Formal Languages and Their Applications

An Introduction via ANTLR and PLY

— Autumn 2021 —

Prof. Dr. Karl Stroetmann

November 13, 2021

These lecture notes, their LaTeX sources, and the programs discussed in these lecture notes are all freely available at

https://github.com/karlstroetmann/Formal-Languages.

As computer science is a very active field, these lecture notes might change from time to time. Provided the program `git` is installed on your computer, the repository containing the lecture notes can be cloned using the command

```
git clone https://github.com/karlstroetmann/Formal-Languages.git
```

.

Once you have cloned the repository, the command

```
git pull
```

can be used to load the current version of these lecture notes from `github`.

As I am frequently updating these lecture notes, there is always the chance that these notes contain typos or even errors. If you spot an error, it would be nice if you could either send a pull request or contact me via email at karl.stroetmann@dhbw-mannheim.de.

Contents

Chapter 1

Introduction and Motivation

This lecture covers both the theory of formal languages as well as some of their applications. In particular, we discuss the construction of scanners, parsers, interpreters, and compilers. Furthermore, we present a number of tools that can be used to build scanners and parsers. In particular, the following tools will be introduced:

1. PLY generates a parser for *Python* programs.
2. ANTLR can generate parsers for various programming languages. In particular, ANTLR can be used to generate parsers for both *Python* and *Java*.

All of these tools are program generators, i.e. they take as input the description of a language and produce a parser as output.

1.1 Basic Definitions

The central notion of this lecture is the notion of a *formal language*, which basically is a set of strings that is defined in some precise mathematical way. In order to be able to define this notion we require some definitions.

Definition 1 (Alphabet) An alphabet Σ is a finite, non-empty set of characters:

$$\Sigma = \{c_1, \cdots, c_n\}.$$

Sometimes, we use the term symbol to denote a character. □

Examples:

(a) $\Sigma = \{0, 1\}$ is an alphabet that can be used to represent binary numbers.

(b) $\Sigma = \{\texttt{a}, \cdots, \texttt{z}, \texttt{A}, \cdots, \texttt{Z}\}$ is the alphabet used for the English language.

(c) The set $\Sigma_{\text{ASCII}} = \{0, 1, \cdots, 127\}$ is known as the ASCII-Alphabet. The numbers are interpreted as letters, digits, punctuation symbols, and control characters. For example, the numbers in the set $\{65, \cdots, 90\}$ represent the letters $\{\texttt{A}, \cdots, \texttt{Z}\}$. ◇

Definition 2 (Strings) Given an alphabet Σ, a string is a list of characters from Σ. In the theory of formal languages, these lists are written without the bracket symbols and without separating commas. Hence if $c_1, \cdots, c_n \in \Sigma$, then we write

$$w = c_1 \cdots c_n \quad \text{instead of} \quad w = [c_1, \cdots, c_n].$$

The empty string is denoted as ε, i.e. we have $\varepsilon = \texttt{""}$. The set of all strings that can be constructed from the alphabet Σ is written as Σ^*. For emphasis, strings are often surrounded by quotation marks. □

Examples:

1. Assume that $\Sigma = \{0, 1\}$. If we define

   $$w_1 := \text{"01110"} \quad \text{and} \quad w_2 := \text{"11001"},$$

   then both $w_1$and w_2 are strings. Therefore we have

   $$w_1 \in \Sigma^* \quad \text{and} \quad w_2 \in \Sigma^*.$$

2. Assume that $\Sigma = \{\mathtt{a}, \cdots, \mathtt{z}\}$. If we define

   $$w := \text{"}\mathtt{example}\text{"},$$

   then we have $w \in \Sigma^*$. ◇

The *length* of a string w is defined as the number of characters composing w. The length of w is written as $|w|$. We use array notation to extract the characters from a string: Given a string w and a natural number $i \leq |w|$, we agree that $w[i]$ denotes the i-th character of the string w. We start to count the characters at 0 as this is the convention used in many many modern programming languages like C, *Java*, and *Python*.

Next, we define the concatenation of two strings w_1 and w_2 as the string w that results from appending the string w_2 at the end of w_1. The concatenation of w_1 and w_2 is written as $w_1 \cdot w_2$ or sometimes even shorter as $w_1 w_2$.

Example: If $\Sigma = \{0, 1\}$ and, furthermore, $w_1 = \text{"01"}$ and $w_2 = \text{"10"}$, then we have

$$w_1 \cdot w_2 = \text{"0110"} \quad \text{and} \quad w_2 \cdot w_1 = \text{"1001"}.$$ ◇

Definition 3 (Formal Language)
If Σ is an alphabet, then a *precisely defined* subset $L \subseteq \Sigma^*$ is called a formal language. □

At this point, your first reaction might be to ask what exactly constitutes a *precisely defined* subset. The idea is to have some kind of unambiguous definition of the subset L that makes up the language. We need to have an unambiguous definition that specifies whether a given string is indeed part of the defined language. To give one negative example, a language like English is not a formal language as their is no precise definition of what constitutes a valid sentence of the English language.

The previous definition is very general. As the lecture proceeds, we will define several specializations of this concept. For us, the two most important specializations are regular languages and context-free languages, because these two categories are the most important in computer science.

Examples:

1. Assume that $\Sigma = \{0, 1\}$. Define

   $$L_{\mathbb{N}} = \{1 \cdot w \mid w \in \Sigma^*\} \cup \{0\}$$

   Then $L_{\mathbb{N}}$ is the language consisting of all strings that can be interpreted as natural numbers given in binary notation. The language contains all strings from Σ^* that start with the character 1 as well as the string 0, which only contains the character 0. For example, we have

   $$\text{"100"} \in L_{\mathbb{N}}, \quad \text{but} \quad 010 \notin L_{\mathbb{N}}.$$

   Let us define a function

   $$\mathit{value} : L_{\mathbb{N}} \to \mathbb{N}$$

   on the set $L_{\mathbb{N}}$. We define $\mathit{value}(w)$ by induction on the length of w. We call $\mathit{value}(w)$ the interpretation of w. The idea is that $\mathit{value}(w)$ computes the number represented by the string w:

   (a) $\mathit{value}(0) = 0$, $\mathit{value}(1) = 1$,

   (b) $|w| > 0 \to \mathit{value}(w0) = 2 \cdot \mathit{value}(w)$,

   (c) $|w| > 0 \to \mathit{value}(w1) = 2 \cdot \mathit{value}(w) + 1$.

2. Again we have $\Sigma = \{0, 1\}$. Define the language $L_{\mathbb{P}}$ to be the set of all strings from $L_{\mathbb{N}}$ that are prime numbers:

   $$L_{\mathbb{P}} := \{w \in L_{\mathbb{N}} \mid \mathit{value}(w) \in \mathbb{P}\}$$

   Here, $\mathbb{P}$ denotes the set of prime numbers, which is the set of all natural numbers p bigger than 1 that have no divisor other than 1 or p:

   $$\mathbb{P} = \Big\{p \in \mathbb{N} \;\Big|\; \{t \in \mathbb{N} \mid \exists k \in \mathbb{N} : k \cdot t = p\} = \{1, p\}\Big\}.$$

3. Define $\Sigma_{\text{ASCII}} = \{0, \cdots, 127\}$. Furthermore, define L_C as the set of all strings of the form

   $$\texttt{char*}\ f\texttt{(char*}\ x\texttt{)}\ \{\ \cdots\ \}$$

   that are, furthermore, valid `C` function definitions. Therefore, L_C contains all those strings that can be interpreted as a `C` function f such that f takes a single argument which is a string and returns a value which is also a string.

4. Define $\Sigma := \Sigma_{\text{ASCII}} \cup \{\dagger\}$, where † is some new symbol that is different from all symbols in Σ_{ASCII}. The universal language L_u is the set of all strings of the form

   $$p\dagger x\dagger y$$

   such that

   (a) $p \in L_C$,
   (b) $x, y \in \Sigma^*_{\text{ASCII}}$,
   (c) if f is the function that is defined by p, then $f(x)$ yields the result y. ◇

The examples given above demonstrate that the notion of a formal language is very broad. While it is easy to recognize the strings of the language $L_{\mathbb{N}}$, it is quite a bit more difficult to decide whether a string is a member of $L_{\mathbb{P}}$ or L_C. Finally, since the halting problem is undecidable, there can be no algorithm that is able to decide whether a string w is an element of the language L_u. However, this language is still semi-decidable: If there is a string w such that $w \in L_u$, then we are able to prove this.

1.2 Overview

My goal in this lecture is to cover the following topics. In order to describe these topics, I will need to use some notions that I cannot define precisely at this point. Don't worry if you don't yet understand these notions, as they will be explained once we get there. Basically, this lecture comes in three parts.

1. In the first part we discuss the theory of regular languages.
   (a) We will start with *regular expressions*. After a formal definition of this notion, we discuss how regular expressions are used in *Python*.
   (b) Next, we show how the tool PLY can be used to generate scanners.
   (c) Then, we turn to the implementation of regular expressions via finite state machines.
   (d) We show how the equivalence of regular expressions can be checked.
   (e) We finish our discussion of regular languages by discussing their limits. In particular, we discuss the Pumping Lemma.
2. In the second part of this lecture we discuss context free languages. Context free languages are a formalism to describe the syntax of programming languages. In particular, the theory of regular languages can be used to implement parsers.
   (a) We discuss the definition of context free grammars. These are used to specify context free languages.
   (b) We discuss the parser generators ANTLR and PLY.
   (c) We present the theory that is necessary to understand the parser generator PLY.
   (d) We proceed to discuss the limits of context free languages.
3. In the last part of this lecture we discuss interpreters and compilers.

1.3 Literature

In addition to these lecture notes there are three books that I would like to recommend:

(a) *Introduction to Automata Theory, Languages, and Computation* [HMU06]

This book is the bible with respect to the theory of formal languages and it contains all the theoretical results discussed in this lecture. Obviously, we will only be able to cover a small part of the results discussed in this book.

(b) *Introduction to the Theory of Computation* [Sip12]

This is another readable introduction to the theory of formal languages. It also discusses the theory of computability, which is not covered in this lecture.

(c) *Compilers — Principles, Techniques and Tools* [ASUL06]

This book is one of the standard references with respect to the theory of compilers. It also covers a fair amount of the theory of formal languages.

1.4 Check your Understanding

(a) Define the notion of an alphabet.

(b) Given an alphabet, define the notion of a string.

(c) Define the notion of a formal language.

Chapter 2

Regular Expressions

Regular expressions are terms that specify those formal languages that are simple enough to be recognized by a so called finite state machine. The concept of finite state machines will be discussed in chapter 4. A regular expression is able to specify

1. the choice between different alternatives,
2. concatenation, and
3. repetition.

Many modern scripting languages are based on regular expression, for example the initial popularity of the programming language *Perl* was largely due to its efficiency in dealing with regular expressions. Today, all modern high-level languages, e.g. *Python*, *Java*, C#, and many others provide extensive libraries to support regular expressions. Furthermore, there are a number of UNIX tools like `grep`, `sed` or `awk` that are based on regular expressions. Hence, every aspiring computer scientist needs to be comfortable with regular expressions. In this chapter we will give a definition of regular expressions that is quite concise and is different from the definition given in most programming languages. The advantage of this concise definition is that it is more convenient for our theoretical analysis of regular languages, which is given in Chapter 4 and Chapter 5. In the next chapter we will present the syntax of regular expressions that is used in *Python*.

2.1 Preliminary Definitions

Before we can define the syntax and semantics of regular expressions, we need some auxiliary definitions.

Definition 4 (Product of Languages) If Σ is an alphabet and $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are formal languages, the product of L_1 and L_2 is written as $L_1 \cdot L_2$ and is defined as the set of all concatenations $w_1 \cdot w_2$ such that $w_1 \in L_1$ and $w_2 \in L_2$, i.e. we have

$$L_1 \cdot L_2 := \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}. \qquad \diamond$$

Example: If $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and L_1 and L_2 are defined as

$$L_1 = \{\mathtt{ab}, \mathtt{bc}\} \quad \text{and} \quad L_2 = \{\mathtt{ac}, \mathtt{cb}\},$$

then the product of L_1 and L_2 is given as

$$L_1 \cdot L_2 = \{\mathtt{abac}, \mathtt{abcb}, \mathtt{bcac}, \mathtt{bccb}\}. \qquad \diamond$$

Definition 5 (Power of a Language) Assume Σ is an alphabet, $L \subseteq \Sigma^*$ is a formal language and $n \in \mathbb{N}$. The n-th power of L is written as L^n and is defined by induction on n.

B.C.: $n = 0$:

$L^0 := \{\varepsilon\}$.

Here ε denotes the empty string, i.e. we have $\varepsilon =$ "".

I.S.: $n \mapsto n + 1$:

$L^{n+1} = L^n \cdot L$ ◇

Example: If $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $L = \{\mathtt{ab}, \mathtt{ba}\}$, we have

(a) $L^0 = \{\varepsilon\}$,

(b) $L^1 = \{\varepsilon\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{ab}, \mathtt{ba}\} = L$,

(c) $L^2 = \{\mathtt{ab}, \mathtt{ba}\} \cdot \{\mathtt{ab}, \mathtt{ba}\} = \{\mathtt{abab}, \mathtt{abba}, \mathtt{baab}, \mathtt{baba}\}$. ◇

Definition 6 (Kleene Closure) Assume that Σ is an Alphabet and $L \subseteq \Sigma^*$ is some formal language. Then the Kleene closure of L is written as L^* and is defined to be the union of all powers L^n for all $n \in \mathbb{N}$:

$$L^* := \bigcup_{n \in \mathbb{N}} L^n = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \cdots.$$

Note that $\varepsilon \in L^*$. Therefore, L^* is never the empty set, not even if $L = \{\}$. □

Example: Assume $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $L = \{\mathtt{a}\}$. Then we have

$$L^* = \{\mathtt{a}^n \mid n \in \mathbb{N}\}.$$

Here $\mathtt{a}^n$ is the string of length n that contains only the letter a. Hence, we have

$$\mathtt{a}^n = \underbrace{\mathtt{a} \cdots \mathtt{a}}_{n}.$$ ◇

Formally, given a string s and an non-negative integer Zahl $n \in \mathbb{N}$, we define the expression s^n by induction on n:

B.C.: $n = 0$

$s^0 := \varepsilon$.

I.S.: $n \mapsto n + 1$

$s^{n+1} := s^n \cdot s$, where $s^n \cdot s$ denotes the concatenation of the strings s^n and s. ◇

The previous example shows that the Kleene closure of a finite language can be infinite. It is easy to see that the Kleene closure of a language L is infinite if L contains at least one string s that is different from the empty string ε.

2.2 The Formal Definition of Regular Expressions

We proceed to define the set of regular expressions given an alphabet Σ. This set is denoted as $\mathtt{RegExp}_\Sigma$ and is defined by induction. Simultaneously, we define the function

$$L : \mathtt{RegExp}_\Sigma \to 2^{\Sigma^*},$$

which interprets every regular expression r as a formal language $L(r) \subseteq \Sigma^*$.[1]

Definition 7 (Regular Expressions) The set $\mathtt{RegExp}_\Sigma$ of regular expressions on the alphabet Σ is defined by induction as follows:

[1] Given a set M the power set of M, i.e. the set of all subsets of M, is denoted as 2^M.

1. $\emptyset \in \texttt{RegExp}_\Sigma$

   The regular expression $\emptyset$ denotes the empty language, we have

   $$L(\emptyset) := \{\}.$$

   In order to avoid confusion we assume that the symbol $\emptyset$ is not a member of the alphabet Σ, i.e. we have $\emptyset \notin \Sigma$.

2. $\varepsilon \in \texttt{RegExp}_\Sigma$

   The regular expression ε denotes the language that only contains the empty string ε:

   $$L(\varepsilon) := \{\varepsilon\}.$$

   Observe that in this equation the two occurrences of ε are interpreted differently: The occurrence of ε on the left hand side of this equation denotes a regular expression, while the occurrence of ε on the right hand side denotes the empty string.

   Furthermore, we assume that $\varepsilon \notin \Sigma$.

3. $c \in \Sigma \rightarrow c \in \texttt{RegExp}_\Sigma$.

   Every character from the alphabet Σ is a regular expression. This expression denotes the language that contains only the string c:

   $$L(c) := \{c\}.$$

   Observe that we identify characters with strings of length one.

4. $r_1 \in \texttt{RegExp}_\Sigma \wedge r_2 \in \texttt{RegExp}_\Sigma \rightarrow r_1 + r_2 \in \texttt{RegExp}_\Sigma$

   Starting from two regular expressions r_1 and r_2 we can use the infix operator "$+$" to build a new regular expression. This regular expression denotes the union of the languages described by r_1 and r_2:

   $$L(r_1 + r_2) := L(r_1) \cup L(r_2).$$

   In order to avoid confusion we have to assume that the symbol "+" does not occur in the alphabet Σ, i.e. we have "+" $\notin \Sigma$.

5. $r_1 \in \texttt{RegExp}_\Sigma \wedge r_2 \in \texttt{RegExp}_\Sigma \rightarrow r_1 \cdot r_2 \in \texttt{RegExp}_\Sigma$

   Starting from the regular expression r_1 and r_2 we can use the infix operator "$\cdot$" to build a new regular expression. This regular expression denotes the product of the languages of r_1 and r_2:

   $$L(r_1 \cdot r_2) := L(r_1) \cdot L(r_2).$$

   Again, in order to avoid confusion we have to assume that the symbol "$\cdot$" does not occur in the alphabet Σ, i.e. we have "$\cdot$" $\notin \Sigma$.

6. $r \in \texttt{RegExp}_\Sigma \rightarrow r^* \in \texttt{RegExp}_\Sigma$

   Given a regular expression r, the postfix operator "*" can be used to create a new regular expression. This new regular expression denotes the Kleene closure of the language described by r:

   $$L(r^*) := \big(L(r)\big)^*.$$

   We have to assume that "*" $\notin \Sigma$.

7. $r \in \texttt{RegExp}_\Sigma \rightarrow (r) \in \texttt{RegExp}_\Sigma$

   Regular expressions can be surrounded by parentheses. This does not change the language denoted by the regular expression:

   $$L\big((r)\big) := L(r).$$

   We have to assume that the parentheses "(" and ")" do not occur in the alphabet Σ, i.e. we have "(" $\notin \Sigma$ and ")" $\notin \Sigma$. $\diamond$

Given the preceding definition it is not clear whether the regular expression

$$\mathtt{a} + \mathtt{b} \cdot \mathtt{c}$$

is to be interpreted as

$$(\mathtt{a} + \mathtt{b}) \cdot \mathtt{c} \quad \text{or} \quad \mathtt{a} + (\mathtt{b} \cdot \mathtt{c}).$$

In order to ensure that regular expressions can be read unambiguously we have to assign operator precedences:

1. The postfix operator "*" has the highest precedence.
2. The precedence of the infix operator "·" is lower than the precedence of "*" but stronger than the precedence of "+".
3. The operator "+" has the lowest precedence.

Using these conventions, the regular expression

$$\mathrm{a} + \mathrm{b} \cdot \mathrm{c}^* \quad \text{is interpreted as} \quad \mathrm{a} + \big(\mathrm{b} \cdot (\mathrm{c}^*)\big).$$

Examples: In the following examples, the alphabet Σ is defined as

$$\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}.$$

1. $r_1 := (\mathtt{a} + \mathtt{b} + \mathtt{c}) \cdot (\mathtt{a} + \mathtt{b} + \mathtt{c})$

   The expression r_1 denotes the set of all strings that have the length 2:

   $$L(r_1) = \big\{w \in \Sigma^* \;\big|\; |w| = 2\big\}.$$

2. $r_2 := (\mathtt{a} + \mathtt{b} + \mathtt{c}) \cdot (\mathtt{a} + \mathtt{b} + \mathtt{c})^*$

   The expression r_2 denotes the set of all strings that have at least the length 1:

   $$L(r_2) = \big\{w \in \Sigma^* \;\big|\; |w| \geq 1\big\}.$$

3. $r_3 := (\mathtt{b} + \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{b} + \mathtt{c})^*$

   The expression r_3 denotes the set of all those strings that have exactly one occurrence of the letter "a". A string containing exactly one "a" is a string that starts with an arbitrary amount of the letters b and c (this is what $(\mathtt{b} + \mathtt{c})^*$ denotes), followed by the letter "a", followed by another substring containing only the letters b and c.

   $$L(r_3) = \Big\{w \in \Sigma^* \;\Big|\; \#\big\{i \in \mathbb{N} \;\big|\; w[i] = \mathtt{a}\big\} = 1\Big\}.$$

   Given a set M, the expression $\#M$ denotes the number of elements of M.

4. $r_4 := (\mathtt{b} + \mathtt{c})^* \cdot \mathtt{a} \cdot (\mathtt{b} + \mathtt{c})^* + (\mathtt{a} + \mathtt{c})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{c})^*$

   The regular expression r_4 denotes the set of all those strings that either contain exactly one occurrence of the letter "a" or exactly one occurrence of the letter "b".

   $$L(r_4) = \Big\{w \in \Sigma^* \;\Big|\; \#\big\{i \in \mathbb{N} \;\big|\; w[i] = \mathtt{a}\big\} = 1\Big\} \cup \Big\{w \in \Sigma^* \;\Big|\; \#\big\{i \in \mathbb{N} \;\big|\; w[i] = \mathtt{b}\big\} = 1\Big\}. \qquad \diamond$$

Remark: The syntax of regular expressions given here is the same as the syntax used in [HMU06]. However, the syntax used for regular expression in programming languages like *Python* is different. We will discuss these differences later. $\diamond$

Exercise 1:

(a) Assume $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Define a regular expression for the language $L \subseteq \Sigma^*$ that consists of those strings that contain at least one occurrence of the letter "a" and one occurrence of the letter "b".

(b) Assume $\Sigma = \{0, 1\}$. Specify a regular expression for the language $L \subseteq \Sigma^*$ that consists of those strings s such that the antepenultimate character is the symbol "1".

(c) Again, we have $\Sigma = \{0, 1\}$. Define a regular expression for the language $L \subseteq \Sigma^*$ containing all those strings that do not contain the substring 110.

Solution: The regular expression r that is sought for can be defined as

$$r = (0 + 1 \cdot 0)^* \cdot 1^*.$$

First, it is quite obvious that the language $L(r)$ does not contain a string w such that w contains the substring 110. This is so because a character 1 that is generated by the part $(0 + 1 \cdot 0)^*$ is immediately followed by a 0. Hence if w contains the substring 110, the first 1 cannot originate from the regular expression $(0 + 1 \cdot 0)^*$. Furthermore, if the first 1 of the substring 110 originates from the regular expression 1^*, then there cannot be a 0 following since the language generated by 1^* contains only ones.

Second, assume that the string w does not contain the substring 110. We have to show that $w \in L(r)$. Now if the character 1 does not occur in the string w, then w is just a bunch of zeros and therefore w can be generated by the regular expression $(0 + 1 \cdot 0)^*$ and hence also by $(0 + 1 \cdot 0)^* \cdot 1^*$. If the string w does contain the character 1, there are two cases.

(a) The first occurrence of 1 is followed by a 0. Then the prefix of w up to and including this 0 is generated by the regular expression $(0 + 1 \cdot 0)^*$. The remaining part of w is shorter and, by induction, can be shown to be generated by $(0 + 1 \cdot 0)^* \cdot 1^*$.

(b) The first occurrence of 1 is followed by another 1. In this case, the rest of w must be made up of ones. Hence, the part of w starting with the first 1 is generated by 1^* and obviously the preceding zeros can all be generated by $(0 + 1 \cdot 0)^*$.

(d) Again, assume $\Sigma = \{0, 1\}$. What is the language L generated by the regular expression

$$(1 + \varepsilon) \cdot (0 \cdot 0^* \cdot 1)^* \cdot 0^*?$$ ◇

Solution: This is the language L such that the strings in L do not contain the substring 11.

2.3 Algebraic Simplification of Regular Expressions

Given two regular expressions r_1 and r_2, we write

$$r_1 \doteq r_2 \quad \text{iff} \quad L(r_1) = L(r_2),$$

i.e. if r_1 and r_2 describe the same language. If the equation $r_1 \doteq r_2$ holds, then we call r_1 and r_2 equivalent. The following algebraic laws apply:

(a) $r_1 + r_2 \doteq r_2 + r_1$

This equation is true because the union operator is commutative for sets:

$$L(r_1 + r_2) = L(r_1) \cup L(r_2) = L(r_2) \cup L(r_1) = L(r_2 + r_1).$$

(b) $(r_1 + r_2) + r_3 \doteq r_1 + (r_2 + r_3)$

This equation is true because the union operator is associative for sets.

(c) $(r_1 \cdot r_2) \cdot r_3 \doteq r_1 \cdot (r_2 \cdot r_3)$

This equation is true because the concatenation of strings is associative, for any strings u, v, and w we have

$$(uv)w = u(vw).$$

This implies

$$\begin{aligned}
L\big((r_1 \cdot r_2) \cdot r_3\big) &= \big\{xw \mid x \in L(r_1 \cdot r_2) \wedge w \in L(r_3)\big\} \\
&= \big\{(uv)w \mid u \in L(r_1) \wedge v \in L(r_2) \wedge w \in L(r_3)\big\} \\
&= \big\{u(vw) \mid u \in L(r_1) \wedge v \in L(r_2) \wedge w \in L(r_3)\big\} \\
&= \big\{uy \mid u \in L(r_1) \wedge y \in L(r_2 \cdot r_3)\big\} \\
&= L\big(r_1 \cdot (r_2 \cdot r_3)\big).
\end{aligned}$$

The following equations are more or less obvious.

(d) $\emptyset \cdot r \doteq r \cdot \emptyset \doteq \emptyset$

(e) $\varepsilon \cdot r \doteq r \cdot \varepsilon \doteq r$

(f) $\emptyset + r \doteq r + \emptyset \doteq r$

(g) $(r_1 + r_2) \cdot r_3 \doteq r_1 \cdot r_3 + r_2 \cdot r_3$

(h) $r_1 \cdot (r_2 + r_3) \doteq r_1 \cdot r_2 + r_1 \cdot r_3$

(i) $r + r \doteq r$, because

$$L(r + r) = L(r) \cup L(r) = L(r).$$

(j) $(r^*)^* \doteq r^*$

We have

$$L(r^*) = \bigcup_{n \in \mathbb{N}} L(r)^n$$

and that implies $L(r) \subseteq L(r^*)$. This holds true if we replace r by r^*. Therefore

$$L(r^*) \subseteq L\big((r^*)^*\big)$$

holds. In order to prove the inclusion

$$L\big((r^*)^*\big) \subseteq L(r^*),$$

we consider the structure of the strings $w \in L\big((r^*)^*\big)$. Because of

$$L\big((r^*)^*\big) = \bigcup_{n \in \mathbb{N}} L(r^*)^n$$

we have $w \in L\big((r^*)^*\big)$ if and only if there is an $n \in \mathbb{N}$ such that there are strings $u_1, \cdots, u_n \in L(r^*)$ satisfying

$$w = u_1 \cdots u_n.$$

Because of $u_i \in L(r^*)$ we find a number $m(i) \in \mathbb{N}$ for every $i \in \{1, \cdots, n\}$ such that for $j = 1, \cdots, m(i)$ there are strings $v_{i,j} \in L(r)$ satisfying

$$u_i = v_{1,i} \cdots v_{m(i),i}.$$

Combining these equations yields

$$w = v_{1,1} \cdots v_{m(1),1} v_{1,2} \cdots v_{m(2),2} \cdots v_{1,n} \cdots v_{m(n),n}.$$

Hence w is a concatenation of strings from the language $L(r)$ and hence we have

$$w \in L\big(r^*\big).$$

This shows the inclusion $L\big((r^*)^*\big) \subseteq L(r^*)$.

(k) $\emptyset^* \doteq \varepsilon$

(l) $\varepsilon^* \doteq \varepsilon$

(m) $r^* \doteq \varepsilon + r^* \cdot r$

(n) $r^* \doteq (\varepsilon + r)^*$

2.4 Check your Understanding

(a) Given a formal language L, what is the definition of L^*?

(b) How is the set $\texttt{RegExp}_\Sigma$ defined?

(c) How is the function $L : \texttt{RegExp}_\Sigma \to 2^{\Sigma^*}$ defined?

(d) Given $r_1, r_2 \in \texttt{RegExp}_\Sigma$, how is the notion $r_1 \doteq r_2$ defined?

Chapter 3

Building Scanners with Ply

After having defined regular expressions we will now get a taste of their power in practice. To this end we discuss the tool PLY, which can generate both *Python scanners* and *Python parsers*. A scanner is a program that splits a given string into a list of *tokens*, where a token is a group of consecutive characters that logically belong together. An example will clarify this. The input for a C-compiler is an ASCII-string that can be interpreted as a valid C program. In order to translate this string into machine language, the C-compiler first groups the different characters into tokens. In the case of a C program, the compiler generates the following tokens:

1. Keywords, a.k.a. reserved words like "`if`", "`while`", "`for`", or "`switch`".
2. Operator symbols like "+", "+=", "<", or "<=".
3. Parentheses like "(", "[", and "{" and the corresponding closing symbols.
4. Constants. The language C distinguishes between three different kinds of constants:
   (a) Numbers, for example the integer "123" or the floating point number "`1.23e2`".
   (b) Strings, which are enclosed in double. For example, "`"hello"`" is a string constant. Note that the character "`"`" is part of the string constant, while the opening and closing quotes surrounding the string constant have been used to separate the string constant from the surrounding text.
   (c) Single letters that are enclosed in single quotes as in "`'a'`".
5. Identifiers that can act as variable names, function names, or type names.
6. Comments, which come in two flavors: Single line comments start with the string "`//`" and extend to the end of the line, while multi line comments start with the string "`/*`" and are ended by "`*/`".
7. So called white space characters. For example the blank character `' '`, the tabulator `'\t'`, the line break `'\n'`, and the carriage return `'\r'` are white space symbols.

   Both white space characters and comments are silently removed by the scanner.

To make things more concrete, Figure 3.1 on page 16 contains a C program that prints the string "`Hello World!`" followed by a newline character. The scanner would transform this program into the following list of tokens:

```
[ "#", "include", "<", "stdio.h", ">", "int", "main", "(", ")", "{",
  "printf", "(", "Hello World!\n", ")", ";" "return", "1", ";", "}"
]
```

Note that the scanner discards the white space characters. They only serve to separate tokens.

Although it is possible to write a scanner manually, it is easier to generate a scanner from a specification with the help of a so called scanner generator. We discuss one such tool in the next section.

```
1  /* Hello World program */
2  #include<stdio.h>
3
4  int main() {
5      printf("Hello World!\n");
6      return 1;
7  }
```

Figure 3.1: A simple C program.

3.1 The Structure of a Ply Scanner Specification

In this section we introduce Python Lex-Yacc which is also known as PLY. As the home page of PLY states, "PLY is an implementation of the lex and yacc parsing tools for *Python*". The tool has been developed by David Beazley. In this section, we only discuss PLY as a scanner generator. In Chapter 10 we will discuss how PLY can be used to generate a parser.

We begin with a simple example. In general, a PLY scanner specification is made up of three parts. Figure 3.2 shows how a scanner is specified that can tokenize arithmetical expressions. This example has been taken from the official PLY documentation.

1. The module `ply.lex` contains the definition of the function `ply.lex.lex()` that is able to generate a scanner. Therefore, this module is imported in line 1.

2. The first part of a scanner specification is the token declaration section. Syntactically, this is just a list containing the names of all tokens. Note that all token names have to start with a capital letter.

   In Figure 3.2 the token declaration section extends from line 3 to line 11.

3. The second part contains the token definitions. There are two kinds of token definitions:

   (a) Immediate token definitions have the following form:

   ```
   t_name = r'regexp'
   ```

   Here *name* has to be one of the names declared in the declaration section and *regexp* is a regular expression using the syntax that is specified in the *Python* `re` module.

   (b) Functional token definitions are syntactically *Python function definitions* and have the following form:

   ```
   def t_name(t):
       r'regexp'
           ⋮
   ```

   Here, the vertical dots " ⋮ " denote any *Python* code, while *name* has to be one of the token names declared in the declaration section and *regexp* is a regular expression.

   The functional token definition shown in line 20–23 takes a token `t` as its argument. This token has the attribute `t.value`, which refers to the string that has been recognized as this token. In this case, this string is a sequence of digits that can be interpreted as a number. In line 22 the function `t_NUMBER` converts this string into a number and stores this number as the attribute `t.value`. Finally, the token `t` itself is returned. This is a typical case where we need a functional token definition since we want to modify the token that is returned.

   In Figure 3.2 the token definitions start in line 13 and end in line 23.

4. The third part deals with the handling of newlines, ignored characters, and scanner errors.

(a) A PLY input file may contain the definition of the function `t_newline`. This function is supposed to deal with newlines contained in the input. The main purpose of this function is to set the counter `t.lexer.lineno`. Every token `t` has the attribute `t.lexer`, which is a reference to the scanner object. In turn, the scanner object has the attribute `lineno`, which is supposed to be an integer containing the number of the line currently scanned. This integer starts at the value 1. Every time a newline is read it should be incremented.

In line 26 the regular expression `r'\+'` matches any positive number of newlines. Hence the counter `lineno` has to be incremented by the length of the string `t.value`.

Note that the function `t_newline` does not return a token.

(b) Line 29 specifies that both blanks and tabs should be ignored by the scanner. Note that the string

"`' \t'`"

is not interpreted as a regular expression but rather as a list of its characters. Furthermore, this is not a raw string and must not be prefixed with the character "`r`", for otherwise the character sequence "`\t`" would not be interpreted as a tab symbol.

(c) The function `t_error` deals with characters that can not be recognized. An error message is printed and the call `t.lexer.skip(1)` discards the character that could not be matched.

5. In line 35 the function `lex.lex` creates the scanner that has been specified.
6. Line 38 shows how data can be fed into this scanner.
7. In order to use this scanner we can just iterate over it as shown in line 40. This iteration scans the input string using the generated scanner and produces the tokens that are recognized by the scanner one by one.

If we run the program shown in Figure 3.2 we get the following output:

```
LexToken(NUMBER,3,1,0)
LexToken(PLUS,'+',1,2)
LexToken(NUMBER,4,1,4)
LexToken(TIMES,'*',1,6)
LexToken(NUMBER,10,1,8)
LexToken(PLUS,'+',1,11)
LexToken(NUMBER,0,1,13)
LexToken(NUMBER,0,1,14)
LexToken(NUMBER,7,1,15)
LexToken(PLUS,'+',1,17)
LexToken(LPAREN,'(',1,19)
LexToken(MINUS,'-',1,20)
LexToken(NUMBER,20,1,21)
LexToken(RPAREN,')',1,23)
LexToken(TIMES,'*',1,25)
LexToken(NUMBER,2,1,27)
```

As we can see the tokens returned by our scanner are objects of class `LexToken`. These objects have four attributes:

1. The first attribute is called `type`. Its value is a string that is the name of one of the declared tokens.
2. The second attribute is called `value`. Normally, this is the string that has been recognized but we are allowed to change this attribute. For example, the function `t_NUMBER` converts the recognized string into an integer value.
3. The third attribute is called `lineno`. This specifies the line number where the token has been found.
4. The last attribute is called `lexpos`. This is a counter that is incremented with every character that is read.

Homework: Install PLY and make sure that the example presented previously works.

```
import ply.lex as lex

tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN'
]

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER(t):
    r'0|[1-9][0-9]*'
    t.value = int(t.value)
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore  = ' \t'

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()

data = '3 + 4 * 10 + 007 + (-20) * 2'
lexer.input(data)

for tok in lexer:
    print(tok)
```

Figure 3.2: A simple scanner Specification for PLY.

3.2 The Syntax of Regular Expressions in Python

In the previous chapter we have defined regular expressions using only a minimal amount of syntax. Using as little syntax as possible is beneficial for our upcoming theoretical investigations of regular expression in the next chapter where we show that regular expressions can be implemented using finite state machines. However, for practical applications it is useful to considerably enrich the syntax of regular expressions that we have seen so far. For this reason, the *Python* module *re* provides a number of abbreviations that enable us to denote complex regular expressions in a more compact form. I have written a short tutorial that introduces the most important features of the regular

expressions defined in the module `re`. As it is best to read this tutorial interactively, this section only contains the reference

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Regexp-Tutorial.ipynb

that points to this tutorial.

3.3 A Complex Example: Evaluating an Exam

This sections presents a more complex example that shows some of the power of PLY. The task at hand is the evaluation of an exam. When I mark an exam I create a file that has a format similar to the example shown in Figure 3.3.

```
1  Class: Algorithms and Complexity
2  Group: TIT09AID
3  MaxPoints = 60
4
5  Exercise:      1. 2. 3. 4. 5. 6.
6  Jim Smith:     9 12 10  6  6  0
7  John Slow[1]:  4  4  2  0  -  -
8  Susi Sorglos:  9 12 12  9  9  6
```

Figure 3.3: Results of an Exam

1. The first line contains the keyword "`Class`", a colon "`:`", and then the name of the lecture.
2. The second line specifies the group that has taken the exam.
3. The third line specifies the number of points that are necessary to obtain the best mark.
4. The fourth line is empty.
5. The fifth line numbers the exercises.
6. After that, there is a table. Every row in this table lists the scores achieved by a student for each of the exercises. The name of each student is at the beginning of each row. The name is followed by a colon and after that there is a list of the scores achieved for each exercise. If an exercise has not been attempted at all, the corresponding column contains a hyphen "–".

