Carl-Fuhlrott-Gymnasium
Wuppertal

Jahrgangsstufe Q1
Schuljahr 2021/2022

Der Aufbau einer Programmiersprache

Lexer, Parser, Compiler und Interpreter

Silas Groh

Betreut von Sonja Sokolović

7. März 2022

Inhaltsverzeichnis

1. Einleitung **4**

2. Grammatik **5**

2.1. Die BNF und EBNF Notationen 5

2.1.1. Repetition 6

2.1.2. Optionalität 6

2.2. Reguläre und kontextfreie Sprachen 7

2.3. Beispiel an *rost* 8

3. Der Lexer: lexikalische Analyse **10**

3.1. Vorgehensweise 10

3.2. Der Lexer von *rost* 11

4. Parser: syntaktische Analyse **14**

4.1. Vorgehensweise 14

4.2. Der Parser von *rost* 15

5. Compiler: Übersetzung in Maschinensprache **17**

5.1. LLVM 17

6. Interpreter: Ausführung des Programms **18**

6.1. Vorgehensweise 18

6.2. Die Interpreter von *rost* und *Roost* 19

7. Abschluss **22**

Abbildungsverzeichnis **23**

Tabellenverzeichnis **23**

Programmblockverzeichnis **23**

Literaturverzeichnis **24**

Anhang A. Lexer von *rost* **25**

Anhang B. Parser von *rost* 27

Anhang C. Interpreter von *rost* 29

Anhang D. Grammatik von *Roost* 31

Abschlusserklärung 33

1. Einleitung

Programmiersprachen sind, ob es einem bewusst ist oder nicht, ein wichtiger Bestandteil unseres Lebens geworden. Vieles läuft digital und wurde somit von jemandem programmiert. Für Außenstehende scheinen Programmiersprachen häufig sehr kompliziert, in Wahrheit erleichtern sie einem Programmierer jedoch massiv die Entwicklung eines Programms. Der Sinn von Programmiersprachen ist es, ein Programm möglichst verständlich für Menschen darzustellen. Diesen Programmtext muss dann aber auch der Computer verstehen, um das Programm auszuführen. Wie das funktioniert, habe ich im Rahmen dieser Facharbeit erforscht.

Die Ausführung von Programmtext ist in mehrere Teile aufgeteilt: vorerst den *Lexer*, welcher für die lexikalische Analyse zuständig ist und den *Parser* für die syntaktische Analyse. Danach wird je nach Programmiersprache unterschieden. Bei kompilierten[1] Sprachen folgt der *Compiler*. Dieser übersetzt den Programmtext in Maschinensprache[2], oder in Fällen wie *Java* oder *Python* in einen sogenannten *Bytecode* der jeweiligen Sprache, welcher wiederum interpretiert oder kompiliert wird. Die anderen Sprachen, darunter auch meine eigene Sprache *Roost*, welche im Rahmen dieser Facharbeit von mir erstellt wurde, werden direkt von einem *Interpreter* ausgeführt. Ein *Interpreter* ist einfacher zu entwickeln, hat allerdings zwei Nachteile. Zum einen muss der *Interpreter* auf jeder Zielmaschine installiert sein, um Programme dieser Sprache ausführen zu können, und zum anderen läuft ein interpretiertes Programm meist nicht so schnell wie ein vorab kompiliertes.

Die von mir erstellte Sprache *Roost* ist in *Rust* geschrieben. Die meisten Programmblöcke in dieser Facharbeit verwenden daher *Rust*(-ähnliche) Syntax, die beschriebene Vorgehensweise sollte aber auf die meisten prozeduralen Programmiersprachen übertragbar sein. Für weitere Informationen zu *Rust* kann die offizielle Internetseite unter `https://rust-lang.org/` besucht werden. Das Logo für *Roost* ist auf dem Titelblatt zu sehen. Weitere Informationen folgen in Kapitel 7.

Zu Beginn werde ich den Begriff „Grammatik“ in Bezug auf Programmiersprachen erklären, da dies zum weiteren Verständnis wichtig ist. Daraufhin folgen der Lexer, Parser, Compiler und Interpreter in dieser Reihenfolge, um dem typischen Aufbau einer Sprache zu folgen. Drei dieser Kapitel sind in allgemeine *Vorgehensweise* und eine beispielhafte *Implementation* aufgeteilt. Zum Schluss folgen Informationen zu den Projekten, die im Rahmen dieser Facharbeit von mir erstellt wurden.

[1] in Maschinensprache übersetzte

[2] Die „Einsen und Nullen“, die der jeweilige Prozessor versteht

2. Grammatik

Bei kommunikativen Sprachen, wie Deutsch und Englisch, bestimmt die Grammatik der jeweiligen Sprache die Struktur der Sätze. Bei Programmiersprachen ist das genauso. Der Aufbau der meisten Programmiersprachen ist anhand einer Grammatik definiert. Diese kann beispielsweise vorgeben, dass auf eine Zahl mit einem + dahinter immer eine weitere Zahl folgen muss.

2.1. Die BNF und EBNF Notationen

Zur Darstellung von diesen Grammatiken existieren verschiedene Notationen[1]. J. Backus und P. Naur veröffentlichten 1960 zur Darstellung der Grammatik der Sprache *Algol 60* die *Backus-Naur Form*, kurz *BNF* [Bac+60]. Diese wurde später von N. Wirth erweitert, um die Darstellung von Repetition[2] und Optionalität ohne Verwendung von Rekursion[3] zu ermöglichen. Diese ist als *EBNF* also *Erweiterte Backus-Naur Form* bekannt [Wir77].

Man nehme folgendes Beispiel:

Programmblock 2.1: Grammatik für einen simplen Satz

```
1 Subjekt     = 'Anna'  | 'Fritz' ;
2 Prädikat    = 'steht' | 'sitzt' ;
3 Satz        = Subjekt , Prädikat , '.' ;
```

Ein `Subjekt` ist hier als entweder *Anna* oder *Fritz* definiert. Der vertikale Strich | trennt hierbei mehrere Möglichkeiten, wovon genau *eine* zutreffen muss. Ein `Prädikat` ist in dieser Sprache entweder *steht* oder *sitzt*. Ein ganzer `Satz` muss nun dem Aufbau `Subjekt , Prädikat , '.'` folgen, also erst ein `Subjekt`, dann ein `Prädikat` und zum Ende ein Punkt. Somit lassen sich in dieser Sprache die folgenden vier Sätze bilden: „Anna steht.“, „Anna sitzt.“, „Fritz steht.“ und „Fritz sitzt.“. Die Teile, die in der Grammatik in Anführungszeichen stehen müssen genau so im Programm erscheinen und werden *Terminalsymbole* genannt.

In *EBNF* werden also verschiedene Regeln definiert. In dem obigen Beispiel sind das `Subjekt`, `Prädikat` und `Satz`. Jede Regel fängt mit einem Namen an, daraufhin folgt ein = und dahinter steht

[1] hier: Schreibweisen / Darstellungsformen
[2] Wiederholung
[3] eine von sich selbst abhängige Definition

die Definition der Regel. Eine Regel endet immer mit einem Semikolon und kann somit auch auf mehrere Zeilen aufgeteilt werden. Die Definition einer Regel besteht immer aus *Symbolen*, welche durch verschiedene Zeichen getrennt werden. Ein Symbol ist entweder ein *Terminalsymbol* oder der Name einer Regel. Letzteres wird dann durch den Inhalt jener Regel substituiert[4]. Das Beispiel aus Programmblock 2.1 ließe sich also auch wie in Programmblock 2.2 darstellen. Die Klammern gruppieren hierbei mehrere Symbole zusammen.

