grammatik legen. So können wir bis zu einem gewissen Grad unabhängig vom Vorlesungsfortschritt eine Aufgabe zu Parsern stellen, ohne dass das Thema der syntaktischen Analyse vollständig abgedeckt sein muss. Nachdem der generierte Parser einen Syntaxbaum erstellt hat, können wir die Struktur dieses Baums durch Bereitstellen von insgesamt 16 Testfällen überprüfen. Diese Tests umfassen nicht nur das Testprogramm, das bereits im Lexer enthalten war, sondern auch eine Reihe von 7 Tests, die sich speziell auf das "dangling else"beziehen.
Mit diesem erstellten Syntaxbaum können wir uns in der nächsten Übungsaufgabe der semantischen Analyse zuwenden.

Aufgabe 4 - Semantische Analyse

Die vierte Aufgabe unseres Übungskonzepts widmet sich der semantischen Analyse. Hierfür stellen wir neben den Strukturen wie dem Syntaxbaum und der Symboltabelle auch eine Datei bereit, in der bereits die Visit-Methoden für den Syntaxbaum vorbereitet sind. Ebenfalls geben wir in der Aufgabenstellung die semantischen Regeln der Sprache C1 an. Das Ziel dieser Übungsaufgabe für die Studierenden besteht darin, diese Regeln mithilfe der Implementation des Visitors zu überprüfen. Mit dieser Aufgabe möchten wir sicherstellen, dass die Studierenden ein Verständnis für die

Funktionen der semantischen Analyse entwickeln. Bei der Konzeption dieser Aufgabe haben wir darüber nachgedacht, ob wir die Unterstützung der Symboltabelle in der Aufgabenstellung entfernen sollen und die Studierenden stattdessen selbstständig die Implementierung der Symboltabelle durchführen lassen sollten. Dies würde den Vorteil bieten, dass die Studierenden sich auch mit dem Aufbau und der Funktionsweise einer Symboltabelle auseinandersetzen könnten. Wir haben uns jedoch vorerst dagegen entschieden. Wir gehen davon aus, dass die Studierenden bereits durch die Verwendung der Symboltabelle in der Aufgabe ein ausreichend umfassendes Verständnis entwickeln werden. Da wir den Aufwand für die Studierenden noch nicht endgültig abschätzen können, möchten wir die eigenständige Implementierung der Symboltabelle als Möglichkeit zur zukünftigen Optimierung der Übungsaufgabe offenlassen. Ein wichtiger Punkt, der sich aus der Implementierung der Symboltabelle durch die Studierenden ergeben würde, ist, dass Aspekte wie die Speicheradressierung von Variablen und Parametern festgelegt werden müssten, um die resultierenden Zwischenbäume (Syntaxbäume als Zwischenrepräsentation zwischen Frontend und Backend des Interpreters) weiterhin automatisiert testen zu können.
Mit dieser Übungsaufgabe haben wir die semantische Analyse behandelt. Damit ist die Vorbereitung abgeschlossen, um in der letzten Teilaufgabe den resultierenden Zwischenbaum zu interpretieren.

Aufgabe 5 - Interpretation

In der fünften und letzten Teilaufgabe sollen die Studierenden den Zwischenbaum interpretieren, den sie in den vorherigen Aufgaben erarbeitet haben. Wie in unserem Übungsinterpreter sollen die Studierenden hierbei auf eine stackbasierte Speicherverwaltung ähnlich der in C zurückgreifen. Neben dem generellen Interpretieren des Quellcodes soll das Lernziel hier auch den Umgang mit dieser Art der Speicherverwaltung umfassen. Hierzu stellen wir eine vorbereitete Rust-Datei bereit, in der die Grundstruktur des Interpreters durch die Methoden des Visitors auf dem Zwischenbaum vorbereitet ist. Da den Studierenden die Funktionalität der einzelnen Knoten des Zwischenbaums bekannt ist, ist in der Aufgabenstellung wenig Bedarf an tiefgreifenden Erläuterungen. Da in dieser Aufgabe keine neue Struktur konstruiert wird, die von einer weiteren Komponente verwendet wird, wollen wir den Studierenden bei der Bearbeitung dieser Aufgabe mehr Freiheiten einräumen als in den vorherigen Aufgaben. Daher ist es in dieser Übungsaufgabe zulässig, beispielhaft die Definition weiterer Strukturen vorzunehmen, wenn dies von den Studierenden benötigt wird. Wie auch in den vorherigen Übungsaufgaben können die Studierenden für die einzelnen Komponenten des Lexers, Parsers und der semantischen Analyse entweder die vorgegebenen Angaben oder ihre eigenen Implementierungen nutzen. Um dies über die Übungsaufgaben hinweg zu unterstützen, liefern wir, wie auch bei dieser Übungsaufgabe, jeweils Testfälle für alle bisherigen Komponenten des Interpreters mit. Somit sind insgesamt 37 (+1 für die Korrektur) Testfälle für den vollständigen Interpreter verfügbar, die sich wie folgt, auf die einzelnen Komponenten des Interpreters verteilen:

- Lexer – 2 (+1 nur für die Korrektur)
- Parser – 17
- Semantische Analyse – 9
- Interpreter – 9

Mit dieser Übungsaufgabe endet das von uns entwickelte Übungskonzept, in dem wir neben einer Einstiegsaufgabe vier weitere Übungsaufgaben zu den einzelnen Komponenten eines Interpreters präsentiert haben.