I have written a *Jupyter notebook* that is able to evaluate data of this kind. You can find the notebook here:

https://github.com/karlstroetmann/Formal-Languages/blob/master/Ply/Exam-Evaluation.ipynb

3.4 Scanner States

Sometimes, regular expressions are not quite enough and it is beneficial for the scanner to have different states. The following example illustrates this. We will develop a program that is able to convert an HTML file into a pure text file. This program is actually quite useful: Some years ago I had a student that was blind. If he read a web page, he would use his Braille display. For him, the HTML markup was of no use so if the markup was removed, he could read web pages faster. In order to develop the program to remove HTML tags, we have to use scanner states. The idea behind scanner states is that the scanner can use different regular expressions for different parts of the input. For example, the header of an HTML file, i.e. the part that is between the `<head>` and `</head>` tags, can just be

[1] You know nothing, John Slow.

skipped. On the other hand, the text inside the <body> and </body> tags needs to be echoed after any remaining tags have been removed. The easiest way to achieve this is by using scanner states and switching between them. The following notebook shows how this can be done:

https://github.com/karlstroetmann/Formal-Languages/blob/master/Ply/Html2Text.ipynb

Exercise 2: The purpose of the following exercise is to transform LaTeX into MathML. LaTeX is a document markup language that is especially well suited to present text that contains mathematical formulæ. In fact, these lecture notes have all been typeset using LaTeX. MathML is the part of Html that deals with the representation of mathematical formulæ. As LaTeX provides a very rich document markup language and we can only afford to spend a few hours on this exercise, we confine ourselves to a small subset of LaTeX. Figure 3.4 on page 20 shows the example input file that we want to transform in Html. If this example file is typeset using LaTeX, it is displayed as shown in Figure 3.5 on page 20. The program that you are going to develop should transform the LaTeX input file into an Html file. For your convenience, all these files are available in the github directory

Exercises/LaTeX2HTML.

This directory contains also the *Jupyter* notebook LaTeX2HTML.ipynb. This notebook contains lots of predefined functions that are useful in order to solve the given task.

```
\documentclass{article}
\begin{document}
The sum of the squares of the first $n$ natural numbers is given as:
$$ \sum\limits_{i=1}^{n} i^{2} = \frac{1}{6} \cdot n \cdot (n+1) \cdot (2\cdot n + 1). $$
According to Pythagoras, the length of the hypotenuse of a right triangle is
the square root of the squares of the length of the two catheti:
$$ c = \sqrt{a^{2} + b^{2}}.  $$
The area of a circle is given as
$$  A = \pi \cdot r^{2},   $$
while its circumference satisfies
$$ C = 2 \cdot \pi \cdot r.  $$
\end{document}
```

Figure 3.4: An example LaTeX input file.

The sum of the squares of the first n natural numbers is given as:

$$\sum_{i=1}^{n} i^2 = \frac{1}{6} \cdot n \cdot (n+1) \cdot (2 \cdot n + 1).$$

According to Pythagoras, the length of the hypotenuse of a right triangle is the square root of the squares of the length of the two catheti:

$$c = \sqrt{a^2 + b^2}.$$

The area A of a circle is given as

$$A = \pi \cdot r^2,$$

while its circumference satisfies

$$C = 2 \cdot \pi \cdot r.$$

Figure 3.5: Output produced by the LaTeX file shown in Figure 3.4

In order to do this exercise, you have to understand a little bit about LaTeX and about MathML. In the following,

we discuss those features of these two language that are needed in order to solve the given problem.

1. A LaTeX input file has the following structure:
   (a) The first line list the type of the document. In our example, it reads
   `\documentclass{article}`.
   This line will be transformed into the following HTML:
   ```
   <html>
   <head>
   <script type="text/javascript"
   src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML">
   </script>
   ```
   Here, the `<script>` tag is necessary in order for the MathML to be displayed correctly.
   (b) The next line has the form:
   `\begin{document}`
   This line precedes the content and should be translated into the tag
   `<body>`.
   (c) After that, the LaTeX file consists of text that contains mathematical formula.
   (d) The LaTeX input file finishes with a line of the form
   `\end{document}`.
   This line should be translated into the tags
   `</body></html>`.
2. In LaTeX, an inline formula is started and ended with a single dollar symbol "$". In MathML, an inline formula is written as
   `<math xmlns="http://www.w3.org/1998/Math/MathML" display='inline'>···</math>`.
   Here, I have used "···" to represent the mathematical content of the formula.
3. In LaTeX, a formula that is displayed in its own line is started and ended with the string "$$". In MathML, these formulæ are called block formulæ and are written as
   `<math xmlns="http://www.w3.org/1998/Math/MathML" display='block'>···</math>`.
   Again, I have used "···" to represent the mathematical content of the formula.
4. While in LaTeX a mathematical variable does not need any special markup, in MathML a mathematical variable is written using the tags `<mi>` and `</mi>`. For example, the mathematical variable n is written as
   `<mi>n</mi>`.
5. While in LaTeX a number does not need any special markup, in MathML a number is written using the tags `<mn>` and `</mn>`. For example, the number 3.14149 is written as
   `<mn>3.14159</mn>`.
6. In LaTeX the mathematical constant π is written using the command "\pi". In MathML, we have to make use of the HTML entity "π" and hence we would write π as
   `<mn>&pi;</mn>`.

7. In LaTeX the multiplication operator "$\cdot$" is written using the command "\cdot". In MathML, we have to make use of the Html entity "⋅" and hence we would write "$\cdot$" as

   <mop>⋅</mop>.

8. While in LaTeX most operator symbols stand for themselves, in MathML an operator is surrounded by the tags <mop> and </mop>. For example, the operator $+$ is written as

   <mop>+</mop>.

9. In LaTeX, raising an expression e to the nth power is done using the operator "^". Furthermore, the exponent should be enclosed in the curly braces "{" and "}". For example, the code to produce the term x^2 is

   x^{2}.

   In MathML, raising an expression to a power is achieved using the tags <msup> and </msup>. For example, in order to display the term x^2, we have to write

   <msup><mi>x</mi><mn>2</mn></msup>.

10. In LaTeX, taking the square root of an expression is done using the command "\sqrt". The argument has to be enclosed in curly braces. For example, in order to produce the output $\sqrt{a+b}$, we have to write

    \sqrt{a+b}.

    In MathML, taking the square root makes use of the tags <msqrt> and </msqrt>. The example shown above can be written as

    <msqrt><mi>a</mi><mop>+</mop><mi>b</mi></msqrt>.

11. In LaTeX, writing a fraction is done using the command "\frac". This command takes two arguments, the numerator and the denominator. Both of these have to be enclosed in curly braces. For example, in order to produce the output $\frac{a+b}{2}$, we have to write

    \frac{a+b}{2}.

    In MathML, a fraction is created via the tags <mfrac> and </mfrac>. Additionally, if the arguments contain more than a single element, each of them has to be enclosed in the tags <mrow> and </mrow>. The example shown above can be written as

    <mfrac><mrow><mi>a</mi><mop>+</mop><mi>b</mi></mrow><mn>2</mn></mfrac>.

12. In LaTeX, writing a sum is done using the command "\sum\limits". This command takes two arguments: The first argument gives the indexing variable together with its lower bound, while the second argument gives the upper bound. The first argument is started using the string "_{" and ended using the string "}", while the second argument is started using the string "^{" and ended using the string "}". For example, in order to produce the output

    $$\sum_{i=1}^{n} i,$$

    we have to write

    \sum\limits_{i=1}^{n} i.

    In MathML, a sum with lower and upper limits is created via the tags <munderover> and </munderover> and the Html entity "&sum". The tag munderover takes three arguments:

    (a) The first argument is the operator, so in this case it is the entity "&sum".
    (b) The second argument initializes the indexing variable of the sum.
    (c) The third argument provides the upper bound.

    The second argument usually contains more than a single item and therefore has to be enclosed in the tags <mrow> and </mrow>. Hence, the example shown above would be written as follows:

```
<munderover>
    <mo>&sum;</mo>
    <mrow>
        <mi>i</mi> <mo>=</mo> <mn>1</mn>
    </mrow>
    <mi>n</mi>
</munderover>
```

Remark: The most important problem that you have to solve is the following: Once you encounter a closing brace "}" you have to know whether this brace closes the argument of a square root, a fraction, a sum, or an exponent. You should be aware that, for example, square roots and fractions can be nested. Hence, it is not enough to have a single variable that remembers whether you are parsing, say, a square root or a fraction. Instead, every time you encounter a string like, e.g.

`\sqrt{` or `\frac{`,

you should store the current state on a stack and set the new state according to whether you have just seen the keyword "`\frac`" or "`\sqrt`" or whatever caused the curly brace to be opened. When you encounter a closing brace "}", you should restore the state to its previous value by looking up this value from the stack. ◇

Chapter 4

Finite State Machines

In the previous chapter we have seen how to generate a scanner using PLY. In this chapter we learn how regular expressions can be implemented using finite state machines, abbreviated as FSMs. There are two kinds of FSMs: The deterministic ones and non-deterministic ones. Although non-deterministic FSMs seem to be more powerful than deterministic FSMs, we will see that every non-deterministic FSM can be transformed into an equivalent deterministic FSM. After proving this result, we show how a regular expression can be translated into an equivalent non-deterministic FSM. Finally, we show that the language recognized by any FSM can be described by an equivalent regular expression. Therefore, the central result of this chapter is the equivalence of finite state machines and regular expressions. Hence, the results proved in this chapter are as follows:

1. The language described by a regular expression can be defined by a non-deterministic FSM.
2. Every non-deterministic FSM can be transformed into an equivalent deterministic FSM.
3. For every deterministic FSM there is an regular expression specifying the language recognized by the FSM.

4.1 Deterministic Finite State Machines

The FSMs that we are going to discuss in this chapter are used to read a string and to decide whether this string is an element of a given language. Hence, the input of these FSMs is a string, while the output is either the value `True` or the value `False`. The name giving feature of an FSM is the fact that an FSM only has a finite number of possible states. Reading a character causes the FSM to change its state. An FSM accepts its input if it has reached a so called accepting state after reading all characters of the input string. Let me explain this idea more precisely:

1. Initialy, the FSM is in a state that is know as the start state.
2. In every step of its computation, the FSM reads one character of the input string s. Every time a character is processed, the state of the FSM might change.
3. Some states of the FSM are designated as accepting states. If the FSM has consumed all characters of the given input string s and the FSM has reached an accepting state, then the input string s is accepted and the FSM returns `True`. Otherwise the FSM returns `False` and the string s is rejected.

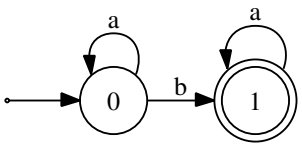

Figure 4.1: An FSM to recognize the language $L(\mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*)$.

Finite state machines are best presented graphically. Figure 4.1 depicts a simple FSM that recognizes those strings that are specified by the regular expression

$$\mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*.$$

This FSM has but two states. These states are called 0 and 1.

1. State 0 is the start state. In Figure 4.1, the start state is indicated by an arrow comming from nowhere that points to it.

   If the FSM is in the state 0 and reads the character "a", then the FSM stays in the state 0. This is specified in the figure by an arrow labeled with the character "a" that both starts and ends in the state 0. On the other hand, if the character "b" is read while the FSM is in state 0, then the FSM switches into the state 1. This is depicted by an arrow labeled with the character "b" that originates from the state 0 and points to the state 1.

2. State 1 is an accepting state. In Figure 4.1 this is specified by the fact that the state 1 is decorated by a double circle.

   If the character "a" is read while the FSM is in state 1, then the FSM does not change its state. On the other hand, if the FSM reads the character "b" while in state 1, then the next state is undefined since there is no arrow originating from state 1 that is labeled with the character "b".

   In general, a FSM dies if it reads a character c in a state s such that there is no transition from s when c is read. In this case, the FSM returns the value `False` to signal that it does not accept the given input string.

Formally, a *finite state machine* is defined as a 5-tuple.

Definition 8 (Fsm) A finite state machine (abbreviated as FSM) is a 5-tuple

$$F = \langle Q, \Sigma, \delta, q_0, A \rangle$$

where the components Q, Σ, δ, q_0, and A have the following properties:

1. Q is the finite set of states.
2. Σ is the input alphabet. Therefore, Σ is a set of characters and the strings read by the FSM F are strings from the set Σ^*.
3. $\delta : Q \times \Sigma \to Q \cup \{\Omega\}$

   is the transition function. For every state $q \in Q$ and for all characters $c \in \Sigma$ the expression $\delta(q, c)$ computes the new state of the FSM F that is reached if F reads the character c while in state q. If $\delta(q, c) = \Omega$, then F dies when it is in state q and the next character is c.

   In the figures depicting FSMs transitions of the form $\delta(q, c) = \Omega$ are not shown.
4. $q_0 \in Q$ is the start state.
5. $A \subseteq Q$ is the set of accepting states. □

Example: The FSM that is shown in Figure 4.1 is formally defined as follows:

$$F = \langle Q, \Sigma, \delta, q_0, A \rangle,$$

where we have:

1. $Q = \{0, 1\}$,
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
3. $\delta = \big\{\langle 0, a\rangle \mapsto 0, \langle 0, b\rangle \mapsto 1, \langle 1, a\rangle \mapsto 1, \langle 1, b\rangle \mapsto \Omega\big\}$,
4. $q_0 = 0$,
5. $A = \{1\}$.

In order to formally define the language $L(F)$ that is accepted by an FSM F we generalize the transition function δ to a new function

$$\delta^* : Q \times \Sigma^* \to Q \cup \{\Omega\}$$

that, instead of a single character, accepts a string as its second argument. The definition of $\delta^*(q, w)$ is given by induction on the string w.

I.A. $w = \varepsilon$: We define

$$\delta^*(q, \varepsilon) := q,$$

because if a deterministic FSM does not read any character, it cannot change its state.

I.S. $w = cv$ where $c \in \Sigma$ and $v \in \Sigma^*$: We define

$$\delta^*(q, cv) := \begin{cases} \delta^*\big(\delta(q, c), v\big) & \text{provided } \delta(q, c) \neq \Omega; \\ \Omega & \text{otherwise.} \end{cases}$$

If F reads the string cv, it first reads the character c. Now if this causes F to change into the state $\delta(q, c)$, then F has to read the string v in the state $\delta(q, c)$. However, if $\delta(q, c)$ is undefined, then $\delta^*(q, cv)$ is undefined too.

Definition 9 (Accepted Language, $L(F)$) For an FSM $F = \langle Q, \Sigma, \delta, q_0, A\rangle$ the language accepted by F is called $L(F)$ and is defined as

$$L(F) := \big\{s \in \Sigma^* \mid \delta^*(q_0, s) \in A\big\}.$$

Hence, the accepted language of F is the set of all those strings that take F from its start state into an accepting state. ◇

Exercise 3: Specify an FSM F such that $L(F)$ is the set of all those strings $s \in \{a, b\}^*$, such that s contains the substring "aba". ◇

Complete Finite State Machines Occasionally it is beneficial for an FSM F to be complete: An FSM

$$F = \langle Q, \Sigma, \delta, q_0, A\rangle,$$

is complete if the transition function δ never returns the undefined value Ω, i.e. we have

$$\delta(q, c) \neq \Omega \quad \text{for all } q \in Q \text{ and } c \in \Sigma.$$

Proposition 10 For every FSM F there exists a complete FSM $\widehat{F}$ that accepts the same language as the FSM F, i.e. we have:

$$L(\widehat{F}) = L(F).$$

Proof: Assume F is given as

$$F = \langle Q, \Sigma, \delta, q_0, A\rangle.$$

The idea is to define $\widehat{F}$ by adding a new state to the set of states Q. This new state is called the dead state. If there is no next state for a given state $q \in Q$ when a character c is processed, i.e. if we have

$$\delta(q, c) = \Omega,$$

then F changes into the dead state. Once F has reached a dead state, it will never leave this state.

The formal definition of the FSM $\widehat{F}$ is done as follows: We introduce a new state ☠ which serves as the dead state. The only requirement is that ☠ $\notin Q$.

1. $\widehat{Q} := Q \cup \{☠\}$,

   the dead state is added to the set Q.

2. $\widehat{\delta} : \widehat{Q} \times \Sigma \to \widehat{Q}$,

   where the function $\widehat{\delta}$ is defined as follows:

   (a) $\delta(q, c) \neq \Omega \to \widehat{\delta}(q, c) = \delta(q, c)$ for all $q \in Q$ and $c \in \Sigma$.

   If the state transition function is defined for the state q and the character c, then $\widehat{\delta}(q, c)$ is the same as $\delta(q, c)$.

   (b) $\delta(q, c) = \Omega \to \widehat{\delta}(q, c) = ☠$ for all $q \in Q$ and $c \in \Sigma$.

   If the state transition function δ is undefined for the state q and the character c, then $\widehat{\delta}(q, c)$ returns the dead state ☠.

   (c) $\widehat{\delta}(☠, c) = ☠$ for all $c \in \Sigma$,

   because there is no escape from death[1].

Hence the FSM $\widehat{F}$ is given as follows:

$$\widehat{F} = \langle \widehat{Q}, \Sigma, \widehat{\delta}, q_0, A \rangle.$$

If F reads a string s without reaching an undefined state, then the behavior of F and $\widehat{F}$ is the same. However, if F reaches an undefined state, then $\widehat{F}$ instead switches into the dead state ☠ and remains in this state regardless of the rest of the input string. As the dead state ☠ is not an accepting state, the languages accepted by F and $\widehat{F}$ are identical. □

Exercise 4: Define an FSM that accepts the language specified by the regular expression

$$r := (\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b}).$$ ◇

Solution: The regular expression r specifies those strings s from the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ such that the antepenultimate character of s is the character "b". In order to recognize this fact, the FSM has to remember the last three characters. As there are eight different possible combinations for the last three characters, the FSM needs to have eight states. Let us number these states 0, 1, 2, $\cdots$, 7. We describe the purpose of these states in the following:

State 0: In this state, the character "b" has not yet been seen. Depending on how many characters have been read, there are four cases:

(a) At least three characters have been read. In this case, the last three characters are "aaa".

(b) Two characters have been read. In this case, the string that has been read so far is the string "aa".

(c) Only one character has been read so far. In this case, the string that has been is "a".

(d) Nothing has yet been read and therefore the string that has been read is ε.

For the remaining states we list the last three characters that have been read without further comment.

State 1: "aab".

This case also covers the cases where the strings "ab" and "b" have been read.

State 2: "aba".

This case also cover the case where the string "ba" has been read.

State 3: "abb".

This case also cover the case where the string "bb" has been read.

State 4: "bab".

State 5: "bba".

State 6: "bbb".

[1] Or, as the disciples of the Drowned Good say: "What is dead may never die".

State 7: "baa".

Obviously, the states 4, 5, 6 and 7 are the accepting states because here the antepenultimate character is the character "b". Next, we construct the transition function δ.

0. First, let us consider the state 0. If the last three characters that have been read are "aaa" and if we read the character "a" next, then the last three characters read will again be "aaa". Hence, we must have

   $$\delta(0, \mathtt{a}) = 0.$$

   However, if instead we read the character "b" in state 0, then the last three characters that have been read are "aab", which is exactly the last three characters that have been read in state 1. Hence we have

   $$\delta(0, \mathtt{b}) = 1.$$

1. Next we consider state 1. If the last three characters are "aab" and we read the character "a" next, then the last three characters are "aba". This corresponds to the state 2. Therefore, we must have

   $$\delta(1, \mathtt{a}) = 2.$$

   If instead we read the character "b" while in state 1, then the last three characters will be "abb", which corresponds to the state number 3. Hence we have

   $$\delta(1, \mathtt{b}) = 3.$$

The remaining transitions are found in a similar way. Figure 4.2 on page 28 shows the resulting FSM. We still have to explain how we have chosen the start state. When the computation starts, the finite state machine has not read any character. In particular, this implies that neither of the last three characters is the character "b". Hence we can use the state 0 as the start state of our FSM.

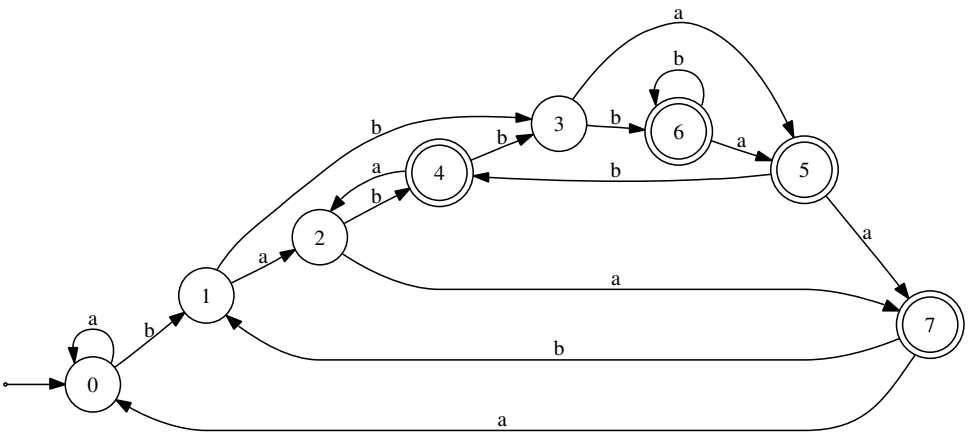

Figure 4.2: An FSM accepting $L\big(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a} + \mathtt{b}) \cdot (\mathtt{a} + \mathtt{b})\big)$.

Remark: There is a nice tool available that can be used to better understand finite state machines. This tool is called JFLAP. It is a *Java* program and is available at

https://www2.cs.duke.edu/csed/jflap.

4.2 Non-Deterministic Finite State Machines

For many applications, the finite state machines introduced in the previous section are unwieldy because they have a large numbers of states. For example, the regular expression to recognize the language

$$L\big((\mathtt{a}+\mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a}+\mathtt{b}) \cdot (\mathtt{a}+\mathtt{b})\big)$$

needs 8 different states since the FSM needs to remember the last three characters that have been read and there are $2^3 = 8$ combinations of these characters. It would be possible to simplify this FSM if the FSM would be permitted to *choose* its next state from a given set of states.

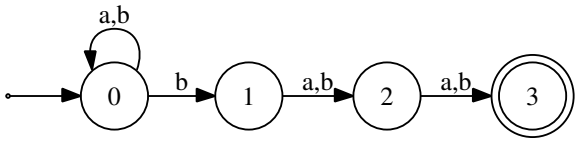

Figure 4.3: A non-deterministic finite state machine to recognize $L\big((\mathtt{a}+\mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a}+\mathtt{b}) \cdot (\mathtt{a}+\mathtt{b})\big)$.

Figure 4.3 presents a non-deterministic finite state machine that accepts the language specified by the regular expression

$$(\mathtt{a}+\mathtt{b})^* \cdot \mathtt{b} \cdot (\mathtt{a}+\mathtt{b}) \cdot (\mathtt{a}+\mathtt{b}).$$

This finite state machine has only 4 different states that are named 0, 1, 2 and 3.

1. 0 is the start state. If the FSM reads the letter a while it is in this state, the FSM will stay in state 0. However, if the FSM reads the character b, then the finite state machine has a choice: It can either stay in state 0, or it might switch to the state 1.
2. In state 1 the finite state machine switches to state 2 if it reads either the character a or the character b.
3. In state 2 the FSM switches to state 3 if it reads either the character a or the character b.
4. State 3 is the accepting state. There is no transition from this state. Hence, if the FSM is in state 3 and there are still characters to read, then the FSM dies.

The finite state machine in Figure 4.3 is non-deterministic because it has to guess the next state if it is in state 0 and reads the character "b". Let us consider a possible *computation* of the FSM when it reads the input "abab":

$$0 \overset{a}{\mapsto} 0 \overset{b}{\mapsto} 1 \overset{a}{\mapsto} 2 \overset{b}{\mapsto} 3$$

In this computation, the FSM has chosen the correct transition when reading the first occurrence of the character "b". If the FSM had stayed in the state 0 instead of switching into the state 1, it would have been impossible to reach the accepting state 3 later because then the computation would have worked out as follows:

$$0 \overset{a}{\mapsto} 0 \overset{b}{\mapsto} 0 \overset{a}{\mapsto} 0 \overset{b}{\mapsto} 1$$

Here, the FSM is in state 1 after consuming the input string "abab" and as state 1 is not an accepting state, the FSM would have falsely rejected the string "abab". Let us consider a different example where the input is the string "bbbbb":

$$0 \overset{b}{\mapsto} 0 \overset{b}{\mapsto} 1 \overset{b}{\mapsto} 2 \overset{b}{\mapsto} 3 \overset{b}{\mapsto} \Omega$$

Here, the FSM has switched to early into the state 1. In this case, the FSM dies when reading the last character "b". If the FSM has stayed in state 0 when reading the second occurrence of the character "b", then it would have correctly accepted the string "bbbbb" since then the computation could have been as follows:

$$0 \overset{b}{\mapsto} 0 \overset{b}{\mapsto} 0 \overset{b}{\mapsto} 1 \overset{b}{\mapsto} 2 \overset{b}{\mapsto} 3.$$

The previous examples show that in order to avoid premature death, the given non-deterministic FSM has to choose its successor state wisely. If F is a non-deterministic FSM and s is a string such that F can, when reading s, choose its successor so that it reaches an accepting state after having read s, then the string s is an element of the language $L(F)$.

It seems that the concept of a non-deterministic FSM is far more powerful than the concept of a deterministic FSM. After all, a non-deterministic FSM appears to have some form of clairvoyance for else it could not guess which

states to choose. However, we will prove in the next section that both deterministic and non-deterministic FSMs have the same power to recognize languages: Every language recognized by a non-deterministic FSM is also recognized by a deterministic FSM. In order to prove this claim, we have to formalize the notion of a non-deterministic FSM. The definition that follows is more general than the informal description of non-deterministic FSMs given so far, as we will allow the FSM to also have ε transitions. An ε transition allows the FSM to switch its state without reading any character. For example, if there is an ε transition from the state 1 into the state 2, we write

$$1 \overset{\varepsilon}{\mapsto} 2.$$

Definition 11 (NFA) A non-deterministic FSM (abbreviated as NFA for non-deterministic automaton) is a 5-tupel

$$\langle Q, \Sigma, \delta, q_0, A \rangle,$$

such that the following holds:

1. Q is the finite set of states.
2. Σ is the input alphabet.
3. δ is a function from $Q \times (\Sigma \cup \{\varepsilon\})$ that assigns a set of states $\delta(q, a) \subseteq Q$ to every pair $\langle q, a \rangle$ from $Q \times (\Sigma \cup \{\varepsilon\})$:

   $$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q.$$

   If $a \in \Sigma$, then $\delta(q, a)$ is the set of states the FSM can switch to after reading the character a in state q. The set $\delta\big(q, \varepsilon\big)$ is the set of states that can be reached from the state q without reading a character.

   As in the deterministic case, δ is called the transition function.
4. $q_0 \in Q$ is the start state.
5. $A \subseteq Q$ is the set of accepting states.

If we have $q_2 \in \delta(q_1, \varepsilon)$, then the FSM has an ε-transition from the state q_1 into the state q_2. This is written as

$$q_1 \overset{\varepsilon}{\mapsto} q_2.$$

If $c \in \Sigma$ and $q_2 \in \delta(q_1, c)$, we write

$$q_1 \overset{c}{\mapsto} q_2.$$ □

In order to distinguish a deterministic FSM from a non-deterministic FSM, deterministic FSMs are also called DFA which is short for deterministic finite automaton.

Example: For the FSM F shown in Figure 4.3 on page 29 we have

$$F = \langle Q, \Sigma, \delta, 0, A \rangle \quad \text{where}$$

1. $Q = \{0, 1, 2, 3\}$.
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.
3. $\delta = \big\{\langle 0, \mathtt{a}\rangle \mapsto \{0\}, \langle 0, \mathtt{b}\rangle \mapsto \{0, 1\}, \langle 0, \varepsilon\rangle \mapsto \{\}, \langle 1, \mathtt{a}\rangle \mapsto \{2\}, \langle 1, \mathtt{b}\rangle \mapsto \{2\}, \langle 1, \varepsilon\rangle \mapsto \{\},$
   $\langle 2, \mathtt{a}\rangle \mapsto \{3\}, \langle 2, \mathtt{b}\rangle \mapsto \{3\}, \langle 2, \varepsilon\rangle \mapsto \{\},$
   $\langle 3, \mathtt{a}\rangle \mapsto \{\}, \langle 3, \mathtt{b}\rangle \mapsto \{\}, \langle 3, \varepsilon\rangle \mapsto \{\}\big\}.$

   It is more convenient to specify the transition function δ as follows:

   $$0 \overset{\mathtt{a}}{\mapsto} 0, \quad 0 \overset{\mathtt{b}}{\mapsto} 0, \quad 0 \overset{\mathtt{b}}{\mapsto} 1, \quad 1 \overset{\mathtt{a}}{\mapsto} 2,$$
   $$1 \overset{\mathtt{b}}{\mapsto} 2, \quad 2 \overset{\mathtt{a}}{\mapsto} 3 \quad \text{and} \quad 2 \overset{\mathtt{b}}{\mapsto} 3.$$
4. The start state is 0.
5. $A = \{3\}$, hence the only accepting state is 3. ◇

In order to formally define how a non-deterministic FSM processes its input we introduce the notion of a configuration of a non-deterministic FSM. A configuration is defined as a pair

$$\langle q, s \rangle$$

where q is a state and s is a string. Here, q is the current state of the FSM and s is the part of the input that has not yet been consumed. We define a binary relation $\rightsquigarrow$ on configurations as follows:

$$\langle q_1, cs \rangle \rightsquigarrow \langle q_2, s \rangle \quad \text{iff} \quad q_1 \overset{c}{\mapsto} q_2, \text{ i.e. if } q_2 \in \delta(q_1, c).$$

Therefore, we have $\langle q_1, cs \rangle \rightsquigarrow \langle q_2, s \rangle$ if and only if the FSM transitions from the state q_1 into the state q_2 when the character c is consumed. Furthermore, we have

$$\langle q_1, s \rangle \rightsquigarrow \langle q_2, s \rangle \quad \text{iff} \quad q_1 \overset{\varepsilon}{\mapsto} q_2, \text{ i.e. if } q_2 \in \delta(q_1, \varepsilon).$$

This accounts for the ε transitions. The reflexive-transitive closure of the relation $\rightsquigarrow$ is written as $\rightsquigarrow^*$. The language accepted by a non-deterministic FSM F is denoted as $L(F)$ and is defined as

$$L(F) := \big\{ s \in \Sigma^* \mid \exists p \in A : \langle q_0, s \rangle \rightsquigarrow^* \langle p, \varepsilon \rangle \big\}.$$

Here, q_0 is the start state and A is the set of accepting states. Hence, a string s is an element of the language $L(F)$, iff there is an accepting state p such that the configuration $\langle p, \varepsilon \rangle$ is reachable from the configuration $\langle q_0, s \rangle$.

Example: The FSM F shown in Figure 4.3 accepts those strings $w \in \{\mathtt{a}, \mathtt{b}\}^*$ such that the antepenultimate character of w is the character "b":

$$L(F) = \big\{ w \in \{\mathtt{a}, \mathtt{b}\}^* \;\big|\; |w| \geq 3 \wedge w[-3] = \mathtt{b} \big\} \qquad \diamond$$

I have found a simulator for non-deterministic finite state machines at the following address:

http://ivanzuzak.info/noam/webapps/fsm_simulator/

Since this simulator is written in *JavaScript* it is even more convenient to use than the *Java* applet for deterministic finite state machines discussed earlier.

Exercise 5: Specify a non-deterministic FSM F such that $L(F)$ is the set of those strings from the language $\{a, b\}^*$ that contain the substring "aba". $\diamond$

4.3 Equivalence of Deterministic and Non-Deterministic FSMs

In this section we show how a non-deterministic FSM

$$F = \langle Q, \Sigma, \delta, q_0, A \rangle$$

can be transformed into a deterministic FSM $\mathit{det}(F)$ such that both FSMs accept the same language, i.e. we have

$$L(F) = L\big(\mathit{det}(F)\big)$$

The idea behind this transformation is that the FSM $\mathit{det}(F)$ has to compute the set of all states that the FSM F could be in. Hence the states of the deterministic FSM $\mathit{det}(F)$ are **sets** of states of the non-deterministic FSM F. A set of these states contains all those states that the non-deterministic FSM F could have reached. Furthermore, a set M of states of the FSM F is an accepting state of the FSM $\mathit{det}(F)$ if the set M contains an accepting state of the FSM F.

In order to present the construction of $\mathit{det}(F)$ we first have to define two auxiliary functions. We start with the ε-closure of a given state. For every state q of the non-deterministic FSM F the function

$$\mathit{ec} : Q \to 2^Q$$

computes the set $\mathit{ec}(q)$ of all those states that the FSM F can reach by ε transitions from the state q. Formally, the set $\mathit{ec}(Q)$ is computed inductively:

B.C.: $q \in \mathit{ec}(q)$.

I.S.: $p \in \mathit{ec}(q) \wedge r \in \delta(p, \varepsilon) \;\rightarrow\; r \in \mathit{ec}(q)$.

If the state p is an element of the ε-closure of the state q and there is an ε-transition from p to some state r, then r is also an element of the ε-transition of q.

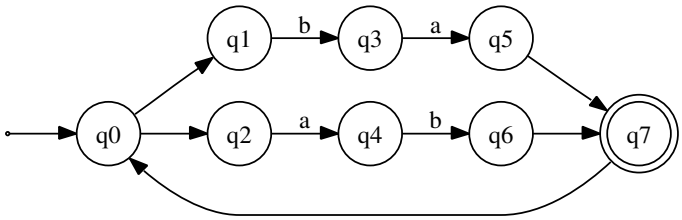

Figure 4.4: A non-deterministic FSM with ε-transitions.

Example: Figure 4.4 shows a non-deterministic FSM with ε-transitions. In the figure, the ε-transitions are shown as unlabelled arrows. We compute the ε-closure for all states:

1. $\mathit{ec}(q_0) = \{q_0, q_1, q_2\}$,
2. $\mathit{ec}(q_1) = \{q_1\}$,
3. $\mathit{ec}(q_2) = \{q_2\}$,
4. $\mathit{ec}(q_3) = \{q_3\}$,
5. $\mathit{ec}(q_4) = \{q_4\}$,
6. $\mathit{ec}(q_5) = \{q_5, q_7, q_0, q_1, q_2\}$,
7. $\mathit{ec}(q_6) = \{q_6, q_7, q_0, q_1, q_2\}$,
8. $\mathit{ec}(q_7) = \{q_7, q_0, q_1, q_2\}$. □

In order to transform a non-deterministic FSM into a deterministic FSM $\mathit{det}(F)$ we have to extend the function $\delta : Q \times \Sigma \rightarrow 2^Q$ into the function

$$\widehat{\delta} : Q \times \Sigma \rightarrow 2^Q.$$

The idea is that given a state q and a character c, $\widehat{\delta}(q, c)$ is the set of all states that the FSM F could reach when it reads the character c in state q and then performs an arbitrary number of ε-transitions. Formally, the definition of $\widehat{\delta}$ is as follows:

$$\widehat{\delta}(q_1, c) := \bigcup \big\{ \mathit{ec}(q_2) \mid q_2 \in \delta(q_1, c) \big\}.$$

This formula is to be read as follows:

(a) For every state $q_2 \in Q$ that can be reached from the state q_1 by reading the character c we compute the ε-closure $\mathit{ec}(q_2)$.

(b) Then we take the union of all these sets $\mathit{ec}(q_2)$.

Example: In continuation of the previous example (shown in Figure 4.4) we have:

1. $\widehat{\delta}(q_0, \mathsf{a}) = \{\}$,

   because in state q_0 there is no transition on reading the character a. Note that in our definition of the function $\widehat{\delta}$ the ε-transitions are done only after the character has been read.

2. $\widehat{\delta}(q_1, \mathtt{b}) = \{q_3\}$,

   because when the letter `'b'` is read in the state q_1 the FSM switches into the state q_3 and the state q_3 has no ε-transitions.

3. $\widehat{\delta}(q_3, \mathtt{a}) = \{q_5, q_7, q_0, q_1, q_2\}$,

   because when the letter `'a'` is read in the state q_3 the FSM switches into the state q_5. From q_5 the states q_7, q_0, q_1 and q_2 are reachable by ε-transitions. ◇

The function $\widehat{\delta}$ maps a state into a set of states. Since the FSM $\mathit{det}(F)$ uses sets of states of the FSM F as its states we need a function that maps sets of states of the FSM F into sets of states. Hence we generalize the function $\widehat{\delta}$ to the function

$$\Delta : 2^Q \times \Sigma \to 2^Q$$

such that for a set M of states and a character c the expression $\Delta(M, c)$ computes the set of all those states that the FSM F could be in if it is in a state from M, then reads the character c, and finally makes some ε-transitions. The formal definition is as follows:

$$\Delta(M, c) := \bigcup \Big\{ \widehat{\delta}(q, c) \mid q \in M \Big\}.$$

This formula is easy to understand: For every state $q \in M$ we compute the set of states that the FSM could be in after reading the character c and doing some ε-transitions. Then we take the union of these sets.

Example: Continuing our previous example (shown in Figure 4.4) we have:

1. $\Delta(\{q_0, q_1, q_2\}, \mathtt{a}) = \{q_4\}$,
2. $\Delta(\{q_0, q_1, q_2\}, \mathtt{b}) = \{q_3\}$,
3. $\Delta(\{q_3\}, \mathtt{a}) = \{q_5, q_7, q_0, q_1, q_2\}$,
4. $\Delta(\{q_3\}, \mathtt{b}) = \{\}$,
5. $\Delta(\{q_4\}, \mathtt{a}) = \{\}$,
6. $\Delta(\{q_4\}, \mathtt{b}) = \{q_6, q_7, q_0, q_1, q_2\}$. ◇

Now we are ready to formally define how the deterministic FSM $\mathit{det}(F)$ is constructed from the non-deterministic FSM $F := \langle Q, \Sigma, \delta, q_0, A \rangle$. We define:

$$\mathit{det}(F) = \langle 2^Q, \Sigma, \Delta, \mathit{ec}(q_0), \widehat{A} \rangle$$

where the components of this tuple are defined as follows:

1. The set of states of $\mathit{det}(F)$ is the set of all subsets of Q and therefore it is equal to the power set 2^Q.

   Later we will see that we do not need all of these subsets. The reason is that the states are those subsets that could be reached from the start state q_0 when some string has been read. In most cases there are some combinations of states that can not be reached and the corresponding sets are not really needed as states.

2. The input alphabet Σ does not change when going from F to $\mathit{det}(F)$. After all, the deterministic FSM $\mathit{det}(F)$ has to recognize the same language as the non-deterministic FSM F.

3. The previously defined function Δ specifies how the set of states changes when a character is read.

4. The start state $\mathtt{ec}(q_0)$ of the non-deterministic FSM $\mathit{det}(F)$ is the set of all states that can be reached from the start state q_0 of the non-deterministic FSM F via ε-transitions.