Programmblock 2.2: Grammatik für einen simplen Satz - Kurzfassung

```
1 Satz = ( 'Anna' | 'Fritz' ) , ( 'steht' | 'sitzt' ) , '.' ;
```

2.1.1. Repetition

Wie bereits erwähnt kann *EBNF* auch Repetition von Symbolen darstellen. Hierzu werden die zu wiederholenden Symbole in geschweiften Klammern gruppiert.

```
1 Programm = 'a' , { 'b' } ;
```

Diese Regel besagt, dass das Programm aus einem ‚a' mit beliebig vielen folgenden ‚b's bestehen muss. *Beliebig viele* beinhaltet auch *kein* mal. Mögliche Programme dieser Sprache sind zum Beispiel „a", „ab", sowie „abbbbbbbbb".

2.1.2. Optionalität

Um zu kennzeichnen, dass ein Symbol oder eine Gruppe von Symbolen, nicht zwingend notwendig, aber möglich ist, werden eckige Klammern verwendet.

```
1 Programm = 'a' , [ 'b' ] ;
```

In dieser Sprache sind nur zwei Programme möglich: „a" und „ab".

[4] ersetzt

2.2. Reguläre und kontextfreie Sprachen

Kontextfreie Sprachen sind Sprachen, bei denen jedes Vorkommen eines nicht-terminalen Symbols[5] in der Definition einer Regel, unabhängig vom Kontext, immer durch diese Regel substituiert werden kann. Dies schließt einige kommunikative Sprachen aus [Shi85]. Jede Sprache, die *EBNF* darstellen kann, ist kontextfrei. Bei Programmiersprachen ist es erwünscht, diese Kontextfreiheit einzuhalten [Wir08]. Das Beispiel von Programmblock 2.1 erfüllt diese Voraussetzung. Zudem gibt es die regulären Sprachen. Diese bilden eine Unterklasse der kontextfreien Sprachen und sind folgendermaßen definiert:

> Eine Sprache ist *regulär*, wenn sich ihre Syntax durch eine einzige EBNF-Regel ohne Rekursion ausdrücken lässt. [Wir08]

Anders könnte man formulieren:

> Eine Sprache ist *regulär*, wenn sich ihre Grammatik mit einer einzigen EBNF-Regel aus ausschließlich Terminalsymbolen darstellen lässt.

Dies trifft ebenfalls auf das Beispiel aus Abschnitt 2.1 zu, da die Kurzfassung in Programmblock 2.2 nur noch aus einer einzigen Regel besteht, welche sich nicht selbst beinhaltet (keine Rekursion) und auch nur noch aus Terminalsymbolen besteht. Für Klarheit und Übersicht der Grammatik lässt sich diese natürlich wieder, wie in Programmblock 2.1, in mehrere Regeln aufteilen. Das folgende Beispiel ist allerdings nur kontextfrei und nicht regulär, da die Definition eines *Ausdrucks* von sich selber abhängt und somit Rekursion aufweist.

```
1  Term        = 'a' | 'b' | 'c' ;
2  Ausdruck    = Term | '(' , Ausdruck , ')' ;
```

Würde man hier versuchen, wie in Programmblock 2.2 alle Regeln durch ihre Definition zu substituieren, würde dies nie gelingen. Das Einsetzen jeder Definition, lässt weiterhin nicht-terminale Symbole in der Definition. Sofern diese erneut eingesetzt werden, müsste sich dieser Prozess unendlich wiederholen.

```
1  Ausdruck = ( 'a' | 'b' | 'c' ) | '(' , ( Term | '(' , Ausdruck , ')' ) , ')' ;
```

Mögliche Programme in dieser Sprache sind unter anderem „a“, „(b)“ und „(((((c)))))“. Die Wichtigkeit dieses Unterschiedes zwischen regulären und kontextfreien Sprachen wird später in Bezug auf die lexikalische und syntaktische Analyse deutlich.

[5] der Name einer Regel

2.3. Beispiel an *rost*

Im Rahmen dieser Facharbeit habe ich für besseres Verständnis selbstständig eine Programmiersprache mit dem Namen *Roost* entwickelt. Da dieses Projekt aber zu groß geworden ist, um es hier verständlich zu erklären, entschied ich mich dazu, eine kleinere Version mit dem Namen *rost* zu erstellen. *rost* ist lediglich ein Rechner, welcher Punkt-vor-Strich-Rechnung sowie Klammersetzung beachtet. Die Grammatik für *rost* ist in Programmblock 2.3 zu sehen.

Programmblock 2.3: Die Grammatik von *rost*

```
(* predefined "lists" *)
DIGIT       = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

(* tokens *)
number      = DIGIT , { DIGIT } , [ '.' , DIGIT , { DIGIT } ] ;

(* nodes *)
Expression  = Term , { ( '+' | '-' ) , Term } ;
Term        = Factor , { ( '*' | '/' | '%' ) , Factor } ;
Factor      = ( '+' | '-' ) , Factor
            | '('  , Expression , ')'
            | number ;
```

In Zeile 5 ist die reguläre Sprache `number` definiert, welche eine einzelne Zahl darstellt. Sie besteht aus einer oder mehreren Ziffern, auf welche ein Punkt folgen kann. Wenn ein Punkt vorhanden ist, muss darauf wieder mindestens eine Ziffer folgen. So können sowohl ganze Zahlen als auch Dezimalzahlen dargestellt werden.

Eine `Expression` ist nicht regulär, da diese einen `Term` enthält, welcher einen `Factor` enthält, welcher wiederum in Zeile 11 eine `Expression` enthält. Mögliche `Expression`s sind unter anderem „42“, „3.1415“, „-10“, „2+2*2“, oder „(2+2)*2“. Für klares Verständnis empfiehlt es sich, alle diese Beispiele einmal anhand der Grammatik selbst zu zerlegen. Die letzten beiden Beispiele sind in den Abbildungen 2.1 und 2.2 als Syntaxbäume dargestellt.

Diese Darstellungen verdeutlichen, dass durch die Grammatik automatisch die Priorität der Rechenoperationen festgelegt wird. Bei „2+2*2“ in Abbildung 2.1 soll später zuerst das Produkt „2*2“ evaluiert[6] und das Ergebnis im Nachhinein zu „2“ addiert werden. Für „(2+2)*2“ (Abb. 2.2) soll allerdings erst „2+2“ berechnet werden, da es in Klammern steht und somit höhere Priorität hat.