In diesem Kapitel haben wir zunächst Entwicklungskriterien definiert, die als Grundlage für die Entwicklung unseres Interpreters und des daraus folgenden Übungskonzepts dienten. Anschließend haben wir über die Entwicklung des Interpreters berichtet, wobei wir auf unsere Designentscheidungen und aufgetretene Probleme eingegangen sind. Abschließend haben wir in diesem Kapitel unsere Entscheidungen bei der Gestaltung des neuen Übungskonzepts erläutert. Im nächsten Kapitel der Arbeit werden wir eine Auswertung und einen Vergleich der jeweiligen Übungskonzepte durchführen.

4. Auswertung

In diesem Kapitel möchten wir eine Auswertung unseres neu entwickelten Übungskonzepts durchführen. Dabei werden wir anhand unserer Entwurfskriterien prüfen, ob wir den in diesen formulierten Anspruch an unsere Umsetzung erreichen konnten. Ebenfalls werden wir im direkten Vergleich mit den bisherigen Übungskonzepten herausarbeiten, durch welche Funktionen und umgesetzte Ansätze unser neuer Ansatz hervorsticht. Zu diesem Zweck wollen wir die Auswertung anhand unserer aufgestellten Entwurfskriterien strukturieren.

4.1. Auswertung der Interaktionskriterien

Zunächst betrachten wir die Untergruppe der Interaktionskriterien. Im Rahmen dieser Kriterien haben wir Maßstäbe für die Interaktion mit der Aufgabenstellung sowie den mitgelieferten Inhalten einer Aufgabe festgelegt. Unser wichtigster Fokus lag dabei auf der Gestaltung der Aufgabenstellung. Im Folgenden beschreiben wir beispielhaft anhand des ersten Übungsblatts die Gestaltung der Aufgabenstellungen. Jedes unserer Übungsblätter folgt einem einheitlichen Erscheinungsbild mit demselben grundsätzlichen Aufbau. Das Aufgabenblatt beginnt mit einem Abschnitt über allgemeine Hinweise, in dem die Formalitäten zur Einreichung festgelegt werden. Wir geben unter anderem vor, dass die fehlerfreie Kompilierung der Abgabe eine Mindestvoraussetzung für die Bewertung darstellt. Im zweiten Abschnitt werden weitere Informationen zur konkreten Bearbeitung vermittelt, einschließlich Ausführung und Testen des Programms. Alle zu bearbeitenden Dateien werden in einer eigenen Sektion übersichtlich aufgelistet. Unter der Überschrift der jeweiligen Aufgabennummer werden die inhaltlichen Vorgaben präsentiert. Dieser Abschnitt beginnt mit einer Definition des Lernziels für das jeweilige Übungsblatt. Anschließend folgt eine Kurzbeschreibung der gestellten Aufgabe, die zusammen mit dem definierten Lernziel den Umfang der Übungsaufgabe verdeutlicht. Die eigentliche Aufgabenstellung wird im Anschluss ausführlich präsentiert. Falls erforderlich, werden weitere Angaben wie die Liste der Tokens für Übungsblatt 2 oder die Grammatikregeln für Blatt 3 angegeben. Im Gegensatz zu den bisherigen Ansätzen, bei denen solche Angaben teilweise nur per Verlinkung eingebunden wurden, haben wir diese Informationen direkt auf dem Aufgabenblatt platziert, um von externen Informationsquellen unabhängig zu sein. Durch diese Strukturierung der Aufgabenblätter konnten wir die in den Interaktionskriterien vorgegebene Form einhalten. Auch die Aufgabenstellungen der bisherigen Ansätze weisen eine ähnliche Struktur auf. Allerdings definieren weder unser neuer Ansatz noch die bisherigen Ansätze explizit ein Lernziel. Abgesehen davon erfüllen aber auch diese Aufgabenblätter die von uns vorgegebenen Kriterien.
Ein weiteres von uns festgelegtes Kriterium betrifft die Bereitstellung der Übungsinhalte. Wie bereits erwähnt, ermöglicht uns die Verwendung von Rust die Nutzung von Cargo zur Aufbereitung der Inhalte. In Cargo-Projekten werden Abhängigkeiten mit bestimmten Versionsnummern angegeben. Beim Bauen oder Ausführen des Projekts werden diese Abhängigkeiten automatisch in den gewünschten Versionen herunterge-

laden und eingebunden. Dies erleichtert die Bereitstellung von Übungsaufgaben, da im Vergleich zur Verwendung von C die manuelle Verwaltung von Abhängigkeiten entfällt. Da zudem der Build-Prozess des Projekts von Cargo übernommen wird, sind zusätzliche, bereitgestellte Makefiles nicht mehr erforderlich. Wie wir sehen können, bietet die Nutzung von Rust im Rahmen der Projektverwaltung bereits einige Vorteile gegenüber der Verwendung von C als Programmiersprache. Im nächsten Teil werden wir uns der Prüfung der Inhaltskriterien widmen.