5. The set of accepting states $\widehat{A}$ is the set of those subsets of Q that contain an accepting state of the FSM F:

   $$\widehat{A} := \big\{ M \in 2^Q \mid M \cap A \neq \{\} \big\}.$$

Exercise 6: Transform the non-deterministic FSM F that is shown in Figure 4.3 on page 29 into the deterministic FSM $det(F)$. ◇

Solution: We start by computing the set of states.

1. As we have $ec(0) = \{0\}$, the start state of $det(F)$ is the set containing 0.
   $$S_0 := ec(0) = \{0\}.$$
2. As we have $\delta(0, \mathtt{a}) = \{0\}$ and there are no ε-transitions we have
   $$\Delta(S_0, \mathtt{a}) = \Delta(\{0\}, \mathtt{a}) = \{0\} = S_0.$$
3. As we have $\delta(0, \mathtt{b}) = \{0, 1\}$ we conclude
   $$S_1 := \Delta(S_0, \mathtt{b}) = \Delta(\{0\}, \mathtt{b}) = \{0, 1\}.$$
4. We have that $\delta(0, \mathtt{a}) = \{0\}$ and $\delta(1, \mathtt{a}) = \{2\}$. Hence
   $$S_2 := \Delta(S_1, \mathtt{a}) = \Delta(\{0, 1\}, \mathtt{a}) = \{0, 2\}.$$
5. We have $\delta(0, \mathtt{b}) \in \{0, 1\}$ and $\delta(1, \mathtt{b}) = \{2\}$. Therefore
   $$S_4 := \Delta(S_1, \mathtt{b}) = \Delta(\{0, 1\}, \mathtt{b}) = \{0, 1, 2\}$$
   Similarly we derive the following:
6. $S_3 := \Delta(S_2, \mathtt{a}) = \Delta(\{0, 2\}, \mathtt{a}) = \{0, 3\}.$
7. $S_5 := \Delta(S_2, \mathtt{b}) = \Delta(\{0, 2\}, \mathtt{b}) = \{0, 1, 3\}.$
8. $S_6 := \Delta(S_4, \mathtt{a}) = \Delta(\{0, 1, 2\}, \mathtt{a}) = \{0, 2, 3\}.$
9. $S_7 := \Delta(S_4, \mathtt{b}) = \Delta(\{0, 1, 2\}, \mathtt{b}) = \{0, 1, 2, 3\}.$
10. $\Delta(S_3, \mathtt{a}) = \Delta(\{0, 3\}, \mathtt{a}) = \{0\} = S_0.$
11. $\Delta(S_3, \mathtt{b}) = \Delta(\{0, 3\}, \mathtt{b}) = \{0, 1\} = S_1.$
12. $\Delta(S_5, \mathtt{a}) = \Delta(\{0, 1, 3\}, \mathtt{a}) = \{0, 2\} = S_2.$
13. $\Delta(S_5, \mathtt{b}) = \Delta(\{0, 1, 3\}, \mathtt{b}) = \{0, 1, 2\} = S_4.$
14. $\Delta(S_6, \mathtt{a}) = \Delta(\{0, 2, 3\}, \mathtt{a}) = \{0, 3\} = S_3.$
15. $\Delta(S_6, \mathtt{b}) = \Delta(\{0, 2, 3\}, \mathtt{b}) = \{0, 1, 3\} = S_5.$
16. $\Delta(S_7, \mathtt{a}) = \Delta(\{0, 1, 2, 3\}, \mathtt{a}) = \{0, 2, 3\} = S_6.$
17. $\Delta(S_7, \mathtt{b}) = \Delta(\{0, 1, 2, 3\}, \mathtt{b}) = \{0, 1, 2, 3\} = S_7.$

These are all possible sets of states that the deterministic FSM $det(F)$ can reach. For a better overview let us summarize the definitions of the individual states of the deterministic FSM:

$$S_0 = \{0\},\ S_1 = \{0, 1\},\ S_2 = \{0, 2\},\ S_3 = \{0, 3\},\ S_4 = \{0, 1, 2\},$$
$$S_5 = \{0, 1, 3\},\ S_6 = \{0, 2, 3\},\ S_7 = \{0, 1, 2, 3\}$$

Therefore the set $\widehat{Q}$ of the deterministic FSM $det(F)$ is given as follows:

$$\widehat{Q} := \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}.$$

The transition function Δ is shown as a table:

Δ	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
a	S_0	S_2	S_3	S_0	S_6	S_2	S_3	S_6
b	S_1	S_4	S_5	S_1	S_7	S_4	S_5	S_7

Finally we recognize that only the sets S_3, S_5, S_6 and S_7 contain the accepting state 3. Therefore we have

$$\widehat{A} := \{S_3, S_5, S_6, S_7\}.$$

Therefore we have now found the deterministic $\textsc{Fsm}$ *det*(F). We have

$$\textit{det}(F) := \langle \widehat{Q}, \Sigma, \Delta, S_0, \widehat{A} \rangle.$$

This $\textsc{Fsm}$ is shown in Figure 4.5 on page 36.

We realize that this deterministic $\textsc{Fsm}$ *det*(F) has 8 different states. The non-deterministic $\textsc{Fsm}$ F has 4 different states $Q = \{0, 1, 2, 3\}$. Hence the power set 2^Q has 16 elements. Why then has the $\textsc{Fsm}$ *det*(F) only 8 and not $2^4 = 16$ states? The reason is that we can only reach those sets of states form the start 0 that contain the state 0 because no matter whether we read an a or a b the $\textsc{Fsm}$ F can always choose to switch to the state 0. Therefore, every set of states that is reachable from the state 0 has to contain the state 0. Therefore, sets that do not contain 0 are not needed as states of the deterministic $\textsc{Fsm}$ *det*(F).

Exercise 7: Transform the non-deterministic $\textsc{Fsm}$ F that is shown in Figure 4.4 on page 32 into an equivalent deterministic $\textsc{Fsm}$ *det*(F). ◇

4.3.1 Implementation

It is straightforward to implement the theory developed so far. The *Jupyter notebook*

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/NFA-2-DFA.ipynb

contains a program that takes a non-deterministic $\textsc{Fsm}$ F and computes the deterministic $\textsc{Fsm}$ $\texttt{det}(F)$.

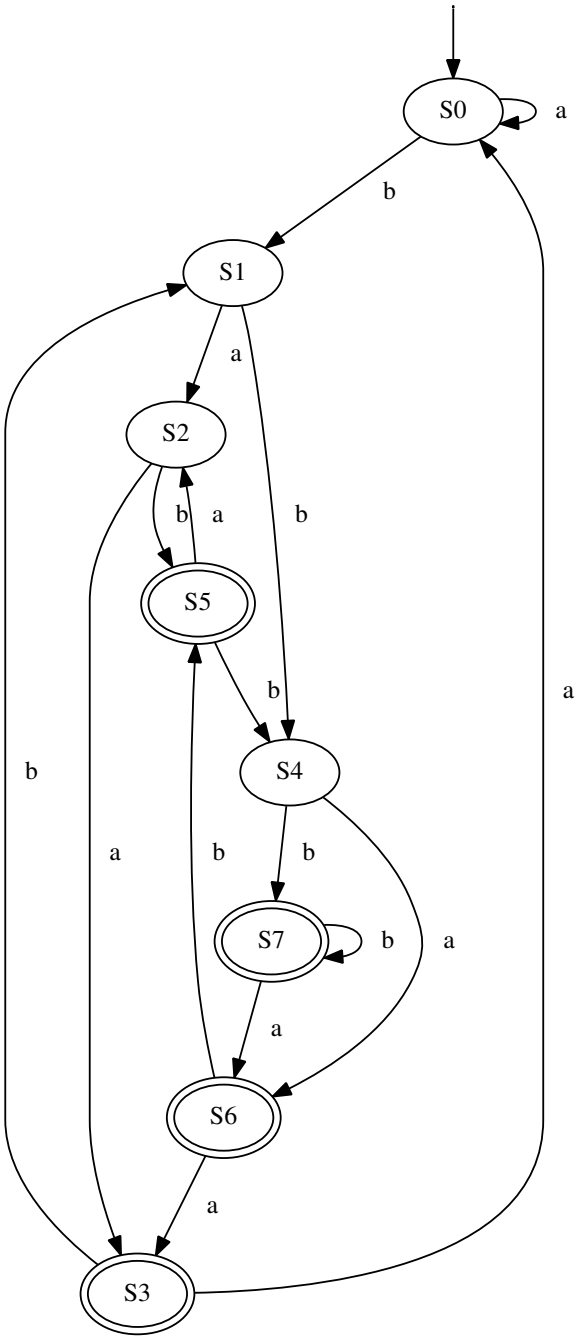

Figure 4.5: The deterministic FSM *det*(F).

4.4 From Regular Expressions to Non-Deterministic Finite State Machines

In this section we show how regular expressions can be implemented as non-deterministic finite state machine. Given a regular expression r we will construct a non-deterministic FSM $A(r)$ that accepts the language that is described by the regular expression r, i.e. we will have

$$L\big(A(r)\big) = L(r).$$

The FSM $A(r)$ is defined by induction on the regular expression r. The FSM $A(r)$ will have the following properties:

1. $A(r)$ does not have a transition into its start state.
2. $A(r)$ has exactly one accepting state. We refer to this state as *accept*$\big(A(r)\big)$. Furthermore, there are no transitions out of this state.

In the following we assume that Σ is the alphabet that has been used when constructing the regular expression r. Then we can define $A(r)$ as follows:

1. The FSM $A(\emptyset)$ is defined as
   $$A(\emptyset) = \big\langle \{q_0, q_1\}, \Sigma, \{\}, q_0, \{q_1\}\big\rangle.$$
   Note that this FSM has no transitions at all.

   

   Figure 4.6: The FSM $A(\emptyset)$.

   Figure 4.6 shows the FSM $A(\emptyset)$. It is obvious that we have $L\big(A(\emptyset)\big) = \{\}$.
2. The FSM $A(\varepsilon)$ is defined as
   $$A(\varepsilon) = \big\langle \{q_0, q_1\}, \Sigma, \big\{\langle q_0, \varepsilon\rangle \mapsto \{q_1\}\big\}, q_0, \{q_1\}\big\rangle.$$

   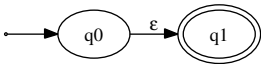

   Figure 4.7: The FSM $A(\varepsilon)$.

   Figure 4.7 shows the FSM $A(\varepsilon)$. We have that $L\big(A(\varepsilon)\big) = \{\text{''}\}$, i.e. the FSM only accepts the empty string.
3. For a letter $c \in \Sigma$ the FSM $A(c)$ is defined as
   $$A(c) = \big\langle \{q_0, q_1\}, \Sigma, \big\{\langle q_0, c\rangle \mapsto \{q_1\}\big\}, q_0, \{q_1\}\big\rangle.$$

   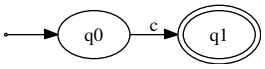

   Figure 4.8: The FSM $A(c)$.

   Figure 4.8 shows $A(c)$. We have that $L\big(A(c)\big) = \{c\}$, i.e. the FSM only accepts the character c.
4. In order to define the FSM $A(r_1 \cdot r_2)$ for the concatenation $r_1 \cdot r_2$ we assume that the states in the FSMs $A(r_1)$ and $A(r_2)$ are different. This can always be achieved by renaming the states of $A(r_2)$. Next, we assume that $A(r_1)$ and $A(r_2)$ have the following form:
   (a) $A(r_1) = \big\langle Q_1, \Sigma, \delta_1, q_1, \{q_2\}\big\rangle$,
   (b) $A(r_2) = \big\langle Q_2, \Sigma, \delta_2, q_3, \{q_4\}\big\rangle$,
   (c) $Q_1 \cap Q_2 = \{\}$.

   Then we can build the FSM $A(r_1 \cdot r_2)$ from $A(r_1)$ and $A(r_2)$ as follows:
   $$A(r_1 \cdot r_2) := \big\langle Q_1 \cup Q_2, \Sigma, \big\{\langle q_2, \varepsilon\rangle \mapsto \{q_3\}\big\} \cup \delta_1 \cup \delta_2, q_1, \{q_4\}\big\rangle$$
   Here, the notation $\{\langle q_2, \varepsilon\rangle \mapsto q_3\} \cup \delta_1 \cup \delta_2$ specifies that $A(r_1 \cdot r_2)$ contains all transitions from both $A(r_1)$

and $A(r_2)$ and, furthermore, contains an ε-transition from q_2 to q_3. Formally, this transition function δ can be specified as follows:

$$\delta(q,c) := \begin{cases} \{q_3\} & \text{if } q = q_2 \text{ and } c = \varepsilon, \\ \delta_1(q,c) & \text{if } q \in Q_1 \text{ and } \langle q, c\rangle \neq \langle q_2, \varepsilon\rangle, \\ \delta_2(q,c) & \text{if } q \in Q_2. \end{cases}$$

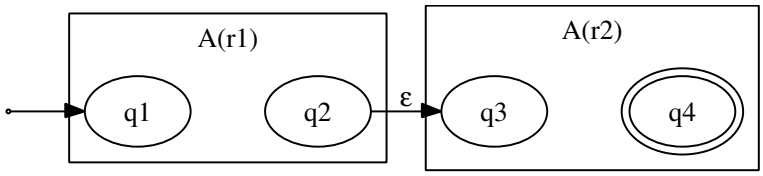

Figure 4.9: The FSM $A(r_1 \cdot r_2)$.

Figure 4.9 shows the FSM $A(r_1 \cdot r_2)$.

Instead of having an ε-transition from q_2 to q_3 we can identify the states q_2 and q_3. The advantage is that the resulting FSM is smaller. We will do this when creating FSMs by hand.

I haven't done this identification in the definition above because both the graphical representation and the implementation get more complicated when we identify these states.

5. In order to define the FSM $A(r_1 + r_2)$ we assume again that the states of the FSMs $A(r_1)$ and $A(r_2)$ are different and that $A(r_1)$ and $A(r_2)$ have the following form:

   (a) $A(r_1) = \big\langle Q_1, \Sigma, \delta_1, q_1, \{q_3\}\big\rangle$,

   (b) $A(r_2) = \big\langle Q_2, \Sigma, \delta_2, q_2, \{q_4\}\big\rangle$,

   (c) $Q_1 \cap Q_2 = \{\}$.

   Then the FSM $A(r_1 + r_2)$ is defined as follows:

$$\big\langle \{q_0, q_5\} \cup Q_1 \cup Q_2, \Sigma, \big\{\langle q_0, \varepsilon\rangle \mapsto \{q_1, q_2\}, \langle q_3, \varepsilon\rangle \mapsto \{q_5\}, \langle q_4, \varepsilon\rangle \mapsto \{q_5\}\big\} \cup \delta_1 \cup \delta_2, q_0, \{q_5\}\big\rangle$$

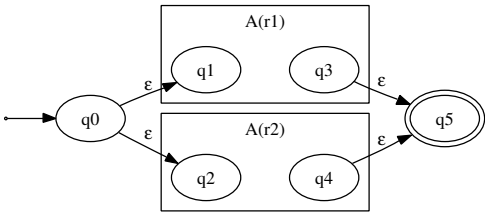

Figure 4.10: The FSM $A(r_1 + r_2)$.

Figure 4.10 shows the FSM $A(r_1 + r_2)$. In addition to the states of $A(r_1)$ and $A(r_2)$ there are two more states:

(a) q_0 is the start state of the FSM $A(r_1 + r_2)$,

(b) q_5 is the only accepting state of the FSM $A(r_1 + r_2)$.

In addition to the transitions of $A(r_1)$ and $A(r_2)$ the FSM $A(r_1 + r_2)$ has four more ε-transitions.

(a) The new start state q_0 has two ε-transitions leading to the start states q_1 and q_2 of the FSMs $A(r_1)$ and $A(r_2)$.

(b) Each of the accepting states q_3 and q_4 of the FSMs $A(r_1)$ and $A(r_2)$ has an ε-transition to the new accepting state q_5.

In order to simplify this FSM we could identify the three states q_0, q_1 and q_2 and the three states q_3, q_4 and q_5. However, the resulting FSM would be more difficult to understand and hence we are not doing this when creating FSMs by hand.

6. In order to define the FSM $A(r^*)$ we assume that

$$A(r) = \langle Q, \Sigma, \delta, q_1, \{q_2\}\rangle.$$

Then $A(r^*)$ is defined as follows:

$$\langle \{q_0, q_3\} \cup Q, \Sigma, \{\langle q_0, \varepsilon\rangle \mapsto \{q_1, q_3\}, \langle q_2, \varepsilon\rangle \mapsto \{q_1, q_3\}\} \cup \delta, q_0, \{q_3\}\rangle.$$

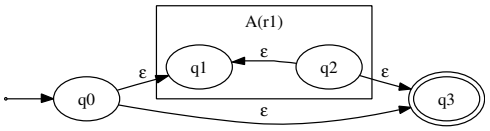

Figure 4.11: The FSM $A(r^*)$.

Figure 4.11 shows the FSM $A(r^*)$. In comparison with $A(r)$ this FSM has two additional states.

(a) q_0 is the start state of $A(r^*)$,

(b) q_3 is the only accepting state of $A(r^*)$.

The FSM $A(r^*)$ has four more ε-transitions than $A(r)$:

(a) The new start state q_0 has ε-transitions to the states q_1 and q_3.

(b) q_2 has an ε-transition back to the state q_1.

(c) q_2 also has an ε-transition to the state q_3.

Attention: If we would identify the two states q_0 and q_1 and the two states q_2 and q_3, then the resulting FSM would no longer be correct!

Exercise 8: Construct a non-deterministic FSM that accepts the language specified by the regular expression

$$\mathtt{a}^* \cdot \mathtt{b}^*.$$

Consider what would happen if you would identify the two states q_0 and q_1 and the two states q_2 and q_3 in step 6 of the construction given above. ◇

Exercise 9: Construct a non-deterministic FSM for the regular expression

$$(\mathtt{a} + \mathtt{b}) \cdot \mathtt{a}^* \cdot \mathtt{b}.$$ ◇

4.4.1 Implementation

The *Jupyter notebook* Regexp-2-NFA.ipynb implements the theory discussed in this section.

4.5 Translating a Deterministic Fsm into a Regular Expression

In this last section we start with a deterministic FSM F and construct a regular expression r such that we have

$$L(r) = L(F).$$

We assume that the FSM F is given as follows:

$$F = \langle \{q_0, q_1, \cdots, q_n\}, \Sigma, \delta, q_0, A\rangle.$$

For every pair of states $\langle p_1, p_2\rangle \in Q \times Q$ we define a regular expression $r(p_1, p_2)$ such that $r(p_1, p_2)$ describes those strings w that take the FSM F from the state p_1 to the state p_2, i.e. we have

$$L\big(r(p_1, p_2)\big) = \big\{w \in \Sigma^* \mid \langle p_1, w\rangle \rightsquigarrow^* \langle p_2, \varepsilon\rangle\big\}.$$

The definition of $r(p_1, p_2)$ is done by first defining auxiliary regular expressions $r^{(k)}(p_1, p_2)$ for all $k = 0, \cdots, n+1$. The regular expression $r^{(k)}(p_1, p_2)$ specifies those strings that take the FSM F from the state p_1 to the state p_2 without visiting a state from the set

$$Q_k := \big\{q_i \mid i \in \{k, \cdots, n\}\big\} = \{q_k, \cdots, q_n\}.$$

To this end we define the ternary relation

$$\mapsto_k \subseteq (Q \times \Sigma^* \times Q).$$

Given two states $p, q \in Q$ and a string w we have that

$$p \overset{w}{\mapsto}_k q$$

holds iff the FSM F switches from the state p to the state q when it reads the string w but on reading w does not switch to a state from the set Q_k in-between. Here, "in-between" specifies that the states p and q may well be elements of the set Q_k, only the states between p and q must not be in Q_k. The formal definition of $p \overset{w}{\mapsto}_k q$ is done by induction on w:

B.C.: $|w| \leq 1$. Then there are two cases:

(a) $p \overset{\varepsilon}{\mapsto}_k p$,
because when the empty string is read we can only reach the state p if we start in the state p.

(b) $\delta(p, c) = q \;\Rightarrow\; p \overset{c}{\mapsto}_k q$,
because when the FSM reads the character c and switches from state p directly to the state q, there are no states "inbetween".

I.S.: $w = cv$ where $|v| \geq 1$.

Here we have: $p \overset{c}{\mapsto} q \wedge q \notin Q_k \wedge q \overset{v}{\mapsto}_k r \Rightarrow p \overset{cv}{\mapsto}_k r$.

Now we are ready to define the regular expressions $r^{(k)}(p_1, p_2)$ for all $k = 0, \cdots, n+1$. This definition will be done such that we have

$$L\big(r^{(k)}(p_1, p_2)\big) = \big\{w \in \Sigma^* \mid p_1 \overset{w}{\mapsto}_k p_2\big\}.$$

The definition of the regular expressions $r^{(k)}(p_1, p_2)$ is done by induction on k.

B.C.: $k = 0$.

Then we have $Q_0 = Q$ and therefore the set Q_0 contains all states. Therefore, when the FSM switches from the state p_1 to the state p_2 it must not visit any states in-between. There are two cases.

(a) $p_1 \neq p_2$: Then we can have $p_1 \overset{w}{\mapsto}_0 p_2$ only if w contains but a single letter. Define

$$\{c_1, \cdots, c_l\} := \{c \in \Sigma \mid \delta(p_1, c) = p_2\}$$

as the set of all letters that take the FSM from the state p_1 to the state p_2. If this set is not empty we define

$$r^{(0)}(p_1, p_2) := c_1 + \cdots + c_l.$$

If this set is empty, then there is no direct transition from p_1 to p_2 and we define

$$r^{(0)}(p_1, p_2) := \emptyset.$$

(b) $p_1 = p_2$: Again we define

$$\{c_1, \cdots, c_l\} := \{c \in \Sigma \mid \delta(p_1, c) = p_1\}.$$

If this set is not empty we have

$$r^{(0)}(p_1, p_1) := c_1 + \cdots + c_l + \varepsilon.$$

Otherwise we have

$$r^{(0)}(p_1, p_1) := \varepsilon.$$

I.S.: $k \mapsto k + 1$.

When compared to the regular expression $r^{(k)}(p_1, p_2)$, the regular expression $r^{(k+1)}(p_1, p_2)$ is allowed to use the state q_k, because q_k is the only element of the set Q_k that is not a member of the set Q_{k+1}. If the FSM reads a string w that switches the state p_1 to the state p_2 without switching into a state from the set Q_{k+1}, then there are two cases.

(a) We already have $p_1 \overset{w}{\mapsto}_k p_2$.

(b) The string w can be written as $w = w_1 s_1 \cdots s_l w_2$ where we have:

- $p_1 \overset{w_1}{\mapsto}_k q_k$,
- $q_k \overset{s_i}{\mapsto}_k q_k$ for all $i = \{1, \cdots, l\}$,
- $q_k \overset{w_2}{\mapsto}_k p_2$.

Therefore we define

$$r^{(k+1)}(p_1, p_2) := r^{(k)}(p_1, p_2) + r^{(k)}(p_1, q_k) \cdot \left(r^{(k)}(q_k, q_k)\right)^* \cdot r^{(k)}(q_k, p_2).$$

Now we are ready to define the regular expressions $r(p_1, p_2)$ for all states p_1 and p_2:

$$r(p_1, p_2) := r^{(n+1)}(p_1, p_2).$$

This regular expression specifies all strings that take the FSM from the state p_1 to the state p_2 without using any state from the set Q_{n+1} in-between. Since we have

$$Q_{n+1} = \{q_i \mid i \in \{0, \cdots, n\} \wedge i \geq n + 1\} = \{\}$$

we know that Q_{n+1} is empty. Therefore the regular expression $r^{(n+1)}(p_1, p_2)$ does not exclude any states when switching from state p_1 to the state p_2.

In order to construct a regular expression that specifies the language accepted by a deterministic FSM F we write the set A of accepting states of F as

$$A = \{t_1, \cdots, t_m\}.$$

Then the regular expression $r(A)$ is defined as

$$r(A) := r(q_0, t_1) + \cdots + r(q_0, t_m).$$

This regular expression specifies those strings that take the FSM F from its start state q_0 into any of its accepting states. □

Exercise 10: Take the FSM shown in Figure 4.1 and construct an equivalent regular expression.

Solution: The FSM has two states: 0 and 1. We start by computing the regular expressions $r^{(k)}(i, j)$ for all $i, j \in \{0, 1\}$ for $k = 0$, 1 und 2:

1. For $k = 0$ we have:
   (a) $r^{(0)}(0, 0) = \mathsf{a} + \varepsilon$,
   (b) $r^{(0)}(0, 1) = \mathsf{b}$,
   (c) $r^{(0)}(1, 0) = \emptyset$,
   (d) $r^{(0)}(1, 1) = \mathsf{a} + \varepsilon$.
2. For $k = 1$ we have:

(a) For $r^{(1)}(0,0)$ we have:

$$\begin{array}{rcl} r^{(1)}(0,0) & = & r^{(0)}(0,0) + r^{(0)}(0,0) \cdot \left(r^{(0)}(0,0)\right)^* \cdot r^{(0)}(0,0) \\ & \doteq & r^{(0)}(0,0) \cdot \left(r^{(0)}(0,0)\right)^* \end{array}$$

In the last step we used the fact that

$$\begin{array}{rcl} r + r \cdot r^* \cdot r & \doteq & r \cdot (\varepsilon + r^* \cdot r) \\ & \doteq & r \cdot r^* \end{array}$$

to simplify the result. If we substitute for $r^{(0)}(0,0)$ the expression $\mathtt{a} + \varepsilon$ we get

$$r^{(1)}(0,0) \doteq (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^*.$$

As we have $(\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* \doteq \mathtt{a}^*$ we have

$$r^{(1)}(0,0) \doteq \mathtt{a}^*.$$

(b) For $r^{(1)}(0,1)$ we have:

$$\begin{array}{rcl} r^{(1)}(0,1) & \doteq & r^{(0)}(0,1) + r^{(0)}(0,0) \cdot \left(r^{(0)}(0,0)\right)^* \cdot r^{(0)}(0,1) \\ & \doteq & \mathtt{b} + (\mathtt{a} + \varepsilon) \cdot (\mathtt{a} + \varepsilon)^* \cdot \mathtt{b} \\ & \doteq & \mathtt{b} + \mathtt{a}^* \cdot \mathtt{b} \\ & \doteq & (\varepsilon + \mathtt{a}^*) \cdot \mathtt{b} \\ & \doteq & \mathtt{a}^* \cdot \mathtt{b} \end{array}$$

(c) For $r^{(1)}(1,0)$ we have:

$$\begin{array}{rcl} r^{(1)}(1,0) & \doteq & r^{(0)}(1,0) + r^{(0)}(1,0) \cdot \left(r^{(0)}(0,0)\right)^* \cdot r^{(0)}(0,0) \\ & \doteq & \emptyset + \emptyset \cdot (\mathtt{a} + \varepsilon)^* \cdot (\mathtt{a} + \varepsilon) \\ & \doteq & \emptyset \end{array}$$

(d) For $r^{(1)}(1,1)$ we have

$$\begin{array}{rcl} r^{(1)}(1,1) & \doteq & r^{(0)}(1,1) + r^{(0)}(1,0) \cdot \left(r^{(0)}(0,0)\right)^* \cdot r^{(0)}(0,1) \\ & \doteq & (\mathtt{a} + \varepsilon) + \emptyset \cdot (\mathtt{a} + \varepsilon)^* \cdot \mathtt{b} \\ & \doteq & (\mathtt{a} + \varepsilon) + \emptyset \\ & \doteq & \mathtt{a} + \varepsilon \end{array}$$

3. For $k = 2$ we only have to compute the regular expression $r^{(2)}(0,1)$:

$$\begin{array}{rcl} r^{(2)}(0,1) & \doteq & r^{(1)}(0,1) + r^{(1)}(0,1) \cdot \left(r^{(1)}(1,1)\right)^* \cdot r^{(1)}(1,1) \\ & \doteq & \mathtt{a}^* \cdot \mathtt{b} + \mathtt{a}^* \cdot \mathtt{b} \cdot (\mathtt{a} + \varepsilon)^* \cdot (\mathtt{a} + \varepsilon) \\ & \doteq & \mathtt{a}^* \cdot \mathtt{b} + \mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^* \\ & \doteq & \mathtt{a}^* \cdot \mathtt{b} \cdot (\varepsilon + \mathtt{a}^*) \\ & \doteq & \mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*. \end{array}$$

As the state 0 is the start state and the state 1 is the only accepting state we have

$$r(F) = r^{(2)}(0,1) \doteq \mathtt{a}^* \cdot \mathtt{b} \cdot \mathtt{a}^*.$$

□

Exercise 11: Take the FSM shown in Figure 4.12 on page 43 and construct a regular expression specifying the same language as this FSM.

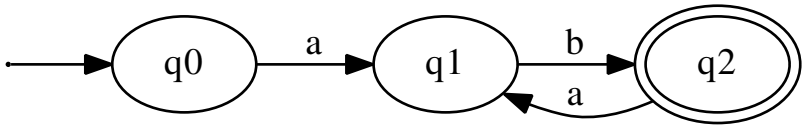

Figure 4.12: A deterministic finite state machine.

4.5.1 Implementation

The *Jupyter notebook*

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/DFA-2-RegExp.ipynb

contains a program that takes a deterministic FSM F and computes a regular expression r such that r specifies the language accepted by F, i.e. we have $L(r) = L(F)$.

4.6 Minimization of Finite State Machines

In this section we show how to minimize the number of states of a deterministic finite state machine

$$F = \langle Q, \Sigma, \delta, q_0, A \rangle.$$

Without loss of generality we want to assume that the FSM F is complete: Therefore, we assume that for each state $q \in Q$ and each letter $c \in \Sigma$ the expression $\delta(q, c)$ returns a state of Q. Our goal is to find a deterministic finite state machine

$$F^- = \langle Q^-, \Sigma, \delta^-, q_0, A^- \rangle,$$

which accepts the same language as F, so

$$L(F^-) = L(F)$$

and for which the number of states of the set Q^- is minimal. We start with the function

$$\delta^* : Q \times \Sigma^* \to Q$$

which extends the function δ by accepting a string instead of a single character as its first argument. The function call $\delta^*(q, s)$ calculates the state p which the FSM F enters if it reads the string s in the state q. As we have assumed the *Fsm* F to be complete, we have that

$$\delta(q, c) \notin \Omega \quad \text{for all } q \in Q \text{ and all } c \in \Sigma$$

and then $\delta^*(q, w)$ is defined by induction on w as follows:

(a) $\delta^*(q, \varepsilon) := q$ for all $q \in Q$.

(b) $\delta^*(q, cw) := \delta^*(\delta(q, c), v)$ for all $q \in Q$, $c \in \Sigma$, and $v \in \Sigma^*$.

Since the function δ^* is a generalization of the function δ, in the future we will not distinguish in the notation between δ and δ^* and write δ for both functions.

Obviously, in a finite state machine $F = \langle Q, \Sigma, \delta, q_0, A \rangle$ we can remove all the states $p \in Q$ which are not reachable from the start state. A state p is reachable iff a string $w \in \Sigma^*$ is given, such that

$$\delta(q_0, w) = p$$

holds. In the following we assume that all states of the FSM F can be reached from the start state.

In general, we can minimize an FSM by identifying certain states. If we look for example at the FSM shown in figure 4.13, we can identify the states q_1 and q_2 as well as q_3 and q_4 without changing the language of the FSM. The central idea in minimizing a FSM is that we compute those states which **must not** be identified and consider all other states as equivalent. States that must not be identified are called separable states. This notion is defined next.

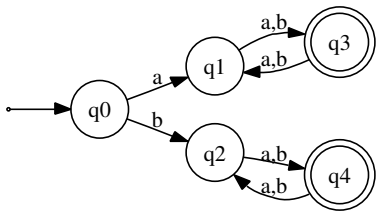

Figure 4.13: A finite state machine with equivalent states.

Definition 12 (Separable States) Assume $F = \langle Q, \Sigma, \delta, q_0, A\rangle$ is a deterministic finite state machine. Two states $p_1, p_2 \in Q$ are called separable if and only if there exists a string $s \in \Sigma^*$ such that either

1. $\delta(p_1, s) \in A$ and $\delta(p_2, s) \notin A$ or
2. $\delta(p_1, s) \notin A$ and $\delta(p_2, s) \in A$

holds. In this case, the string s separates p_1 and p_2. □

If two states p_1 and p_2 are separable, then it is obvious that theses states must not be identified. We define an equivalence relation $\sim$ on the set Q of all states by setting

$$p_1 \sim p_2 \quad \text{iff} \quad \forall s \in \Sigma^* : \big(\delta(p_1, s) \in A \leftrightarrow \delta(p_2, s) \in A\big).$$

Hence, two states p_1 and p_2 are considered to be equivalent iff they are not separable. The claim is that we can identify all pairs $\langle p_1, p_2\rangle$ of equivalent states. The identification of two states p_1 and p_2 is done by removing the state p_2 from the set Q and changing the transition function δ in a way that the new version of δ will return p_1 in all those cases where the old version of δ had returned p_2.

The question remains how we can determine which states are distinguishable. One possibility is to create a set V with pairs of states. We add the pair $\langle p, q\rangle$ to the set V if we have discovered that p and q are distinguishable. We recognize p and q as distinguishable if there is a letter $c \in \Sigma$ and two states s and t that are already known to be distinguishable such that

$$\delta(p, c) = s, \quad \delta(q, c) = t, \quad \text{and} \quad \langle s, t\rangle \in V$$

holds. This idea leads to an algorithm that uses two steps:

(a) First we initialize V with all the pairs $\langle p, q\rangle$, for which either p is an accepting state and q is not an accepting state, or q is an accepting state and p is not an accepting state, because an accepting state can be distinguished from a non-accepting state by the empty string ε:

$$V := \big\{\langle p, q\rangle \in Q \times Q \mid (p \in A \wedge q \notin A) \vee (p \notin A \wedge q \in A)\big\}$$

This can also be written as the union of two Cartesian products. We have

$$V = A \times (Q \backslash A) \cup (Q \backslash A) \times A.$$

(b) As long as we find a new pair $\langle p, q\rangle \in Q \times Q$ for which there is a letter c such that the states $\delta(p, c)$ and $\delta(q, c)$ are already distinguishable, we add this pair to the set V:

```
while ∃⟨p,q⟩ ∈ Q × Q : ∃c ∈ Σ : ⟨δ(p,c), δ(q,c)⟩ ∈ V ∧ ⟨p,q⟩ ∉ V
   choose ⟨p,q⟩ ∈ Q × Q such that ⟨δ(p,c), δ(q,c)⟩ ∈ V ∧ ⟨p,q⟩ ∉ V
      V := V ∪ { ⟨p,q⟩, ⟨q,p⟩ };
```

If we have found all pairs $\langle p, q\rangle$ of distinguishable states, then we can identify all states p and q which are not distinguishable and therefore $\langle p, q\rangle \notin V$. The FSM constructed in this way is, in fact, minimal.

Exercise 12: We consider the FSM shown in figure 4.13 and apply the algorithm described above to this FSM. We

use a table for this. The columns and rows of this table are numbered with the different states. If we have recognized in the first step that the states i and j are distinguishable, we insert a 1 in this table in the i-th row and the j-th column. Furthermore, since the states i and j can also be distinguished from the states j and i, we also insert a 1 in the j-th row and the i-th column.

1. In the first step we recognize that the two accepting states q_3 and q_4 are distinguishable from all non-accepting states. So the pairs $\langle q_0, q_3\rangle$, $\langle q_0, q_4\rangle$, $\langle q_1, q_3\rangle$, $\langle q_1, q_4\rangle$, $\langle q_2, q_3\rangle$ and $\langle q_2, q_4\rangle$ are distinguishable. So the table now has the following configuration:

	q_0	q_1	q_2	q_3	q_4
q_0				1	1
q_1				1	1
q_2				1	1
q_3	1	1	1		
q_4	1	1	1		

2. Next we see that the states q_0 and q_1 are distinguishable because

   $$\delta(q_0, a) = q_1, \quad \delta(q_1, a) = q_3 \quad \text{and} \quad q_1 \not\sim q_3.$$

   Also we see that the states q_0 and q_2 are distinguishable, because

   $$\delta(q_0, b) = q_2, \quad \delta(q_2, b) = q_4 \quad \text{and} \quad q_2 \not\sim q_4.$$

   Since we have found in the second step that $q_0 \not\sim q_1$ and $q_0 \not\sim q_2$ holds true, we will insert a 2 at the corresponding positions in the table. Therefore the table now has the following form:

	q_0	q_1	q_2	q_3	q_4
q_0		2	2	1	1
q_1	2			1	1
q_2	2			1	1
q_3	1	1	1		
q_4	1	1	1		

3. Now we do not find any more pairs of distinguishable states, because if we take a look at the pair $\langle q_1, q_2\rangle$ we can see that

   $$\delta(q_1, \mathsf{a}) = q_3 \quad \text{and} \quad \delta(q_2, \mathsf{a}) = q_4,$$

   but because the states q_3 and q_4 are not distinguishable so far, this does not yield a new distinguishable pair. Neither does

   $$\delta(q_1, \mathsf{b}) = q_3 \quad \text{and} \quad \delta(q_2, \mathsf{b}) = q_4$$

   yield a new distinguishable pair. At this point, the two states q_3 and q_4 remain. We find

   $$\delta(q_3, c) = q_1 \quad \text{and} \quad \delta(q_4, c) = q_2 \quad \text{for all } c \in \{\mathsf{a}, \mathsf{b}\}$$

   and because the states q_1 and q_2 are not yet known to be distinguishable, we have not found any new distinguishable states. So we can read the equivalent states from the table, it holds that:

   (a) $q_1 \sim q_2$
   (b) $q_3 \sim q_4$

   Figure 4.14 shows the corresponding reduced finite FSM.

Exercise 13: Construct the minimal deterministic FSM that recognizes the language $L\big(a \cdot (b \cdot a)^*\big)$. Perform the following steps:

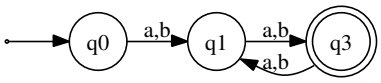

Figure 4.14: The reduced FSM.

(a) Construct a non-deterministic FSM which recognizes this language.

(b) Transform this non-deterministic FSM into a deterministic FSM.

(c) Minimize the number of states of this FSM with the algorithm given above.