Die Regel `Factor = ( '+' | '-' ) , Factor` in Zeile 6 erlaubt das Setzen von + und - Zeichen vor einen *Factor*, ohne eine ganze *Expression* bilden zu müssen. Konkret sind damit also

[6] ausgewertet

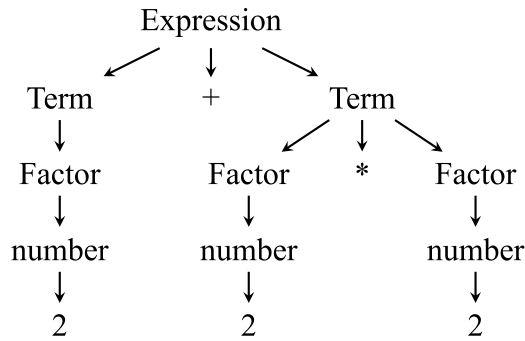

Abbildung 2.1.: Der Syntaxbaum für „2+2*2“

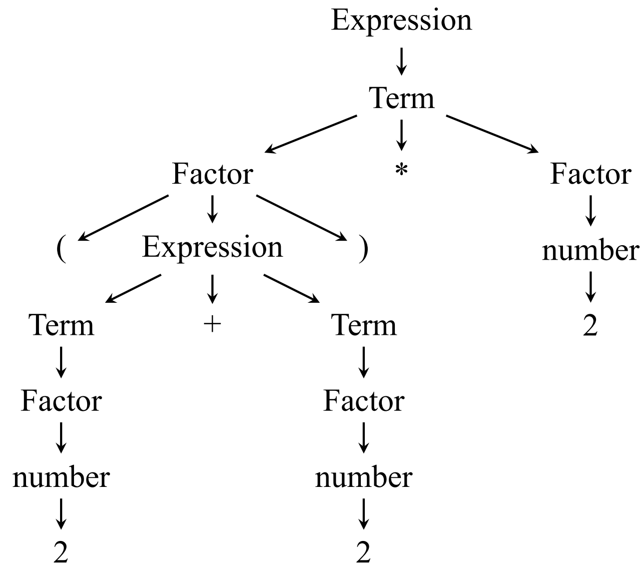

Abbildung 2.2.: Der Syntaxbaum für „(2+2)*2“

Programme wie „+5“, „-(3+2)“ und „--6“ möglich.

Das `%` Zeichen in Zeile 9 steht für die sogenannte *Modulo* Operation, welche den Rest einer Ganzzahldivision berechnet. Diese ist in der Priorität gleichgestellt mit Multiplikation und Division.

3. Der Lexer: lexikalische Analyse

Die bereits besprochene Grammatik ist für die ersten beiden Teile des Programmablaufs, den *Lexer* und *Parser*, von Bedeutung. Der Lexer nimmt als Eingabe einzelne Zeichen des Programmtextes und teilt diese anhand der regulären Regeln der Grammatik in *Tokens* auf. Der Parser verwendet diese *Tokens* und analysiert sie anhand der nicht regulären Grammatikregeln, um daraus einen *Syntaxbaum* zu bilden. Dies ist in Tabelle 3.1 übersichtlich dargestellt.

Ein *Parser* könnte gleichzeitig auch die Funktion des *Lexers* übernehmen, da reguläre Sprachen eine Unterklasse der komplexen Sprachen sind und ein Parser für komplexe Sprachen ausgelegt ist. Weil die regulären Regeln allerdings simpler sind, ist es einfacher und effizienter, diese separat zu scannen [WG84].

3.1. Vorgehensweise

Der *Lexer* liest den Programmtext Zeichen für Zeichen. Daher wird eine Variable, beispielsweise `current_char`, verwendet, um das zuletzt gelesene Zeichen zu speichern. Zudem wird eine Funktion benötigt, die das nächste Zeichen liest und den gelesenen Wert `current_char` zuweist. Diese heißt beispielsweise `next()`. Sollte der Programmtext einen Fehler aufweisen, wird eine Funktion zum Verarbeiten dieses Fehlers aufgerufen, hier `error()`. Zuletzt muss es eine Möglichkeit geben, anhand des aktuellen Zeichens festzustellen, zu welcher Regel dieses gehört. Hier gibt es die Funktion `first(expr)`, welche eine Liste von allen möglichen Zeichen zurückgibt, mit denen die Regel `expr` beginnen kann. Den verschiedenen regulären Grammatikregeln kann dann jeweils eine Prozedur zugewiesen werden, wie in Tabelle 3.2 dargestellt (vgl. [Wir08]).

Da der *Parser* nur *Tokens* als Eingabe akzeptiert, muss der *Lexer* den gesamten Programmtext in *Tokens* aufteilen. Bisher sind allerdings nur einige Tokens abgedeckt, nämlich diese, die durch reguläre Regeln in der Grammatik definiert sind. Im Fall von *rost* ist das nur `number`. Die übrigen

Tabelle 3.1.: Vergleich Lexer und Parser (nach [Wir08])

Prozess	Eingabeelement	Algorithmus	Grammatik	Ausgabeelement
Lexikalische Analyse	Zeichen	Lexer	regulär	Token
Syntaktische Analyse	Token	Parser	kontextfrei	Node

Tabelle 3.2.: Zuweisung von regulärer Grammatik zu Prozeduren (nach [Wir08])

Grammatik	Prozedur(Grammatik)
`'x'`	`if current_char == 'x' { next() } else { error() }`
`( expr )`	`Prozedur(expr)`
`[ expr ]`	`if current_char in first(expr) { Prozedur(expr) }`
`{ expr }`	`while current_char in first(expr) { Prozedur(expr) }`
`expr_1 , expr_2 , ... , expr_n`	`Prozedur(expr_1); Prozedur(expr_2); ... ; Prozedur(expr_n)`

`expr_1 | expr_2 | ... | expr_n`	`if current_char in first(expr_1) { Prozedur(expr_1) }`
	`else if current_char in first(expr_2) { Prozedur(expr_2) }`
	`...`
	`else if current_char in first(expr_n) { Prozedur(expr_n) }`
	`else { error() }`

Tokens sind alle *Terminalsymbole*, welche in den nicht-regulären Regeldefinitionen vorkommen [Wir08].

Ein *Token* hat immer einen Typ und einen Wert [Bal18]. Der Typ ist beispielsweise `Number` für eine Zahl, oder `Plus` für ein +. Der Wert ist nur für die nicht-terminalen *Tokens* wichtig (hier nur `Number`), da diese nicht immer den gleichen Wert haben.

Bei komplizierteren Grammatiken mag es notwendig sein, mehr als nur ein Zeichen vorauszusehen.

3.2. Der Lexer von *rost*

Der vollständige Programmtext des Lexers kann in Anhang A eingesehen werden. In diesem Unterkapitel werden einzelne Ausschnitte genauer erklärt.

Die *Token* Typen sind in *rost* in einer `enum`[1] gespeichert. Diese ist in Programmblock 3.1 zu sehen. Ein weiterer Token Typ, `EOF`[2], der weder ein *Terminalsymbol*, noch eine reguläre Grammatik ist, wird ebenfalls definiert. Dieser wird als letztes Token erscheinen, um dem *Parser* das Ende zu markieren. Ein gesamtes *Token* wird dann wie in Programmblock 3.2 dargestellt. Der Wert eines *Tokens* wird in einem `String` gespeichert.

Ein *Lexer* speichert zum einen den `input`, also den Programmtext, den `current_char` und den `current_char_index`, der speichert an welcher Position des `input`s sich `current_char` befindet (siehe Programmblock 3.3).

Die Funktion `scan()` startet das Scannen des Programms und gibt eine Liste von *Tokens* zu-

[1]kurz für *enumeration*, Deutsch: *Auflistung*
[2]kurz für *end of file*, Deutsch: *Ende der Datei*

Programmblock 3.1: Token Typen in *rost*

```
2  pub enum TokenType {
3      LParen,   // '('
4      RParen,   // ')'
5      Plus,     // '+'
6      Minus,    // '-'
7      Multiply, // '*'
8      Divide,   // '/'
9      Modulo,   // '%'
10
11     Number,
12
13     EOF,
14 }
```

Programmblock 3.2: Token in *rost*

```
17 pub struct Token {
18     pub token_type: TokenType,
19     pub value: String,
20 }
```

rück. Solange es noch ein weiteres Zeichen im Programmtext gibt, wird anhand dieses Zeichens ein Token erzeugt. Wenn das aktuelle Zeichen ein Leerzeichen, HT[3] oder CR[4] ist, soll er dieses ignorieren und zum nächsten Zeichen gehen. Wenn das Zeichen ein einzelnes Token darstellt, wird dieses Token durch die Funktion `make_single_char()` erstellt. Wenn das Zeichen eine Ziffer ist, soll durch die Funktion `make_number()` versucht werden, ein `Number` Token zu erstellen. Bei anderen Zeichen soll ein Fehler angezeigt werden, dass das Zeichen an dieser Stelle nicht erwartet wurde. Hierzu wurde in diesem Fall `panic!()` verwendet, welches die Ausführung des gesamten Programms mit einer Fehlermeldung stoppt (siehe Programmblock 3.4).