4.2. Auswertung der Inhaltskriterien

Die von uns in Abschnitt 3.1 definierten Inhaltskriterien beschäftigen sich mit der Wahl der Programmiersprache für die Umsetzung des Zielprogramms sowie dem Umfang der vermittelten Inhalte. Dabei haben wir bereits herausgearbeitet, welche Vorteile Rust gegenüber C bietet. Wie wir im Verlauf der Implementation unseres Übungsinterpreters festgestellt haben, bietet Rust unter anderem den Vorteil einer umfangreichen Standardbibliothek. Die Studierenden können sich somit während der Programmieraufgaben stärker auf die vermittelten Inhalte konzentrieren und müssen sich nicht so stark mit der Implementierung grundlegender Strukturen wie dem Stack und dem Syntaxbaum auseinandersetzen. Durch die Wahl der Sprache Rust konnten wir den Fokus der Übungsaufgaben erfolgreich auf die zu vermittelnden Inhalte im Bereich des Compilerbaus lenken.
Wie wir in den Inhaltskriterien festgelegt haben, konnten wir fünf Übungsaufgaben entwickeln, von denen vier die einzelnen Teilbereiche eines Interpreters abdecken: die lexikografische Analyse, die syntaktische Analyse, die semantische Analyse und die Interpretation. Diese Struktur ermöglicht es, nach einer Einführung in das Themenfeld und die verwendete Sprache einen vollständigen Übungsinterpreter zu implementieren. Im Gegensatz zum bisherigen Ansatz in Rust können wir in unserem Übungskonzept auch die Interpretation des eingegebenen Programmcodes ermöglichen. Dadurch können wir eine größere Bandbreite an Themen im Bereich des Compilerbaus abdecken. Durch die geschickte Nutzung von externen Bibliotheken und Entwicklungsverfahren wie der Verwendung von Design Patterns konnten wir bei gleichbleibendem Arbeitsaufwand die gesamte Struktur des Interpreters abdecken. Unser neu entwickelter Ansatz bietet somit auf inhaltlicher Ebene einen klaren Vorteil gegenüber der bisherigen Rust-Lösung, während wir durch die Vorteile von Rust auch strukturelle Vorteile im Vergleich zum C-Ansatz erzielen konnten. Als nächsten Schritt wollen wir die definierten Entwicklungskriterien besprechen.

4.3. Auswertung der Entwicklungskriterien

Im Rahmen der Entwicklungskriterien haben wir Kriterien definiert, anhand derer wir die grundlegende Struktur bei der Entwicklung unseres Interpreters lenken konnten. Da dieser Übungsinterpreter die Grundlage für die später abgeleiteten Aufgaben bildete, konnten wir so auch indirekt Einfluss auf die Übungsaufgaben nehmen. Ein wichtiger Bestandteil der Entwicklungskriterien war die strikte Auswahl von

Kriterien für die Auswahl von externen Bibliotheken. Wie auch in den bisherigen Übungsansätzen in C und Rust haben wir uns dafür entschieden, für den Scanner und auch den Parsergenerator auf externe Bibliotheken zurückzugreifen. Dabei konnten wir durch unsere Kriterien sowohl inhaltlich geeignete als auch einfach zu nutzende Crates auswählen. Für den Scanner haben wir uns, wie im Rust-Ansatz, für Logos entschieden. Diese Bibliothek zeichnet sich durch eine gute Lesbarkeit der hinterlegten Token und eine umfassende Dokumentation aus. Diese Wahl führte im Vergleich zum C-Ansatz, der die Bibliothek Flex verwendet, zu einer besseren Übersicht im Programm und einer strukturellen Vereinfachung beim Eintragen der Tokens.
Die zweite von uns extern genutzte Bibliothek ist LALRPOP und wird in unserem Ansatz für die Generierung des Parsers verwendet. Die von LALRPOP erwartete Eingabe orientiert sich syntaktisch stark an der eigenen Syntax von Rust. Dies trägt zur Lesbarkeit des Codes bei. Ebenfalls bietet LALRPOP durch die einfache Auswertung einzelner Regelteile die Möglichkeit, einen Syntaxbaum aus der Tokenfolge zu konstruieren. Wie wir im Laufe der Entwicklung des Interpreters festgestellt haben, akzeptiert LALRPOP keine mehrdeutigen Ableitungen. Um dies anzupassen, haben wir die Grammatik wie in Abschnitt 3.2 bearbeitet. Bison bot für diese Problemstellung die Möglichkeit, Präferenzen zu definieren. Da uns bei der Nutzung von LALRPOP für unsere Übungsaufgaben diese Funktion nicht zur Verfügung steht, eröffnet sich stattdessen jedoch für uns die Möglichkeit eine Aufgabe zu formulieren, in welcher sich die Studierenden mit diesem Umformungsprozess auseinandersetzen. Damit können die Studierenden ihr Wissen im Umgang mit Grammatiken vertiefen. Auch in diesem Fall konnten wir durch die Auswahl einer Bibliothek, unter Zuhilfenahme unserer Entwicklungskriterien, eine geeignete Wahl treffen, die insbesondere die Struktur des Interpreters und der zu bearbeitenden Übungsaufgaben verbessert. Im folgenden Abschnitt wollen wir die letzte Kategorie der Entwurfskriterien betrachten.