4.6.1 Implementation

The *Jupyter notebook*

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Minimize.ipynb

contains a program that takes a deterministic FSM F and returns an equivalent FSM that has the minimum number of states.

4.7 Conclusion

In this chapter we have shown that the concept of a deterministic finite state machine and a regular expression are equivalent.

(a) Every deterministic finite state machine can be translated into an equivalent regular expression.

(b) Every regular expression can be translated into an equivalent non-deterministic FSM.

(c) Every non-deterministic FSM can be transformed into an equivalent deterministic FSM.

Furthermore, we have shown that every deterministic finite state machine can be minimized.

Historical Remark Stephen C. Kleene (1909 – 1994) has shown in 1956 that the concepts of finite state machines and regular expression have the same strength [Kle56].

4.8 Check your Understanding

(a) How are deterministic and non-deterministic finite state machines defined?

(b) Given a non-deterministic FSM F, can you convert it into a deterministic FSM $\texttt{det}(F)$ that accepts the same language as the FSM F?

(c) Given a regular expression r, can you describe the steps necessary to construct a non-deterministic FSM F that accepts the language specified by r?

(d) Given a deterministic FSM F can you describe how to construct a regular expression r that specifies the language accepted by F?

(e) How do we mimimize a finite state machine?

Chapter 5

The Theory of Regular Languages

A formal language $L \subseteq \Sigma^*$ is called a regular language if there is a regular expression r such that the language L is specified by r, i.e. if

$$L = L(r)$$

holds. In Chapter 4 we have shown that the regular languages are those languages that are recognized by a finite state machine. In this chapter, we show that regular languages have certain closure properties:

(a) The union $L_1 \cup L_2$ of two regular languages L_1 and L_2 is a regular language.

(b) The intersection $L_1 \cap L_2$ of two regular languages L_1 und L_2 is a regular language.

(c) The complement $\Sigma^* \backslash L$ of a regular language L is a regular language.

As an application of these closure properties we will then show how it is possible to decide whether two regular expressions are equivalent, i.e. we present an algorithm that takes two regular expressions r_1 and r_2 as input and checks, whether

$$r_1 \doteq r_2$$

holds. After that, we discuss the limits of regular languages. To this end, we prove the *pumping lemma*. Using the pumping lemma we will be able to show that, for example, the language

$$\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$$

is not regular. To summarize, this chapter

- discusses closure properties of regular languages
- presents an algorithm for checking the equivalence of regular expressions, and
- shows that certain languages are not regular.

5.1 Closure Properties of Regular Languages

In this section we show that regular languages are closed under the Boolean operations of union, intersection and complement. We start with the union.

Proposition 13 If L_1 and L_2 are regular languages, then the union $L_1 \cup L_2$ is a regular language, too.

Proof: As L_1 and L_2 are regular languages, there exist regular expressions r_1 and r_2 such that

$$L_1 = L(r_1) \quad \text{and} \quad L_2 = L(r_2)$$

holds. We define $r := r_1 + r_2$. Then we have

$$L(r) = L(r_1 + r_2) = L(r_1) \cup L(r_2) = L_1 \cup L_2.$$

Since $L_1 \cup L_2$ is a language that is generated by a regular expression, it is a regular language. □

Proposition 14 If L_1 and L_2 are regular languages, then the intersection $L_1 \cap L_2$ is a regular language, too.

Proof: While the proof of Proposition 13 follows directly from the definition of regular expressions, we have to do a little more work now. In the previous chapter we saw that for every regular expression r there is an equivalent deterministic finite state machine F, which accepts the language specified by r, and we can also assume that this FSM is complete. Let r_1 and r_2 be the regular expressions that define the languages L_1 and L_2, i.e. we have

$$L_1 = L(r_1) \quad \text{und} \quad L_2 = L(r_2).$$

First, we construct two complete deterministic FSMs F_1 and F_2 that accept these languages, i.e. we have

$$L(F_1) = L_1 \quad \text{and} \quad L(F_2) = L_2.$$

Our goal is to construct a finite state machine F such that F accepts the language $L_1 \cap L_2$ and nothing else. As every finite state machine can be converted into an equivalent regular expression we will then have shown that the language $L_1 \cap L_2$ is regular. We will use the FSMs F_1 and F_2 to construct F. Assume that

$$F_1 = \langle Q_1, \Sigma, \delta_1, q_1, A_1 \rangle \quad \text{and} \quad F_2 = \langle Q_2, \Sigma, \delta_2, q_2, A_2 \rangle$$

holds. We define F as the generalized Cartesian product of F_1 und F_2 as follows:

$$F := F_1 \times F_2 := \langle Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, \langle q_1, q_2 \rangle, A_1 \times A_2 \rangle,$$

where the state transition function

$$\delta_1 \times \delta_2 : (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$$

is defined as

$$(\delta_1 \times \delta_2)\big(\langle p_1, p_2 \rangle, c\big) := \big\langle \delta_1(p_1, c), \delta_2(p_2, c) \big\rangle.$$

Effectively, the FSM F runs the FSM F_1 and F_2 in parallel. In order to do so, the states of F are pairs of the form $\langle p_1, p_2 \rangle$ where p_1 is a state from F_1 and p_2 is a state from F_2 and the transition function $\delta_1 \times \delta_2$ computes the state that follows on $\langle p_1, p_2 \rangle$ when a character c is read, by simultaneously computing the next states of p_1 and p_2 in the FSMs F_1 and F_2 when these FSMs read the character c. A string s is accepted by F if and only if both F_1 and F_2 accept s. Therefore, the set A of accepting states of the FSM F is defined as:

$$A := \big\{ \langle p_1, p_2 \rangle \in Q_1 \times Q_2 \mid p_1 \in A_1 \wedge p_2 \in A_2 \big\} = A_1 \times A_2.$$

Then for all $s \in \Sigma^*$ we have:

$$\begin{array}{ll} & s \in L(F) \\ \Leftrightarrow & (\delta_1 \times \delta_2)(\langle q_1, q_2 \rangle, s) \in A \\ \Leftrightarrow & \langle \delta_1(q_1, s), \delta_2(q_2, s) \rangle \in A_1 \times A_2 \\ \Leftrightarrow & \delta_1(q_1, s) \in A_1 \wedge \delta_2(q_2, s) \in A_2 \\ \Leftrightarrow & s \in L(F_1) \wedge s \in L(F_2) \\ \Leftrightarrow & s \in L(F_1) \cap L(F_2) \\ \Leftrightarrow & s \in L_1 \cap L_2 \end{array}$$

Therefore we have shown that

$$L(F) = L_1 \cap L_2$$

and this completes the proof. □

Remark: In principle it would be possible to define a function

$$\wedge : \mathit{RegExp} \times \mathit{RegExp} \to \mathit{RegExp}$$

that takes two regular expressions r_1 and r_2 and returns a regular expression $r_1 \wedge r_2$ such that we have

$$L(r_1 \wedge r_2) = L(r_1) \cap L(r_2).$$

To compute $r_1 \wedge r_2$ we would first compute non-deterministic FSMs F_1 and F_2 such that

$$L(F_1) = L(r_1) \quad \text{and} \quad L(F_2) = L(r_2)$$

holds. Then we would transform the FSMs F_1 and F_2 into equivalent deterministic FSMs $\texttt{det}(F_1)$ and $\texttt{det}(F_2)$. After that, we would build the extended Cartesian product of $\texttt{det}(F_1)$ and $\texttt{det}(F_2)$ as shown above. Finally, we would convert the FSM $\texttt{det}(F_1) \times \texttt{det}(F_2)$ into an equivalent regular expression. However, most of the time the resulting regular expression would be absurdly large. Therefore, it is not practical to implement the function $\wedge$. $\diamond$

Proposition 15 If L is a regular language with the alphabet Σ, then the complement of L, which is defined as the language $\Sigma^* \backslash L$, is regular.

Proof: We assume that r is a regular expression describing the language L, i.e. $L = L(r)$. We construct a non-deterministic FSM F such that $L(F) = L(r) = L$. We transform this non-deterministic FSM into the deterministic FSM $\texttt{det}(F)$ as discussed in the previous chapter. Assume that we have

$$\texttt{det}(F) = \langle Q, \Sigma, \delta, q_0, A \rangle.$$

We define the deterministic FSM $\overline{\texttt{det}(F)}$ as follows:

$$\overline{\texttt{det}(F)} = \langle Q, \Sigma, \delta, q_0, Q \backslash A \rangle.$$

Then we have

$$\begin{array}{lll} & w \in L\Big(\overline{\texttt{det}(F)}\Big) \\ \Leftrightarrow & \delta(q_0, w) \in Q \backslash A \\ \Leftrightarrow & \delta(q_0, w) \notin A \\ \Leftrightarrow & w \notin L\big(\texttt{det}(F)\big) \\ \Leftrightarrow & w \notin L(F) \\ \Leftrightarrow & w \notin L(r) \\ \Leftrightarrow & w \notin L \\ \Leftrightarrow & w \in \Sigma^* \backslash L \end{array}$$

This shows that the language $\Sigma^* \backslash L$ is accepted by the FSM $\overline{\texttt{det}(F)}$ and hence it is a regular language. $\square$

Corollary 16 If L_1 and L_2 are regular languages over the common alphabet Σ, then the set difference $L_1 \backslash L_2$ is a regular language.

Proof: A string w is a member of $L_1 \backslash L_2$ iff w is a member of L_1 and w is also a member of the complement of L_2. Therefore we have

$$L_1 \backslash L_2 = L_1 \cap (\Sigma^* \backslash L_2),$$

The previous proposition shows that for a regular language L_2 the complement $\Sigma^* \backslash L_2$ is also a regular language. Since the intersection of regular languages is regular, too, we see that $L_1 \backslash L_2$ is also regular. $\square$

All in all we have now shown that regular languages are closed under Boolean set operations.

Exercise 14: Assume Σ to be some alphabet. For a string $s = c_1 c_2 \cdots c_{n-1} c_n \in \Sigma^*$ the reversal of s is written s^R and it is defined as

$$s^R := c_n c_{n-1} \cdots c_2 c_1.$$

For example, if $s = \texttt{abc}$, then $s^R = \texttt{cba}$. The reversal L^R of a language $L \subseteq \Sigma^*$ is defined as

$$L^R := \{s^R \mid s \in L\}.$$

Next, assume that the language $L \subseteq \Sigma^*$ is regular. Prove that then L^R is a regular language, too. $\diamond$

5.2 Recognizing Empty Languages

In this section we develop an algorithm that takes a deterministic FSM

$$F = \langle Q, \Sigma, \delta, q_0, A \rangle$$

as its input and then checks, whether the language accepted by F is empty, i.e. it checks whether $L(F) = \{\}$. To this end we interpret the FSM F as a directed graph. The nodes of this graph are the states of the set Q and for two states q_1 and q_2 there is an edge q_1 from to q_2 iff there exists a character $c \in \Sigma$, such that $\delta(q_1, c) = q_2$. The language $L(F)$ is empty if and only if this graph has no path that starts in the state q_0 and leads to an accepting state.

Therefore, in order to answer the question whether $L(F) = \{\}$ holds, we have to compute the set R of all those states that are reachable from the start state q_0. The computation of R can be done iteratively as follows:

1. $q_0 \in R$.
2. $p_1 \in R \wedge \delta(p_1, c) = p_2 \;\rightarrow\; p_2 \in R$.

   This last step is repeated until there are no more states that can be added to the set R.

Then we have $L(F) = \{\}$ if and only if none of the accepting states is reachable, i.e. we have

$$L(F) = \{\} \;\Leftrightarrow\; R \cap A = \{\}.$$

Hence we now have an algorithm for checking whether $L(F) = \{\}$ holds: We compute the states that are reachable from the start state q_0 and then we check whether this set contains any accepting states.

Remark: If the regular language L is not specified via an FSM F, but rather is defined via a regular expression r, then there is a simple recursive algorithm for checking whether $L(r)$ is empty:

1. $L(\emptyset) = \{\}$.
2. $L(\varepsilon) \neq \{\}$.
3. $L(c) \neq \{\}$ for all $c \in \Sigma$.
4. $L(r_1 \cdot r_2) = \{\} \;\Leftrightarrow\; L(r_1) = \{\} \vee L(r_2) = \{\}$.
5. $L(r_1 + r_2) = \{\} \;\Leftrightarrow\; L(r_1) = \{\} \wedge L(r_2) = \{\}$.
6. $L(r^*) \neq \{\}$. ◇

5.3 Equivalence of Regular Expressions

In Chapter 2 we had defined two regular expressions r_1 and r_2 to be equivalent (written $r_1 \doteq r_2$), if the languages specified by r_1 and r_2 are identical:

$$r_1 \doteq r_2 \stackrel{\text{def}}{\Longleftrightarrow} L(r_1) = L(r_2).$$

In this section, we present an algorithm that receives two regular expressions r_1 and r_2 as input and then checks whether $r_1 \doteq r_2$ holds.

Theorem 17 If r_1 and r_2 are regular expressions, then the question whether $r_1 \doteq r_2$ holds is decidable.

Proof: We present an algorithm that decides whether $L(r_1) = L(r_2)$ holds. First, we observe that the sets $L(r_1)$ and $L(r_2)$ are identical iff the set differences $L(r_2) \backslash L(r_1)$ and $L(r_1) \backslash L(r_2)$ are both empty:

$$\begin{array}{lcl} L(r_1) = L(r_2) & \Leftrightarrow & L(r_1) \subseteq L(r_2) \;\wedge\; L(r_2) \subseteq L(r_1) \\ & \Leftrightarrow & L(r_1) \backslash L(r_2) = \{\} \;\wedge\; L(r_2) \backslash L(r_1) = \{\} \end{array}$$

Next, assume that F_1 and F_2 are deterministic FSMS such that

$$L(F_1) = L(r_1) \quad \text{and} \quad L(F_2) = L(r_2)$$

holds. We have seen in Chapter 4 how F_1 and F_2 can be constructed from r_1 and r_2. According to the corollary 16 the languages $L(r_1)\backslash L(r_2)$ and $L(r_2)\backslash L(r_1)$ are regular and we have seen how to construct FSMs $F_{1,2}$ and $F_{2,1}$ such that

$$L(r_1)\backslash L(r_2) = L(F_{1,2}) \quad \text{and} \quad L(r_2)\backslash L(r_1) = L(F_{2,1})$$

holds. Hence we have

$$r_1 \doteq r_2 \;\Leftrightarrow\; L(F_{1,2}) = \{\} \wedge L(F_{2,1}) = \{\}$$

and according to Section 5.2 this question is decidable by checking whether any of the accepting states of $F_{1,2}$ or $F_{2,1}$ are reachable from the start state. □

Remark: The Jupyter notebook `Equivalence.ipynb`, which is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Python/Equivalence.ipynb

implements the theory discussed in this section.

5.4 Limits of Regular Languages

In this section we present a theorem that can be used to show that certain languages are not regular. This theorem is known as the pumping lemma for regular languages.

Theorem 18 (Pumping Lemma for Regular Languages, Michael Rabin and Dana Scott 1959, [RS59])
Assume L is a regular language. Then there exists a positive natural number $n \in \mathbb{N}$ such that every string $s \in L$ that has a length of at least n can be split into three substrings u, v, and w such that the following holds:

1. $s = uvw$,
2. $v \neq \varepsilon$,
3. $|uv| \leq n$,
4. $\forall h \in \mathbb{N} : uv^h w \in L$.

This theorem can be written as a single formula: If L is a regular language, then

$$\exists n \in \mathbb{N} : \Big(n > 0 \wedge \forall s \in L : \big(|s| \geq n \rightarrow \exists u, v, w \in \Sigma^* : s = uvw \wedge v \neq \varepsilon \wedge |uv| \leq n \wedge \forall h \in \mathbb{N} : uv^h w \in L\big)\Big).$$

Proof: As L is a regular language, there exists a deterministic FSM

$$F = \langle Q, \Sigma, \delta, q_0, A \rangle,$$

such that $L = L(F)$. The number n whose existence is claimed in the Pumping Lemma is defined as the number of states of F:

$$n := \mathit{card}(Q).$$

Next, assume a string $s \in L$ is given such that $|s| \geq n$. Define $m := |s|$. Then there are m characters c_i such that

$$s = c_1 c_2 \cdots c_m.$$

Since $|s| \geq n$, we have $m \geq n$. On reading the characters c_i the FSM changes its states as follows:

$$q_0 \xmapsto{c_1} q_1 \xmapsto{c_2} q_2 \xmapsto{c_3} \cdots \xmapsto{c_m} q_m$$

and since we have $s \in L$ we conclude that q_m must be an accepting state, i.e. $q_m \in A$. As $m \geq n$ and n is the total number of states of F, not all of the states

$$q_0, q_1, q_2, \cdots, q_m$$

can be different. Because of

$$\mathit{card}\big(\{0, 1, \cdots, n\}\big) = n + 1$$

we know, that even in the list

$$[q_0, q_1, q_2, \cdots, q_n]$$

at least one state has to occur at least twice. Hence there are natural numbers $k, l \in \{0, \cdots, n\}$ such that

$$q_k = q_l \wedge k < l.$$

Next, we define the strings u, v, and w as follows:

$$u := c_1 \cdots c_k, \quad v := c_{k+1} \cdots c_l, \quad \text{and} \quad w := c_{l+1} \cdots c_m.$$

As $k < l$ we have that $v \neq \varepsilon$ and $l \leq n$ implies $|uv| \leq n$. Furthermore, we have the following:

1. Reading the string u changes the state of the FSM F from the start state q_0 to the state q_k, we have
$$q_0 \overset{u}{\longmapsto} q_k. \tag{5.1}$$
2. Reading the string v changes the state of the FSM F from the state q_k to the state q_l. As we have $q_l = q_k$, this implies
$$q_k \overset{v}{\longmapsto} q_k. \tag{5.2}$$
3. Reading the string w changes the state of the FSM F from the state $q_l = q_k$ to the accepting state q_m:
$$q_k \overset{w}{\longmapsto} q_m. \tag{5.3}$$

From $q_k \overset{v}{\longmapsto} q_k$ we conclude

$$q_k \overset{v}{\longmapsto} q_k \overset{v}{\longmapsto} q_k, \quad \text{hence} \quad q_k \overset{v^2}{\longmapsto} q_k.$$

As we can repeat reading v in state q_k any number of times, we have

$$q_k \overset{v^h}{\longmapsto} q_k \quad \text{for all } h \in \mathbb{N}. \tag{5.4}$$

Combining the equations (5.1), (5.3), and (5.4) we have

$$q_0 \overset{u}{\longmapsto} q_k \overset{v^h}{\longmapsto} q_k \overset{w}{\longmapsto} q_m.$$

This can be condensed to

$$q_0 \overset{uv^hw}{\longmapsto} q_m$$

and since the state q_m is an accepting state we conclude that $uv^hw \in L$ holds for any $h \in \mathbb{N}$. □

Proposition 19 The alphabet Σ is defined as $\Sigma = \{$ "a" , "b" $\}$. Define the language L as the set of all strings of the form $\mathsf{a}^k\mathsf{b}^k$ where k is some natural number:

$$L = \big\{\mathsf{a}^k\mathsf{b}^k \mid k \in \mathbb{N}\big\}.$$

Then the language L is not regular.

Proof: The proof is a proof by contradiction. We assume that L is a regular language. According to the Pumping Lemma there exists a fixed natural number $n > 0$ such that every $s \in L$ that satisfies $|s| \geq n$ can be written as

$$s = uvw$$

where, furthermore, we have that

$$|uv| \leq n, \quad v \neq \varepsilon, \quad \text{and} \quad \forall h \in \mathbb{N} : uv^hw \in L$$

holds. Let us define the string s as

$$s := \mathtt{a}^n\mathtt{b}^n.$$

Obviously we have $|s| = 2 \cdot n \geq n$. Hence there are strings u, v, and w such that

$$\mathtt{a}^n\mathtt{b}^n = uvw, \quad |uv| \leq n, \quad v \neq \varepsilon, \quad \text{and} \quad \forall h \in \mathbb{N} : uv^hw \in L.$$

As $|uv| \leq n$, the string uv is a prefix not only of s but even of $\mathtt{a}^n$. Therefore, and since $v \neq \varepsilon$ we know that the string v must have the form

$$v = \mathtt{a}^k \quad \text{for some } k \in \mathbb{N}.$$

If we take the formula $\forall h \in \mathbb{N} : uv^hw \in L$ and set $h := 0$, we conclude that

$$uw \in L. \tag{5.5}$$

In order to facilitate our argument, we define the function

$$\mathit{count} : \Sigma^* \times \Sigma \to \mathbb{N}.$$

Given a string t and a character c the function $\mathit{count}(t, c)$ counts how often the character c occurs in the string t. For the language L we have

$$t \in L \Rightarrow \mathit{count}(t, \text{“}\mathtt{a}\text{”}) = \mathit{count}(t, \text{“}\mathtt{b}\text{”}).$$

On one hand we have:

$$\begin{array}{lcl}
\mathit{count}(uw, \text{“}\mathtt{a}\text{”}) & = & \mathit{count}(uvw, \text{“}\mathtt{a}\text{”}) - \mathit{count}(v, \text{“}\mathtt{a}\text{”}) \\
& = & \mathit{count}(s, \text{“}\mathtt{a}\text{”}) - \mathit{count}(v, \text{“}\mathtt{a}\text{”}) \\
& = & \mathit{count}(\mathtt{a}^n\mathtt{b}^n, \text{“}\mathtt{a}\text{”}) - \mathit{count}(\mathtt{a}^k, \text{“}\mathtt{a}\text{”}) \\
& = & n - k \\
& < & n
\end{array}$$

But on the other hand we have

$$\begin{array}{lcl}
\mathit{count}(uw, \text{“}\mathtt{b}\text{”}) & = & \mathit{count}(uvw, \text{“}\mathtt{b}\text{”}) - \mathit{count}(v, \text{“}\mathtt{b}\text{”}) \\
& = & \mathit{count}(s, \text{“}\mathtt{b}\text{”}) - \mathit{count}(v, \text{“}\mathtt{b}\text{”}) \\
& = & \mathit{count}(\mathtt{a}^n\mathtt{b}^n, \text{“}\mathtt{b}\text{”}) - \mathit{count}(\mathtt{a}^k, \text{“}\mathtt{b}\text{”}) \\
& = & n - 0 \\
& = & n
\end{array}$$

Therefore, we have

$$\mathit{count}(uw, \text{“}\mathtt{a}\text{”}) < \mathit{count}(uw, \text{“}\mathtt{b}\text{”})$$

and this shows that the string uw is not a member of the language L because for all strings in L the number of occurrences of the character “a” is the same as the number of occurrences of the character “b”. This contradiction shows that the language L cannot be regular. □

Exercise 15: Let us define the language L as the set of all those strings that contain an equal number of occurrences of the characters “a” and “b”:

$$L := \left\{w \in \{\texttt{'a'}, \texttt{'b'}\}^* \;\middle|\; \texttt{count}(w, \texttt{'a'}) = \texttt{count}(w, \texttt{'b'})\right\}$$

Prove that the language L is not regular. ◇

Remark: As the language L defined in the previous exercise is not regular we can conclude that regular expressions are unable to check even such simple questions as to whether the parentheses in an expression are balanced. Therefore, the concept of regular expressions is not strong enough to describe the syntax of a programming language. The next chapter introduces the notion of context-free languages. These languages are powerful enough to describe modern programming languages.

Exercise 16: The language L_{square} is the set of all strings of the form $\mathtt{a}^n$ where n is a square, we have

$$L_{\text{square}} = \left\{\mathtt{a}^m \;\middle|\; \exists k \in \mathbb{N} : m = k^2\right\}$$

Prove that the language L_{square} is not a regular language. ◇

Solution: The proof is a proof by contradiction. We assume that L_{square} is a regular language. According to the Pumping Lemma there exists an natural number $n > 0$ such that every string $s \in L_{\text{square}}$ such that $|s| \geq n$ can be split into three parts u, v, and w such that we have:

1. $s = uvw$,
2. $|uv| \leq n$,
3. $v \neq \varepsilon$,
4. $\forall h \in \mathbb{N} : uv^h w \in L_{\text{square}}$.

Let us define $s := a^{n^2}$. We have $s \in L_{\text{square}}$ and we see that

$$|s| = \left|a^{n^2}\right| = n^2 \geq n.$$

Hence there have to be strings u, v and w such that $s = uvw$ and u, v, and w have the properties specified above. As 'a' is the only character that occurs in s, the strings u, v, and w also contain only this character. Hence there must be natural numbers x, y, and z such that

$$u = a^x,\ v = a^y \text{ und } w = a^z$$

holds. Then we have the following.

(a) The partition $s = uvw$ has the form $a^{n^2} = a^x a^y a^z$ and hence we have

$$n^2 = x + y + z. \tag{5.6}$$

(b) The inequality $|uv| \leq n$ implies $x + y \leq n$, which implies

$$y \leq n. \tag{5.7}$$

(c) From the condition $v \neq \varepsilon$ we get

$$y > 0. \tag{5.8}$$

(d) The formula $\forall h \in \mathbb{N} : uv^h w \in L_{\text{square}}$ implies

$$\forall h \in \mathbb{N} : a^x a^{y \cdot h} a^z \in L_{\text{square}}. \tag{5.9}$$

In particular, this must hold for $h = 2$:

$$a^x a^{y \cdot 2} a^z \in L_{\text{square}}.$$

According to the definition of L_{square} there is a natural number k such that

$$x + 2 \cdot y + z = k^2. \tag{5.10}$$

If we add y on both sides of equation (5.6) we get

$$n^2 + y = x + 2 \cdot y + z = k^2.$$

Because of $y > 0$ this implies

$$n < k. \tag{5.11}$$

On the other hand we have

$$\begin{array}{lcll} k^2 & = & x + 2 \cdot y + z & \text{according to (5.10)} \\ & = & x + y + z + y & \\ & \leq & x + y + z + n & \text{according to (5.7)} \\ & = & n^2 + n & \text{according to (5.6)} \\ & < & n^2 + 2 \cdot n + 1 & \text{since } n + 1 > 0 \\ & = & (n+1)^2 & \end{array}$$

This shows that $k^2 < (n+1)^2$ holds and therefore we have

$$k < n + 1. \tag{5.12}$$

The inequalities (5.11) and (5.12) imply

$$n < k < n + 1.$$

Since k is a natural number and n is also a natural number this is a contradiction because there is no natural number between n and $n + 1$. □

Exercise 17: Define $\Sigma := \{\mathsf{a}\}$. Prove that the language

$$L := \{\mathsf{a}^p \mid p \text{ is a prime number}\}$$

is not regular. ◇

Proof: The proof is done by contradiction. Assume that L is regular. According to the Pumping Lemma there is a positive natural number n such that all strings $s \in L$ that have a length of at least n can be written as

$$s = uvw$$

where, furthermore, the substrings u, v, and w have the following properties:

1. $v \neq \varepsilon$,
2. $|uv| \leq n$, and
3. $\forall h \in \mathbb{N} : uv^h w \in L$.

Let us choose a prime number p such that $p \geq n + 2$. Since there are infinitely many prime numbers, this is always possible. Next, define $s := \mathsf{a}^p$. Then we have $|s| = p \geq n$ and we can use the Pumping Lemma to conclude that there must be substrings u, v, and w such that

$$\mathsf{a}^p = uvw$$

holds. Hence the substrings u, v, and w can only contain the letter "a". Therefore there exist natural numbers x, y, and z such that

$$u = \mathsf{a}^x, \quad v = \mathsf{a}^y, \quad \text{and} \quad w = \mathsf{a}^z.$$

We can conclude that x, y, and z have the following properties:

1. $x + y + z = p$,
2. $y \neq 0$,
3. $x + y \leq n$,
4. $\forall h \in \mathbb{N} : x + h \cdot y + z \in \mathbb{P}$.

   Here, $\mathbb{P}$ denotes the set of prime numbers.

If we substitute $h := x + z$ in the last formula, we get

$$x + (x + z) \cdot y + z \in \mathbb{P}.$$

Because of $x + (x + z) \cdot y + z = (x + z) \cdot (1 + y)$ this leads to

$$(x + z) \cdot (1 + y) \in \mathbb{P}.$$

This is impossible. As $y > 0$ the factor $1 + y$ is different from 1 and because of $x + y \leq n$, $x + y + z = p$, and $p \geq n + 2$ it follows that $z \geq 2$. Hence the factor $x + z$ is also greater than 1. But then the product $(x + z) \cdot (1 + y)$ is not a prime number and this contradiction shows that L can not be regular. □

Exercise 18: The language L_{power} is the set of all strings of the form $\mathtt{a}^n$ where n is a power of 2, i.e. we have

$$L_{\text{power}} = \left\{ \mathtt{a}^{2^k} \mid k \in \mathbb{N} \right\}$$

Prove that the language L_{power} is not regular. ◇

5.5 Check your Understanding

(a) What are the closure properties of regular languages?

(b) How can we check whether two regular expressions are equivalent?

(c) Specify the exact wording of the the Pumping Lemma.

(d) Can you prove that the language

$$L := \left\{ w \in \{\texttt{'a'}, \texttt{'b'}\}^* \;\middle|\; \texttt{count}(w, \texttt{'a'}) = \texttt{count}(w, \texttt{'b'}) \right\}$$

is not regular?

Chapter 6

Context-Free Languages

In this chapter we present the notion of a *context-free language*. This concept is much more powerful than the notion of a regular language. The syntax of most modern programming languages can be described by a context-free language. Furthermore, checking whether a string is a member of a context-free language structures the string into a recursive structure known as a parse tree. These parse trees are the basis for understanding the meaning of a string that is to be interpreted as a program fragment. A program that checks whether a given string is an element of a context-free language is called a parser. The task of a parser is to build a parse tree from a given string. Parsing is therefore the first step in an interpreter or a compiler. In this chapter, we first define the notion of context-free languages. Next, we discuss parse trees. We conclude this chapter by introducing top down parsing, which is one of the less complex algorithms that are available for parsing a string.

6.1 Context-Free Grammars

Context-free languages are used to describe programming languages. Context-free languages are formal languages like regular languages, but they are much more expressive than regular languages. When we process a program, we not only want decide whether the program is syntactically correct, but we also want to understand the *structure* of the program. The process of *structuring* is also referred to as parsing and the program that does this structuring is called a parser. As input a parser usually does not receive the text of a program, but instead a sequence of so-called terminals, which are also called tokens. These tokens are generated by a scanner, which uses regular expressions to split the program text into single words, which we call tokens in this context.

The parser receives a sequence of tokens from the scanner and has the task of constructing a so-called syntax tree. For this purpose the parser uses a grammar which specifies how the input is to be structured. As an example, consider parsing arithmetic expressions. We define the set *ArithExpr* of arithmetic expressions inductively. In order to correctly represent the structure of arithmetic expressions, we define simultaneously the sets *Product* and *Factor*. The set *Product* contains arithmetic expressions that are expressions representing products and quotients and the set *Factor* contains numbers and parenthesized expressions. The definition of these additional sets is necessary in order to get the precedences of the operators right. The basic building blocks of arithmetic expressions are variables, numbers, the operator symbols "+", "−", "∗", "/", and the bracket symbols "(" and ")". Based on these symbols the inductive definition of the sets *Factor*, *Product* and *ArithExpr* proceeds as follows:

1. Each number is a factor:

   $$C \in \mathit{number} \Rightarrow C \in \mathit{Factor}.$$

2. Each variable is a factor:

   $$V \in \mathit{variable} \Rightarrow V \in \mathit{Factor}.$$

3. If A is an arithmetic expression and we enclose this expression in parentheses we get an expression that we can use as a factor:

   $$A \in \mathit{ArithExpr} \Rightarrow \text{ "(" } A \text{ ")" } \in \mathit{Factor}.$$

A word about notation: while in the above formula A is a meta-variable that stands for an arbitrary arithmetic expression, the strings "(" and ")" are to be interpreted literally and are therefore enclosed in quotation marks. The quotation marks itself are not part of the arithmetic expression but only serve the notation.

4. If F is a factor, then F is also a product:

$$F \in \mathit{Factor} \Rightarrow F \in \mathit{Product}.$$

5. If P is a product and if F is a factor, then the strings P "*" F and P "/" F are also products:

$$P \in \mathit{Product} \wedge F \in \mathit{Factor} \Rightarrow P \text{"*"} F \in \mathit{Product} \ \wedge \ P \text{"/"} F \in \mathit{Product}.$$

6. Each product is also an arithmetic expression

$$P \in \mathit{Product} \Rightarrow P \in \mathit{ArithExpr}.$$

7. If A is an arithmetic expression and P is a product, then the strings A "+" P and A "–" P are arithmetic expressions:

$$A \in \mathit{ArithExpr} \wedge P \in \mathit{Product} \Rightarrow A \text{"+"} P \in \mathit{ArithExpr} \ \wedge \ A \text{"–"} P \in \mathit{ArithExpr}.$$

The rules given above define the sets *Factor*, *Product* and *ArithExpr* by mutual recursion. We can write this definition in terms of so-called grammar rules much more more compactly:

$$\begin{array}{rcl}
\mathit{arithExpr} & \to & \mathit{arithExpr} \text{ "+" } \mathit{product} \\
\mathit{arithExpr} & \to & \mathit{arithExpr} \text{ "–" } \mathit{product} \\
\mathit{arithExpr} & \to & \mathit{product} \\
\mathit{product} & \to & \mathit{product} \text{ "*" } \mathit{factor} \\
\mathit{product} & \to & \mathit{product} \text{ "/" } \mathit{factor} \\
\mathit{product} & \to & \mathit{factor} \\
\mathit{factor} & \to & \text{"(" } \mathit{arithExpr} \text{ ")"} \\
\mathit{factor} & \to & \text{VARIABLE} \\
\mathit{factor} & \to & \text{NUMBER}
\end{array}$$

We refer to the expressions on the left side of a grammar rule as syntactic variables or as non-terminals. All other expressions are called terminals. We write syntactic variables in lowercase, because that is the convention in the parser generators ANTLR and PLY, which we will introduce later. In the literature, however, it is often the other way around. There the syntactic variables are capitalized and the terminals are written in lower case. Occasionally a syntactic variable is also called a syntactic category.

In the example, *arithExpr*, *product* and *factor* are the syntactic variables. The remaining expressions, in our case NUMBER, VARIABLE and the characters "+", "–", "*", "/", "(" and ")" are the terminals or tokens. So these are exactly the characters that do not appear on the left side of a grammar rule. There are two types of terminals:

1. Operator symbols and separators, such as "/" and "(".

   These terminals are used literally.

2. Tokens such as NUMBER or VARIABLE also have a value associated with them. In the case of NUMBER this is a number, in the case of VARIABLE this is a string containing the name of the variable. We will always write these kinds of token with capital letters to distinguish them from the syntactic variables.

Usually, grammar rules are rendered in a more compact notation than the one presented above. For our example, this notation looks like this:

$$\begin{array}{rcl}
\mathit{arithExpr} & \to & \mathit{arithExpr} \text{ "+" } \mathit{product} \mid \mathit{arithExpr} \text{ "–" } \mathit{product} \mid \mathit{product} \\
\mathit{product} & \to & \mathit{product} \text{ "*" } \mathit{factor} \mid \mathit{product} \text{ "/" } \mathit{factor} \mid \mathit{factor} \\
\mathit{factor} & \to & \text{"(" } \mathit{arithExpr} \text{ ")"} \mid \text{NUMBER} \mid \text{VARIABLE}
\end{array}$$

So here the individual alternatives of a rule are separated by the metacharacter "|". Following the example given above, we now present the formal definition for the notion of a context-free grammar.

Definition 20 (Context-Free Grammar) A context-free grammar G is a 4-tuple

$$G = \langle V, T, R, S \rangle,$$

where

1. V is a set of names which we call syntactic variable or non-terminals.

   In the example above we have

   $$V = \{\mathit{arithExpr}, \mathit{product}, \mathit{factor}\}.$$

2. T is a set of names that we call terminals. The sets T and V are disjoint, so

   $$T \cap V = \emptyset$$

   holds. In the example given above we have

   $$T = \{\text{NUMBER}, \text{VARIABLE}, \text{“+”}, \text{“-”}, \text{“*”}, \text{“/”}, \text{“(”}, \text{“)”}\}.$$

3. R is the set of grammar rules. Formally, a grammar rule is a pair of the form $\langle A, \alpha \rangle$:

   (a) The first component of this pair is a syntactic variable:

   $$A \in V.$$

   (b) The second component is a string built from syntactic variables and terminals:

   $$\alpha \in (V \cup T)^*.$$

   Therefore, we have

   $$R \subseteq V \times (V \cup T)^*.$$

   If $\langle x, \alpha \rangle$ is a rule, we write this rule as

   $$x \to \alpha.$$

   In the example given above we have defined the first rule as

   $$\mathit{arithExpr} \to \mathit{arithExpr}\ \text{“+”}\ \mathit{product}$$

   Formally, this rule stands for the pair

   $$\big\langle \mathit{arithExpr}, [\mathit{arithExpr}, \text{“+”}, \mathit{product}] \big\rangle.$$

4. S is an element of the set V, which we define as the start symbol. In the example above, *arithExpr* is the start symbol. ◇

6.1.1 Derivations

Next, we want to determine which language is defined by a given grammar G. To do this, we first define the notion of a derivation-step. Assume that

1. $G = \langle V, T, R, S \rangle$ is a grammar,
2. $a \in V$ is a syntactic variable,
3. $\alpha a \beta \in (V \cup T)^*$ is a string of terminals and syntactic variables, containing the variable a, and
4. $(a \to \gamma) \in R$ is a rule.