[3]Horizontal Tab
[4]Carriage Return; wird von Windows zusätzlich zu LF für Zeilenumbrüche verwendet

Programmblock 3.3: Attribute des *rost* Lexers

```
3  pub struct Lexer {
4      input: String,
5      current_char: Option<char>,
6      current_char_index: usize,
7  }
```

Programmblock 3.4: Funktion `scan()` des *rost* Lexers

```
19  pub fn scan(&mut self) -> Vec<Token> {
20      let mut tokens = vec![];
21
22      while let Some(current_char) = self.current_char {
23          match current_char {
24              ' ' | '\t' | '\r' => self.next(),
25              '(' => tokens.push(self.make_single_char(TokenType::LParen,   "(")),
26              ')' => tokens.push(self.make_single_char(TokenType::RParen,   ")")),
27              '+' => tokens.push(self.make_single_char(TokenType::Plus,     "+")),
28              '-' => tokens.push(self.make_single_char(TokenType::Minus,    "-")),
29              '*' => tokens.push(self.make_single_char(TokenType::Multiply, "*")),
30              '/' => tokens.push(self.make_single_char(TokenType::Divide,   "/")),
31              '%' => tokens.push(self.make_single_char(TokenType::Modulo,   "%")),
32              _ => {
33                  if current_char.is_ascii_digit() {
34                      tokens.push(self.make_number());
35                  } else {
36                      panic!("SyntaxError: Illegal character `{}`", current_char);
37                  }
38              }
39          }
40      }
41      tokens.push(Token::new(TokenType::EOF, String::new()));
42
43      return tokens;
44  }
```

Hier wird auch in Zeile 24 die Funktion `next()` benutzt. Diese erhöht den `current_char_index` um 1 und setzt `current_char` zu dem Zeichen des Programmtexts an dieser Stelle.

```
46  fn next(&mut self) {
47      self.current_char_index += 1;
48      self.current_char = self.input.chars().nth(self.current_char_index);
49  }
```

Die Funktion `make_number()` (einsehbar in Anhang A) benutzt die in Tabelle 3.2 gezeigten Prozeduren, um die in Programmblock 2.3 gezeigte Grammatik für `number` einzuhalten. Es wird vorab ein neuer leerer *String* erstellt, welcher den Wert des Tokens halten soll. Danach wird das aktuelle Zeichen diesem String hinzugefügt und zum nächsten Zeichen gesprungen. Dies soll wiederholt werden, solange das aktuelle Zeichen eine Ziffer ist. Für den Fall, dass danach ein Punkt folgt, sollen dieser und die folgenden Ziffern auf dieselbe Weise ebenfalls dem String hinzugefügt werden.

4. Parser: syntaktische Analyse

Der *Parser* hat die Aufgabe, die nicht-regulären Regeln der Grammatik zu erkennen. Er bekommt als Eingabe die Tokens vom Lexer und gibt einen Syntaxbaum aus.

Der hier dargestellte Parser benutzt die „Methode des rekursiven Abstiegs“ [Wir08], auch „recursive descent parsing“ [Nac21] genannt. Diese fällt unter die Gruppe der *Top-down* Parser [Knu71]. Im Gegensatz dazu gibt es auch *Bottom-up* Parser, die nicht bei der obersten Node anfangen, sondern ganz unten im Baum. Außerdem gibt es sowohl im Lexer, als auch im Parser, die Möglichkeit, mehr als nur ein Zeichen oder Token vorauszusehen. Damit ließen sich weitere Grammatiken implementieren, allerdings mit dem Nachteil der Verlangsamung [Wir08].

Häufig wird der Lexer auch nicht so strikt vom Parser abgetrennt. Stattdessen hält der Parser nicht eine Liste von Tokens, sondern einen Lexer, und ruft bei Bedarf eine Funktion des Lexers auf, die ein weiteres Token erzeugt und zurückgibt [Bal18]. Die Abtrennung wurde hier für besseres Verständnis vorgenommen.

4.1. Vorgehensweise

Ein Parser unterscheidet sich tatsächlich nur bedingt von einem Lexer. Der einzige Unterschied ist das Vorkommen von Rekursion. Während beim Lexer einfach getestet werden konnte, ob das aktuelle Zeichen dem jeweiligen Terminalsymbol entspricht, muss es beim Parser für die nicht-terminalen Symbole weitere Funktionen geben. Jede dieser Funktionen kann sich selber oder andere Funktionen rekursiv aufrufen und wirft einen Fehler, wenn der Programmtext die Grammatik nicht einhält [Wir08]. Es gibt für jede nicht-reguläre Regel in der Grammatik eine Funktion, die jeweils eine *Node*[1] des Syntaxbaums zurückgibt. Diese *Node* hält die relevanten Informationen für die jeweilige Regel. Die Node für die Regel `Term = '[' , Term , ']' | number ;` würde beispielsweise so aussehen:

```
1  enum Term {
2      Recursive { contained: Term }, // Hält eine weitere Node vom Typ Term
3      Number { value: decimal }, // Hält eine beliebige Dezimalzahl
4  }
```

[1] Knotenpunkt

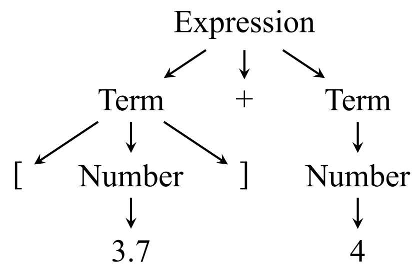

Abbildung 4.1.: Der Syntaxbaum für „[3.7]+4“

Andernfalls würde eine Node für die Regel `Expression = Term , '+' , Term ;`, die nur eine Möglichkeit aufweist, folgendermaßen aussehen:

```
struct Expression {
    left: Term,
    right: Term,
}
```

Das gesamte Programm wird am Ende aus nur einer Node bestehen, der weitere Nodes untergeordnet sind. Daher kann die Ausgabe des Parsers in einem Syntaxbaum dargestellt werden, bei dem die Nodes durch Pfeile verbunden sind. Zwei Beispiele dafür gab es bereits in den Abbildungen 2.1 und 2.2, ein weiteres für die eben genannte Grammatik ist in Abbildung 4.1 zu sehen.

4.2. Der Parser von *rost*

Der vollständige Programmtext des Parsers kann in Anhang B eingesehen werden. In diesem Unterkapitel werden einzelne Ausschnitte genauer erklärt.

Die Definitionen der Nodes in *rost* sind in Programmblock 4.1 gezeigt. Das `Box<>` in den Zeilen 6 und 16, sowie das `#[derive(Clone)]` vor jeder Node können hier zum Verständnis ignoriert werden. Zum Speichern des jeweiligen Operators wird der Typ des passenden Tokens gespeichert, hier in Zeilen 7, 12 und 16.