4.4. Auswertung der Designkriterien

Die Designkriterien ermöglichten es uns, Kriterien festzulegen, mit denen wir die Struktur des Interpreters beeinflussen konnten. Einer der wichtigsten Punkte dabei war die Vorgabe, eine möglichst hohe Modularität des Interpreters zu ermöglichen. Modularität ist ein wichtiges Ziel und ein grundlegendes Prinzip der Softwareentwicklung. Durch die Modularisierung von Softwareprojekten kann die Lesbarkeit und das Verständnis erhöht werden. Weitere Vorteile sind eine erleichterte Testbarkeit der Software und auch die Tatsache, dass Komponenten in modularen Softwareprojekten leicht ersetzt oder wiederverwendet werden können. Auch für unser Projekt war die Modularität ein wichtiger Faktor. Dadurch konnten nicht nur wir einfacher Aufgaben aus dem Übungsinterpreter ableiten, sondern auch für die Studierenden eine bessere Übersicht in dem von ihnen schrittweise aufgebauten Projekt schaffen. Zu diesem Zweck haben wir jede der vier Komponenten des Interpreters jeweils eigenständig voneinander entwickelt. Die vier Module sind nur über die Ein- und Ausgabestrukturen miteinander verbunden. Im bisherigen C- und auch im Rust-Ansatz werden die syntaktische Analyse und die semantische Analyse in derselben Struktur implementiert,

welche über mehrere Aufgaben erweitert wird. Ein Austausch des Parsergenerators ist hier beispielsweise nicht möglich. Wir konnten durch die Trennung der einzelnen Komponenten auch eine nachvollziehbare Struktur eines Interpreters herausarbeiten. Ein weiteres Ziel, das wir in den Designkriterien festgelegt haben, ist die Testbarkeit des Projekts. Dies ist nicht nur ein wichtiger Faktor, um die Qualität einer Software zu gewährleisten, sondern auch relevant für die Bewertung der Übungsaufgaben. Im Rahmen der Modularisierung des Projekts konnten wir für jede der Komponenten des Interpreters eine eigene Testumgebung erstellen und insgesamt über 37 Testfälle bereitstellen. Diese dienen bei den Übungsaufgaben sowohl den Studierenden als Leitlinie für ihre Implementierung als auch können sie zur Korrektur der eingereichten Lösungen genutzt werden.

4.5. Ansätze im direkten Vergleich

Im folgenden Abschnitt wollen wir anhand des Parsers exemplarisch die Vorteile unseres Übungskonzepts im Vergleich zu den bisherigen Lösungen auf der Grundlage der Entwurfskriterien präsentieren. Wie in Abschnitt 3.2 diskutiert, nutzen wir als Parsergenerator die externe Bibliothek LALRPOP.

Durch die Verwendung einer rust-ähnlichen Syntax konnten wir ein gutes Verständnis für den strukturellen Aufbau der Regeln erreichen. Aus diesem Grund haben wir in Listing 13 die Regel zur Ableitung eines Funktionskopfes und die Erzeugung eines entsprechenden Knotens im Syntaxbaum dargestellt.

```
67 functiondefinition: FunctionDefinition = {
68     <typ:type_rule> <id:ID> "(" <params:parameterlist> ")"
69     "{" <stats:statementlist> "}" => FunctionDefinition{
70         func_type: typ,
71         func_name: id,
72         func_param: params,
73         func_body: FunctionBody{
74             body: stats
75         },
76     },
77 };
```

Listing 13.: Beispielhafte Darstellung der Regel `functiondefinition` in LALRPOP, welche den Kopf einer Funktion ableitet und den passenden Knoten des Syntaxbaums zurückliefert. Entnommen aus unserem Interpreter, *c1_pars.LALRPOP*

```
functiondefinition:
type ID[name] {
    let name =$name.unwrap_name();

    [...]

    $$ = Value::Tree(function_node(name));
}
'(' opt_parameterlist[params] ')' '{' statementlist[body] '}' {
    let mut function_node = $3.unwrap_tree();
    if let Tree(params) = $params {
        function_node.push_node(params);
    }
    let body = $body.unwrap_tree();
    function_node.push_node(body);
    $$ = Tree(function_node);

    [...]
}
```