Then the string $\alpha a \beta$ can be converted by a derivation step into the string $\alpha \gamma \beta$, so we replace an occurrence of the syntactic variable a by the right-hand side of the rule $a \to \gamma$. We write this derivation step as

$$\alpha a \beta \Rightarrow_G \alpha \gamma \beta.$$

If the grammar G used is clear from the context, then the index G is omitted and we write $\Rightarrow$ in place of $\Rightarrow_G$. The

transitive and reflexive closure of the relation $\Rightarrow_G$ is denoted by $\Rightarrow_G^*$. If we want to express that the derivation of the string w from the non-terminal a consists of n derivation steps we write

$$a \Rightarrow^n w.$$

We give an example:

$$\begin{array}{rcl}
\textit{arithExpr} & \Rightarrow & \textit{arithExpr}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textit{product}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textit{product}\ \text{“*”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textit{factor}\ \text{“*”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textit{factor}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textit{product} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textit{factor} \\
& \Rightarrow & \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}
\end{array}$$

So we have shown that

$$\textit{arithExpr} \Rightarrow^* \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}$$

or more precisely

$$\textit{arithExpr} \Rightarrow^8 \textsc{Number}\ \text{“*”}\ \textsc{Number}\ \text{“+”}\ \textsc{Number}$$

applies. If we replace the terminal NUMBER by different numbers, we have thus shown that the string

$$2 * 3 + 4$$

is an arithmetic expression. In general, we define the language $L(G)$ defined by a grammar G as the set of all strings which on the one hand consist only of terminals and which on the on the other hand can be derived from the start symbol S of the grammar, i.e. we have

$$L(G) := \big\{w \in T^* \mid S \Rightarrow^* w\big\}.$$

Example: The language

$$L = \{(^n)^n \mid n \in \mathbb{N}\}$$

is generated by the grammar

$$G = \big\langle \{s\}, \{\text{“(”}, \text{“)”}\}, R, s \big\rangle,$$

where the rules R are given as follows:

$$s \rightarrow\ \text{“(”}\ s\ \text{“)”}\ \mid \varepsilon.$$

Proof: We first show that each word $w \in L$ can be derived from the start symbol s:

If $w \in L$, then $s \Rightarrow^* w$.

So let $w_n = (^n)^n$. We show by induction over $n \in \mathbb{N}$ that $w_n \in L(G)$.

B.C.: $n = 0$.

We have $w_0 = \varepsilon$. Since the grammar contains the rule $s \rightarrow \varepsilon$ we have

$$s \Rightarrow \varepsilon,$$

Therefore $w_0 \in L(G)$ holds.

I.S.: $n \mapsto n + 1$.

The string w_{n+1} is of the form $w_{n+1} =$ “(” w_n “)” , where the string w_n is of course also in L. So, according to the induction hypotheses, there is a derivation of w_n:

$$s \Rightarrow^* w_n.$$

Overall, we then have the derivation

$$s \Rightarrow \text{ “(” } s \text{ “)” } \Rightarrow^* \text{ “(” } w_n \text{ “)” } = w_{n+1}.$$

Therefore $w_{n+1} \in L(G)$.

Next, we show that every word w that can be derived from s is an element of the language L. We perform the proof by induction on the number $n \in \mathbb{N}$ of the derivation steps:

B.C.: $n = 1$.

The only derivation of a string built from terminals that consists of only one step is

$$s \Rightarrow \varepsilon.$$

Consequently, $w = \varepsilon$ must hold and because of $\varepsilon = (^0)^0 \in L$ we have $w \in L$.

I.S.: $n \mapsto n + 1$.

If the derivative consists of more than one step, then the derivative must have the following form:

$$s \Rightarrow \text{ “(” } s \text{ “)” } \Rightarrow^n w$$

From this it follows that

$$w = \text{ “(” } v \text{ “)” } \quad \text{and} \quad s \Rightarrow^n v.$$

By i.h. then $v \in L$ holds. Thus there exists $k \in \mathbb{N}$ with $v = (^k)^k$. So we have

$$w = \text{ “(” } v \text{ “)” } = ((^k)^k) = (^{k+1})^{k+1} \in L.$$ □

Exercise 19: We define for $w \in \Sigma^*$ and $c \in \Sigma$ the function

$$\mathit{count}(w, c).$$

The function counts how many times the letter c occurs in the word w. The definition is done by induction on the string w.

B.C.: $w = \varepsilon$.

We set

$$\mathit{count}(\varepsilon, c) := 0.$$

I.S.: $w = dv$ with $d \in \Sigma$ and $v \in \Sigma^*$.

Then $\mathit{count}(dv, c)$ is defined by case distinction:

$$\mathit{count}(dv, c) := \begin{cases} \mathit{count}(v, c) + 1 & \text{if } c = d; \\ \mathit{count}(v, c) & \text{falls } c \neq d. \end{cases}$$ ◇

We define $\Sigma = \{\text{“A”}, \text{“B”}\}$ and define the language L as the set of words $w \in \Sigma^*$ in which the letters “A” and “B” occur with the same frequency:

$$L := \{w \in \Sigma^* \mid \mathit{count}(w, \text{“A”}) = \mathit{count}(w, \text{“B”})\}$$

Give a grammar G such that $L = L(G)$ and prove that this grammar indeed generates L. ◇

Exercise 20: Define $\Sigma := \{\text{“A”}, \text{“B”}\}$. In the previous chapter, we have already defined the reversal of a string $w = c_1 c_2 \cdots c_{n-1} c_n \in \Sigma^*$ as the string

$$w^R := c_n c_{n-1} \cdots c_2 c_1.$$

A string $w \in \Sigma^*$ is called a palindrome if the string is identical to its reversal, i.e. if

$$w = w^R$$

holds true. For example, the strings

$$w_1 = \texttt{ABABA} \quad \text{and} \quad w_2 = \texttt{ABBA}$$

are both palindromes, while the string ABB is not a palindrome. The language of palindromes $L_{\text{palindrome}}$ is the set of all strings in Σ^* that are palindromes, i.e. we have

$$L_{\text{palindrome}} := \{ w \in \Sigma^* \mid w = w^R \}.$$

(a) Prove that the language $L_{\text{palindrome}}$ is a context-free language.

(b) Prove that the language $L_{\text{palindrome}}$ is not regular. ◇

6.1.2 Parse Trees

Using a grammar G, we can not only tell whether a given string s is an element of the language $L(G)$ generated by the grammar, we can also structure the string by building a parse tree. If a grammar is

$$G = \langle V, T, R, S \rangle$$

given, a parse tree for this grammar is a tree satisfying the following conditions:

1. The tree consists of two types of nodes:
   (a) The leaf nodes are those nodes that have no outgoing edges.
   (b) The inner nodes are all those nodes that have outgoing edges.
2. Each inner node is labeled with a variable.
3. Each leaf node is labeled with a terminal or with a variable.
4. If a leaf node is labeled with a variable a, then the grammar contains a rule of the form
   $$a \to \varepsilon.$$
5. If an inner node is labeled with a variable a and the children of of this node are labeled with the symbols X_1, X_2, $\cdots$, X_n, then the grammar G contains a rule of the form
   $$a \to X_1 X_2 \cdots X_n.$$

If we read the leave nodes of a parse tree from left to right, they yield a word that is derived from the grammar G. Figure 6.1 shows a parse tree for the word "2*3+4". It is derived using the grammar given above for arithmetic expressions.

Since trees of the type shown in Figure 6.1 become too large very quickly we simplify these trees using the following rules:

1. Is n an interior node labeled with the variable A and among the children of this node there is exactly one child labeled with a terminal o then we remove this child and label the node n instead with the terminal o.
2. If an inner node has only one child, we replace that node with its child.

We call the tree obtained in this way the abstract syntax tree. Figure 6.2 shows the abstract syntax tree that results from the tree in Figure 6.1. The structure stored in this tree is exactly what we need to evaluate the arithmetic expression "2*3+4", because the tree shows us the order for evaluating the operators.

6.1.3 Ambiguous Grammars

The grammar given at the beginning of section 6.1 to describe arithmetic expressions seems very complicated because it uses three different syntactic categories: *arithExpr*, *product*, and *factor*. We introduce a simpler grammar G which describes the same language:

$$G = \big\langle \{\mathit{expr}\}, \{\text{NUMBER}, \text{VARIABLE}, \text{"+"}, \text{"-"}, \text{"*"}, \text{"/"}, \text{"("}, \text{")"}\}, R, \mathit{expr} \big\rangle,$$

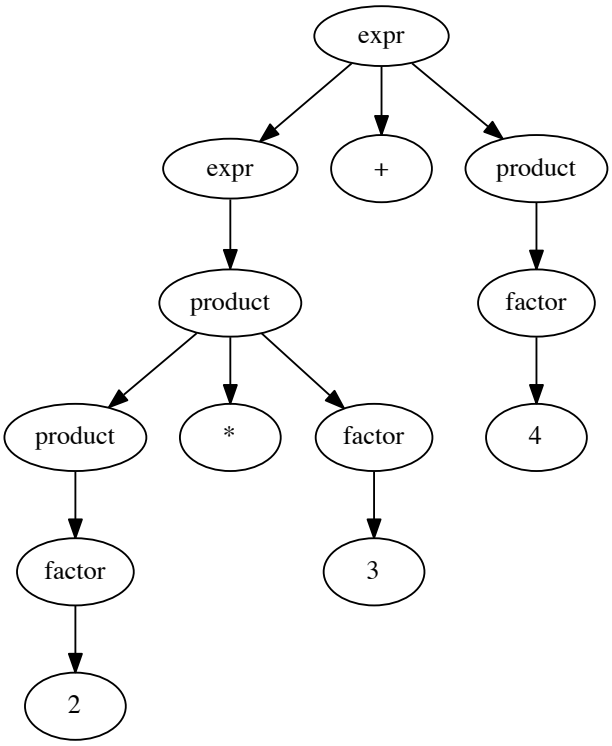

Figure 6.1: A Parse Tree for the String "2*3+4".

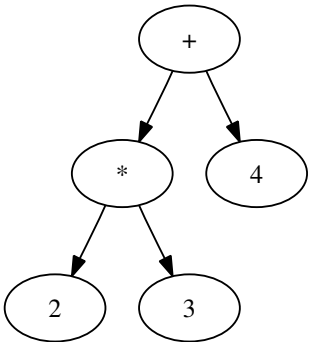

Figure 6.2: Ein abstrakter Syntax-Baum für den String "2*3+4".

The rules R are given as follows:

$$
\begin{array}{rcl}
\textit{expr} & \to & \textit{expr}\ \text{“+”}\ \textit{expr} \\
& | & \textit{expr}\ \text{“−”}\ \textit{expr} \\
& | & \textit{expr}\ \text{“*”}\ \textit{expr} \\
& | & \textit{expr}\ \text{“/”}\ \textit{expr} \\
& | & \text{“(”}\ \textit{expr}\ \text{“)”} \\
& | & \text{NUMBER} \\
& | & \text{VARIABLE}
\end{array}
$$

In order to show that the string "2*3+4" is in the grammar generated by this language, we give the following derivation:

$$\begin{aligned} \textit{expr} &\Rightarrow \textit{expr}\ \text{“+”}\ \textit{expr} \\ &\Rightarrow \textit{expr}\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\ &\Rightarrow 2\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\ &\Rightarrow 2\ \text{“*”}\ 3\ \text{“+”}\ \textit{expr} \\ &\Rightarrow 2\ \text{“*”}\ 3\ \text{“+”}\ 4 \end{aligned}$$

This derivation corresponds to the abstract syntax tree shown in Fig. 6.2 is shown. However, there is another derivation of the string "2*3+4" with this grammar:

$$\begin{aligned} \textit{expr} &\Rightarrow \textit{expr}\ \text{“*”}\ \textit{expr} \\ &\Rightarrow \textit{expr}\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\ &\Rightarrow 2\ \text{“*”}\ \textit{expr}\ \text{“+”}\ \textit{expr} \\ &\Rightarrow 2\ \text{“*”}\ 3\ \text{“+”}\ \textit{expr} \\ &\Rightarrow 2\ \text{“*”}\ 3\ \text{“+”}\ 4 \end{aligned}$$

This derivation corresponds to the abstract syntax tree shown in Fig. 6.3. In this derivation, the string "2*3+4" is apparently taken to be a product, which contradicts the convention that the operator "*" binds stronger than the operator "+". If we were to evaluate the string using the last syntax tree, we would obviously get the wrong result! The

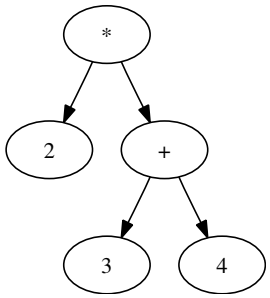

Figure 6.3: another abstract syntax tree for the string "2*3+4"

reason for this problem is the fact that the last specified grammar is ambiguous. An ambiguous grammar is unsuitable for parsing. Unfortunately, the question of whether a given grammar is ambiguous is, in general, not decidable: It can be shown that this question is equivalent to the Post correspondence problem. Since Post's correspondence problem has been shown to be undecidable, the question whether a grammar is ambiguous is also unsolvable. Proofs of these claims can be found, for example, in the book by Hopcroft, Motwani, and Ullman [HMU06].

6.2 Top-Down Parser

In this section, we present a method that can be used to conveniently parse a whole range of grammars. The basic idea is simple: In order to parse a string w using a grammar rule of the form

$$a \to X_1 X_2 \cdots X_n$$

we try to parse an X_1 first. In doing so, we decompose the string w into the form $w = w_1 r_1$ such that $w_1 \in L(X_1)$ holds. Then we try to find an X_2 in the residual string r_1, thus decomposing r_1 as $r_1 = w_2 r_2$ where $w_2 \in L(X_2)$ holds. Continuing this process, we end up with the string w being split as

$$w = w_1 w_2 \cdots w_n \quad \text{with } w_i \in L(X_i) \text{ for all } i = 1, \cdots, n.$$

Unfortunately, this procedure does not work when the grammar is left-recursive, that is, a rule has the form

$$a \to a\,\beta$$

because then to parse an a we would immediately try again to parse an a and thus we would be stuck in an infinite loop. There are two ways to deal with this kind of problem:

(a) We can rewrite the grammar so that it no longer is left-recursive.

(b) A simpler method is to extend the notion of a context-free grammar. We will use the notion of a so called extended Backus Naur form grammar (abbreviated as EBNF-grammar). Theoretically, the expressive power of EBNF grammars is the same as the expressive power of context-free grammars. In practice, however, it turns out that the construction of top-down parsers for EBNF grammars is easier, because in an EBNF grammar the left recursion can be replaced by iteration.

In the rest of this chapter we will discuss these two procedures in more detail using the grammar for arithmetic expressions as an example.

6.2.1 Rewriting a Grammar to Eliminate Left Recursion

In the following, assume that a grammar $G = \langle V, T, R, S\rangle$ is given, $a \in V$ is a syntactic variable and the Greek letters β and γ stand for any strings consisting of syntactic variables and tokens, i.e. we have $\beta, \gamma \in (V \cup T)^*$. If a is defined by the two rules

$$\begin{array}{lcl} a & \to & a\beta \\ & | & \gamma \end{array}$$

then a derivation of a, where we always replace the syntactic variable a first, has the form

$$a \Rightarrow a\beta \Rightarrow a\beta\beta \Rightarrow a\beta\beta\beta \Rightarrow \cdots \Rightarrow a\beta^n \Rightarrow \gamma\beta^n.$$

Thus we see that the language $L(a)$ described by the syntactic variable a consists of all the strings that can be derived from the expression $\gamma\beta^n$:

$$L(a) = \bigl\{w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma\beta^n \Rightarrow^* w\bigr\}.$$

This language can also be described by the following rules for a:

$$\begin{array}{lcl} a & \to & \gamma b \\ b & \to & \beta b \\ & | & \varepsilon \end{array}$$

Here we have introduced the auxiliary variable b. The derivations resulting from the syntactic variable b have the form

$$b \Rightarrow \beta b \Rightarrow \beta\beta b \Rightarrow \cdots \Rightarrow \beta^n b \Rightarrow \beta^n.$$

Hence the variable b describes the language

$$L(b) = \bigl\{w \in \Sigma \mid \exists n \in \mathbb{N} : \beta^n \Rightarrow^* w\bigr\}.$$

Thus it is clear that with the grammar rules given above we have

$$L(a) = \bigl\{w \in \Sigma^* \mid \exists n \in \mathbb{N} : \gamma\beta^n \Rightarrow^* w\bigr\}.$$

To remove the left recursion from the grammar shown in Figure 6.4 on page 66, we need to generalize the example given above. We now consider the general case and assume that a non-terminal a is defined by rules of the following form:

$$
\begin{array}{lcl}
a & \to & a\beta_1 \\
& | & a\beta_2 \\
& \vdots & \vdots \\
& | & a\beta_k \\
& | & \gamma_1 \\
& \vdots & \vdots \\
& | & \gamma_l
\end{array}
$$

We can reduce this case to the first case by introducing two auxiliary variables b and c:

$$
\begin{array}{lcl}
a & \to & ab \mid c \\
b & \to & \beta_1 \mid \cdots \mid \beta_k \\
c & \to & \gamma_1 \mid \cdots \mid \gamma_l
\end{array}
$$

Then we can rewrite this grammar by introducing a new auxiliary variable, let's call it l for list, and get

$$
\begin{array}{lcl}
a & \to & c\ l \\
l & \to & b\ l \mid \varepsilon.
\end{array}
$$

At this point, the auxiliary variables b and c can now be eliminated. This yields to the following grammar rules:

$$
\begin{array}{lcl}
a & \to & \gamma_1\ l \mid \gamma_2\ l \mid \cdots \mid \gamma_l\ l \\
l & \to & \beta_1\ l \mid \beta_2\ l \mid \cdots \mid \beta_k\ l \mid \varepsilon
\end{array}
$$

expr	→	*expr* "+" *product*
	|	*expr* "−" *product*
	|	*product*
product	→	*product* "∗" *factor*
	|	*product* "/" *factor*
	|	*factor*
factor	→	"(" *expr* ")"
	|	NUMBER

Figure 6.4: left-recursive grammar for arithmetic expressions

If we apply this procedure to the grammar for arithmetic expressions shown in Figure 6.4, we obtain the grammar shown in Figure 6.5. The variables *exprRest* and *productRest* can be interpreted as follows:

1. *exprRest* describes a list of the form.

   op product $\cdots$ *op product*,

   where *op* $\in$ { "+" , "−" }.

2. *productRest* describes a list of the form

   op factor $\cdots$ *op factor*,

   where *op* $\in$ { "∗" , "/" } holds.

$$
\begin{array}{rcl}
\textit{expr} & \rightarrow & \textit{product}\ \ \textit{exprRest} \\
\textit{exprRest} & \rightarrow & \text{“+”}\ \textit{product}\ \ \textit{exprRest} \\
 & | & \text{“−”}\ \textit{product}\ \ \textit{exprRest} \\
 & | & \varepsilon \\
\textit{product} & \rightarrow & \textit{factor}\ \ \textit{productRest} \\
\textit{productRest} & \rightarrow & \text{“*”}\ \textit{factor}\ \ \textit{productRest} \\
 & | & \text{“/”}\ \textit{factor}\ \ \textit{productRest} \\
 & | & \varepsilon \\
\textit{factor} & \rightarrow & \text{“(”}\ \textit{expr}\ \text{“)”} \\
 & | & \textsc{Number}
\end{array}
$$

Figure 6.5: Grammatik für arithmetische Ausdrücke ohne Links-Rekursion.

Exercise 21:

(a) The following grammar describes regular expressions:

$$
\begin{array}{rcl}
\textit{regExp} & \rightarrow & \textit{regExp}\ \text{“+”}\ \textit{regExp} \\
 & | & \textit{regExp}\ \textit{regExp} \\
 & | & \textit{regExp}\ \text{“*”} \\
 & | & \text{“(”}\ \textit{regExp}\ \text{“)”} \\
 & | & \textsc{letter}
\end{array}
$$

This grammar uses only the syntactic variable $\{\textit{regExp}\}$ and the following Terminals

$$\{\text{“+”}, \text{“*”}, \text{“(”}, \text{“)”}, \textsc{letter}\}.$$

Since the grammar is ambiguous, this grammar is unsuitable for parsing. Transform this grammar into an unambiguous grammar where the postfix operator “*” binds more strongly than the concatenation of two regular expressions, while the “+” operator binds weaker than concatenation. Use the grammar for arithmetic expressions as a guide and introduce suitable new syntactic variables.

(b) Remove the left recursion from the grammar created in part (a) of this task. ◇

6.2.2 Implementing a Top Down Parser in Python

Now we are ready to implement a parser for recognizing arithmetic expressions. We will use the grammar that is shown in Figure 6.5 on page 67. Before we can implement the parser, we need a scanner. We will use a hand-coded scanner that is shown in Figure 6.6 on page 68. The function `tokenize` implemented in this scanner receives a string `s` as argument and returns a list of tokens. The string `s` is supposed to represent an arithmetical expression. In order to understand the implementation, you need to know the following:

(a) We need to set the flag `re.VERBOSE` in our call of the function `findall` below because otherwise we are not able to format the regular expression `lexSpec` the way we have done it.

```
import re

def tokenize(s):
    lexSpec = r'''([ \t]+)        |  # blanks and tabs
                  ([1-9][0-9]*|0) |  # number
                  ([()])          |  # parentheses
                  ([-+*/])        |  # arithmetical operators
                  (.)                # unrecognized character
                '''
    tokenList = re.findall(lexSpec, s, re.VERBOSE)
    result    = []
    for ws, number, parenthesis, operator, error in tokenList:
        if ws:        # skip blanks and tabs
            pass
        if number:
            result += [ number ]
        if parenthesis:
            result += [ parenthesis ]
        if operator:
            result += [ operator ]
        if error:
            result += [ f'ERROR({error})']
    return result
```

Figure 6.6: A scanner for arithmetic expressions.

(b) The regular expression `lexSpec` contains 5 parenthesized groups. Therefore, `findall` returns a list of 5-tuples where the 5 components correspond to the 5 groups of the regular expression. As the 5 groups are non-overlapping, exactly one of the 5 components will be a non-empty string.

Figure 6.7 on page 69 shows an implementation of a recursive descent parser in PYTHON.

(a) The main function is the function `parse`. This function takes a string s representing an arithmetic expression. This string is tokenized using the function `tokenize`. The function `tokenize` turns a string into a list of tokens. For example, the expression

```
tokenize('(1 + 2) * 3')
```

returns the result

```
['(', 1, '+', 2, ')', '*', 3].
```

This list of tokens is then parsed by the function `parseExpr`. That function returns a pair:

(a) The first component is the value of the arithmetic expression.

(b) The second component is the list of those tokens that have not been consumed when parsing the expression. Of course, on a successful parse this list should be empty.

(b) The function `parseExpr` implements the grammar rule

$$\textit{expr} \rightarrow \textit{product}\ \ \textit{exprRest}.$$

It takes a token list `TL` as input. It will return a pair of the form

$$(v, \texttt{Rest}),$$

```
def parse(s):
    TL           = tokenize(s)
    result, Rest = parseExpr(TL)
    assert Rest == [], f'Parse Error: could not parse {TL}'
    return result

def parseExpr(TL):
    product, Rest = parseProduct(TL)
    return parseExprRest(product, Rest)

def parseExprRest(Sum, TL):
    if TL == []:
        return Sum, []
    elif TL[0] == '+':
        product, Rest = parseProduct(TL[1:])
        return parseExprRest(Sum + product, Rest)
    elif TL[0] == '-':
        product, Rest = parseProduct(TL[1:])
        return parseExprRest(Sum - product, Rest)
    else:
        return Sum, TL

def parseProduct(TL):
    factor, Rest = parseFactor(TL)
    return parseProductRest(factor, Rest)

def parseProductRest(product, TL):
    if TL == []:
        return product, []
    elif TL[0] == '*':
        factor, Rest = parseFactor(TL[1:])
        return parseProductRest(product * factor, Rest)
    elif TL[0] == '/':
        factor, Rest = parseFactor(TL[1:])
        return parseProductRest(product / factor, Rest)
    else:
        return product, TL

def parseFactor(TL):
    if TL[0] == '(':
        expr, Rest = parseExpr(TL[1:])
        assert Rest[0] == ')', 'Parse Error: expected ")"'
        return expr, Rest[1:]
    else:
        return int(TL[0]), TL[1:]
```

Figure 6.7: A top down parser for arithmetic expressions.

where v is the value of the arithmetic expression that has been parsed, while `Rest` is the list of the remaining tokens. For example, the expression

```
parseExpr(['(', 1, '+', 2, ')', '*', 3, ')', '*', 2])
```

returns the result

```
[9, [')', '*', 2]].
```

Here, the part `['(', 1, '+', 2, ')', '*', 3]` has been parsed and evaluated as the number 9 and `[')', '*', 2]` is the list of tokens that have not yet been processed.

In order to parse an arithmetic expression, the function first parses a *product* and then it tries to parse the remaining tokens as an *exprRest*. The function `parseExprRest` that is used to parse an *exprRest* needs two arguments:

(a) The first argument is the value of the product that has been parsed by the function `parseProduct`.

(b) The second argument is the list of tokens that can be used.

To understand the mechanics of `parseExpr`, consider the evaluation of

```
[1, '*', 2, '+', 3].
```

Here, the function `parseProduct` will return the result

```
(2, ['+', 3]),
```

where 2 is the result of parsing and evaluating the token list `[1, '*', 2]`, while `['+', 3]` is the part of the input token list that is not used by `parseProduct`. Next, the list `['+', 3]` needs to be parsed as the rest of an expression and then 3 needs to be added to 2.

(c) The function `parseExprRest` takes a number and a list of tokens. It implements the following grammar rules:

$$\begin{array}{lcl} \textit{exprRest} & \to & \text{“+”}\ \textit{product}\ \ \textit{exprRest} \\ & \mid & \text{“−”}\ \textit{product}\ \ \textit{exprRest} \\ & \mid & \varepsilon \end{array}$$

Therefore, it checks whether the first token is either “+” or “−”. If the token is “+”, it parses a *product*, adds the result of this product to the `sum` of values parsed already and proceeds to parse the rest of the tokens.

The case that the first token is “−” is similar to the previous case. If the next token is neither “+” nor “−”, then it could be either the token “)” or else it might be the case that the list of tokens is already exhausted. In either case, the rule

$$\textit{exprRest} \to \varepsilon$$

is used. Therefore, in that case we have not consumed any tokens and hence the input argument is already the result.

(d) The function `parseProduct` implements the rule

$$\textit{product} \to \textit{factor}\ \ \textit{exprRest}.$$

The implementation is similar to the implementation of *parseExpr*.

(e) The function `parseProductRest` implements the rules

$$\begin{array}{lcl} \textit{productRest} & \to & \text{“*”}\ \textit{factor}\ \ \textit{productRest} \\ & \mid & \text{“/”}\ \textit{factor}\ \ \textit{productRest} \\ & \mid & \varepsilon \end{array}$$

The implementation is similar to the implementation of *parseExprRest*.

(f) The function `parseFactor` implements the rules

$$\begin{array}{lcl} \textit{factor} & \rightarrow & \text{“(” } \textit{expr} \text{ “)”} \\ & | & \textsc{Number} \end{array}$$

Therefore, we first check whether the next token is “(” because in that case, we have to use the first grammar rule, otherwise we use the second.

The parser shown in Figure 6.7 does not contain any error handling. Appropriate error handling will be discussed once we have covered the theory of top-down parsing.

Exercise 22: In Exercise 21 on page 67 you have developed a grammar for regular expressions that does not contain left recursion. Implement a top down parser for this grammar. The resulting grammar should return a nested tuple that represents a regular expression. ◇

6.2.3 Implementing a Recursive Descent Parser that Uses an EBNF Grammar

The previous solution to parse an arithmetical expression was not completely satisfying: The reason is that we did not really fix the problem of left recursion but rather cured the symptoms. The underlying reason for left recursion is that context free grammars are not that convenient to describe the structure of programming languages since a description of this structure needs both recursion and iteration, but context-free grammars provide no direct way to describe iteration. Rather, they simulate iteration via recursion. Let us therefore extend the power of context-free languages slightly by admitting regular expression on the right hand side of grammar rules. These new type of grammars are known as *extended Backus Naur form* grammars, which is abbreviated as EBNF grammars. An EBNF grammar admits the operators “*”, “?”, and “+” on the right hand side of a grammar rule. The meaning of these operators is the same as when these operators are used in the regular expressions of the programming language *Python*. Furthermore, the right hand side of a grammar rule can be structured using parentheses.

$$\begin{array}{rcl} \text{expr} & \rightarrow & \text{product } \big((\texttt{'+'} \mid \texttt{'-'})\ \text{product}\big)^* \\ \text{product} & \rightarrow & \text{factor } \big((\texttt{'*'} \mid \texttt{'/'})\ \text{factor}\big)^* \\ \text{factor} & \rightarrow & \texttt{'('}\ \text{expr}\ \texttt{')'} \\ & | & \texttt{NUMBER} \end{array}$$

Figure 6.8: EBNF grammar for arithmetical expressions.

It can be shown that the languages described by EBNF grammars are still context-free languages. Therefore, these operators do not change the expressive power of context-free grammars. However, it is often much more convenient to describe a language using an EBNF grammar rather than using a context-free grammar. Figure 6.8 displays an EBNF grammar for arithmetical expressions.

Obviously, the grammar in Figure 6.8 is more concise than the context-free grammar shown in Figure 6.5 on page 67. For example, the first rule clearly expresses that an arithmetical expression is a list of products that are separated by the operators “+” and “−”.
Figure 6.9 shows a recursive descent parser that implements this grammar.

1. The function `parseExpr` recognizes a `product` in line 2. The value of this `product` is stored in the variable `result` together with the list `Rest` of those tokens that have not been consumed yet. If the list `Rest` is not empty and the first token in this list is either the operator “+” or the operator “−”, then the function `parseExpr` tries to recognize more products. These are added to or subtracted from the `result` computed so far in line 7 or 9. If there are no more products to be parsed, the `while` loop terminates and the function returns the `result` together with the list of the remaining tokens `Rest`.

2. The function `parseProduct` recognizes a `factor` in line 13. The value of this `factor` is stored in the variable `result` together with the list `Rest` of those tokens that have not been consumed yet. If the list `Rest` is not empty and the first token in this list is either the operator "*" or the operator "/", then the function `parseProduct` tries to recognize more factors. The `result` computed so far is multiplied with or divided by these factors in line 18 or 20. If there are no more products to be parsed, the `while` loop terminates and the function returns the `result` together with the list `Rest` of tokens that have not been consumed.
3. The function `parseFactor` recognizes a `factor`. This is either an expression in parentheses or a number.
   - If the first token is a an opening parenthesis, the function tries to parse an expression next. This expression has to be followed by a closing parenthesis. The tokens following this closing parenthesis are not consumed but rather are returned together with the result of evaluating the expression.
   - If the first token is a number, this number is returned together with the list of all those tokens that have not been consumed.

```
1   def parseExpr(TL):
2       result, Rest = parseProduct(TL)
3       while len(Rest) > 1 and Rest[0] in {'+', '-'}:
4           operator = Rest[0]
5           arg, Rest = parseProduct(Rest[1:])
6           if operator == '+':
7               result += arg
8           else:               # operator == '-':
9               result -= arg
10      return result, Rest
11
12  def parseProduct(TL):
13      result, Rest = parseFactor(TL)
14      while len(Rest) > 1 and Rest[0] in {'*', '/'}:
15          operator = Rest[0]
16          arg, Rest = parseFactor(Rest[1:])
17          if operator == '*':
18              result *= arg
19          else:               # operator == '/':
20              result /= arg
21      return result, Rest
22
23  def parseFactor(TL):
24      if TL[0] == '(':
25          expr, Rest = parseExpr(TL[1:])
26          assert Rest[0] == ')', "ERROR: ')' expected, got {Rest[0]}"
27          return expr, Rest[1:]
28      else:
29          assert isinstance(TL[0], int), "ERROR: Number expected, got {TL[0]}"
30          return TL[0], TL[1:]
```

Figure 6.9: A recursive descent parser for the grammar in Figure 6.8.

Exercise 23: In Exercise 21 on page 67 you have developed an EBNF grammar for regular expressions. Implement a top down parser for this grammar. The resulting grammar should return a nested tuple that represents a regular expression. ◇

Historical Notes The language ALGOL [Bac59, NBB+60] was the first programming language with a syntax that was based on an EBNF grammar.

6.3 Check your Understanding

(a) Define the concept of a context-free grammar.

(b) Assume that G is a context-free grammar. How is the language $L(G)$ defined?

(c) What is the definition of a parse tree?

(d) How do we transform a parse tree into an abstract syntax tree?

(e) What are ambiguous grammars?

(f) How can a context free grammar that contains left recursion be transformed into an equivalent grammar that does not contain left recursion.

(g) How does a top-down parser work?

(h) Why do we have to eliminate left recursion from a grammar in order to build a top-down parser?

(i) Define the notion of an EBNF grammar.

Chapter 7

Introducing ANTLR

If your task is to implement a parser, it is best to use one of the tools that is available for this purpose. The wikipedia page "Comparison of parser generators" shows the large number of parser generators that are available. A parser generator is a program that takes a grammar as its input and generates a parser that can parse according to this grammar. In my opinion, the parser generator that is both most mature[1] and most powerful is ANTLR [Par12, PHF14]. The name is short for *another tool for language recognition*[2]. ANTLR can be downloaded at

`http://www.antlr.org`.

In this lecture we will use ANTLR version 4.9.2, which was released in March 2021. The tool ANTLR is written in *Java* and therefore its main target language is *Java*. However, ANTLR can also be used to generate parsers for the programming languages `C++`, `C#`, *Python*, *JavaScript*, *Go*, and *Swift*. We will introduce ANTLR via some examples that demonstrate the most important features of this tool. For a discussion of all the features offered by ANTLR I recommend the book "The Definitive ANTLR Reference" by Terrence Parr [Par12]. However, this book only covers the generation of *Java* parsers.

7.1 A Parser for Arithmetic Expressions

We start with a parser for arithmetic expressions. This parser will do nothing fancy, it will just check whether a given string has the format of an arithmetic expression and adheres to the grammar that is shown in Figure 6.4 on page 66. If we use ANTLR we can implement this grammar as shown in Figure 7.1. We discuss the implementation line by line.

1. In line 1 the keyword `grammar` specifies the name of the grammar. In this case the grammar is called `Expr`. This grammar has to be stored in a file with the name "`Expr.g4`". The rule is that the file name has to be the same name as the name of the grammar and the file extension has to be "`.g4`".

2. Line 3 gives the grammar rule for the non-terminal *start*. In the last chapter we would have used the notation

   $$start \rightarrow expr$$

   to specify this rule. With ANTLR the left and right part of a grammar rule are separated by a colon. ANTLR terminates every grammar rule with the character "`;`".

3. Line 6–9 give the grammar rules for the non-terminal *expr*. Note that we have to enclose the terminals "+" and "−" in single quotes. The different alternatives for the non-terminal `expr` are separated by the character "|".

4. Similarly, the lines 11–14 and 16–18 show the grammar rules for the non-terminals *product* and *factor*.

[1] The first version of ANTLR became available in 1992 and ANTLR 4.0 was released in 2012.

[2] The name ANTLR can also be interpreted as an acronym for anti LR, where "LR" is short for *LR* parser. LR parsers are a kind of *bottom-up parsers*. We will discuss these parsers later in chapter 9.

```
1  grammar Expr;
2
3  start   : expr
4          ;
5
6  expr    : expr '+' product
7          | expr '-' product
8          | product
9          ;
10
11 product : product '*' factor
12         | product '/' factor
13         | factor
14         ;
15
16 factor  : '(' expr ')'
17         | NUMBER
18         ;
19
20 NUMBER  : '0'|[1-9][0-9]*;
21 WS      : [ \t\n\r] -> skip;
```

Figure 7.1: ANTLR grammar for arithmetical expressions.

5. With ANTLR the grammar and the specification of the tokens can be given in the same file. In order to be able to distinguish terminals and non-terminals, terminals have to begin with an upper case letter,[3] while non-terminals start with a lower case letter. Therefore, "NUMBER" is the name of a terminal.

6. In line 20 the lexical specification of the non-terminal NUMBER is given by a regular expression. The regular expression

   ```
   '0'|[1-9][0-9]*;
   ```

   describes a sequence of digits. This sequence can only start with the digit "0" when "0" is not followed by any other digit.

   Notice that we have to enclose the first occurrence of "0" in single quotes. On the other hand, we must not put the digits occurring in the square brackets "[" and "]" in quotes, since these occur inside a range and characters inside a range must never be quoted.

7. Line 21 defines the terminal WS, where the name WS is short for *white space*. This terminal specifies a single character that is either a blank, a tabulator, a line break, or a carriage return. The lexical specification of the terminal WS is followed by the operator "->" which in turn is followed by a semantic action. The semantic action "skip", which is executed once a white space character has been recognized, simply discards the white space character. Therefore, the net effect of line 21 is to discard all white space characters.

   In most programming languages[4], white space has no purpose other than that of separating tokens. Therefore, most ANTLR parsers will have a scanner rule that is similar to the rule shown in line 21.

To conclude, the lines 3–18 specify the grammar, while the lines 20–21 specify the lexical structure. If the grammar that is shown in Figure 7.1 is stored in the file Expr.g4 we can generate a parser by using the following command:

[3] It is a convention that the names of non-terminals consist of only upper case letters, but this is not required.

[4] Unfortunately, *Python* is an exception to this rule. Therefore, parsing *Python* is considerably harder than parsing languages like C or *Java*.

```
java -jar /usr/local/lib/antlr-4.9.2-complete.jar -Dlanguage=Python3 Expr.g4
```

Of course, this only works if the file `antlr-4.9.2-complete.jar` is stored in the directory `/usr/local/lib/`. Among others, `Antlr` will then generate the following files:

1. `ExprParser.py`

   This file contains the parser.

2. `ExprLexer.py`

   This is the scanner.

3. ANTLR generates some more files. However, these are not relevant for us.

In order to run the parser we need a driver program. Figure 7.2 shows such a program.

```
1  from ExprLexer  import ExprLexer
2  from ExprParser import ExprParser
3
4  import antlr4
5
6  def parse_string(string):
7      inputStream = antlr4.InputStream(string)
8      lexer       = ExprLexer(inputStream)
9      tokenStream = antlr4.CommonTokenStream(lexer)
10     parser      = ExprParser(tokenStream)
11     parser.start()
```

Figure 7.2: Driver program for the parser generated by ANTLR.