Der Parser hält eine Liste der Tokens vom Programmtext in `tokens`, den aktuellen Token in `current_token` und die Stelle des aktuellen Tokens in der Liste in `current_token_index`.

Durch den ähnlichen Aufbau zum Lexer weist auch die Funktion `next()` Gemeinsamkeiten auf, mit dem Unterschied, dass diese ein Token vom Typ `EOF` zurückgibt, sobald die Länge der Token-Liste überschritten ist.

Die Hauptfunktion `parse()` in Zeile 20 ruft die Funktion `expression()` auf, da das gesamte Programm aus einer einzigen `Expression` bestehen soll. Wenn daraufhin kein `EOF` Token folgt,

Programmblock 4.1: Node Definitionen von *rost*

```
4  #[derive(Clone)]
5  pub struct Expression {
6      pub term: Box<Term>,
7      pub following: Vec<(TokenType, Term)>,
8  }
9  #[derive(Clone)]
10 pub struct Term {
11     pub factor: Factor,
12     pub following: Vec<(TokenType, Factor)>,
13 }
14 #[derive(Clone)]
15 pub enum Factor {
16     Unary(TokenType, Box<Factor>),
17     Expression(Expression),
18     Number(BigDecimal),
19 }
```

```
4  pub struct Parser {
5      tokens: Vec<Token>,
6      current_token: Token,
7      current_token_index: usize,
8  }
```

soll ein Fehler geworfen werden. Die restlichen Funktionen sind ähnlich zum Lexer aufgebaut. Interessant ist hierbei noch die Funktion `factor()`, da die Regel für `Factor` drei verschiedene Möglichkeiten aufweist. Da wir bereits beim Schreiben der Grammatik darauf geachtet haben, dass die ersten Zeichen und somit auch die ersten Tokens aller drei Möglichkeiten disjunkt[2] sind, lässt sich an diesen einfach erkennen, welche der drei Möglichkeiten zutrifft.

[2] ohne Überschneidung

5. Compiler: Übersetzung in Maschinensprache

Wie in der Einleitung beschrieben, werden manche Programmiersprachen vorab von einem *Compiler* kompiliert und manche von einem anderem Programm, dem *Interpreter*, interpretiert und somit sofort ausgeführt. Weil aber nicht jeder Prozessor gleich aufgebaut ist und nicht jedes Betriebssystem gleich funktioniert, ist Maschinensprache längst nicht auf jedem Gerät gleich. Daher muss ein *Compiler* viele verschiedene Zielplattformen unterstützen, was eine Menge Arbeit bei der Erstellung des Compilers bedeutet. Um dies zu umgehen, haben Sprachen wie *Java* und *Python* einen eigenen sogenannten *Bytecode*, zu dem das Programm vorerst kompiliert wird. Dieser *Bytecode* wird dann von einem *Interpreter* oder einem weiteren Compiler auf der Zielmaschine ausgeführt.

5.1. LLVM

Alternativ zu einem *Bytecode* benutzen viele Compiler der heute bekannten kompilierten Sprachen, beispielsweise der *Rust* Compiler, der *clang* C und C++ Compiler sowie der *Glasgow Haskell Compiler (GHC)* das Hilfsmittel *LLVM*[1]. Dieses nimmt ein Programm in Form der *LLVM Intermediate Representation*, kurz *LLVM IR* [Sar15], und übernimmt von da an die Kompilierung zu den verschiedenen Zielsystemen. Die Idee ist also ähnlich zu der von *Bytecode*, die Anpassung an das Zielsystem passiert aber bereits bei der Kompilierung eines Programms und nicht erst bei der Ausführung. Somit ist der Prozess des Kompilierens in zwei Teile aufgeteilt. Einmal das *frontend*, mit den Aufgaben des Lexers, des Parsers und den Syntaxbaum in eine *IR* umzuwandeln, und das *backend*, das diese *IR* benutzt um Maschinensprache zu erzeugen [Nac21]. *LLVM* ist ein solches *backend*, weshalb nur noch das *frontend* selber erstellt werden muss.

Der Einfachheit halber habe ich sowohl für *rost*, als auch für *Roost* nur einen *Interpreter* erstellt. Aus diesem Grund werde ich auch nicht weiter auf Compiler eingehen.

[1] *LLVM* ist kein Akronym, sondern der vollständige Eigenname

6. Interpreter: Ausführung des Programms

Die Aufgabe eines Interpreters ist es, das Programm sofort, ohne vorige Kompilierung, auszuführen. Die Abtrennung zwischen Compiler und Interpreter ist aber nicht genau zu erkennen. Ein purer Compiler erstellt aus dem Programmtext sofort Maschinensprache; ein purer Interpreter führt das Programm ohne voriges Verarbeiten des Syntaxbaums sofort aus. Die meisten Implementationen von Programmiersprachen entsprechen allerdings keinem dieser beiden Extrema. Zum Beispiel wird erst zu einem *Bytecode* oder einer *IR*[1] kompiliert. Außerdem gibt es sogenannte *JIT*[2] Compiler, bei denen während der Ausführung die Maschinensprache erzeugt wird [Bal18].

Hier werde ich das Prinzip eines sogenannten *Tree-walking Interpreters* [Bal18] zeigen, welches das direkteste und unkomplizierteste ist.

6.1. Vorgehensweise

Der Interpreter hat, genau wie der Parser, für jede Node eine Funktion. Jede Funktion nimmt als Argument eine Instanz der Node und gibt einen uniformen Datentypen zurück. In dem Fall eines Rechners, wie *rost*, ist dieser Datentyp eine Zahl. Bei komplexeren Programmiersprachen mit mehreren Datentypen kann eine `enum` benutzt werden, welche Werte verschiedener Datentypen speichern kann. Diese enum für *Roost* ist in Block 6.1 zu sehen.

Eine Funktion `run()` nimmt die oberste Node des Syntaxbaums und ruft die entsprechende Funktion für diese Node auf. Funktionen für Statement Nodes, also diese, die keinen Wert zurückgeben, sollen bloß das jeweils Notwendige ausführen. Bei Expression Nodes hingegen soll zu der Ausführung auch noch ein Wert zurückgegeben werden. Was ein Statement und was eine Expression ist, hängt von der Sprache ab. So ist zum Beispiel ein `if` in Sprachen wie *Java* oder *C* ein Statement, aber in *Rust* und *Kotlin* eine Expression. Insgesamt ist der Großteil des Verhaltens einer Programmiersprache vom Interpreter beziehungsweise Compiler abhängig. (vgl. [Bal18])

Soll es in der erstellten Sprache möglich sein, Funktionen oder Variablen zu definieren, müs-

[1] *Intermediate Representation*
[2] *Just In Time*; Deutsch: *gerade rechtzeitig*

Programmblock 6.1: Datentypen in *Roost*

```
13 pub enum Value {
14     Number(Decimal),
15     Bool(bool),
16     String(String),
17     Range(Decimal, Decimal),
18     Function(Vec<String>, Statements),
19     BuiltIn,
20     Null,
21 }
```

sen diese auch gespeichert werden. Hierzu bietet sich eine `HashMap`[3] mit Strings als Schlüsseln und `Value`s als zugehörigen Werten an [Bal18]. Je nach gewünschter Implementation können beispielsweise mehrere Scopes, also `HashMap`s, in einer Liste gespeichert werden und beim Suchen eines Variablenwertes im obersten Scope anfangend immer tiefer gesucht werden. Vordefinierte eingebaute Variablen und Funktionen können so in einem vordefinierten Scope unter allen anderen Scopes gespeichert werden, also bei Index `0` in der Liste.