Listing 14.: Beispielhafte Darstellung der Regel `functiondefinition` in Bison, welche den Kopf einer Funktion ableitet und den passenden Knoten des Syntaxbaums zurückliefert. Da im bisherigen Rust-Ansatz der Syntaxbaum erst als Teil der semantischen Analyse konstruiert wird, haben wir eine um die Inhalte der semantischen Analyse gekürzte Fassung der Regel von diesem Übungsblatt dargestellt. Entnommen aus den Musterlösungen des bisherigen Rust-Ansatzes, Übungsblatt 5, *minako_syntax.y*

Im Vergleich dazu haben wir in Listing 14 die ähnliche Regel aus der Bison-Eingabedatei dargestellt.
In beiden Listings wird ein Funktionskopf mit `type`, `name`, `params` und einem `body` abgeleitet, und ein Knoten des Syntaxbaums wird zurückgegeben. Wie leicht zu erkennen ist, ist der Auszug aus unserer neuen Lösung kompakter und dennoch übersichtlicher. Dazu trägt auch die verwendete Form des Syntaxbaums bei. Da wir in unserem neuen Syntaxbaum die Knotentypen aufeinander abgestimmt haben, können wir so auch die Struktur der Grammatik zuverlässig abbilden. Dies ist mit dem Bison-Ansatz nicht möglich, da der Knotentyp der Rückgabe einer Regel nicht überwacht wird. Somit trägt unsere Implementation durch die Vorgaben in den Regeln und Knotentypen zur sichereren Strukturierung des Syntaxbaums bei.

Wie wir nach der Betrachtung der bisherigen und unserer neuen Übungskonzepte anhand der Entwurfskriterien gesehen haben, erfüllt das von uns im Rahmen dieser Arbeit neu entwickelte Übungskonzept diese Entwurfskriterien. Ebenfalls konnten wir

feststellen, dass wir durch die vorher festgelegten Kriterien mehrere Verbesserungen gegenüber den alten Übungskonzepten erreichen konnten. So bieten wir mehrere inhaltlich aufeinander abgestimmte Übungsaufgaben, in denen ein vollständiger Übungsinterpreter entwickelt wird. Durch hohe Modularität, die Ausnutzung von Vorteilen der Sprache Rust sowie eine umfangreiche Testsuite konnten wir die Übersichtlichkeit und Nachvollziehbarkeit der gestellten Übungsaufgaben im Vergleich zu den bisherigen Lösungen erhöhen. Ebenso konnten wir durch die neue Strukturierung und Gestaltung der Übungsaufgaben ein inhaltlich aufeinander abgestimmtes und durchdachtes Übungskonzept erstellen, das Wissen aus verschiedenen Teilbereichen des Compilerbaus vermittelt.
Im nächsten und letzten Abschnitt der Arbeit wollen wir die Ergebnisse abschließend zusammenfassen und einen Ausblick auf mögliche Erweiterungen des Konzepts geben.

5. Fazit

Abschließend möchten wir zunächst zusammenfassen, was wir im Rahmen dieser Arbeit erarbeitet haben. Außerdem wollen wir einen Ausblick über weitere Möglichkeiten geben, den in dieser Arbeit erstellten Ansatz zu erweitern.

5.1. Zusammenfassung

Als Ziel der Arbeit haben wir uns gesetzt, einen Compiler in der Sprache Rust zu entwickeln, der zu Übungszwecken in der Lehre eingesetzt werden kann. In Kapitel 1 haben wir daher zunächst erläutert, aus welchen Gründen wir diese Zielsetzung aufgegriffen haben. Des Weiteren haben wir an dieser Stelle bereits einen Einblick in die theoretischen Hintergründe sowie den Aufbau eines Compilers gegeben. Im zweiten Abschnitt haben wir die aktuellen Umsetzungen beschrieben. Dabei haben wir uns ebenfalls mit dem Wissensstand der Zielgruppe beschäftigt, die an diesem Übungskonzept teilnimmt. Danach sind wir auf die beiden Ansätze eingegangen, die zum Zeitpunkt der Arbeit im Übungsbetrieb eingesetzt wurden. Wir haben den Aufbau der einzelnen Übungsaufgaben besprochen und im Laufe des Abschnitts Probleme herausgearbeitet. Abschließend konnten wir auf dieser Grundlage Kriterien erarbeiten, die wir zur Vermeidung dieser Problemstellungen in unserem neu entwickelten Ansatz berücksichtigen wollen.
Im dritten Abschnitt der Arbeit haben wir den von uns entwickelten neuen Ansatz präsentiert. Wir haben dafür zunächst Entwurfskriterien aufgestellt, durch die wir einen qualitativ hochwertigen Ansatz entwickeln wollen. Ebenso haben wir uns an dieser Stelle, unter Berücksichtigung der bisherigen Übungskonzepte, dazu entschieden, den Fokus unserer Arbeit von einem vollständigen Compiler auf einen Interpreter zu lenken. Im darauf folgenden Abschnitt haben wir schließlich die Entwicklung des Übungsinterpreters dokumentiert. Dabei haben wir unsere Designentscheidungen jeweils anhand der zuvor bestimmten Entwurfskriterien begründet. Nach der Vorstellung des Interpreters haben wir beschrieben, wie wir aus diesem Übungsinterpreter ein neues Übungskonzept mit insgesamt fünf Übungsaufgaben abgeleitet haben, in denen nach einer Einführung in die verwendete Sprache schrittweise ein vollständiger Übungsinterpreter konstruiert wird. Auch hier haben die zuvor definierten Entwurfskriterien eine Rolle gespielt. Im vierten Abschnitt der Arbeit haben wir schließlich unser neu entwickeltes Übungskonzept anhand der Entwurfskriterien ausgewertet. Dabei haben wir auch Vergleiche zu den bisherigen Übungskonzepten gezogen. So konnten wir zeigen, dass unser neu entwickeltes Konzept mehrere Vorteile gegenüber den beiden bisher verwendeten Übungskonzepten aufweist.