1. In Line 1 and 2 we import the scanner and the parser that has been generated by ANTLR.
2. Line 7 transforms the input from a string into an object of class `InputStream`. This object is then used to create the scanner `lexer`.
3. Using this scanner we create an object of class `CommonTokenStream`. This object is then fed into the parser in line 10.
4. The parser is called in line 11. In order to call the parser we have to invoke the method `start`. Here `start` is the name of the non-terminal that is to be recognized

If we want to test our parser we use the command:

```
parse_string("1 + 2 * 3 - 4")
```

This will run without errors, showing that the input string adheres to the specification of the grammar. If we want to see the parse tree instead, we have to use the `TestRig` provided by ANTLR. In order to use the `TestRig`, we first have to create a *Java*-based parser using the command

```
java -jar /usr/local/lib/antlr-4.9.2-complete.jar -Dlanguage=Java Expr.g4
```

The generated *Java* files have to be compiled with the following command:

```
javac -cp .:/usr/local/lib/antlr-4.9.2-complete.jar *.java
```

After that we can generate a parse tree using the command

```
java -cp .:/usr/local/lib/antlr-4.9.2-complete.jar org.antlr.v4.gui.TestRig Expr start -gui
```

This command will open a tree viewer that shows the parse tree of any input we have typed in response to this command. For the input string "`1 + 2 * 3 - 4`" this tree looks as shown in Figure 7.3.

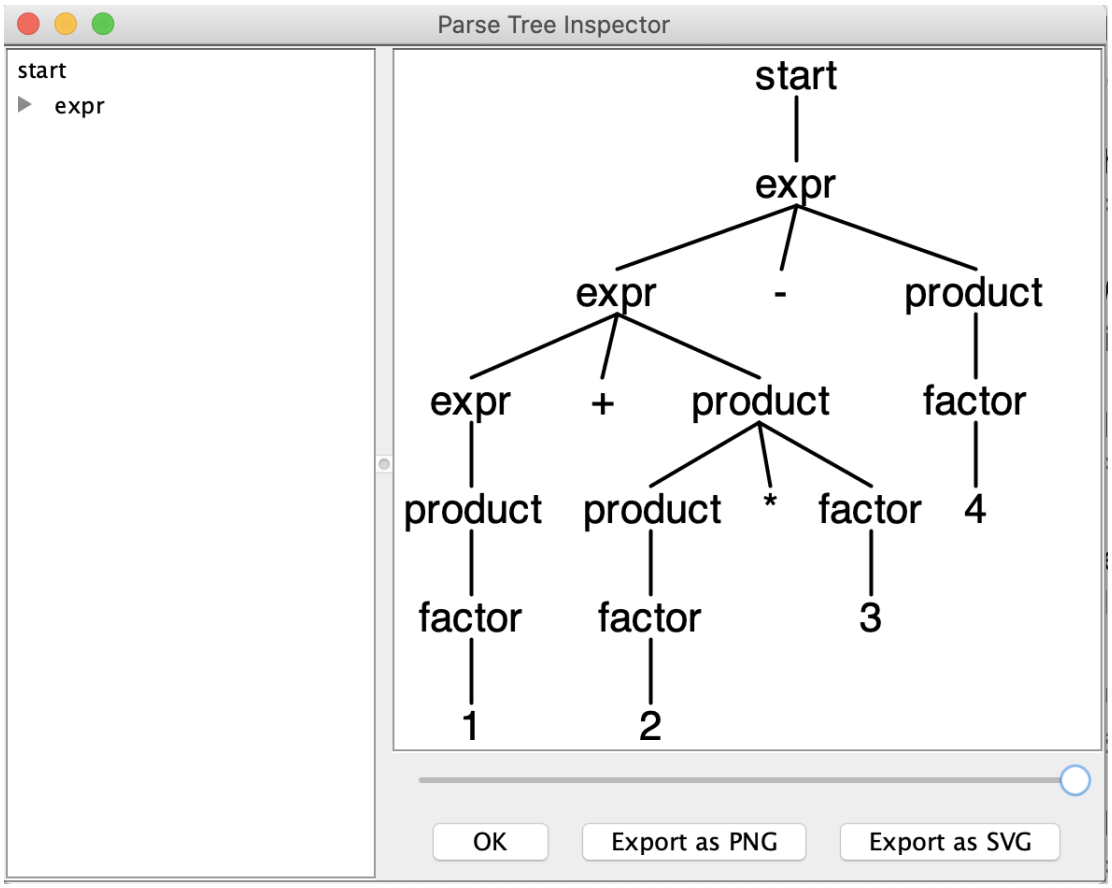

Figure 7.3: Parse tree for the string "`1 + 2 * 3 - 4`".

7.2 Evaluation of Arithmetical Expressions

The last example isn't very exciting as the arithmetical expressions that the parser reads are not evaluated. In this section we show how arithmetical expressions can be evaluated using ANTLR generated parsers. We proceed in two steps:

1. Firstly, we present a grammar for a small symbolic calculator.

   Here, the attribute *symbolic* means that the calculator supports the use of variables.

2. Secondly, we show how this grammar can be extended with actions so that the expressions can be evaluated.

Figure 7.4 presents the grammar. In comparison with grammar for arithmetical expressions that was shown in Figure 7.1 there are the following changes:

1. The start-symbol `start` now recognizes a list of statements. Note that ANTLR provides the postfix operator "+" to specify non-empty sequences. The postfix operators "*" and "?" are also supported. They have the same semantics as in regular expressions.

2. A `statement` is either an assignment or an expression.

```
grammar Program;

start: statement+ ;

statement
    : IDENTIFIER ':=' expr ';'
    | expr ';'
    ;

expr: expr '+' product
    | expr '-' product
    | product
    ;

product
    : product '*' factor
    | product '/' factor
    | factor
    ;

factor
    : 'sqrt' '(' expr ')'
    | '(' expr ')'
    | FLOAT
    | IDENTIFIER

    ;

IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*;
FLOAT     : '0'([.][0-9]+)?
          | [1-9][0-9]*([.][0-9]+)?
          ;
WS        : [ \t\n\r] -> skip;
```

Figure 7.4: A grammar for a symbolic calculator.

3. The rules for expressions and products are the same as previously.
4. The rule for the non-terminal `factor` has changed.
   (a) Expressions are now allowed to contain calls of the function `sqrt`, which is supposed to compute the square root of a given number.
   (b) We can still put expressions in parenthesis.
   (c) Instead of integers the grammar now supports floating point numbers. These are recognized by the terminal `FLOAT`.
   (d) Expressions can also contain variables. These are recognized by the terminal `IDENTIFIER`.
5. The terminal `IDENTIFIER` recognizes variable names. A variable name starts with a letter from the Latin alphabet, i.e. a character from the range `[a-z]`. This letter can be either upper or lower case. The remaining characters of a variable name can be either letters from the Latin alphabet, digits, or the underscore.

6. The terminal FLOAT recognizes floating point numbers, i.e. numbers that have an optional fractional part like 1.23. Note that the dot "." has to be enclosed in square brackets because otherwise it would match any character that is different from a newline.

A parser for this grammar is able to parse strings like the following:

```
x := 2.3 * 3.4; y := sqrt(2 * x); z := x * x + y * y; sqrt(z);
```

Our next task is to develop an interpreter for the language specified by the grammar shown in Figure 7.4. Figure 7.5 shows how an interpreter can be implemented.

```
1  grammar Calculator;
2
3  @header {
4  import math
5  }
6
7  start: statement+ ;
8
9  statement
10     : IDENTIFIER ':=' expr ';' {self.Values[$IDENTIFIER.text] = $expr.result}
11     | expr ';'                 {print($expr.result)                          }
12     ;
13
14 expr returns[result]
15     : e=expr '+' p=product {$result = $e.result + $p.result}
16     | e=expr '-' p=product {$result = $e.result - $p.result}
17     | p=product            {$result = $p.result            }
18     ;
19
20 product returns[result]
21     : p=product '*' f=factor {$result = $p.result * $f.result}
22     | p=product '/' f=factor {$result = $p.result / $f.result}
23     | f=factor               {$result = $f.result            }
24     ;
25
26 factor returns[result]
27     : '(' expr ')'        {$result = $expr.result                }
28     | FLOAT               {$result = float($FLOAT.text)          }
29     | IDENTIFIER          {$result = self.Values[$IDENTIFIER.text]}
30     | 'sqrt' '(' expr ')' {$result = math.sqrt($expr.result)     }
31     ;
32
33 IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*;
34 FLOAT     : '0'([.][0-9]+)?
35           | [1-9][0-9]*([.][0-9]+)?;
36 WS        : [ \t\n\r] -> skip;
```

Figure 7.5: An interpreter for evaluating arithmetical expressions.

1. As we have to use the function sqrt that is located in the module math we need to import this module into our parser. This is achieved by the header declaration shown in line 3–5. The header declaration is started

with the keyword `header` that is followed by an opening brace. It ends at the matching closing brace. ANTLR puts all the code that is between the braces at the beginning of the generated parser.

Note that we have to be careful to **not** indent the code. The reason is that *Python* is very picky about indentation.

2. If the parser recognizes an assignment of the form

   `IDENTIFIER` := *expr*;

   it evaluates the expression *expr* and then stores the result of this evaluation in the dictionary `Values`. In order to do this, the grammar rule is followed by some action code that is enclosed in curly braces. This code will be executed once the parser has recognized the assignment.

   The dictionary `Values` that is used to store the values assigned to variable names is a member of the parser object that is generated by ANTLR. We can refer to this object via the variable `self`. The value that is computed for the expression *expr* is available as the member `$expr.result`. The fact that this member has the name `result` is due to the `returns`-specification in line 14.

3. Line 11 deals with the case where the parser has seen an expression followed by a semicolon. In this case, the result of the evaluation of the expression is printed.

4. The `returns`-declaration in line 14 specifies that when the parser reads an expression, i.e. a string that has the syntactical form specified by the grammar rule for `expr` it will return the result of this evaluation in the member variable `result`.

5. Line 15 deals with the case that the parser has found a sum of the form

   expr '+' *product*

   In order to evaluate this expression, the parser has to compute the sum of the *expr* and the *product*. The notation "`e=expr`" assigns the result of evaluating *expr* to the variable `e` and "`p=product`" assigns the result of evaluating *product* to the variable `p`. The action code adds these variables and assigns the sum to the variable `result`. Note that all these variable names have to prefixed by a dollar symbol.

6. Line 29 shows how a string representing a floating point number is converted into a floating point number. The expression `$FLOAT.text` references the string that is matched by the regular expression `FLOAT` defined in line 34–35.

7. Similarly, in line 29 the expression `$IDENTIFIER.text` references the string that is matched by the regular expression `IDENTIFIER` defined in line 33. This string is then used as a key in the dictionary `Values` that stores the values of the variables.

Next, we need a driver program to call the parser that ANTLR generates from the grammar shown in Figure 7.5. Figure 7.6 shows how we can utilize the parser and scanner that is generated by ANTLR.

(a) Line 6 creates a parser. Since the constructor `CalculatorParser` is called with the argument `None`, this parser is not yet connected to a scanner.

(b) Line 7 creates and sets the member variable `Values` for this parser. This member variable is a dictionary. This dictionary associates variables with their values. Initially, this dictionary is empty.

(c) Line 8 reads a line of input. As long as there still is input, the while loop in line 9 will process this input.

(d) Line 10 transforms the string that has been read into a stream that is suitable for ANTLR.

(e) Line 11 creates a scanner for this input stream.

(f) Line 12 transforms this input stream into a token stream via the previously generated scanner.

(g) Line 13 connects the token stream to the parser.

(h) Line 14 starts the parser. The parser will now consume and process the given line of input.

```
1  from CalculatorLexer  import CalculatorLexer
2  from CalculatorParser import CalculatorParser
3  import antlr4
4
5  def main():
6      parser        = CalculatorParser(None)
7      parser.Values = {}
8      line          = input('> ')
9      while line != '':
10         input_stream  = antlr4.InputStream(line)
11         lexer         = CalculatorLexer(input_stream)
12         token_stream  = antlr4.CommonTokenStream(lexer)
13         parser.setInputStream(token_stream)
14         parser.start()
15         line = input('> ')
16     return parser.Values
```

Figure 7.6: A program to utilize the generated parser.

(i) Line 15 reads the next line of input. If a non-empty line is read, the while loop proceeds.

(j) When there is no more input, line 16 returns the dictionary associating variables with values.

7.3 Generating Abstract Syntax Trees with Antlr

The evaluation of arithmetical expressions was relatively easy as it is possible to evaluate an arithmetical expression directly via semantic actions that are embedded in the grammar. If the problem is more complex than the evaluation of an expression it is usually easier to first generate an abstract syntax tree and then use the syntax tree to solve the problem at hand. We demonstrate this approach using the problem of symbolic differentiation. For example, if the task is to find the derivative of the expression $x \cdot \ln(x)$ with respect to x, then the product rule tells us that

$$\frac{\mathrm{d}}{\mathrm{d}x}\big(x \cdot \ln(x)\big) = 1 \cdot \ln(x) + x \cdot \frac{1}{x}.$$

As the arithmetical expressions that we want to differentiate contain the function symbols for the natural logarithm and for exponentiation in addition to the four arithmetical operators we have to modify the grammar given in the last section. Figure 7.7 shows the grammar that we are going to use for symbolic differentiation.

7.3.1 Implementing the Parser

Figure 7.8 shows how the grammar from Figure 7.7 is implemented with ANTLR.

1. The return specification "`returns [result]`" in line 3 specifies that the expression object that is generated when the parser parses the non-terminal `expr` has a member variable with the name `result`. When referring to this variable inside a semantic action we have to prefix the variable name with the dollar symbol as shown below.

2. Line 4 recognizes a sum of the form

   $$e + p$$

   where e is an expression and p is a product. Hence the parser has to recognize an expression e, followed by the symbol "+" followed by a product p. The abstract syntax tree when reading e is stored in the variable

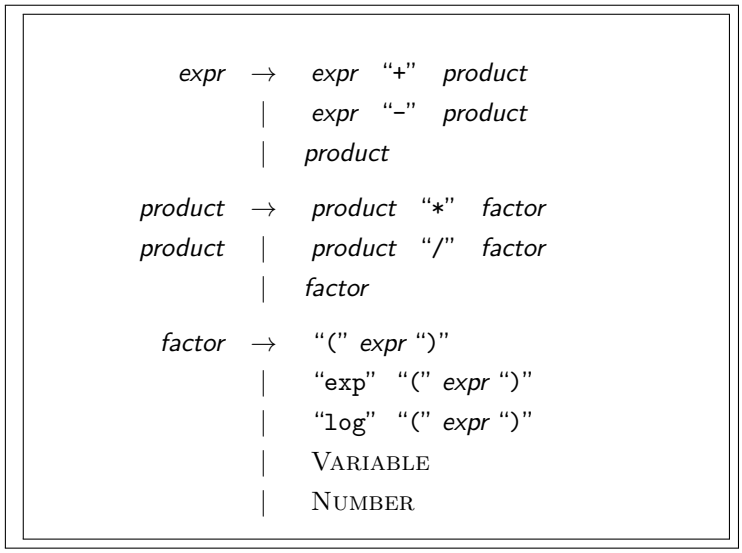

Figure 7.7: A grammar for arithmetical expressions with exponential function and logarithm.

```
1   grammar Differentiator;
2
3   expr returns [result]
4       : e=expr '+' p=product {$result = ('+', $e.result, $p.result)}
5       | e=expr '-' p=product {$result = ('-', $e.result, $p.result)}
6       | p=product            {$result = $p.result                  }
7       ;
8
9   product returns [result]
10      : p=product '*' f=factor {$result = ('*', $p.result, $f.result)}
11      | p=product '/' f=factor {$result = ('/', $p.result, $f.result)}
12      | f=factor               {$result = $f.result                  }
13      ;
14
15  factor returns [result]
16      : '(' e=expr ')'       {$result = $e.result;        }
17      | 'exp' '(' e=expr ')' {$result = ('exp', $e.result)}
18      | 'ln'  '(' e=expr ')' {$result = ('ln' , $e.result)}
19      | v=VAR                {$result = $v.text           }
20      | n=NUMBER             {$result = int($n.text)      }
21      ;
22
23  VAR    : [a-zA-Z][a-zA-Z0-9]*;
24  NUMBER : '0'|[1-9][0-9]*;
25  WS     : [ \t\n\r] -> skip;
```

Figure 7.8: ANTLR implementation of the grammar form Figure 7.7.

`$e.result`, while the syntax tree for the product is stored in the variable `$p.result`. To build a syntax tree

for the sum $e + p$ we create the triple

```
('+', $e.result, $p.result)
```

and assign this triple to the variable `$result`.

3. The remaining grammar rules work in a similar way.
4. In line 23 we can get the actual text that is matched by the terminal `VAR` by writing `$v.text`.
5. In line 24 we have to convert the string recognized by the terminal `NUMBER` into an integer by calling the function `int`.

Our second task is to implement symbolic differentiation. As we have discussed this topic already in our lecture on logic, we confine ourselves with presenting the function `diff` that is shown in Figure 7.9. This function takes a nested tuple representing the abstract syntax tree of an arithmetic expression and computes the derivative with respect to variable `x`.

```
1  def diff(e):
2      if isinstance(e, int):
3          return '0'
4      if e[0] == '+':
5          f , g  = e[1:]
6          fs, gs = diff(f), diff(g)
7          return ('+', fs, gs)
8      if e[0] == '-':
9          f , g  = e[1:]
10         fs, gs = diff(f), diff(g)
11         return ('-', fs, gs)
12     if e[0] == '*':
13         f , g  = e[1:]
14         fs, gs = diff(f), diff(g)
15         return ('+', ('*', fs, g), ('*', f, gs))
16     if e[0] == '/':
17         f , g  = e[1:]
18         fs, gs = diff(f), diff(g)
19         return ('/', ('-', ('*', fs, g), ('*', f, gs)), ('*', g, g))
20     if e[0] == 'ln':
21         f  = e[1]
22         fs = diff(f)
23         return ('/', fs, f)
24     if e[0] == 'exp':
25         f  = e[1]
26         fs = diff(f)
27         return ('*', fs, e)
28     if e == 'x':
29         return '1'
30     return '0'
```

Figure 7.9: A function to compute the derivative of a given expression.

Finally, Figure 7.10 shows how the parser can be invoked. Invoking the parser in line 10 creates an abstract syntax tree that is stored in the variable `context.result` in line 10. This abstract syntax tree is then used as input to the function `diff`.

```
def main():
    parser = DifferentiatorParser(None)
    line   = input('> ')
    while line != '':
        input_stream  = antlr4.InputStream(line)
        lexer         = DifferentiatorLexer(input_stream)
        token_stream  = antlr4.CommonTokenStream(lexer)
        parser.setInputStream(token_stream)
        term = parser.expr()
        d = diff(term.result)
        print(d)
        line = input('> ')
```

Figure 7.10: A driver program for the grammar shown in Figure 7.8

Exercise 24: The github directory associated with this lecture contains the file

Exercises/Grammar2HTML-Antlr/c-grammar.g

that specifies the syntax of the programming language C.

(a) Your first task is to create a context-free grammar that specifies the syntax used to denote the grammar rules given in the file `c-grammar.g`.

(b) Next, you should develop an ANTLR parser that is capable of reading the file `c-grammar.g`.

(c) Once you have tested this parser you should add actions to the grammar so that an abstract syntax tree is generated. The aim is to convert this abstract syntax tree into HTML.

Remark:

(a) The directory

Exercises/Grammar2HTML

contains the notebook Grammar-2-HTML.ipynb that contains a number of function that can be used to transform an abstract syntax tree of a grammar into HTML.

(b) ANTLR provides a negation operator that is written as "~". This operator is handy when matching quoted strings. For example, the token definition

```
STRING : '"' ~('"')+ '"';
```

can be used to match strings that are enclosed in double quotes.

(c) For historical reasons, ANTLR treats the string "`rule`" as a keyword. Therefore, it is not possible to have a syntactical variable that is called "`rule`".

(d) The string "`grammar`" is a keyword in ANTLR. Therefore, you can not use this string as the name of a syntactical variable either.

(e) Your solution should extend the template:

Exercises/Grammar2HTML-Antlr/Grammar.g4

The first five lines of this templates should not be changed!

7.4 Implementing a Simple Interpreter

ANTLR unterstützt EBNF-Grammatiken. Beispielsweise werden die Postfix-Operatoren "*" und "+" unterstützt. Wir demonstrieren die Verwendung dieser Operatoren, indem wir mit Hilfe des Parser-Generators ANTLR einen Interpreter für eine einfache Programmiersprache entwickeln. Abbildung 7.11 zeigt die ANTLR-Grammatik der Programmier-Sprache, für die wir in diesem Abschnitt einen Interpreter entwickeln. Die Syntax dieser Sprache ist an die Syntax der Sprache C angelehnt.

```
1  grammar Pure;
2
3  program  : statement+
4           ;
5  statement: VAR ':=' expr ';'
6           | 'print' '(' expr ')' ';'
7           | 'if'    '(' boolExpr ')' '{' statement* '}'
8           | 'while' '(' boolExpr ')' '{' statement* '}'
9           ;
10 boolExpr : expr '==' expr
11          | expr '<'  expr
12          ;
13 expr     : expr '+' product
14          | expr '-' product
15          | product
16          ;
17 product  : product '*' factor
18          | product '/' factor
19          | factor
20          ;
21 factor   : 'read' '(' ')'
22          | '(' expr ')'
23          | VAR
24          | NUMBER
25          ;
26 VAR      : [a-zA-Z][a-zA-Z_0-9]*;
27 NUMBER   : '0'|[1-9][0-9]*;
28 MULTI_COMMENT : '/*' .*? '*/' -> skip;
29 LINE_COMMENT  : '//' ~('\n')* -> skip;
30 WS           : [ \t\n\r]     -> skip;
```

Figure 7.11: ANTLR-Grammatik für eine einfache Programmier-Sprache.

Die Befehle der zu implementierenden Sprache sind Zuweisungen, Print-Befehle, `if`-Abfragen, sowie `while`-Schleifen. Abbildung 7.12 zeigt ein Beispiel-Programm, das dieser Grammatik entspricht. Dieses Programm liest zunächst eine Zahl ein, die in der Variablen `n` gespeichert wird. Anschließend wird mit Hilfe einer `while`-Schleife die Summe

$$\sum_{i=1}^{n^2} i$$

berechnet und am Ende des Programms ausgegeben.

```
n := read();
s := 0;
i := 0;
while (i < n * n) {
    i := i + 1;
    s := s + i;
}
print(s);
```

Figure 7.12: Ein Programm zur Berechnung der Summe $\sum_{i=0}^{n^2} i$.

Ähnlich wie bei unserem Programm zum symbolischen Differenzieren werden wir die einzelnen Befehle als geschachtelte Tupel darstellen. Das in Abbildung 7.12 gezeigte Programm wird dabei durch das in Abbildung 7.13 gezeigte geschachtelte Tupel dargestellt.

```
('program',
    ('read', 'n'),
    (':=', 's', 0),
    (':=', 'i', 0),
    ('while', ('<',  'i', ('*', 'n', 'n')),
        (':=', 'i', ('+', 'i', 1)),
        (':=', 's', ('+', 's', 'i'))
    ),
    ('print', 's')
)
```

Figure 7.13: Die geschachtelte Liste, die das Programm aus Abbildung 7.12 repräsentiert.

Abbildung 7.14 zeigt die Implementierung des Parsers mit dem Werkzeug ANTLR.

1. Das Start-Symbol der Grammatik ist die Variable `program`. Beim Parsen dieser Variable gibt der Parser ein Tupel von Befehlen in der Variable `stmnt_list` zurück. Dazu initialisieren wir die Variable `SL` als Liste, die nur den String `'program'` enthält. Anschließend wird jeder Befehl, der erkannt wird, an diese Liste angehängt. Schließlich wird diese Liste in ein Tupel umgewandelt und in der Attribut-Variable `stmnt_list` gespeichert.

2. Die syntaktische Variable `statement` beschreibt die verschiedenen Befehle, die in unserer einfachen Sprache unterstützt werden.

   (a) Die einfachsten Befehle sind die Zuweisungen. Diese haben die Form:

   $$v := e;$$

   Hierbei ist v eine Variable und e ist ein arithmetischer Ausdruck. Eine solche Zuweisung wird durch das geschachtelte Tupel

   $$(\text{':='}, v, e)$$

   dargestellt.

   (b) Der Befehl zur Ausgabe eines Wertes hat die Form:

   $$\texttt{print}(e);$$

   Die Aufgabe dieses Befehl ist es, den Wert des Ausdrucks e zu berechnen und auszugeben. Dieser Befehl

```
grammar Simple;

program returns [stmnt_list]
    @init {SL = []}
    : (s = statement {SL.append($s.stmnt)})+ {$stmnt_list = SL}
    ;

statement returns [stmnt]
    @init {SL = []}
    : v = VAR ':=' e = expr ';'       {$stmnt = (':=', $v.text, $e.result)}
    | 'print' '(' r = expr ')' ';'    {$stmnt = ('print', $r.result)}
    | 'if' '(' b = boolExpr ')' '{' (l = statement {SL.append($l.stmnt) })* '}'
      {$stmnt = ('if', $b.result, SL)}
    | 'while' '(' b = boolExpr ')' '{' (l = statement {SL.append($l.stmnt) })* '}'
      {$stmnt = ('while', $b.result, SL)}
    ;

boolExpr returns [result]
    : l = expr '==' r = expr {$result = ('==', $l.result, $r.result)}
    | l = expr '<'  r = expr {$result = ('<',  $l.result, $r.result)}
    ;

expr returns [result]
    : e = expr '+' p = product {$result = ('+', $e.result, $p.result)}
    | e = expr '-' p = product {$result = ('-', $e.result, $p.result)}
    | p = product              {$result = $p.result}
    ;

product returns [result]
    : p = product '*' f = factor {$result = ('*', $p.result, $f.result)}
    | p = product '/' f = factor {$result = ('/', $p.result, $f.result)}
    | f = factor                 {$result = $f.result}
    ;

factor returns [result]
    : 'read' '(' ')' {$stmnt = ('read',)}
    | '(' expr ')'   {$result = $expr.result}
    | v = VAR        {$result = $v.text}
    | n = NUMBER     {$result = int($n.text)}
    ;
```

Figure 7.14: ANTLR-Spezifikation der Grammatik.

wird durch das geschachtelte Tupel

$$(\texttt{'print'}, v)$$

dargestellt.

(c) Ein Test hat die Syntax:

$$\texttt{if (}\ b\ \texttt{) \{}\ \textit{statements}\ \texttt{\}}$$

Hierbei ist b ein Ausdruck, dessen Auswertung `True` oder `False` ergibt und `statements` ist eine Liste von

Befehlen, die ausgeführt werden, falls b den Wert `True` hat. Dieser Befehl wird durch das geschachtelte Tupel

$$(\texttt{'if'}, b, \textit{statements})$$

dargestellt.

(d) Eine Schleife hat die Syntax:

`while (` b `) {` *statements* `}`

Hierbei ist b ein Ausdruck, dessen Auswertung `True` oder `False` ergibt und `statements` ist eine Liste von Befehlen, die ausgeführt werden, solange b den Wert `True` hat. Dieser Befehl wird durch das geschachtelte Tupel

$$(\texttt{'while'}, b, \textit{statements})$$

dargestellt.

3. Die syntaktische Variable `boolExpr` beschreibt einen Ausdruck, der einen Boole'schen Wert annimmt. Es gibt zwei Möglichkeiten einen solchen Wert zu erzeugen.

   (a) Ein Ausdruck der Form

   $$l == r$$

   testet, ob die Auswertungen von l und r die selben Werte ergeben. Dieser Befehl Ausdruck wird durch das geschachtelte Tupel

   $$(\texttt{'=='}, l, r)$$

   dargestellt.

   (b) Ein Ausdruck der Form

   $$l < r$$

   testet, ob die Auswertung von l einen Wert ergibt, der kleiner ist als der Wert, der sich bei der Auswertung von r ergibt. Dieser Befehl Ausdruck wird durch das geschachtelte Tupel

   $$(\texttt{'<'}, l, r)$$

   dargestellt.

4. In analoger Weise beschreiben die Variablen `expr`, `product` und `factor` arithmetische Ausdrücke. Da wir dies bereits hinlänglich früher diskutiert haben, gehen wir an dieser Stelle nicht weiter auf die Grammatik-Regeln ein, durch die diese Variablen definiert werden.

```
41
42  VAR    : [a-zA-Z][a-zA-Z_0-9]*;
43  NUMBER : '0'|[1-9][0-9]*;
44
45  MULTI_COMMENT : '/*' .*? '*/' -> skip;
46  LINE_COMMENT  : '//' ~('\n')* -> skip;
47  WS            : [ \t\n\r]     -> skip;
```

Figure 7.15: ANTLR-Spezifikation der verschiedenen Token.

Die Spezifikation der Token ist in Abbildung 7.15 gezeigt. Der Scanner unterscheidet im Wesentlichen zwischen Variablen und Zahlen. Variablen beginnen mit einem großen oder kleinen Buchstaben, auf den dann zusätzlich Ziffern und der Unterstrich folgen können. Folgen von Ziffern werden als Zahlen interpretiert. Enthält eine solche Folge mehr als ein Zeichen, so darf die erste Ziffer nicht 0 sein. Darüber hinaus entfernt der Scanner Whitespace

und Kommentare. Außerdem haben wir bei der Spezifikation von mehrzeiligen Kommentaren die sogenannte *non-greedy*-Version des Operators "*" benutzt. Die non-greedy-Version des Operators "*" wird als "*?" geschrieben und matched sowenig wie möglich. Daher steht der reguläre Ausdruck

```
'/*' .*?  '*/'
```

für einen String, der mit der Zeichenkette "/*", mit der Zeichenkette "*/" endet und außerdem so kurz wie möglich ist. Dadurch werden in einer Zeile der Form

```
/* Hugo */ i := i + 1; /* Anton */
```

zwei getrennte Kommentare erkannt.

```
def main(file):
    with open(file, 'r') as handle:
        program_text = handle.read()
    input_stream  = antlr4.InputStream(program_text)
    lexer         = SimpleLexer(input_stream)
    token_stream  = antlr4.CommonTokenStream(lexer)
    parser        = SimpleParser(token_stream)
    result        = parser.program()
    Statements    = result.stmnt_list
    execute_tuple(Statements)
```

Figure 7.16: Die Funktion `main`.

```
def execute_tuple(Statement_List, Values={}):
    for stmnt in Statement_List:
        execute(stmnt, Values)

def execute(stmnt, Values):
    op = stmnt[0]
    if stmnt == 'program':
        pass
    elif op == ':=':
        _, var, value = stmnt
        Values[var] = evaluate(value, Values)
    elif op == 'print':
        _, expr = stmnt
        print(evaluate(expr, Values))
    elif op == 'if':
        _, test, *SL = stmnt
        if evaluate(test, Values):
            execute_tuple(SL, Values)
    elif op == 'while':
        _, test, *SL = stmnt
        while evaluate(test, Values):
            execute_tuple(SL, Values)
    else:
        assert False, f'{stmnt} unexpected'
```

Figure 7.17: Die Funktion `execute`.

Abbildung 7.16 zeigt die Funktion `main`, die als Eingabe den Namen einer Datei erhält, die ein Programm unserer einfachen Programmiersprache enthält. Dieses Programm wird geparsed und dadurch in das geschachtelte Tupel `Statements` umgewandelt. Die Funktion `execute_tuple` führt die einzelnen Befehle in dem Tupel `Statements` aus. Dazu verwendet sie die Funktion `execute`, die einen einzelnen Befehl ausführen kann. Abbildung 7.17 zeigt die Implementierung der Funktion `execute`. Diese Implementierung besteht im Wesentlichen aus eine großen Fallunterscheidung nach der Art des auszuführenden Befehls.

1. Zunächst prüfen wir, ob `statement` der String `'program'` ist, der den Beginn des Programms markiert.[5] Da es sich hier nur um eine Markierung und nicht um einen echten Befehl handelt, ist nichts weiter zu tun.

2. Falls es sich bei dem Befehl um eine Zuweisung der Form

   ```
   (':=, var, value)
   ```

   handelt, wird der Wert des Ausdrucks `value` mit Hilfe der Funktion `evaluate` berechnet. Dieser Wert wird dann in dem Dictionary `Values` unter dem Schlüssel `var` gespeichert.

3. Falls es sich bei dem Befehl um eine Leseoperation der Form

   ```
   ('read', var)
   ```

   handelt, so wird mit Hilfe der *Python*-Funktion `input` ein String gelesen. Dieser String wird in eine ganze Zahl umgewandelt. Diese Zahl wird dann in dem Dictionary `Values` unter dem Schlüssel `var` gespeichert.

4. Falls es sich bei dem Befehl um eine Operation der Form

   ```
   ('print', expr)
   ```

   handelt, so wird zunächst der Ausdruck `expr` mit Hilfe der Funktion `evaluate` ausgewertet. Der Dabei erhaltene Wert wird dann ausgegeben.

5. Falls der Befehl die Form

   ('if', test, s_1, $\cdots$, s_n)

   hat, so ist `test` ein Boole'scher Ausdruck und (s_1, $\cdots$, s_n) ist ein Tupel von Befehlen, das in der Variable `SL` gespeichert wird. In diesem Fall wird zunächst der Ausdruck `test` mit Hilfe der Funktion `evaluate` ausgewertet. Wenn diese Auswertung den Wert `True` ergibt, werden anschließend die Befehle in dem Tupel `SL` der Reihe nach ausgeführt.

6. Falls der Befehl die Form

   ('while', test, s_1, $\cdots$, s_n)

   hat, so ist `test` ein Boole'scher Ausdruck und (s_1, $\cdots$, s_n) ist ein Tupel von Befehlen, das in der Variable `SL` gespeichert wird. In diesem Fall wird zunächst der Ausdruck `test` mit Hilfe der Funktion `evaluate` ausgewertet. Wenn diese Auswertung den Wert `True` ergibt, werden anschließend die Befehle in dem Tupel `SL` der Reihe nach ausgeführt. Anschließend wird wieder der Ausdruck `test` ausgewertet. Ist das Ergebnis `False`, so ist die Auswertung des Befehls beendet. Andernfalls werden Die Befehle in der Liste `SL` solange ausgeführt, bis die Auswertung von `test` `False` ergibt.

[5] Diese Markierung wird nur für die Darstellung des Programms als abstrakter Syntaxbaum benötigt.

```
def evaluate(expr, Values):
    if isinstance(expr, int):
        return expr
    if isinstance(expr, str):
        return Values[expr]
    op = expr[0]
    if op == 'read':
        return int(input('Please enter a natural number:'))
    if op == '==':
        _, lhs, rhs = expr
        return evaluate(lhs, Values) == evaluate(rhs, Values)
    if op == '<':
        _, lhs, rhs = expr
        return evaluate(lhs, Values) < evaluate(rhs, Values)
    if op == '+':
        _, lhs, rhs = expr
        return evaluate(lhs, Values) + evaluate(rhs, Values)
    if op == '-':
        _, lhs, rhs = expr
        return evaluate(lhs, Values) - evaluate(rhs, Values)
    if op == '*':
        _, lhs, rhs = expr
        return evaluate(lhs, Values) * evaluate(rhs, Values)
    if op == '/':
        _, lhs, rhs = expr
        return evaluate(lhs, Values) / evaluate(rhs, Values)
    assert False, f'{stmnt} unexpected'
```

Figure 7.18: Die Funktion `evaluate`.

Abbildung 7.18 zeigt die Implementierung der Funktion `evaluate`. Diese Funktion erhält als Eingabe einen arithmetischen Ausdruck und ein Dictionary, in dem die Werte der Variablen abgelegt sind.

(a) Falls es sich bei dem auszuwertenden Ausdruck um eine Zahl handelt, geben wir diese Zahl als Ergebnis zurück.

(b) Falls es sich bei dem auszuwertenden Ausdruck um eine Variable handelt, so schlagen wir den Wert dieser Variable in dem Dictionary `Values` nach und geben wir diesen Wert als Ergebnis zurück.

(c) Falls es sich bei dem auszuwertenden Ausdruck um einen Aufruf der Funktion `read` handelt, fordern wir den Benutzer auf, eine natürliche Zahl einzugeben. Den vom Benutzer eingegebenen String wandeln wir dann noch in eine Zahl um.

(d) Falls es sich bei dem auszuwertenden Ausdruck um einen Boole'schen Ausdruck der Form

```
('==', lhs, rhs)
```

handelt, werten wir die Ausdrücke `lhs` und `rhs` rekursiv aus und geben genau dann `True` zurück, wenn sich für beide Ausdrücke der selbe Wert ergibt.

(e) Falls es sich bei dem auszuwertenden Ausdruck um einen Boole'schen Ausdruck der Form

```
('<', lhs, rhs)
```

handelt, werten wir die Ausdrücke `lhs` und `rhs` rekursiv aus und geben genau dann `True` zurück, wenn der Wert, der sich bei der Auswertung von `lhs` ergibt, kleiner als der Wert ist, der sich bei der Auswertung von `rhs` ergibt.

(f) Falls es sich bei dem auszuwertenden Ausdruck um eine Summe der Form

```
('+', lhs, rhs)
```

handelt, werten wir die Ausdrücke `lhs` und `rhs` rekursiv aus und geben die Summe dieser Werte zurück.

(g) Die Auswertung der arithmetischen Operatoren `'-'`, `'*'` und `'/'` verläuft analog zur Auswertung des Operators `'+'`.

Aufgabe 25:

(a) Erweitern Sie den Interpreter so, dass auch der Operator "<=" unterstützt wird.

(b) Erweitern Sie den Interpreter um `for`-Schleifen.

(c) Erweitern Sie den Interpreter um die logischen Operatoren "&&" für das logische *Und*, "||" für das logische *Oder* und "!" für die Negation. Dabei soll der Operator "!" am stärksten und der Operator "||" am schwächsten binden.

(d) Erweitern Sie die Syntax der arithmetischen Ausdrücke so, dass auch vordefinierte mathematische Funktionen wie `exp`() oder `ln`() benutzt werden können.

(e) Erweitern Sie den Interpreter so, dass auch benutzerdefinierte Funktionen möglich werden.