Die Implementation von `return`, `continue` und `break` Statements erfordert außerdem die Möglichkeit, die Ausführung teilweise zu stoppen und bis zu einem bestimmten Punkt im Baum zurückzukehren. Eine mögliche Lösung dafür ist, nicht sofort einen `Value` zurückzugeben, sondern einen `Result` Typen, der optional einen `Value` enthalten kann, aber auch speichert, ob eines dieser besonderen Statements aufgerufen wurde.

6.2. Die Interpreter von *rost* und *Roost*

Zu dem *rost* Interpreter gibt es tatsächlich nicht viel zu sagen, da dieser nur einfache Rechenoperationen durchführt. Der vollständige Programmtext kann aber in Anhang C eingesehen werden. Stattdessen werde ich kurz auf den deutlich komplexeren Interpreter von *Roost* eingehen. Die Grammatik für *Roost* ist in Anhang D einsehbar. Dabei sind folgende zwei Dinge noch nicht erklärt:

1. Text zwischen zwei `?` erklärt eine beliebige Sequenz, die anders nicht darstellbar ist. `CHAR = ? any UTF8 character ? ;` bedeutet also: ein beliebiges Zeichen des UTF-8 Zeichensatzes.
2. Ein `-` in EBNF steht für *außer*. `CHAR - '"'` heißt also: ein beliebiges Zeichen, *außer* `"`.

Zu Beginn ist das Erzeugen eines Interpreters interessant, da hierbei ein paar eingebaute Funktionen definiert werden.

[3] auch *Dictionary*, *Map* oder *Hash*; eine Sammlung von Schlüsseln zu Werten

```
72      pub fn new(start_node: Statements, stdout: OUT, exit: EXIT) -> Self {
73          return Interpreter {
74              start_node,
75              scopes: vec![HashMap::from([
76                  (String::from("print"), Value::BuiltIn),
77                  (String::from("printl"), Value::BuiltIn),
78                  (String::from("typeOf"), Value::BuiltIn),
79                  (String::from("exit"), Value::BuiltIn),
80              ])],
81              current_scope_index: 0,
82              stdout,
83              exit,
84          };
85      }
```

Beim Aufruf einer dieser Funktionen wird dann entsprechend eine selbst definierte Funktion aufgerufen. Der wichtige Teil ist hierbei in den Zeilen 552 bis 556, in denen die Funktionen aus dem `built_in` Modul aufgerufen werden.

```
543  Value::BuiltIn => {
544      let mut args: Vec<Value> = vec![];
545      for arg in &node.args {
546          result.register(self.visit_expression(&arg)?);
547          should_return!(result);
548          args.push(result.value.clone().unwrap());
549      }
550
551      let value = match node.identifier.as_str() {
552          "print" => built_in::print(args, &mut self.stdout, node.start.clone(), node.
     end.clone(), false),
553          "printl" => built_in::print(args, &mut self.stdout, node.start.clone(), node.
     end.clone(), true),
554          "typeOf" => built_in::type_of(args, node.start.clone(), node.end.clone()),
555          "exit" => built_in::exit(args, &mut self.exit, node.start.clone(), node.end.
     clone()),
556          _ => panic!(),
557      }?;
558      result.success(Some(value));
559      return Ok(result);
560  },
```

Um mit den Scopes umzugehen, in denen Variablen und Funktionen gespeichert werden, gibt es einige Hilfsfunktionen. Je nachdem, wie sich die gewünschte Sprache verhalten soll, müssten diese natürlich verändert werden und es müssten möglicherweise sogar eigene Scopes nur für Funktionen existieren.

Die Funktion `push_scope()` fügt der Scope-Liste einen neuen leeren Scope hinzu und ändert

Programmblock 6.2: Scope Hilfsfunktionen im *Roost* Interpreter

```
 96     fn push_scope(&mut self) {
 97         self.scopes.push(HashMap::new());
 98         self.current_scope_index += 1;
 99     }
100 
101     fn pop_scope(&mut self) {
102         self.scopes.pop();
103         self.current_scope_index -= 1;
104     }
105 
106     fn current_scope(&mut self) -> &mut HashMap<String, Value> {
107         return &mut self.scopes[self.current_scope_index];
108     }
109 
110     fn find_var(&self, name: &String, start_loc: Location, end_loc: Location) ->
    Result<&Value> {
111         let mut scope = self.current_scope_index;
112         loop {
113             if self.scopes[scope].contains_key(name) {
114                 return Ok(self.scopes[scope].get(name).unwrap());
115             }
116             if scope == 0 {
117                 error!(ReferenceError, start_loc, end_loc, "Variable or function with
    name '{}' not found", name);
118             }
119             scope -= 1;
120         }
121     }
```

dementsprechend den `current_scope_index`. Die Funktion `pop_scope()` entfernt den obersten Scope und damit auch alle darin gespeicherten Variablen. `current_scope()` gibt den aktuellen Scope zurück und erleichtert somit das Zugreifen auf diesen. Zuletzt ist in Zeile 110 die Funktion `find_var()` definiert. Diese nimmt einen Variablennamen und durchsucht die Scopes, bei dem aktuellen[4] anfangend, nach einer Variable mit diesem Namen. Wird eine gefunden, wird der entsprechende Wert zurückgegeben. Ansonsten soll ein Fehler geworfen werden.

[4] also hier der letzte Index der Liste

7. Abschluss

Wenn noch weiteres Interesse an den *rost* und *Roost* Projekten bestehen sollte, können die vollständigen Quelltexte dieser und weiterer Projekte auf GitHub unter den Links in Tabelle 7.1 eingesehen werden. Die letzte Spalte gibt jeweils den letzten Commit nach jetzigem Stand an.

Außerdem biete ich sowohl für *rost* als auch für *Roost* interaktive Internetseiten an, auf denen man die jeweiligen Interpreter mit eigenen Programmen ausprobieren kann. Diese sind unter `https://rost.rubixdev.de` und `https://roost.rubixdev.de` erreichbar.

Ein kurzes Beispielprogramm in der Sprache *Roost* zur Berechnung der ersten 42 Zahlen der Fibonacci-Folge[1] ist in Programmblock 7.1 zu sehen.

Programmblock 7.1: Berechnung der Fibonacci-Folge in *Roost*

```
1   var previous = 0
2   var current = 1
3   var next = null
4
5   printl('fib(1) = ' + current)
6   for (n in 2..=42) {
7       next = previous + current
8       previous = current
9       current = next
10      printl('fib('+n+') = ' + current)
11  }
```

[1] Eine Folge von Zahlen, bei der jede Zahl die Summe der beiden Vorgänger ist

Tabelle 7.1.: Quelltexte der Projekte

Projekt	URL	Commit
rost wie im Anhang	`https://github.com/RubixDev/rost/tree/simple`	`723e717`
rost weitergeführt	`https://github.com/RubixDev/rost`	`df30fed`
rost Internetseite	`https://github.com/RubixDev/rost-web`	`06bde37`
Roost	`https://github.com/RubixDev/roost`	`d0e907d`
Roost Internetseite	`https://github.com/RubixDev/roost-web`	`404cd01`

Abbildungsverzeichnis

2.1. Der Syntaxbaum für „2+2*2“ 9
2.2. Der Syntaxbaum für „(2+2)*2“ 9

4.1. Der Syntaxbaum für „[3.7]+4“ 15

Tabellenverzeichnis

3.1. Vergleich Lexer und Parser (nach [Wir08]) 10
3.2. Zuweisung von regulärer Grammatik zu Prozeduren (nach [Wir08]) 11

7.1. Quelltexte der Projekte 22

Programmblockverzeichnis

2.1. Grammatik für einen simplen Satz 5
2.2. Grammatik für einen simplen Satz - Kurzfassung 6
2.3. Die Grammatik von *rost* 8

3.1. Token Typen in *rost* 12
3.2. Token in *rost* 12
3.3. Attribute des *rost* Lexers 12
3.4. Funktion `scan()` des *rost* Lexers 13

4.1. Node Definitionen von *rost* 16

6.1. Datentypen in *Roost* 19
6.2. Scope Hilfsfunktionen im *Roost* Interpreter 21

7.1. Berechnung der Fibonacci-Folge in *Roost* 22

Literaturverzeichnis

[Bac+60] J. W. Backus u. a. „Report on the Algorithmic Language ALGOL 60“. In: *Comm. ACM* 3 (Mai 1960), S. 299–314. ISSN: 0001-0782.