5.2. Ausblick

Wie wir im Verlauf der Arbeit feststellen mussten, würde eine vollständige Implementierung des Compilers zum jetzigen Zeitpunkt den vorgesehenen Arbeitsaufwand

der Übungsaufgaben überschreiten. Dennoch sind wir der Meinung, dass die Erweiterung des Interpreters um die Codegenerierung eine sinnvolle Verbesserung in der Zukunft darstellen könnte. Zu diesem Zweck würde sich die Bibliothek phoron_asm"[21] anbieten. Diese Crate führt die assemblerähnliche Sprache phoronëin, aus der der Jasmin-Assembler[22] Klassen für die Java-Virtual-Maschine generieren kann. Mithilfe dieser Bibliothek könnte der in dieser Arbeit erstellte Interpreter zu einem Compiler erweitert werden. Es böte sich auch an, eine alternative fünfte oder sogar sechste Übungsaufgabe zu entwerfen.
Da wir unser gesamtes Übungskonzept im Rahmen dieser Arbeit nicht durch eine praktische Anwendung im Übungsbetrieb auf den Prüfstand stellen konnten, sind die neuen Übungsaufgaben möglicherweise noch verbesserungswürdig. Die von uns entwickelten Aufgaben bieten jedoch ausreichend Spielraum, um sie je nach Bedarf anpassen zu können und so ein gut geeignetes praktisches Übungskonzept für die Vorlesung Compilerbauanzubieten.

[21] `https://crates.io/crates/phoron_asm`
[22] `https://jasmin.sourceforge.net/`

Literatur

[Bac80] John Backus. Programming in america in the 1950s—some personal impressions. In *A History of Computing in the twentieth century*, pages 125–135. Elsevier, 1980.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[CT11] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[Hof22] Dirk W Hoffmann. *Theoretische Informatik*. Carl Hanser Verlag GmbH Co KG, 2022.

[Jun20] Ralf Jung. Understanding and evolving the rust programming language. 2020.

[JZ08] Maggie Johnson and Julie Zelenski. Lexical analysis. *CS143. Stanford University. Stanford, California*, 25, 2008.

[LSUA06] Monica Lam, Ravi Sethi, Jeffrey D Ullman, and Alfred Aho. Compilers: principles, techniques, and tools. *Pearson Education*, 2006.

[Muc97] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[NLNL94] Sven Naumann, Hagen Langer, Sven Naumann, and Hagen Langer. *Was ist Parsing?* Springer, 1994.

[Rusa] Rust book: Section “references and borrowing”. `https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html`. Accessed: 2023-08-08.

[Rusb] The rust community’s crate registry. `https://crates.io/`. Accessed: 2023-08-08.

[Rusc] Rust crate: Logo. `https://crates.io/crates/logos`. Accessed: 2023-08-08.

[Rusd] Rust documentation: “primitive type reference”. `https://doc.rust-lang.org/std/primitive.reference.html#`. Accessed: 2023-08-08.

[Ruse] Rust documentation: “the manifest format”. https://doc.rust-lang.org/cargo/reference/manifest.html#the-manifest-format. Accessed: 2023-08-08.

[Rusf] Rust documentation: “unsafe rust”. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html. Accessed: 2023-08-15.

[Rusg] The rust teams. “rust programming language”. https://www.rust-lang.org/. Accessed: 2023-04-13.

[SODa] Stack overflow: Stack overflow developer survey 2016. https://survey.stackoverflow.co/2016. Accessed: 2023-09-11.

[SODb] Stack overflow: Stack overflow developer survey 2022. https://survey.stackoverflow.co/2022. Accessed: 2023-09-11.

[SPOa] Fachspezifische studien- und prüfungsordnung (2015), kombibachelor informatik, hu berlin. https://gremien.hu-berlin.de/de/amb/2015/31/31_2015_AMB_Informatik_KombiBA_DRUCK.pdf. Accessed: 2023-08-09.