Hinweis: Jetzt müssen Sie zwischen lokalen und globalen Variablen unterscheiden. Daher reicht es nicht mehr, die Belegungen der Variablen in einem global definierten Dictionary zu verwalten. ◇

Chapter 8

Earley-Parser

In diesem Kapitel stellen wir ein effizientes Verfahren vor, mit dem es möglich ist, für eine beliebige vorgegebene kontextfreie Grammatik

$$G = \langle V, \Sigma, R, S \rangle \quad \text{und einen vorgegebenen String } s \in \Sigma^*$$

zu entscheiden, ob s ein Element der Sprache $L(G)$ ist, ob also $s \in L(G)$ gilt. Der Algorithmus, den wir gleich diskutieren werden, wurde 1970 von Jay Earley publiziert [Ear70]. Neben dem Algorithmus von Earley gibt es noch den Cocke-Younger-Kasami-Algorithmus, in der Literatur auch als CYK-Algorithmus bekannt, der unabhängig von John Cocke [CS70], Daniel H. Younger [You67] und Tadao Kasami [Kas65] entdeckt wurde. Der CYK-Algorithmus ist nur anwendbar, wenn die Grammatik in Chomsky-Normalform vorliegt. Da es sehr aufwendig ist, eine Grammatik in Chomsky-Normalform zu transformieren, wird der CYK-Algorithmus in der Praxis nicht eingesetzt. Demgegenüber kann der von Earley angegebene Algorithmus auf beliebige kontextfreie Grammatiken angewendet werden. Im allgemeinen Fall hat dieser Algorithmus die Komplexität $\mathcal{O}(n^3)$, aber falls die vorgegebene Grammatik eindeutig ist, dann ist die Komplexität lediglich $\mathcal{O}(n^2)$. Geschickte Implementierungen von Earley's Algorithmus erreichen für viele praktisch relevante Grammatiken sogar eine lineare Laufzeit. Im Gegensatz hat der CYK-Algorithmus **immer** die Komplexität $\mathcal{O}(n^3)$. Earley's Algorithmus hat sowohl für $LL(k)$-Grammatiken als auch für $LR(1)$-Grammatiken, die wir in einem späteren Kapitel analysieren werden, nur eine lineare Laufzeit.

Dieses Kapitel gliedert sich in die folgenden Abschnitte:

(a) Zunächst skizzieren wir die Theorie, die Earley's Algorithmus zu Grunde liegt.

(b) Danach geben wir eine einfache Implementierung des Algorithmus in *Python* an.

8.1 Der Algorithmus von Earley

Der zentrale Begriff des von Earley angegebenen Algorithmus ist der Begriff des Earley-Objekts, das wie folgt definiert ist:

Definition 21 (Earley-Objekt) Gegeben sei eine kontextfreie Grammatik $G = \langle V, \Sigma, R, S \rangle$ und ein String $s = x_1 x_2 \cdots x_n \in \Sigma^*$ der Länge n. Wir bezeichnen ein Paar der Form

$$\langle A \to \alpha \bullet \beta, k \rangle$$

genau dann als ein Earley-Objekt, falls folgendes gilt:

(a) $(A \to \alpha\beta) \in R$ und

(b) $k \in \{0, 1, \cdots, n\}$. □

Erklärung: Ein Earley-Objekt beschreibt einen Zustand, in dem ein Parser sich befinden kann. Ein Earley-Parser, der einen String $x_1 \cdots x_n$ parsen soll, verwaltet $n + 1$ Mengen von Earley-Objekten. Diese Mengen bezeichnen wir mit

$$Q_0, Q_1, \cdots, Q_n.$$

Die Interpretation von

$$\langle A \to \alpha \bullet \beta, k\rangle \in Q_j \quad \text{mit } j \geq k$$

ist dann wie folgt:

1. Der Parser versucht die Regel $A \to \alpha\beta$ auf den Teilstring $x_{k+1} \cdots x_n$ anzuwenden und am Anfang dieses Teilstrings ein A mit Hilfe der Regel $A \to \alpha\beta$ zu erkennen.
2. Am Anfang des Teilstrings $x_{k+1} \cdots x_j$ hat der Parser bereits α erkannt, es gilt also
$$\alpha \Rightarrow^* x_{k+1} \cdots x_j.$$
3. Folglich versucht der Parser am Anfang des Teilstrings $x_{j+1} \cdots x_n$ ein β zu erkennen.

Der Algorithmus von Earley verwaltet für $j = 0, 1, \cdots, n$ Mengen Q_j von Earley-Objekten, die den Zustand beschreiben, in dem der Parser ist, wenn der Teilstring $x_1 \cdots x_j$ verarbeitet ist. Zu Beginn des Algorithmus wird der Grammatik ein neues Start-Symbol $\widehat{S}$ sowie die Regel $\widehat{S} \to S$ hinzugefügt. Die Menge Q_0 wird definiert als

$$Q_0 := \big\{\langle \widehat{S} \to \bullet S, 0\rangle\big\},$$

denn der Parser soll ja das Start-Symbol S am Anfang des Strings $x_1 \cdots x_n$ erkennen. Die restlichen Mengen Q_j sind für $j = 1, \cdots, n$ zunächst leer. Die Mengen Q_j werden nun durch die folgenden drei Operationen so lange wie möglich erweitert:

1. *Lese-Operation*

   Falls der Zustand Q_j ein Earley-Objekt der Form $\langle A \to \beta \bullet a\gamma, k\rangle$ enthält, wobei a ein Terminal ist, so versucht der Parser, die rechte Seite der Regel $A \to \beta a\gamma$ zu erkennen und hat bis zur Position j bereits den Teil β erkannt. Folgt auf dieses β nun, wie in der Regel $A \to \beta a\gamma$ vorgesehen, an der Position $j+1$ das Terminal a, so muss der Parser nach der Position $j+1$ nur noch γ erkennen. Daher wird in diesem Fall das Earley-Objekt
   $$\langle A \to \beta a \bullet \gamma, k\rangle$$
   dem Zustand Q_{j+1} hinzugefügt:
   $$\langle A \to \beta \bullet a\gamma, k\rangle \in Q_j \wedge x_{j+1} = a \;\Rightarrow\; Q_{j+1} := Q_{j+1} \cup \big\{\langle A \to \beta a \bullet \gamma, k\rangle\big\}.$$
2. *Vorhersage-Operation*

   Falls der Zustand Q_j ein Earley-Objekt der Form $\langle A \to \beta \bullet C\delta, k\rangle$ enthält, wobei C eine syntaktische Variable ist, so versucht der Parser im Zustand Q_j den Teilstring $C\delta$ zu erkennen. Dazu muss der Parser an diesem Punkt ein C erkennen. Wir fügen daher für jede Regel $C \to \gamma$ der Grammatik das Earley-Objekt $\langle C \to \bullet\gamma, j\rangle$ zu der Menge Q_j hinzu:
   $$\langle A \to \beta \bullet C\delta, k\rangle \in Q_j \wedge (C \to \gamma) \in R \;\Rightarrow\; Q_j := Q_j \cup \big\{\langle C \to \bullet\gamma, j\rangle\big\}.$$
3. *Vervollständigungs-Operation*

   Falls der Zustand Q_i ein Earley-Objekt der Form $\langle C \to \gamma\bullet, j\rangle$ enthält und weiter der Zustand Q_j ein Earley-Objekt der Form $\langle A \to \beta\bullet C\delta, k\rangle$ enthält, dann hat der Parser im Zustand Q_j versucht, ein C zu parsen und das C ist im Zustand Q_i erkannt worden. Daher fügen wir dem Zustand Q_i nun das Earley-Objekt $\langle A \to \beta C \bullet \delta, k\rangle$ hinzu:
   $$\langle C \to \gamma\bullet, j\rangle \in Q_i \wedge \langle A \to \beta \bullet C\delta, k\rangle \in Q_j \;\Rightarrow\; Q_i := Q_i \cup \big\{\langle A \to \beta C \bullet \delta, k\rangle\big\}.$$

Der Algorithmus von Earley, der einen String der Form $s = x_1 \cdots x_n$ parsen will, funktioniert nun so:

1. Wir initialisieren die Zustände Q_i wie folgt:
   $$Q_0 := \big\{\langle \widehat{S} \to \bullet S, 0\rangle\big\},$$
   $$Q_i := \big\{\big\} \quad \text{für } i = 1, \cdots, n.$$
2. Anschließend lassen wir in einer Schleife i von 0 bis n laufen und führen die folgenden Schritte durch:

(a) Wir vergrößern Q_i mit der Vervollständigungs-Operation so lange, bis mit dieser Operation keine neuen Earley-Objekte mehr gefunden werden können.

(b) Anschließend vergrößern wir Q_i mit Hilfe der Vorhersage-Operation. Diese Operation wird ebenfalls so lange durchgeführt, wie neue Earley-Objekte gefunden werden.

(c) Falls $i < n$ ist, wenden wir die Lese-Operation auf Q_i an und initialisierend damit Q_{i+1}.

Falls die betrachtete Grammatik G auch ε-Regeln enthält, also Regeln der Form

$$C \to \varepsilon,$$

dann kann es passieren, dass durch die Anwendung einer Vorhersage-Operation eine neue Anwendung der Vervollständigungs-Operation möglich wird. In diesem Fall müssen Vorhersage-Operation und Vervollständigungs-Operation so lange iteriert werden, bis durch Anwendung dieser beiden Operationen keine neuen Earley-Objekte mehr erzeugt werden können.

3. Falls nach Beendigung des Algorithmus die Menge Q_n das Earley-Objekt $\langle \widehat{S} \to S\bullet, 0\rangle$ enthält, dann war das Parsen erfolgreich und der String $x_1 \cdots x_n$ liegt in der von der Grammatik erzeugten Sprache.

Beispiel: Abbildung 8.1 zeigt eine vereinfachte Grammatik für arithmetische Ausdrücke, die nur aus den Zahlen "1", "2" und "3" und den beiden Operator-Symbolen "+" und "*" aufgebaut ist. Die Menge T der Terminale dieser Grammatik ist also durch

$$T = \{\ \text{“1”}\ ,\ \text{“2”}\ ,\ \text{“3”}\ ,\ \text{“+”}\ ,\ \text{“*”}\ \}$$

gegeben. Wir zeigen, wie sich der String "1+2*3" mit dieser Grammatik und dem Algorithmus von Earley parsen lässt. In der folgenden Darstellung werden wir die syntaktische Variable `expr` mit dem Buchstaben E abkürzen, für `prod` schreiben wir P und für `fact` verwenden wir die Abkürzung F.

```
1   expr : expr '+' prod
2        | prod
3        ;
4
5   prod : prod '*' fact
6        | fact
7        ;
8
9   fact : '1'
10       | '2'
11       | '3'
12       ;
```

Figure 8.1: Eine vereinfachte Grammatik für arithmetische Ausdrücke.

1. Wir initialisieren Q_0 als

$$Q_0 = \{\langle \widehat{S} \to \bullet\, E, 0\rangle\}.$$

Die Mengen Q_1, Q_2, Q_3, Q_4 und Q_5 sind zunächst alle leer. Wenden wir die Vervollständigungs-Operation auf Q_0 an, so finden wir keine neuen Earley-Objekte.

Anschließend wenden wir die Vorhersage-Operation auf das Earley-Objekt $\langle \widehat{S} \to \bullet\, E, 0\rangle$ an. Dadurch werden der Menge Q_0 zunächst die beiden Earley-Objekte

$$\langle E \to \bullet\, E \text{“+”} P, 0\rangle \quad \text{und} \quad \langle E \to \bullet\, P, 0\rangle$$

hinzugefügt. Auf das Earley-Objekt $\langle E \to \bullet\, P, 0\rangle$ können wir die Vorhersage-Operation ein weiteres Mal anwenden und erhalten dann die beiden neuen Earley-Objekte

$$\langle P \to \bullet\ P\text{“*”}\,F, 0\rangle \quad \text{und} \quad \langle P \to \bullet\ F, 0\rangle.$$

Wenden wir auf das Earley-Objekt $\langle P \to \bullet\ F, 0\rangle$ die Vorhersage-Operation an, so erhalten wir schießlich noch die folgenden Earley-Objekte in Q_0:

$$\langle F \to \bullet\text{“1”}, 0\rangle, \quad \langle F \to \bullet\text{“2”}, 0\rangle, \quad \text{und} \quad \langle F \to \bullet\text{“3”}, 0\rangle.$$

Insgesamt enthält Q_0 nun die folgenden Earley-Objekte:

(a) $\langle \widehat{S} \to \bullet\ E, 0\rangle$,
(b) $\langle E \to \bullet\ E\text{“+”}\,P, 0\rangle$
(c) $\langle E \to \bullet\ P, 0\rangle$,
(d) $\langle P \to \bullet\ P\text{“*”}\,F, 0\rangle$,
(e) $\langle P \to \bullet\ F, 0\rangle$,
(f) $\langle F \to \bullet\text{“1”}, 0\rangle$,
(g) $\langle F \to \bullet\text{“2”}, 0\rangle$,
(h) $\langle F \to \bullet\text{“3”}, 0\rangle$.

Jetzt wenden wir die Lese-Operation auf Q_0 an. Da das erste Zeichen des zu parsenden Strings eine “1” ist, hat die Menge Q_1 danach die folgende Form:

$$Q_1 = \big\{\langle F \to \text{“1”}\bullet, 0\rangle\big\}.$$

2. Nun setzen wir $i = 1$ und wenden zunächst auf Q_1 die Vervollständigungs-Operation an. Aufgrund des Earley-Objekts $\langle F \to \text{“1”}\bullet, 0\rangle$ in Q_1 suchen wir in Q_0 ein Earley-Objekt, bei dem die Markierung “•” vor der Variablen F steht. Wir finden das Earley-Objekt $\langle P \to \bullet\ F, 0\rangle$. Daher fügen wir nun Q_1 das Earley-Objekt

   $$\langle P \to F\ \bullet, 0\rangle$$

   hinzu. Hierauf können wir wieder die Vervollständigungs-Operation anwenden und finden (nach mehrmaliger Anwendung) für Q_1 insgesamt die folgenden Earley-Objekte:

   (a) $\langle P \to F\ \bullet, 0\rangle$,
   (b) $\langle P \to P\ \bullet\ \text{“*”}\,F, 0\rangle$,
   (c) $\langle E \to P\ \bullet, 0\rangle$,
   (d) $\langle E \to E\ \bullet\ \text{“+”}\,P, 0\rangle$,
   (e) $\langle \widehat{S} \to E\ \bullet, 0\rangle$.

   Als nächstes wenden wir auf diese Earley-Objekte die Vorhersage-Operation an. Da das Markierungs-Zeichen “•” aber in keinem der in Q_i auftretenden Earley-Objekte vor einer Variablen steht, ergeben sich hierbei keine neuen Earley-Objekte.

   Als letztes wenden wir die Lese-Operation auf Q_1 an. Da in dem String “1+2*3” das Zeichen “+” an der Position 2 liegt ist und Q_1 das Earley-Objekt

   $$\langle E \to E\ \bullet\ \text{“+”}\,P, 0\rangle$$

   enthält, fügen wir in Q_2 das Earley-Objekt

   $$\langle E \to E\text{“+”}\ \bullet\ P, 0\rangle$$

   ein.

3. Nun setzen wir $i = 2$ und wenden zunächst auf Q_2 die Vervollständigungs-Operation an. Zu diesem Zeitpunkt gilt

   $$Q_2 = \{\langle E \to E\text{“+”}\ \bullet\ P, 0\rangle\}.$$

   Da in dem einzigen Earley-Objekt, das hier auftritt, das Markierungs-Zeichen “•” nicht am Ende der Grammatik-Regel steht, finden wir durch die Vervollständigungs-Operation in diesem Schritt keine neuen Earley-Objekte.

Als nächstes wenden wir auf Q_2 die Vorhersage-Operation an. Da das Markierungs-Zeichen vor der Variablen P steht, finden wir zunächst die beiden Earley-Objekte

$$\langle P \to \bullet\ F, 2\rangle \quad \text{und} \quad \langle P \to \bullet\ P\ \text{“*”}\ F, 2\rangle.$$

Da in dem ersten Earley-Objekt das Markierungs-Zeichen vor der Variablen F steht, kann die Vorhersage-Operation ein weiteres Mal angewendet werden und wir finden noch die folgenden Earley-Objekte:

(a) $\langle F \to \bullet\text{“1”}, 2\rangle$,
(b) $\langle F \to \bullet\text{“2”}, 2\rangle$,
(c) $\langle F \to \bullet\text{“3”}, 2\rangle$.

Als letztes wenden wir die Lese-Operation auf Q_2 an. Da das dritte Zeichen in dem zu lesenden String “1+2*3” die Ziffer “2” ist, hat Q_3 nun die Form

$$Q_3 = \{\langle F \to \text{“2”}\bullet, 2\rangle\}.$$

4. Wir setzen $i = 3$ und wenden auf Q_3 die Vervollständigungs-Operation an. Dadurch fügen wir

$$\langle P \to F\ \bullet, 2\rangle$$

in Q_3 ein. Hier können wir ein weiteres Mal die Vervollständigungs-Operation anwenden. Durch iterierte Anwendung der Vervollständigungs-Operation erhalten wir zusätzlich die folgenden Earley-Objekte:

(a) $\langle P \to P \bullet \text{“*”}\ F, 2\rangle$,
(b) $\langle E \to E\ \text{“+”}\ P\ \bullet, 0\rangle$,
(c) $\langle E \to E\ \bullet\ \text{“+”}\ P, 0\rangle$
(d) $\langle \widehat{S} \to E\ \bullet, 0\rangle$.

Als letztes wenden wir die Lese-Operation an. Da der nächste zu lesende Buchstabe das Zeichen “*” ist, erhalten wir

$$Q_4 = \{\langle P \to P\ \text{“*”}\ \bullet\ F, 2\rangle\}.$$

5. Wir setzen $i = 4$. Die Vervollständigungs-Operation liefert keine neuen Earley-Objekte. Die Vorhersage-Operation liefert folgende Earley-Objekte:

(a) $\langle F \to \bullet\text{“1”}, 4\rangle$,
(b) $\langle F \to \bullet\text{“2”}, 4\rangle$,
(c) $\langle F \to \bullet\text{“3”}, 4\rangle$.

Da das nächste Zeichen die Ziffer “3” ist, liefert die Lese-Operation für Q_5:

$$Q_5 = \langle F \to \text{“3”}\bullet, 4\rangle\}.$$

6. Wir setzen $i = 5$. Die Vervollständigungs-Operation liefert nacheinander die folgenden Earley-Objekte:

(a) $\langle P \to P\ \text{“*”}\ F\ \bullet, 2\rangle$,
(b) $\langle E \to E\ \text{“+”}\ P\ \bullet, 0\rangle$,
(c) $\langle P \to P\ \bullet\ \text{“*”}\ F, 2\rangle$,
(d) $\langle E \to E\ \bullet\ \text{“+”}\ P, 0\rangle$
(e) $\langle \widehat{S} \to E\ \bullet, 0\rangle$.

Da die Menge Q_5 das Earley-Objekt $\langle \widehat{S} \to E\ \bullet, 0\rangle$ enthält, können wir schließen, dass der String “1+2*3” tatsächlich in der von der Grammatik erzeugten Sprache liegt.

Aufgabe 26: Zeigen Sie, dass der String “1*2+3” in der Sprache der Grammatik liegt, die in Abbildung 8.1 gezeigt wird. Benutzen Sie dazu den von Earley angegebenen Algorithmus.

8.2 Implementing Earley's Algorithm in Python

The *Jupyter* notebook

https://github.com/karlstroetmann/Formal-Languages/blob/master/ANTLR4-Python/Earley-Parser/Earley-Parser.ipynb

contains an implementation of Earley's algorithm.

Chapter 9

Bottom-Up-Parser

Bei der Konstruktion eines Parsers gibt es generell zwei Möglichkeiten: Wir können Top-Down oder Bottom-Up vorgehen. Den Top-Down-Ansatz haben wir bereits diskutiert. In diesem Kapitel erläutern wir nun den Bottom-Up-Ansatz. Dazu stellen wir im nächsten Abschnitt das allgemeine Konzept vor, das einem Bottom-Up-Parser zu Grunde liegt. Im darauf folgenden Abschnitt zeigen wir, wie Bottom-Up-Parser implementiert werden können und stellen als eine Implementierungsmöglichkeit die Shift-Reduce-Parser vor. Ein Shift-Reduce-Parser arbeitet mit Hilfe einer Tabelle, in der hinterlegt ist, wie der Parser in einem bestimmten Zustand die Eingaben verarbeiten muss. Die Theorie, wie eine solche Tabelle sinnvoll mit Informationen gefüllt werden kann, entwickeln wir dann in dem folgenden Abschnitt: Zunächst diskutieren wir die SLR-Parser (simple LR-Parser). Dies ist die einfachste Klasse von Shift-Reduce-Parsern. Das Konzept der SLR-Parser ist leider für die Praxis nicht mächtig genug. Daher verfeinern wir dieses Konzept und erhalten so die Klasse der kanonischen LR-Parser. Da die Tabellen für LR-Parser in der Praxis häufig groß werden, vereinfachen wir diese Tabellen etwas und erhalten dann das Konzept der LALR-Parser, das von der Mächtigkeit zwischen dem Konzept der SLR-Parser und dem Konzept der LR-Parser liegt. In dem folgenden Kapitel werden wir dann den Parser-Generator PLY diskutieren, der ein LALR-Parser ist.

9.1 Bottom-Up-Parser

Die mit ANTLR erstellten Parser sind sogenannte Top-Down-Parser: Ausgehend von dem Start-Symbol der Grammatik wurde versucht, eine gegebene Eingabe durch Anwendung der verschiedenen Grammatik-Regeln zu parsen. Die Parser, die wir nun entwickeln werden, sind Bottom-Up-Parser. Bei einem solchen Parser ist die Idee, dass wir von dem zu parsenden String ausgehen und dort Terminale anhand der rechten Seiten der Grammatik-Regeln zusammenfassen. Wir geben ein Beispiel und versuchen den String "`1 + 2 * 3`" mit der Grammatik, die durch die Regeln

$$
\begin{array}{lcl}
E & \to & E \text{ “+” } P \mid P \\
P & \to & P \text{ “*” } F \mid F \\
F & \to & \text{“1”} \mid \text{“2”} \mid \text{“3”}
\end{array}
$$

gegeben ist, zu parsen. Dazu suchen wir in diesem String Teilstrings, die den rechten Seiten von Grammatikregeln entsprechen, wobei wir den String von links nach rechts durchsuchen. Auf diese Art versuchen wir, einen Parse-Baum rückwärts von unten aufzubauen:

$$
\begin{array}{lcll}
\texttt{1 + 2 * 3} & \Leftarrow & F \texttt{ + 2 * 3} & (\text{Regel: } F \to \text{“1”}) \\
& \Leftarrow & P \texttt{ + 2 * 3} & (\text{Regel: } P \to F) \\
& \Leftarrow & E \texttt{ + 2 * 3} & (\text{Regel: } E \to P) \\
& \Leftarrow & E \texttt{ + } F \texttt{ * 3} & (\text{Regel: } F \to \text{“2”}) \\
& \Leftarrow & E \texttt{ + } P \texttt{ * 3} & (\text{Regel: } P \to F) \\
& \Leftarrow & E \texttt{ + } P \texttt{ * } F & (\text{Regel: } F \to \text{“3”}) \\
& \Leftarrow & E \texttt{ + } P & (\text{Regel: } P \to P \text{ “*” } F) \\
& \Leftarrow & E & (\text{Regel: } E \to E \text{ “+” } P)
\end{array}
$$

Im ersten Schritt haben wir beispielsweise die Grammatik-Regel $F \to$ "1" benutzt, um den String "1" durch F zu ersetzen und dabei dann den String "F + 2 * 3" erhalten. Im zweiten Schritt haben wir die Regel $P \to F$ benutzt, um F durch P zu ersetzen. Auf diese Art und Weise haben wir am Ende den ursprünglichen String "1 + 2 * 3" auf E zurück geführt. Wir können an dieser Stelle zwei Beobachtungen machen:

1. Wir ersetzen bei unserem Vorgehen immer den am weitesten links stehenden Teilstring, der ersetzt werden kann, wenn wir den anfangs gegebenen String auf das Start-Symbol der Grammatik zurück führen wollen.

2. Schreiben wir die Ableitung, die wir rückwärts konstruiert haben, noch einmal in der richtigen Reihenfolge hin, so erhalten wir:

$$\begin{array}{rcl} E & \Rightarrow & E \texttt{ + } P \\ & \Rightarrow & E \texttt{ + } P \texttt{ * } F \\ & \Rightarrow & E \texttt{ + } P \texttt{ * 3} \\ & \Rightarrow & E \texttt{ + } F \texttt{ * 3} \\ & \Rightarrow & E \texttt{ + 2 * 3} \\ & \Rightarrow & P \texttt{ + 2 * 3} \\ & \Rightarrow & F \texttt{ + 2 * 3} \\ & \Rightarrow & \texttt{1 + 2 * 3} \end{array}$$

   Wir sehen hier, dass bei dieser Ableitung immer die am weitesten rechts stehende syntaktische Variable ersetzt worden ist. Eine derartige Ableitung wird als Rechts-Ableitung bezeichnet.

   Im Gegensatz dazu ist es bei den Ableitungen, die ein Top-Down-Parser erzeugt, genau umgekehrt: Dort wird immer die am weitesten links stehende syntaktische Variable ersetzt. Die mit einem solchen Parser erzeugten Ableitungen heißen daher Links-Ableitungen.

Die obigen beiden Beobachtungen sind der Grund, weshalb die Parser, die wir in diesem Kapitel diskutieren, als LR-Parser bezeichnet werden. Das L steht für left to right und beschreibt die Vorgehensweise, dass der String immer von links nach rechts durchsucht wird, während das R für reverse rightmost derivation steht und ausdrückt, dass solche Parser eine Rechts-Ableitung rückwärts konstruieren.

Bei der Implementierung eines LR-Parsers stellen sich zwei Fragen:

(a) Welche Teilstrings ersetzen wir?

(b) Welche Regeln verwenden wir dabei?

Die Beantwortung dieser Fragen ist im Allgemeinen nicht trivial. Zwar gehen wir die Strings immer von links nach rechts durch, aber damit ist noch nicht unbeding klar, welchen Teilstring wir ersetzen, denn die potentiell zu ersetzenden Teilstrings können sich durchaus überlappen. Betrachten wir beispielsweise das Zwischenergebnis

$$E \texttt{ + } P \texttt{ * 3},$$

das wir oben im fünften Schritt erhalten haben. Hier könnten wir den Teilstring "P" mit Hilfe der Regel

$$E \to P$$

durch "E" ersetzen. Dann würden wir den String

$$E \texttt{ + } E \texttt{ * 3}$$

erhalten. Die einzigen Reduktionen, die wir jetzt noch durchführen können, führen über die Zwischenergebnisse E + E * F und E + E * P zu dem String

$$E \texttt{ + } E \texttt{ * } E,$$

der sich dann aber mit der oben angegebenen Grammatik nicht mehr reduzieren lässt. Die Antwort auf die obigen Fragen, welchen Teilstring wir ersetzen und welche Regel wir verwenden, setzt einiges an Theorie voraus, die wir in den folgenden Abschnitten entwickeln werden.

9.2 Shift-Reduce-Parser

Shift-reduce parsing is one way to implement bottom up parsing. Assume a grammar $G = \langle V, T, R, S \rangle$ is given. A shift-reduce parser is defined as a 4-Tuple

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto} \rangle$$

where

1. Q is the set of states of the shift-reduce parser.

   At first, states are purely abstract.

2. $q_0 \in Q$ is the start state.

3. *action* is a function taking two arguments. The first argument is a state $q \in Q$ and the second argument is a terminal $t \in T$. The result of this function is an element from the set

   $$\mathit{Action} := \big\{\langle \texttt{shift}, q\rangle \mid q \in Q\big\} \cup \big\{\langle \texttt{reduce}, r\rangle \mid r \in R\big\} \cup \big\{\texttt{accept}\big\} \cup \big\{\texttt{error}\big\}.$$

   Here `shift`, `reduce`, `accept`, and `error` are strings that serve to distinguish the different kinds of result of the function *action*. Therefore the signature of the function *action* is given as follows:

   $$\mathit{action} : Q \times T \to \mathit{Action}.$$

4. *goto* is a function that takes a state $q \in Q$ and a syntactical variable $v \in V$ and computes a new state. Therefore the signature of *goto* is as follows:

   $$\mathit{goto} : Q \times V \to Q.$$

A shift-reduce parser uses two stacks:

(a) *States* is a stack of states from the set Q:

$$\mathit{States} \in \mathit{Stack}\langle Q \rangle.$$

(b) *Symbols* is a stack of grammar symbols, i.e. this stack contains both terminals and syntactical variables:

$$\mathit{Symbols} \in \mathit{Stack}\langle T \cup V \rangle.$$

In order to simplify the exposition of shift-reduce parsing we assume that the set T of terminals contains the special symbol "EOF" (short for end of file). This symbol is assumed to occur at the end of the input string but does not occur elsewhere.

In order to understand how a shift-reduce parser works we introduce the notion of a parser configuration. A parser configuration is a triple of the form

$$\langle \mathit{States}, \mathit{Symbols}, \mathit{Tokens} \rangle$$

where *States* and *Symbols* are the aforementioned stacks of states and grammar symbols, while *Tokens* is the rest of the tokens from the input string that have not been processed. The stack *States* always starts with the start state q_0 and has a length that is one more than the length of the stack *Symbols*. If the input string that is to be parsed has the form

$$[t_1, \cdots, t_n]$$

and we have already reduced the first part $[t_1, \cdots, t_k]$ of the input string to produce the symbols

$$[X_1, \cdots, X_m],$$

while $[t_{k+1}, \cdots, t_n]$ is the part of the input string that still needs to be processed, then we have

$$\mathit{States} = [q_0, q_1 \cdots, q_m], \quad \mathit{Symbols} = [X_1, \cdots, X_m], \quad \text{and} \quad \mathit{Tokens} = [t_{k+1}, \cdots, t_n, \texttt{EOF}],$$

and the parser configuration $\langle \mathit{States}, \mathit{Symbols}, \mathit{Tokens} \rangle$ is written as

$$q_0, q_1, \cdots, q_m \mid X_1, \cdots . X_m \mid t_{k+1}, \cdots, t_n, \texttt{EOF}.$$

Shift-reduce parsing starts out with the configuration

$$q_0 \mid\mid t_1, \cdots, t_n, \texttt{EOF}.$$

Then, parsing proceeds iteratively. If the current configuration is

$$q_0, q_1, \cdots, q_m \mid X_1, \cdots, X_m \mid t_{k+1}, \cdots, t_n, \texttt{EOF},$$

then there is a case distinction according to the value of *action*(q_m, t_{k+1}).

(a) If *action*$(q_m, t_{k+1}) =$ *error*, then we know that the given string $t_1 \cdots, t_n$ is not generated by the given grammar and parsing is aborted with an error message.

(b) If *action*$(q_m, t_{k+1}) =$ *accept*, then we must have $t_{k+1} = \texttt{EOF}$ and we also must have $X_1 \cdots X_m = S$. In this case, we have reduced the string $t_1 \cdots t_m$ to the start symbol S of the given grammar and parsing finishes with success.

(c) If *action*$(q_m, t_{k+1}) = \langle \textit{shift}, q \rangle$, then the current configuration is changed into the new configuration

$$q_0, q_1, \cdots, q_m, q \mid X_1, \cdots, X_m, t_{k+1} \mid t_{k+2}, \cdots, t_n, \texttt{EOF},$$

i.e. the next token t_{k+1} is moved from the unread input to the top of the symbol stack and the new state q is pushed onto the stack *States*.

(d) If *action*$(q_m, t_{k+1}) = \langle \textit{reduce}, r \rangle$, where r is a grammar rule, then the grammar rule r must have the form

$$A \to X_{m-k} \cdots X_m,$$

i.e. the right hand side of the grammar rule matches the end of the stack *Symbols*. In this case, the symbols stack is reduced with this grammar rule, i.e. the symbols $X_{m-k} \cdots X_m$ are replaced by A. Furthermore, in the stack *States* the states $q_{m-k} \cdots q_m$ are replaced by the state *goto*(q_{m-k-1}, A). Therefore, the configuration changes as follows:

$$q_0, q_1, \cdots, q_{m-k-1}, \textit{goto}(q_{m-k-1}, A) \mid X_1, \cdots, X_{m-k-1}, A \mid t_{k+1}, \cdots, t_n, \texttt{EOF}.$$

```
1  class ShiftReduceParser():
2      def __init__(self, actionTable, gotoTable):
3          self.mActionTable = actionTable
4          self.mGotoTable   = gotoTable
```

Figure 9.1: Implementation of a shift-reduce parser in *Python*

The class `Shift-Reduce-Parser-Pure` that is shown in Figure 9.1 on page 102 displays the class `ShiftReduceParser`, which maintains two dictionaries.

(a) `mActionTable` stores the function *action* : $Q \times T \to$ *Action*.

(b) `mGotoTable` stores the function *goto* : $Q \times V \to Q$.

Figure 9.2 on page 103 shows the implementation of the method *parse* that implements shift-reduce parsing. This method assumes that the function *action* is coded as a dictionary that is stored in the member variable `mActionTable`. The function *goto* is also represented as a dictionary relation. It is stored in the member variable `mGotoTable`. The method *parse* is called with one argument *TL*. This is the list of tokens that have to be parsed. We append the special token "EOF" at the end of this list. The invocation *parse*(*TL*) returns `True` if the token list *TL* can be parsed successfully and `false` otherwise. The implementation of *parse* works as follows:

1. The variable *index* points to the next token in the token list that is to be read. Therefore, this variable is initialized to 0.

```
1   def parse(self, TL):
2       index   = 0      # points to next token
3       Symbols = []     # stack of symbols
4       States  = ['s0'] # stack of states, s0 is start state
5       TL     += ['EOF']
6       while True:
7           q = States[-1]
8           t = TL[index]
9           p = self.mActionTable.get((q, t), 'error')
10          if p == 'error':
11              return False
12          elif p == 'accept':
13              return True
14          elif p[0] == 'shift':
15              s = p[1]
16              Symbols += [t]
17              States  += [s]
18              index   += 1
19          elif p[0] == 'reduce':
20              head, body = p[1]
21              n       = len(body)
22              Symbols = Symbols[:-n]
23              States  = States [:-n]
24              Symbols = Symbols + [head]
25              state   = States[-1]
26              States += [ self.mGotoTable[state, head] ]
```

Figure 9.2: Implementation of a shift-reduce parser in *Python*

2. The variable *Symbols* stores the stack of symbols. The top of this stack is at the end of this list. Initially, the stack of symbols is empty.

3. The variable *States* is the stack of states. The start state is assumed to be the state "s0". Therefore this stack is initialized to contain only this state.

4. The main loop of the parser

   - sets the variable q to the current state,
   - initializes t to the next token, and then
   - sets p by looking up the appropriate action in the action table. Therefore p is equal to $\mathit{action}(q, t)$.

   What happens next depends on this value of $\mathit{action}(q, t)$.

   (a) $\mathit{action}(q, t) = \mathit{error}$.
   In this case the parser has found a syntax error and returns `False`.

   (b) $\mathit{action}(q, t) = \texttt{accept}$.
   In this case parsing is successful and therefore the function returns `True`.

   (c) $\mathit{action}(q, t) = \langle \texttt{shift}, s \rangle$.
   In this case, the token t is pushed onto the symbol stack in line 16, while the state s is pushed onto the stack of states. Furthermore, the variable *index* is incremented to point to the next unread token.

(d) $\mathit{action}(q, t) = \langle \texttt{reduce}, A \to X_1 \cdots X_n \rangle$.

In this case, we use the grammar rule

$$r = (A \to X_1 \cdots X_n)$$

to reduce the symbol stack. The variable *head* represents the left hand side A of this rule, while the list $[X_1, \cdots, X_n]$ is represented by the variable *body*.

In this case, it can be shown that the symbols X_1, $\cdots$, X_n are on top of the symbol stack. As we are going to reduce the symbol stack with the rule r, we remove these n symbols from the symbol stack and replace them with the variable A.

Furthermore, we have to remove n states from the stack of states. After that, we set *state* to the state that is then on top of the stack of states. Next, the new state $\mathit{goto}(\mathit{state}, A)$ is put on top of the stack of states in line 26.

In order to make the function *parse* work we have to provide an implementation of the functions *action* and *goto*. The tables 9.1 and 9.2 show these functions for the grammar given in Figure 9.3. For this grammar, there are 16 different states, which have been baptized as s_0, s_1, $\cdots$, s_{15}. The tables use two different abbreviations:

1. $\langle \mathit{shft}, s_i \rangle$ is short for $\langle \texttt{shift}, s_i \rangle$.
2. $\langle \mathit{rdc}, r_i \rangle$ is short for $\langle \texttt{reduce}, r_i \rangle$, where r_i denothes the grammar rule number i. Here, we have numbered the rules as follows:
   (a) $r_1 = (\mathit{expr} \to \mathit{expr}$ "+" $\mathit{product})$
   (b) $r_2 = (\mathit{expr} \to \mathit{expr}$ "−" $\mathit{product})$
   (c) $r_3 = (\mathit{expr} \to \mathit{product})$
   (d) $r_4 = (\mathit{product} \to \mathit{product}$ "∗" $\mathit{factor})$
   (e) $r_5 = (\mathit{product} \to \mathit{product}$ "/" $\mathit{factor})$
   (f) $r_6 = (\mathit{product} \to \mathit{factor})$
   (g) $r_7 = (\mathit{factor} \to$ "(" expr ")" $)$
   (h) $r_8 = (\mathit{factor} \to \textsc{Number})$

The corresponding grammar is shown in Figure 9.3. The definition of the grammar rules and the coding of the functions *action* and *goto* is shown in the Figures 9.4, 9.6, and 9.5 on the following pages. Of course, at present we do not have any idea how the functions *action* and *goto* are computed. This requires some theory that will be presented in the next section.

expr	→	*expr* "+" *product*
	|	*expr* "−" *product*
	|	*product*
product	→	*product* "∗" *factor*
	|	*product* "/" *factor*
	|	*factor*
factor	→	"(" *expr* ")"
	|	NUMBER

Figure 9.3: A grammar for arithmetical expressions.