[Knu71] Donald E. Knuth. „Top-down syntax analysis“. In: *Acta Informatica* 1 (Juni 1971), S. 79–110. ISSN: 1432-0525.

[Wir77] Niklaus Wirth. „Modula: A language for modular multiprogramming“. In: *Software: Practice and Experience* 7 (1977), S. 3–35.

[WG84] William M. Waite und Gerhard Goos. *Compiler construction*. Texts and monographs in computer science. New York [u.a.]: Springer, 1984. ISBN: 3-540-90821-8; 0-387-90821-8.

[Shi85] Stuart M. Shieber. „Evidence Against the Context-Freeness of Natural Language“. In: *Linguistics and Philosophy* 8 (Aug. 1985), S. 333–343. ISSN: 1573-0549.

[Wir08] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. 2., bearb. Aufl. München [u.a.]: Oldenbourg, 2008. ISBN: 978-3-486-58581-0.

[Sar15] Suyog Sarda. *LLVM essentials : become familiar with the LLVM infrastructure and start using LLVM libraries to design a computer*. Birmingham: Packt Publishing, 2015. ISBN: 978-1-78528-080-1.

[Bal18] Thorsten Ball. *Writing an Interpreter in Go*. United States: Thorsten Ball, 2018. ISBN: 978-3982016115.

[Nac21] Kai Nacke. *Learn LLVM 12 A Beginner's Guide to Learning LLVM Compiler Tools and Core Libraries with C*. Birmingham: Packt Publishing, Limited, 2021. ISBN: 978-1-83921-350-2.

A. Lexer von *rost*

```rust
use crate::tokens::{Token, TokenType};

pub struct Lexer {
    input: String,
    current_char: Option<char>,
    current_char_index: usize,
}

impl Lexer {
    pub fn new(input: String) -> Self {
        let first_char = input.chars().nth(0);
        return Lexer {
            input,
            current_char: first_char,
            current_char_index: 0,
        };
    }

    pub fn scan(&mut self) -> Vec<Token> {
        let mut tokens = vec![];

        while let Some(current_char) = self.current_char {
            match current_char {
                ' ' | '\t' | '\r' => self.next(),
                '(' => tokens.push(self.make_single_char(TokenType::LParen,   "(")),
                ')' => tokens.push(self.make_single_char(TokenType::RParen,   ")")),
                '+' => tokens.push(self.make_single_char(TokenType::Plus,     "+")),
                '-' => tokens.push(self.make_single_char(TokenType::Minus,    "-")),
                '*' => tokens.push(self.make_single_char(TokenType::Multiply, "*")),
                '/' => tokens.push(self.make_single_char(TokenType::Divide,   "/")),
                '%' => tokens.push(self.make_single_char(TokenType::Modulo,   "%")),
                _ => {
                    if current_char.is_ascii_digit() {
                        tokens.push(self.make_number());
                    } else {
                        panic!("SyntaxError: Illegal character `{}`", current_char);
                    }
                }
            }
        }
        tokens.push(Token::new(TokenType::EOF, String::new()));

        return tokens;
    }

```

```
46      fn next(&mut self) {
47          self.current_char_index += 1;
48          self.current_char = self.input.chars().nth(self.current_char_index);
49      }
50
51      // ---------------------------------------------
52
53      fn make_single_char(&mut self, token_type: TokenType, value: &str) -> Token {
54          self.next();
55          return Token::new(token_type, value.to_string());
56      }
57
58      fn make_number(&mut self) -> Token {
59          let mut number = String::new();
60          number.push(self.current_char.unwrap());
61          self.next();
62
63          while self.current_char != None && self.current_char.unwrap().is_ascii_digit()
 {
64              number.push(self.current_char.unwrap());
65              self.next();
66          }
67
68          if self.current_char == Some('.') {
69              number.push('.');
70              self.next();
71              if self.current_char != None && self.current_char.unwrap().is_ascii_digit
() {
72                  number.push(self.current_char.unwrap());
73                  self.next();
74
75                  while self.current_char != None && self.current_char.unwrap().
is_ascii_digit() {
76                      number.push(self.current_char.unwrap());
77                      self.next();
78                  }
79              } else {
80                  panic!("SyntaxError: Expected digit after decimal point");
81              }
82          }
83
84          return Token::new(TokenType::Number, number);
85      }
86  }
```

B. Parser von *rost*

```
use bigdecimal::BigDecimal;
use crate::{tokens::{Token, TokenType}, nodes::{Expression, Term, Factor}};

pub struct Parser {
    tokens: Vec<Token>,
    current_token: Token,
    current_token_index: usize,
}

impl Parser {
    pub fn new(tokens: Vec<Token>) -> Parser {
        let first_token = tokens[0].clone();
        return Parser {
            tokens,
            current_token: first_token,
            current_token_index: 0,
        }
    }

    pub fn parse(&mut self) -> Expression {
        let expression = self.expression();
        if self.current_token.token_type != TokenType::EOF
            { panic!("SyntaxError: Expected end of file"); }
        return expression;
    }

    fn next(&mut self) {
        self.current_token_index += 1;
        self.current_token = self.tokens
            .get(self.current_token_index)
            .unwrap_or(&Token::new(TokenType::EOF, String::new()))
            .clone();
    }

    fn expression(&mut self) -> Expression {
        let term = self.term();

        let mut following = vec![];
        while [
            TokenType::Plus,
            TokenType::Minus,
        ].contains(&self.current_token.token_type) {
            let operator = self.current_token.token_type.clone();
            self.next();
            following.push((operator, self.term()));
```

```
        }

        return Expression { term: Box::new(term), following };
    }

    fn term(&mut self) -> Term {
        let factor = self.factor();

        let mut following = vec![];
        while [
            TokenType::Multiply,
            TokenType::Divide,
            TokenType::Modulo,
        ].contains(&self.current_token.token_type) {
            let operator = self.current_token.token_type.clone();
            self.next();
            following.push((operator, self.factor()));
        }

        return Term { factor, following };
    }

    fn factor(&mut self) -> Factor {
        if self.current_token.token_type == TokenType::LParen {
            self.next();
            let expression = self.expression();
            if self.current_token.token_type != TokenType::RParen
                { panic!("SyntaxError: Expected `)`, got `{}`", self.current_token.value); }
            self.next();
            return Factor::Expression(expression);
        }

        if self.current_token.token_type == TokenType::Number {
            let num = Factor::Number(self.current_token.value.parse::<BigDecimal>().unwrap());
            self.next();
            return num;
        }

        if [
            TokenType::Plus,
            TokenType::Minus,
        ].contains(&self.current_token.token_type) {
            let operator = self.current_token.token_type.clone();
            self.next();
            return Factor::Unary(operator, Box::new(self.factor()));
        }

        panic!("SyntaxError: Expected expression");
    }
}
```