[SPOb] Fachspezifische studien- und prüfungsordnung (2015), monobachelor informatik, hu berlin. https://gremien.hu-berlin.de/de/amb/2022/9/09_2022_spo_monoba_informatik_druck.pdf. Accessed: 2023-08-09.

[SPOc] Fachspezifische studien- und prüfungsordnung (2022), monobachelor informatik, hu berlin. https://gremien.hu-berlin.de/de/amb/2015/13/13_2015_AMB_Monobachelor_Informatik_DRUCK.pdf. Accessed: 2023-08-09.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[Wir13] Niklaus Wirth. *Compilerbau: Eine Einführung*, volume 36. Springer-Verlag, 2013.

[WSH13] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.

A. Anhang

A.1. C1 Tokens

```
-- Schlüsselwörter
KW_BOOLEAN          "bool"
KW_DO               "do"
KW_ELSE             "else"
KW_FLOAT            "float"
KW_FOR              "for"
KW_IF               "if"
KW_INT              "int"
KW_PRINTF           "printf"
KW_RETURN           "return"
KW_VOID             "void"
KW_WHILE            "while"

-- Operatoren
PLUS                "+"
MINUS               "-"
ASTERISK            "*"
SLASH               "/"
ASSIGN              "="
EQ                  "=="
NEQ                 "!="
LSS                 "<"
GRT                 ">"
LEQ                 "<="
GEQ                 ">="
AND                 "&&"
OR                  "||"

-- Sonstige Token
COMMA               ","
SEMICOLON           ";"
LPAREN              "("
RPAREN              ")"
LBRACE              "{"
RBRACE              "}"

-- Termvariablen
CONST_INT           {INTEGER}
CONST_FLOAT         {FLOAT} ( [eE] ([-+])? {INTEGER} )?
                    | {INTEGER} [eE] ([-+])? {INTEGER}
```

```
CONST_BOOLEAN      "true" | "false"
CONST_STRING       "\"" [^\n\"]* "\""
ID                 ({LETTER})+ ({DIGIT} | {LETTER})*

-- "Pseudotoken" (nur zur Konstruktion anderer Token) --
DIGIT              [0-9]
INTEGER            {DIGIT}+
FLOAT              {INTEGER} "." {INTEGER} | "." {INTEGER}
LETTER             [a-zA-Z]

-- Kommentare --
C-Kommentare       "/*" <comment> "*/"
C++-Kommentare     "//" <comment> "\n"
```

Listing 15.: Liste der möglichen Token in der verwendeten Sprache C1.

A.2. C1 Grammatik

```
program             ::= ( declassignment ";" | functiondefinition )*

functiondefinition  ::= type id "(" ( parameterlist )? ")" "{"
                        statementlist "}"
parameterlist       ::= type id ( "," type id )*
functioncall        ::= id "(" ( assignment ( "," assignment )* )? ")"

statementlist       ::= ( block )*
block               ::= "{" statementlist "}"
                      | statement

statement           ::= ifstatement
                      | forstatement
                      | whilestatement
                      | simple_statement
simple_statement        ::= returnstatement ";"
                      | dowhilestatement ";"
                      | printf ";"
                      | declassignment ";"
                      | statassignment ";"
                      | functioncall ";"
                      | ";"

statblock           ::= "{" statementlist "}"
                      | statement

ifstatement         ::= <KW_IF> "(" assignment ")" statblock (
                        <KW_ELSE> statblock )?
forstatement        ::= <KW_FOR> "(" ( statassignment |
                        declassignment ) ";" expr ";"
                        statassignment ")" statblock
dowhilestatement    ::= <KW_DO> statblock <KW_WHILE> "(" assignment ")"
whilestatement      ::= <KW_WHILE> "(" assignment ")" statblock
returnstatement     ::= <KW_RETURN> ( assignment )?
printf              ::= <KW_PRINTF> "(" (assignment | CONST_STRING) ")"
declassignment      ::= type id ( "=" assignment )?
type                ::= <KW_BOOLEAN>
                      | <KW_FLOAT>
                      | <KW_INT>
                      | <KW_VOID>

statassignment      ::= id "=" assignment
```

```
assignment          ::= id "=" assignment
                      | expr
expr                ::= simpexpr ( "==" simpexpr | "!=" simpexpr |
                          "<=" simpexpr | ">=" simpexpr |
                          "<" simpexpr | ">" simpexpr )?
simpexpr            ::= ( "-" term | term ) ( "+" term |
                          "-" term | "||" term )*
term                ::= factor ( "*" factor | "/" factor |
                          "&&" factor )*
factor              ::= <CONST_INT>
                      | <CONST_FLOAT>
                      | <CONST_BOOLEAN>
                      | functioncall
                      | id
                      | "(" assignment ")"
id                  ::= <ID>
```

Listing 16.: Grammatik der verwendeten Sprache C1.

A.3. C(-1) Grammatik

```
program             ::= ( functiondefinition )* <EOF>

functiondefinition  ::= type <ID> "(" ")" "{" statementlist "}"
functioncall        ::= <ID> "(" ")"

statementlist       ::= ( block )*
block               ::= "{" statementlist "}"
                      | statement
statement           ::= ifstatement
                      | returnstatement ";"
                      | printf ";"
                      | statassignment ";"
                      | functioncall ";"

ifstatement         ::= <KW_IF> "(" assignment ")" block
returnstatement     ::= <KW_RETURN> ( assignment )?

printf              ::= <KW_PRINTF> "(" assignment ")"
type                ::= <KW_BOOLEAN>
                      | <KW_FLOAT>
                      | <KW_INT>
                      | <KW_VOID>

statassignment      ::= <ID> "=" assignment
assignment          ::= ( ( <ID> "=" assignment ) | expr )
expr                ::= simpexpr ( ( "==" | "!=" | "<=" | ">=" | "<" | ">" ) simpexpr
simpexpr            ::= ( "-" )? term ( ( "+" | "-" | "||" ) term )*
term                ::= factor ( ( "*" | "/" | "&&" ) factor )*
factor              ::= <CONST_INT>
                      | <CONST_FLOAT>
                      | <CONST_BOOLEAN>
                      | functioncall
                      | <ID>
                      | "(" assignment ")"
```

Listing 17.: Grammatik der für den Handparser verwendeten Sprache C(-1).

A.4. C1 Grammatik - Eindeutig

```
program             ::= ( declassignment ";" | functiondefinition )*

functiondefinition  ::= type id "(" ( parameterlist )? ")" "{"
                        statementlist "}"
parameterlist       ::= type id ( "," type id )*
functioncall        ::= id "(" ( assignment ( "," assignment )* )? ")"

statementlist       ::= ( block_if )*
block_if            ::= statement
                      | "{" statementlist "}"
block_no_if         ::= statement_no_if
                      | "{" statementlist "}"
simple_statement        ::= returnstatement ";"
                      | printf ";"
                      | declassignment ";"
                      | statassignment ";"
                      | functioncall ";"
                      | dowhilestatement ";"
                      | ";"

statement_no_if     ::= simple_statement
                      | forstatement_no_if
                      | whilestatement_no_if
                      | ifelsestatement

statement           ::= simple_statement
                      | forstatement
                      | whilestatement
                      | ifstatement

statblock_no_if     ::= "{" statementlist "}"
                      | statement_no_if

statblock           ::= "{" statementlist "}"
                      | statement

ifelsestatement     ::= <KW_IF> "(" assignment ")" statblock_no_if
                        <KW_ELSE> statblock

ifstatement         ::= <KW_IF> "(" assignment ")" statblock
                      | <KW_IF> "(" assignment ")" statblock_no_if
                        <KW_ELSE> statblock
```

```
forstatement          ::= <KW_FOR> "(" ( statassignment |
                          declassignment ) ";" expr ";"
                          statassignment ")" statblock
whilestatement        ::= <KW_WHILE> "(" assignment ")" statblock

forstatement_no_if    ::= <KW_FOR> "(" ( statassignment |
                          declassignment ) ";" expr ";"
                          statassignment ")" statblock_no_if
whilestatement_no_if::= <KW_WHILE> "(" assignment ")" statblock_no_if

dowhilestatement      ::= <KW_DO> statblock <KW_WHILE> "(" assignment ")"
returnstatement       ::= <KW_RETURN> ( assignment )?
printf                ::= <KW_PRINTF> "(" (assignment | CONST_STRING) ")"
declassignment        ::= type id ( "=" assignment )?
type                  ::= <KW_BOOLEAN>
                        | <KW_FLOAT>
                        | <KW_INT>
                        | <KW_VOID>

statassignment        ::= id "=" assignment
assignment            ::= id "=" assignment
                        | expr
expr                  ::= simpexpr ( "==" simpexpr | "!=" simpexpr |
                          "<=" simpexpr | ">=" simpexpr |
                          "<" simpexpr | ">" simpexpr )?
simpexpr              ::= ( "-" term | term ) ( "+" term |
                          "-" term | "||" term )*
term                  ::= factor ( "*" factor | "/" factor |
                          "&&" factor )*
factor                ::= <CONST_INT>
                        | <CONST_FLOAT>
                        | <CONST_BOOLEAN>
                        | functioncall
                        | id
                        | "(" assignment ")"
id                    ::= <ID>
```

Listing 18.: Grammatik der verwendeten Sprache C1, in welche die Mehrdeutigkeiten durch das Dangling-Else aufgelöst wurden.

A.5. Dateizugriff

Alle Inhalte (Compiler, Aufgabe, Musterlösungen & Testfälle), welche im Rahmen dieser Arbeit erstellt wurden, stehen sowohl unter `https://gitlab.informatik.hu-berlin.de/bucherda/ma-compiler/-/commit/5788e4c7f3239cf988a36985d4ac8fe5b944f28b` als auch auf dem Beigefügten Datenträger zur Einsicht zur Verfügung. Der Datenträger enthält ebenfalls ein digitale Fassung dieser Arbeit.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 7. November 2023 .....