C. Interpreter von *rost*

```
1  use bigdecimal::BigDecimal;
2  use crate::{nodes::{Expression, Term, Factor}, tokens::TokenType};
3  
4  pub struct Interpreter {}
5  impl Interpreter {
6      pub fn run(nodes: Expression) {
7          let interpreter = Interpreter {};
8          println!("{}", interpreter.visit_expression(&nodes));
9      }
10 
11     fn visit_expression(&self, node: &Expression) -> BigDecimal {
12         let mut base = self.visit_term(&*node.term);
13 
14         for (operator, term) in &node.following {
15             let other = self.visit_term(&term);
16             match operator {
17                 TokenType::Plus => { base += other },
18                 TokenType::Minus => { base -= other },
19                 _ => panic!(),
20             }
21         }
22 
23         return base;
24     }
25 
26     fn visit_term(&self, node: &Term) -> BigDecimal {
27         let mut base = self.visit_factor(&node.factor);
28 
29         for (operator, factor) in &node.following {
30             let other = self.visit_factor(&factor);
31             match operator {
32                 TokenType::Multiply => { base *= other },
33                 TokenType::Divide => { base = base / other },
34                 TokenType::Modulo => { base = base % other },
35                 _ => panic!(),
36             }
37         }
38 
39         return base;
40     }
41 
42     fn visit_factor(&self, node: &Factor) -> BigDecimal {
43         return match node {
44             Factor::Unary(operator, factor) => {
45                 let base = self.visit_factor(factor);
```

```
46                  match operator {
47                      TokenType::Plus => base,
48                      TokenType::Minus => -base,
49                      _ => panic!(),
50                  }
51              },
52              Factor::Expression(expression) => self.visit_expression(expression),
53              Factor::Number(number) => number.clone(),
54          };
55      }
56  }
```

D. Grammatik von *Roost*

```
(* predefined "lists" *)
LETTER_     = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
            | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
            | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd'
            | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n'
            | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x'
            | 'y' | 'z' | '_' ;
DIGIT       = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
OCTAL       = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ;
HEX         = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
            | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd'
            | 'e' | 'f' ;
CHAR        = ? any UTF8 character ? ;
ESCAPE_CHAR = '\' | "'" | '"' | 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v' ;

(* tokens *)
eol     = ? line break ? | ';' ;
bool    = 'true' | 'false' ;
string  = '"' , [ { CHAR - '"' - '\' } , { escape , { CHAR - '"' - '\' } } ] , '"'
        | "'" , [ { CHAR - "'" - '\' } , { escape , { CHAR - "'" - '\' } } ] , "'" ;
escape  = '\' , ( ESCAPE_CHAR
                | OCTAL , OCTAL , OCTAL
                | 'x' , HEX , HEX
                | 'u' , HEX , HEX , HEX , HEX
                | 'U' , HEX , HEX , HEX , HEX , HEX , HEX , HEX , HEX ) ;
identifier  = LETTER_ , { LETTER_ | DIGIT } ;
number      = DIGIT , { DIGIT | '_' } , [ '.' , DIGIT , { DIGIT | '_' } ]
            | '.' , DIGIT , { DIGIT , | '_' } ;

Arguments       = '(' , [ Expression , { ',' , Expression } ] , ')' ;
ArgumentNames   = '(' , [ identifier , { ',' , identifier } ] , ')' ;

(* expression nodes *)
Expression  = OrExpr , [ '..' , [ '=' ] , OrExpr ] ;
OrExpr      = AndExpr , { '|' , AndExpr } ;
AndExpr     = EqExpr , { '&' , EqExpr } ;
EqExpr      = RelExpr , [ ( '==' | '!=' ) , RelExpr ] ;
RelExpr     = AddExpr , [ ( '<' | '>' | '<=' | '>=' ) , AddExpr ] ;
AddExpr     = MulExpr , { ( '+' | '-' ) , MulExpr } ;
MulExpr     = UnaryExpr , { ( '*' | '/' | '%' | '\' ) , UnaryExpr } ;
UnaryExpr   = ( '+' | '-' | '!' ) , UnaryExpr
              | PowExpr ;
PowExpr     = Atom , [ '**' , UnaryExpr ] ;
Atom        = number
            | bool
```

```
46              | string
47              | identifier
48              | CallExpr
49              | IfExpr
50              | 'null'
51              | '(' , Expression , ')'
52              | FunExpr
53              | '{' , Statements , '}' ;
54 CallExpr     = identifier , Arguments ;
55 IfExpr       = 'if' , '(' , Expression , ')' , Block , [ { eol } , 'else' , Block ] ;
56 FunExpr      = 'fun' , ArgumentNames , Block ;
57
58 (* statement nodes *)
59 Statement        = DeclareStmt
60                  | AssignStmt
61                  | LoopStmt
62                  | WhileStmt
63                  | ForStmt
64                  | FunStmt
65                  | Expression
66                  | BreakStmt
67                  | ContinueStmt
68                  | ReturnStmt ;
69 DeclareStmt      = 'var' , identifier , '=' , Expression ;
70 AssignStmt       = identifier , ( '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '\=' | '**='
    ) , Expression ;
71 LoopStmt         = 'loop' , Block ;
72 WhileStmt        = 'while' , '(' , Expression , ')' , Block ;
73 ForStmt          = 'for' , '(' , identifier , 'in' , Expression , ')' , Block ;
74 FunStmt          = 'fun' , identifier , ArgumentNames , Block ;
75 BreakStmt        = 'break' ;
76 ContinueStmt     = 'continue' ;
77 ReturnStmt       = 'return' , [ Expression ] ;
78
79 (* other nodes *)
80 Statements   = { eol } , [ Statement , { eol , { eol } , Statement } ] , { eol } ;
81 Block        = { eol } , ( '{' , Statements , '}' | Statement ) ;
82 Program      = Statements ;
```

Abschlusserklärung

Abschließend bedanke ich mich noch bei allen Personen, die mich bei der Anfertigung dieser Facharbeit unterstützt haben. Der größte Dank geht dabei an meine Informatik-Fachlehrerin Frau Sonja Sokolović, die die gesamte Zeit über stetig behilflich war und diese Facharbeit mehrfach auf den Inhalt, die Struktur und die Darstellung geprüft hat. Außerdem bedanke ich mich bei Mik Müller für das Kreieren des Logos von *Roost* auf der Titelseite. Zuletzt gilt mein Dank noch meinen Eltern, meinem Bruder und erneut Mik Müller für das Prüfen der sprachlichen Richtigkeit und Darstellung.

Hiermit versichere ich, dass ich diese Arbeit selbstständig angefertigt, keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder dem Inhalt nach aus anderen Werken entnommen wurden, in jedem einzelnen Fall mit genauer Quellenangabe kenntlich gemacht habe. Verwendete Informationen aus dem Internet sind der Arbeit als Ausdruck im Anhang beigefügt. Ich bin damit einverstanden, dass die von mir verfasste Facharbeit der schulinternen Öffentlichkeit in der Bibliothek der Schule zugänglich gemacht wird.

25. Februar 2022, Wuppertal — Silas Groh

Ort, Datum — Unterschrift