State	EOF	+	-	*	/	(	)	NUMBER
s_0						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_1	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$	$\langle rdc, r_6 \rangle$		$\langle rdc, r_6 \rangle$	
s_2	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$	$\langle rdc, r_8 \rangle$		$\langle rdc, r_8 \rangle$	
s_3	$\langle rdc, r_3 \rangle$	$\langle rdc, r_3 \rangle$	$\langle rdc, r_3 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_3 \rangle$	
s_4	*accept*	$\langle shft, s_8 \rangle$	$\langle shft, s_9 \rangle$					
s_5						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_6		$\langle shft, s_8 \rangle$	$\langle shft, s_9 \rangle$				$\langle shft, s_7 \rangle$	
s_7	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$	$\langle rdc, r_7 \rangle$		$\langle rdc, r_7 \rangle$	
s_8						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_9						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_{10}	$\langle rdc, r_2 \rangle$	$\langle rdc, r_2 \rangle$	$\langle rdc, r_2 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_2 \rangle$	
s_{11}						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_{12}						$\langle shft, s_5 \rangle$		$\langle shft, s_2 \rangle$
s_{13}	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$	$\langle rdc, r_4 \rangle$		$\langle rdc, r_4 \rangle$	
s_{14}	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$	$\langle rdc, r_5 \rangle$		$\langle rdc, r_5 \rangle$	
s_{15}	$\langle rdc, r_1 \rangle$	$\langle rdc, r_1 \rangle$	$\langle rdc, r_1 \rangle$	$\langle shft, s_{12} \rangle$	$\langle shft, s_{11} \rangle$		$\langle rdc, r_1 \rangle$	

Table 9.1: The function *action*().

State	*expr*	*product*	*factor*
s_0	s_4	s_3	s_1
s_1			
s_2			
s_3			
s_4			
s_5	s_6	s_3	s_1
s_6			
s_7			
s_8		s_{15}	s_1
s_9		s_{10}	s_1
s_{10}			
s_{11}			s_{14}
s_{12}			s_{13}
s_{13}			
s_{14}			
s_{15}			

Table 9.2: The function *goto*().

```
r1 = ('E', ('E', '+', 'P'))
r2 = ('E', ('E', '-', 'P'))
r3 = ('E', ('P'))

r4 = ('P', ('P', '*', 'F'))
r5 = ('P', ('P', '/', 'F'))
r6 = ('P', ('F'))

r7 = ('F', ('(', 'E', ')'))
r8 = ('F', ('int',))
```

Figure 9.4: Grammar rules coded in *Python*.

```
gotoTable   := {};

gotoTable["s0", "E"] := "s4";
gotoTable["s0", "P"] := "s3";
gotoTable["s0", "F"] := "s1";

gotoTable["s5", "E"] := "s6";
gotoTable["s5", "P"] := "s3";
gotoTable["s5", "F"] := "s1";

gotoTable["s8", "P"] := "s15";
gotoTable["s8", "F"] := "s1";

gotoTable["s9", "P"] := "s10";
gotoTable["s9", "F"] := "s1";

gotoTable["s11", "F"] := "s14";
gotoTable["s12", "F"] := "s13";
```

Figure 9.5: Goto table coded in PYTHON.

```
actionTable = {}

actionTable['s0', '('  ] = ('shift', 's5');  actionTable['s8', '('  ] = ('shift', 's5')
actionTable['s0', 'int'] = ('shift', 's2');  actionTable['s8', 'int'] = ('shift', 's2')

actionTable['s1', 'EOF'] = ('reduce', r6);   actionTable['s9', '('  ] = ('shift', 's5')
actionTable['s1', '+'  ] = ('reduce', r6);   actionTable['s9', 'int'] = ('shift', 's2')
actionTable['s1', '-'  ] = ('reduce', r6);
actionTable['s1', '*'  ] = ('reduce', r6);   actionTable['s10', 'EOF'] = ('reduce', r2)
actionTable['s1', '/'  ] = ('reduce', r6);   actionTable['s10', '+' ] = ('reduce', r2)
actionTable['s1', ')'  ] = ('reduce', r6);   actionTable['s10', '-' ] = ('reduce', r2)
                                             actionTable['s10', '*' ] = ('shift', 's12')
actionTable['s2', 'EOF'] = ('reduce', r8);   actionTable['s10', '/' ] = ('shift', 's11')
actionTable['s2', '+'  ] = ('reduce', r8);   actionTable['s10', ')' ] = ('reduce', r2)
actionTable['s2', '-'  ] = ('reduce', r8);
actionTable['s2', '*'  ] = ('reduce', r8);   actionTable['s11', '('  ] = ('shift', 's5')
actionTable['s2', '/'  ] = ('reduce', r8);   actionTable['s11', 'int'] = ('shift', 's2')
actionTable['s2', ')'  ] = ('reduce', r8);
                                             actionTable['s12', '('  ] = ('shift', 's5')
actionTable['s3', 'EOF'] = ('reduce', r3);   actionTable['s12', 'int'] = ('shift', 's2')
actionTable['s3', '+'  ] = ('reduce', r3);
actionTable['s3', '-'  ] = ('reduce', r3);   actionTable['s13', 'EOF'] = ('reduce', r4)
actionTable['s3', '*'  ] = ('shift', 's12'); actionTable['s13', '+'  ] = ('reduce', r4)
actionTable['s3', '/'  ] = ('shift', 's11'); actionTable['s13', '-'  ] = ('reduce', r4)
actionTable['s3', ')'  ] = ('reduce', r3);   actionTable['s13', '*'  ] = ('reduce', r4)
                                             actionTable['s13', '/'  ] = ('reduce', r4)
actionTable['s4', 'EOF'] = 'accept';         actionTable['s13', ')'  ] = ('reduce', r4)
actionTable['s4', '+'  ] = ('shift', 's8');
actionTable['s4', '-'  ] = ('shift', 's9');  actionTable['s14', 'EOF'] = ('reduce', r5)
                                             actionTable['s14', '+'  ] = ('reduce', r5)
actionTable['s5', '('  ] = ('shift', 's5');  actionTable['s14', '-'  ] = ('reduce', r5)
actionTable['s5', 'int'] = ('shift', 's2');  actionTable['s14', '*'  ] = ('reduce', r5)
                                             actionTable['s14', '/'  ] = ('reduce', r5)
actionTable['s6', '+'  ] = ('shift', 's8');  actionTable['s14', ')'  ] = ('reduce', r5)
actionTable['s6', '-'  ] = ('shift', 's9');
actionTable['s6', ')'  ] = ('shift', 's7');  actionTable['s15', 'EOF'] = ('reduce', r1)
                                             actionTable['s15', '+'  ] = ('reduce', r1)
actionTable['s7', 'EOF'] = ('reduce', r7);   actionTable['s15', '-'  ] = ('reduce', r1)
actionTable['s7', '+'  ] = ('reduce', r7);   actionTable['s15', '*'  ] = ('shift', 's12')
actionTable['s7', '-'  ] = ('reduce', r7);   actionTable['s15', '/'  ] = ('shift', 's11')
actionTable['s7', '*'  ] = ('reduce', r7);   actionTable['s15', ')'  ] = ('reduce', r1)
actionTable['s7', '/'  ] = ('reduce', r7);
actionTable['s7', ')'  ] = ('reduce', r7);
```

Figure 9.6: Action table coded in PYTHON.

9.3 SLR-Parser

In diesem Abschnitt zeigen wir, wie wir für eine gegebene kontextfreie Grammatik G die im letzten Abschnitt verwendeten Funktionen

$$\mathit{action} : Q \times T \to \mathit{Action} \quad \text{and} \quad \mathit{goto} : Q \times V \to Q$$

berechnen können. Dazu klären wir als erstes, welche Informationen die in der Menge Q enthaltenen Zustände enthalten sollen. Wir werden diese Zustände so definieren, dass sie die Information enthalten, welche Regel der Shift-Reduce-Parser anzuwenden versucht, welcher Teil der rechten Seite einer Grammatik-Regel bereits erkannt worden ist und was noch erwartet wird. Zu diesem Zweck definieren wir den Begriff einer markierten Regel. In der englischen Originalliteratur [Knu65] wird hier unglücklicherweise der inhaltsleere Begriff "item" verwendet.

Definition 22 (markierte Regel) Eine markierte Regel einer Grammatik $G = \langle V, T, R, s\rangle$ ist ein Tripel

$$\langle a, \beta, \gamma\rangle,$$

für das gilt

$$(a \to \beta\gamma) \in R.$$

Wir schreiben eine markierte Regel der Form $\langle a, \beta, \gamma\rangle$ als

$$a \to \beta \bullet \gamma.$$ □

Die markierte Regel $a \to \beta \bullet \gamma$ drückt aus, dass der Parser versucht, mit der Regel $a \to \beta\gamma$ ein a zu parsen, dabei schon β gesehen hat und als nächstes versucht, γ zu erkennen. Das Zeichen $\bullet$ markiert also die Position innerhalb der rechten Seite der Regel, bis zu der wir die rechte Seite der Regel schon gelesen haben. Die Idee ist jetzt, dass wir die Zustände eines SLR-Parsers als Mengen von markierten Regeln darstellen. Um diese Idee zu veranschaulichen, betrachten wir ein konkretes Beispiel: Wir gehen von der in Abbildung 9.3 auf Seite 104 gezeigten Grammatik für arithmetische Ausdrücke aus, wobei wir diese Grammatik noch um ein neues Start-Symbol $\widehat{s}$ und die Regel

$$\widehat{s} \to \mathit{expr}\ \$$$

erweitern. Der Start-Zustand enthält offenbar die markierte Regel

$$\widehat{s} \to \varepsilon \bullet \mathit{expr}\,\$,$$

denn am Anfang versuchen wir ja, das Start-Symbol $\widehat{s}$ herzuleiten. Die Komponente ε drückt aus, dass wir bisher noch nichts verarbeitet haben. Neben dieser markierten Regel muss der Start-Zustand dann außerdem die markierten Regeln

1. $\mathit{expr} \to \varepsilon \bullet \mathit{expr}$ '+' $\mathit{product}$,
2. $\mathit{expr} \to \varepsilon \bullet \mathit{expr}$ '−' $\mathit{product}$ und
3. $\mathit{expr} \to \varepsilon \bullet \mathit{product}$

enthalten, denn es könnte ja beispielsweise sein, dass wir die Regel

$$\mathit{expr} \to \mathit{expr}\text{ '+' }\mathit{product}$$

verwenden müssen, um die gesuchte *expr* herzuleiten. Genauso gut könnte es natürlich sein, dass wir stattdessen die Regel

$$\mathit{expr} \to \mathit{product}$$

benutzen müssen. Das erklärt, warum wir die markierte Regel

$$\mathit{expr} \to \varepsilon \bullet \mathit{product}$$

in den Start-Zustand aufnehmen müssen, denn da wir am Anfang noch gar nicht wissen können, welche Regel wir benötigen, muss der Start-Zustand daher alle diese Regeln enthalten. Haben wir erst die markierte Regel

$$\mathit{expr} \to \varepsilon \bullet \mathit{product}$$

zum Start-Zustand hinzugefügt, so sehen wir, dass wir eventuell als nächstes ein *product* lesen müssen. Daher sehen wir, dass der Start-Zustand außerdem noch die folgenden markierten Regeln enthält:

4. *product* $\rightarrow \bullet$ *product* '*' *factor*,
5. *product* $\rightarrow \bullet$ *product* '/' *factor*,
6. *product* $\rightarrow \bullet$ *factor*,

   Nun zeigt die sechste Regel, dass wir eventuell als erstes einen *factor* lesen werden. Daher fügen wir zu dem Start-Zustand auch die folgenden beiden markierten Regeln hinzu:

7. *factor* $\rightarrow \bullet$ '(' *expr* ')' ,
8. *factor* $\rightarrow \bullet$ NUMBER.

Insgesamt sehen wir, dass der Start-Zustand aus einer Menge mit 8 markierten Regeln besteht. Das oben gezeigte System, aus einer gegebenen Regel weitere Regeln abzuleiten, formalisieren wir in dem Begriff des Abschlusses einer Menge von markierten Regeln.

Definition 23 (closure($\mathcal{M}$)) Es sei $\mathcal{M}$ eine Menge markierter Regeln. Dann definieren wir den Abschluss dieser Menge als die kleinste Menge $\mathcal{K}$ markierter Regeln, für die folgendes gilt:

1. $\mathcal{M} \subseteq \mathcal{K}$,

   der Abschluss umfasst also die ursprüngliche Regel-Menge.

2. Ist einerseits

   $$a \rightarrow \beta \bullet c\,\delta$$

   eine markierte Regel aus der Menge $\mathcal{K}$, wobei c eine syntaktische Variable ist, und ist andererseits

   $$c \rightarrow \gamma$$

   eine Grammatik-Regel der zu Grunde liegenden Grammatik G, so ist auch die markierte Regel

   $$c \rightarrow \bullet\gamma$$

   ein Element der Menge $\mathcal{K}$. Als Formel schreibt sich dies wie folgt:

   $$(a \rightarrow \beta \bullet c\,\delta) \in \mathcal{K} \;\wedge\; (c \rightarrow \gamma) \in R \;\Rightarrow\; (c \rightarrow \bullet\gamma) \in \mathcal{K}$$

Die so definierte Menge $\mathcal{K}$ ist eindeutig bestimmt und wird im Folgenden mit *closure*($\mathcal{M}$) bezeichnet. ◇

Bemerkung: Wenn Sie sich an den Earley-Algorithmus erinnern, dann sehen Sie, dass bei der Berechnung des Abschlusses dieselbe Berechnung wie bei der Vorhersage-Operation des Earley-Algorithmus durchgeführt wird. ◇

Für eine gegebene Menge $\mathcal{M}$ von markierten Regeln, kann die Berechnung von $\mathcal{K} :=$ *closure*($\mathcal{M}$) iterativ erfolgen:

1. Zunächst setzen wir $\mathcal{K} := \mathcal{M}$.
2. Anschließend suchen wir alle Regeln der Form

   $$a \rightarrow \beta \bullet c\,\delta$$

   aus der Menge $\mathcal{K}$, für die c eine syntaktische Variable ist und fügen dann für alle Regeln der Form $c \rightarrow \gamma$ die neue markierte Regel

   $$c \rightarrow \bullet\gamma$$

   in die Menge $\mathcal{K}$ ein. Dieser Schritt wird solange iteriert, bis keine neuen Regeln mehr gefunden werden.

Beispiel: Wir gehen von der in Abbildung 9.3 auf Seite 104 gezeigten Grammatik für arithmetische Ausdrücke aus und betrachten die Menge

$$\mathcal{M} := \{\mathit{product} \to \mathit{product}\ \text{'*'} \bullet \mathit{factor}\}$$

Für die Menge $\mathit{closure}(\mathcal{M})$ finden wir dann

$$\begin{array}{lcll} \mathit{closure}(\mathcal{M}) & = & \{ & \mathit{product} \to \mathit{product}\ \text{'*'} \bullet \mathit{factor}, \\ & & & \mathit{factor} \to \bullet\ \text{'('}\ \mathit{expr}\ \text{')'}, \\ & & & \mathit{factor} \to \bullet\ \textsc{Number} \\ & & \}. & \end{array}$$ ◇

Unser Ziel ist es, für eine gegebene kontextfreie Grammatik $G = \langle V, T, R, s\rangle$ einen Shift-Reduce-Parser

$$P = \langle Q, q_0, \mathit{action}, \mathit{goto}\rangle$$

zu definieren. Um dieses Ziel zu erreichen, müssen wir als erstes festlegen, wie wir die Zustände der Menge Q definieren wollen, denn dann funktioniert die Definition der restlichen Komponenten fast von alleine. Wir hatten oben schon gesagt, dass wir die Zustände als Mengen von markierten Regeln definieren. Wir definieren zunächst

$$\Gamma := \{a \to \beta \bullet \gamma \mid (a \to \beta\gamma) \in R\}$$

als die Menge aller markierten Regeln der Grammatik. Nun ist es allerdings nicht sinnvoll, beliebige Teilmengen von Γ als Zustände zuzulassen: Eine Teilmenge $\mathcal{M} \subseteq \Gamma$ kommt nur dann als Zustand in Betracht, wenn die Menge $\mathcal{M}$ unter der Funktion *closure*() abgeschlossen ist, wenn also $\mathit{closure}(\mathcal{M}) = \mathcal{M}$ gilt. Wir definieren daher

$$Q := \{\mathcal{M} \in 2^\Gamma \mid \mathit{closure}(\mathcal{M}) = \mathcal{M}\}.$$

Die Interpretation der Mengen $\mathcal{M} \in Q$ ist die, dass ein Zustand $\mathcal{M}$ genau die markierten Regeln enthält, die in der durch den Zustand beschriebenen Situation angewendet werden können.

Zur Vereinfachung der folgenden Konstruktionen erweitern wir die Grammatik $G = \langle V, T, R, s\rangle$ durch Einführung eines neuen Start-Symbols $\widehat{s}$ und eines neuen Tokens \$ zu der Grammatik

$$\widehat{G} = \Big\langle V \cup \{\widehat{s}\}, T \cup \{\$\}, R \cup \{\widehat{s} \to s\,\$\}, \widehat{s}\Big\rangle.$$

Das Token \$ steht dabei für das Ende der Eingabe. Die Grammatik $\widehat{G}$ bezeichnen wir als die augmentierte Grammatik. Die Verwendung der augmentierten Grammatik ermöglicht die nun folgende Definition des Start-Zustands. Wir setzen nämlich:

$$q_0 := \mathit{closure}\Big(\{\widehat{s} \to \bullet s\,\$\}\Big).$$

Als nächstes konstruieren wir die Funktion *goto*(). Die Definition lautet:

$$\mathit{goto}(\mathcal{M}, c) := \mathit{closure}\Big(\{a \to \beta\,c \bullet \delta \mid (a \to \beta \bullet c\,\delta) \in \mathcal{M}\}\Big).$$

Um diese Definition zu verstehen, nehmen wir an, dass der Parser in einem Zustand ist, in dem er versucht, ein a mit Hilfe der Regel $a \to \beta\,c\,\delta$ zu erkennen und dass dabei bereits der Teilstring β erkannt wurde. Dieser Zustand wird durch die markierte Regel

$$a \to \beta \bullet c\,\delta$$

beschrieben. Wird nun ein c erkannt, so kann der Parser von dem Zustand, der die Regel $a \to \beta \bullet c\,\delta$ enthält in einen Zustand, der die Regel $a \to \beta\,c \bullet \delta$ enthält, übergehen. Daher erhalten wir die oben angegebene Definition der Funktion $\mathit{goto}(\mathcal{M}, c)$. Für die gleich folgende Definition der Funktion $\mathit{action}(\mathcal{M}, t)$ ist es nützlich, die Definition der Funktion *goto* auf Terminale zu erweitern. Für Terminale $t \in T$ setzen wir:

$$\mathit{goto}(\mathcal{M}, t) := \mathit{closure}\Big(\{a \to \beta\,t \bullet \delta \mid (a \to \beta \bullet t\,\delta) \in \mathcal{M}\}\Big).$$

Bevor wir die Funktion *action* berechnen können, müssen wir zwei weitere Funktionen definieren, die Funktionen *First* und *Follow*.

9.3.1 Die Funktionen First und Follow

In diesem Abschnitt definieren wir die beiden Funktionen *First* und *Follow*. Diese Funktionen werden zur Berechnung der Funktion *action* benötigt. Wir betrachten dazu eine kontextfreie Grammatik $G = \langle V, T, R, s \rangle$ als vorgegeben.

(a) Für eine syntaktische Variable a berechnet *First*(a) die Menge aller Token, mit denen ein String w beginnen kann, der von der Variable a abgeleitet wird, für den also $a \Rightarrow_G^* w$ gilt.

(b) Für eine syntaktische Variable a berechnet *Follow*(a) die Menge der Token, die auf ein a in einem von s abgeleiteten String folgen können, d.h. $t \in$ *Follow*(a) falls es eine Ableitung der folgenden Form gibt:

$s \Rightarrow_G^* w\, a\, t\, r$. Hier sind w und r Token-Strings, während t ein einzelnes Token bezeichnet.

Definition 24 (ε-erzeugend) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik und *a* sei eine syntaktische Variable, also $a \in V$. Die Variable *a* heißt ε-erzeugend genau dann, wenn

$$a \Rightarrow^* \varepsilon$$

gilt, also dann, wenn sich aus der Variablen *a* das leere Wort ableiten lässt. Wir schreiben *nullable*(a) wenn die Variable *a* als ε-erzeugend nachgewiesen ist. ◇

Beispiele:

(a) Bei der in Abbildung 6.5 auf Seite 67 gezeigten Grammatik sind offenbar die Variablen *exprRest* und *productRest* ε-erzeugend.

(b) Wir betrachten nun ein weniger offensichtliches Beispiel. Die Grammatik G enthalte die folgenden Regeln:

$$S \to a\ b\ c$$
$$a \to \ 'X'\ b \mid a\ 'Y'\ \mid b\ c$$
$$b \to \ 'X'\ b \mid a\ 'Y'\ \mid c\ c$$
$$c \to a\ b\ c \mid \varepsilon$$

Zunächst ist offenbar die Variable *c* ε-erzeugend. Dann sehen wir, dass aufgrund der Regel $b \to c\ c$ auch *b* ε-erzeugend ist und daraus folgt wegen der Regel $a \to b\ c$, dass auch *a* ε-erzeugend ist. Schließlich erkennen wir S als ε-erzeugend, denn die erste Regel lautet

$$S \to a\ b\ c$$

und hier sind alle Variablen auf der rechten Seite der Regel bereits als ε-erzeugende Variablen nachgewiesen worden.

Definition 25 (First()) Es sei $G = \langle V, T, R, s \rangle$ eine kontextfreie Grammatik und $a \in V$. Dann definieren wir *First*(a) als die Menge aller der Token t, mit denen ein von *a* abgeleitetes Wort beginnen kann:

$$\mathit{First}(a) := \{t \in T \mid \exists \gamma \in (V \cup T)^* : a \Rightarrow^* t\,\gamma\}.$$

Die Definition der Funktion *First*() kann wie folgt auf Strings aus $(V \cup T)^*$ erweitert werden:

1. $\mathit{First}(\varepsilon) = \{\}$.
2. $\mathit{First}(t\beta) = \{t\}$ if $t \in T$.
3. $\mathit{First}(a\beta) = \begin{cases} \mathit{First}(a) \cup \mathit{First}(\beta) & \text{if } a \Rightarrow^* \varepsilon; \\ \mathit{First}(a) & \text{otherwise.} \end{cases}$

If *a* is a variable of G and the rules defining *a* are given as

$$a \to \alpha_1 \mid \cdots \mid \alpha_n,$$

then we have

$$\mathit{First}(a) = \bigcup_{i=1}^{n} \mathit{First}(\alpha_i).$$ ◇

Remark: Note that the definitions of the function *First*(a) for variables $a \in V$ and the function *First*(α) for strings $\alpha \in (V \cup T)^*$ are mutually recursive. The computation of *First*(a) is best done via a fixpoint computation: Start by setting *First*(a) := $\{\}$ for all variables $a \in V$ and then continue to iterate the equations defining *First*(a) until none of the sets *First*(a) changes any more. The next example clarifies this idea.

Beispiel: Wir können für die Variablen a der in Abbildung 6.5 gezeigten Grammatik die Mengen *First*(a) iterativ berechnen. Wir berechnen die Funktion *First*(a) für die einzelnen Variablen a am besten so, dass wir mit den Variablen beginnen, die in der Hierarchie ganz unten stehen.

1. Zunächst folgt aus den Regeln

   $$\mathit{factor} \rightarrow \text{'('}\ \mathit{expr}\ \text{')'} \mid \textsc{Number},$$

   dass jeder von *factor* abgeleitete String entweder mit einer öffnenden Klammer oder einer Zahl beginnt:

   $$\mathit{First}(\mathit{factor}) = \{\ \text{'('}, \textsc{Number}\ \}.$$

2. Analog folgt aus den Regeln

   $$\mathit{productRest} \rightarrow \text{'*'}\ \mathit{factor}\ \ \mathit{productRest} \mid \text{'/'}\ \mathit{factor}\ \ \mathit{productRest} \mid \varepsilon,$$

   dass ein *productRest* entweder mit dem Zeichen "*" oder "/" beginnt:

   $$\mathit{First}(\mathit{productRest}) = \{\ \text{'*'}, \text{'/'}\ \}$$

3. Die Regel für die Variable *product* lautet

   $$\mathit{product} \rightarrow \mathit{factor}\ \ \mathit{productRest}.$$

   Da die Variable *factor* nicht ε erzeugend ist, sehen wir, dass die Menge *First*(*product*) mit der Menge *First*(*factor*) übereinstimmt:

   $$\mathit{First}(\mathit{product}) = \{\ \text{'('}, \textsc{Number}\ \}.$$

4. Aus den Regeln

   $$\mathit{exprRest} \rightarrow \text{'+'}\ \mathit{product}\ \ \mathit{exprRest} \mid \text{'-'}\ \mathit{product}\ \ \mathit{exprRest} \mid \varepsilon$$

   können wir *First*(*exprRest*) wie folgt berechnen:

   $$\mathit{First}(\mathit{exprRest}) = \{\ \text{'+'}, \text{'-'}\ \}.$$

5. Weiter folgt aus der Regel

   $$\mathit{expr} \rightarrow \mathit{product}\ \ \mathit{exprRest}$$

   und der Tatsache, dass *product* nicht ε-erzeugend ist, dass die Menge *First*(*expr*) mit der Menge *First*(*product*) übereinstimmt:

   $$\mathit{First}(\mathit{expr}) = \{\ \text{'('}, \textsc{Number}\ \}.$$

Since we have computed the sets *First*(a) in a clever order, we did not have to perform a proper fixpoint iteration in this example. ◇

Definition 26 (Follow()) Es sei $G = \langle V, T, R, S \rangle$ eine kontextfreie Grammatik und $a \in V$. Bei der Berechnung von *Follow*() wird die Grammatik zunächst abgeändert, indem wir das Symbol \$ als neues Symbol zu der Menge T der Terminale hinzufügen. Zu den Variablen wird das neue Symbol $\widehat{S}$ hinzugefügt, das auch gleichzeitig das neue Start-Symbol der Grammatik ist. Zu der Menge R der Regeln fügen wir die folgende Regel neu hinzu:

$$\widehat{S} \rightarrow S\ \$.$$

Weiter definieren wir

$$\widehat{T} := T \cup \{\$\}.$$

Die so veränderte Grammatik bezeichnen wir als die augmentierte Grammatik. Dann definieren wir *Follow*(a) als die Menge aller der Token t, die in einer Ableitung auf a folgen können:

$$\mathit{Follow}(a) := \left\{ t \in \widehat{T} \mid \exists \beta, \gamma \in (V \cup \widehat{T})^* : \widehat{S} \Rightarrow^* \beta\, a\, t\, \gamma \right\}.$$

Wenn sich aus dem Start-Symbol $\widehat{S}$ also irgendwie ein String $\beta\, a\, t\, \gamma$ ableiten lässt, bei dem das Token t auf die Variable a folgt, dann ist t ein Element der Menge *Follow*(a). ◇

Beispiel: Wir untersuchen wieder die in Abbildung 6.5 gezeigte Grammatik für arithmetische Ausdrücke.

1. Aufgrund der neu hinzugefügten Regel

   $$\widehat{S} \to \mathit{expr}\ \$$$

   muss die Menge *Follow*(*expr*) das Zeichen $ enthalten. Aufgrund der Regel

   $$\mathit{factor} \to \text{'('}\ \mathit{expr}\ \text{')'}$$

   muss die Menge *Follow*(*expr*) außerdem das Zeichen ')' enthalten. Also haben wir insgesamt

   $$\mathit{Follow}(\mathit{expr}) = \{\$,\ \text{')'}\}.$$

2. Aufgrund der Regel

   $$\mathit{expr} \to \mathit{product}\ \ \mathit{exprRest}$$

   wissen wir, dass alle Terminale, die auf ein *expr* folgen können, auch auf ein *exprRest* folgen können, womit wir schon mal wissen, dass *Follow*(*exprRest*) die Token $ und ')' enthält. Da *exprRest* sonst nur am Ende der Regeln vorkommt, die *exprRest* definieren, sind das auch schon alle Token, die auf *exprRest* folgen können und wir haben

   $$\mathit{Follow}(\mathit{exprRest}) = \{\$,\ \text{')'}\}.$$

3. Die Regeln

   $$\mathit{exprRest} \to \text{'+'}\ \mathit{product}\ \ \mathit{exprRest} \mid \text{'−'}\ \mathit{product}\ \ \mathit{exprRest}$$

   zeigen, dass auf ein *product* alle Elemente aus *First*(*exprRest*) folgen können, aber das ist noch nicht alles: Da die Variable *exprRest* ε-erzeugend ist, können zusätzlich auf *product* auch alle Token folgen, die auf *exprRest* folgen. Damit haben wir insgesamt

   $$\mathit{Follow}(\mathit{product}) = \{\ \text{'+'}\ ,\ \text{'−'}\ ,\$,\ \text{')'}\ \}.$$

4. Die Regel

   $$\mathit{product} \to \mathit{factor}\ \ \mathit{productRest}$$

   zeigt, dass alle Terminale, die auf ein *product* folgen können, auch auf ein *productRest* folgen können. Da *productRest* sonst nur am Ende der Regeln vorkommt, die *productRest* definieren, sind das auch schon alle Token, die auf *productRest* folgen können und wir haben insgesamt

   $$\mathit{Follow}(\mathit{productRest}) = \{\ \text{'+'}\ ,\ \text{'−'}\ ,\$,\ \text{')'}\ \}.$$

5. Die Regeln

   $$\mathit{productRest} \to \text{'*'}\ \mathit{factor}\ \ \mathit{productRest} \mid \text{'/'}\ \mathit{factor}\ \ \mathit{productRest}$$

   zeigen, dass auf ein *factor* alle Elemente aus *First*(*productRest*) folgen können, aber das ist noch nicht alles: Da die Variable *productRest* ε-erzeugend ist, können zusätzlich auf *factor* auch alle Token folgen, die auf *productRest* folgen. Damit haben wir insgesamt

   $$\mathit{Follow}(\mathit{factor}) = \{\ \text{'*'}\ ,\ \text{'/'}\ ,\ \text{'+'}\ ,\ \text{'−'}\ ,\$,\ \text{')'}\ \}.$$ □

Das letzte Beispiel zeigt, dass die Berechnung des Prädikats *nullable*() und die Berechnung der Mengen *First*(a) und *Follow*(a) für eine syntaktische Variable a eng miteinander verbunden sind. Es sei

$$a \to Y_1 Y_2 \cdots Y_k$$

eine Grammatik-Regel. Dann bestehen zwischen dem Prädikat `nullable()` und den beiden Funktionen *First*() und *Follow*() die folgenden Beziehungen:

1. $\forall t \in T : \neg\, \mathit{nullable}(t)$.
2. $k = 0 \Rightarrow \mathit{nullable}(a)$.
3. $\big(\forall i \in \{1, \cdots, k\} : \mathit{nullable}(Y_i)\big) \Rightarrow \mathit{nullable}(a)$.
   Setzen wir hier $k = 0$ so sehen wir, dass 2. ein Spezialfall von 3. ist.
4. $\mathit{First}(Y_1) \subseteq \mathit{First}(a)$.
5. $\big(\forall j \in \{1, \cdots, i-1\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{First}(Y_i) \subseteq \mathit{First}(a)$.
   Setzen wir oben $i = 1$, so sehen wir, dass 4. ein Spezialfall von 5. ist.
6. $\mathit{Follow}(a) \subseteq \mathit{Follow}(Y_k)$.
7. $\big(\forall j \in \{i+1, \cdots, k\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{Follow}(a) \subseteq \mathit{Follow}(Y_i)$.
   Setzen wir hier $i = k$ so sehen wir, dass 6. ein Spezialfall von 7. ist.
8. $\forall i \in \{1, \cdots, k-1\} : \mathit{First}(Y_{i+1}) \subseteq \mathit{Follow}(Y_i)$.
9. $\big(\forall j \in \{i+1, \cdots, l-1\} : \mathit{nullable}(Y_j)\big) \Rightarrow \mathit{First}(Y_l) \subseteq \mathit{Follow}(Y_i)$.
   Setzen wir hier $l = i + 1$ so sehen wir, dass 8. ein Spezialfall von 9. ist.

Mit Hilfe dieser Beziehungen können *nullable*(), *First*() und *Follow*() iterativ über eine Fixpunkt-Iteration berechnet werden:

1. Zunächst werden die Funktionen *First*(a) und *Follow*(a) für jede syntaktische Variable a mit der leeren Menge initialisiert. Das Prädikat *nullable*(a) wird für jede syntaktische Variable auf `false` gesetzt.
2. Anschließend werden die oben angegebenen Regeln so lange angewendet, wie sich durch die Anwendung Änderungen ergeben.

9.3.2 Die Berechnung der Funktion action

Als Letztes spezifizieren wir, wie die Funktion *action*$(\mathcal{M}, t)$ für eine Menge von markierten Regeln $\mathcal{M}$ und ein Token t berechnet wird. Bei der Definition von *action*$(\mathcal{M}, t)$ unterscheiden wir vier Fälle.

1. Falls $\mathcal{M}$ eine markierte Regel der Form $a \to \beta \bullet t\, \delta$ enthält und $t \neq \$$ ist, dann setzen wir

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t) \rangle,$$

   denn in diesem Fall versucht der Parser ein a mit Hilfe der Regel $a \to \beta\, t \delta$ zu erkennen und hat von der rechten Seite dieser Regel bereits β erkannt. Ist nun das nächste Token im Eingabe-String tatsächlich das Token t, so kann der Parser dieses t lesen und geht dabei von dem Zustand $a \to \beta \bullet t\, \delta$ in den Zustand $a \to \beta\, t \bullet \delta$ über, der von der Funktion *goto*$(\mathcal{M}, t)$ berechnet wird. Insgesamt haben wir also

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t) \rangle \quad \text{falls} \quad (a \to \beta \bullet t\, \delta) \in \mathcal{M} \text{ und } t \neq \$.$$

2. Falls $\mathcal{M}$ eine markierte Regel der Form $a \to \beta\bullet$ enthält und wenn zusätzlich $t \in \mathit{Follow}(a)$ gilt, dann setzen wir

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, a \to \beta \rangle,$$

   denn in diesem Fall versucht der Parser ein a mit Hilfe der Regel $a \to \beta$ zu erkennen und hat bereits β erkannt. Ist nun das nächste Token im Eingabe-String das Token t und ist darüber hinaus t ein Token, dass auf a folgen kann, gilt also $t \in \mathit{Follow}(a)$, so kann der Parser die Regel $a \to \beta$ anwenden und den Symbol-Stack mit dieser Regel reduzieren. Wir haben dann

   $$\mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, a \to \beta \rangle \quad \text{falls } (a \to \beta\bullet) \in \mathcal{M},\ a \neq \widehat{s} \text{ und } t \in \mathit{Follow}(a) \text{ gilt.}$$

3. Falls $\mathcal{M}$ die markierte Regel $\widehat{s} \to s \bullet \$$ enthält und wir den zu parsenden String vollständig gelesen haben, setzen wir

   $$\mathit{action}(\mathcal{M}, \$) := \texttt{accept},$$

   denn in diesem Fall versucht der Parser, $\widehat{s}$ mit Hilfe der Regel $\widehat{s} \to s\,\$$ zu erkennen und hat also bereits s erkannt. Ist nun das nächste Token im Eingabe-String das Datei-Ende-Zeichen \$, so liegt der zu parsende String in der durch die Grammatik G spezifizierten Sprache $L(G)$. Wir haben daher

   $$\mathit{action}(\mathcal{M}, \$) := \texttt{accept}, \quad \text{falls } (\widehat{s} \to s \bullet \$) \in \mathcal{M}.$$

4. In den restlichen Fällen setzen wir

   $$\mathit{action}(\mathcal{M}, t) := \texttt{error}.$$

Zwischen den ersten beiden Regeln kann es Konflikte geben. Wir unterscheiden zwischen zwei Arten von Konflikten.

1. Ein *Shift-Reduce-Konflikt* tritt auf, wenn sowohl der erste Fall als auch der zweite Fall vorliegt. In diesem Fall enthält die Menge $\mathcal{M}$ also zum einen eine markierte Regel der Form

   $$a \to \beta \bullet t\,\gamma,$$

   zum anderen enthält $\mathcal{M}$ eine Regel der Form

   $$c \to \delta\bullet \quad \text{mit } t \in \mathit{Follow}(c).$$

   Wenn dann das nächste Token den Wert t hat, ist nicht klar, ob dieses Token auf den Symbol-Stack geschoben und der Parser in einen Zustand mit der markierten Regel $a \to \beta\,t \bullet \gamma$ übergehen soll, oder ob stattdessen der Symbol-Stack mit der Regel $c \to \delta$ reduziert werden muss.

2. Ein *Reduce-Reduce-Konflikt* liegt vor, wenn die Menge $\mathcal{M}$ zwei verschiedene markierte Regeln der Form

   $$c_1 \to \gamma_1\bullet \quad \text{und} \quad c_2 \to \gamma_2\bullet$$

   enthält und wenn gleichzeitig $t \in \mathit{Follow}(c_1) \cap \mathit{Follow}(c_2)$ ist, denn dann ist nicht klar, welche der beiden Regeln der Parser anwenden soll, wenn das nächste zu lesende Token den Wert t hat.

Falls einer dieser beiden Konflikte auftritt, dann sagen wir, dass die Grammatik keine SLR-Grammatik ist. Eine solche Grammatik kann mit Hilfe eines SLR-Parsers nicht geparst werden. Wir werden später noch Beispiele für die beiden Arten von Konflikten angeben, aber zunächst wollen wir eine Grammatik untersuchen, bei der keine Konflikte auftreten und wollen für diese Grammatik die Funktionen *goto*() und *action*() auch tatsächlich berechnen. Wir nehmen als Grundlage die in Abbildung 9.3 gezeigte Grammatik. Da die syntaktische Variable *expr* auf der rechten Seite von Grammatik-Regeln auftritt, definieren wir *start* als neues Start-Symbol und fügen in der Grammatik die Regel

$$\mathit{start} \to \mathit{expr}\,\$$$

ein. Dieser Schritt entspricht dem früher diskutierten Augmentieren der Grammatik. Als erstes berechnen wir die Menge der Zustände Q. Wir hatten dafür oben die folgende Formel angegeben:

$$Q := \big\{\mathcal{M} \in 2^{\Gamma} \mid \mathit{closure}(\mathcal{M}) = \mathcal{M}\big\}.$$

Diese Menge enthält allerdings auch Zustände, die von dem Start-Zustand über die Funktion *goto*() gar nicht erreicht werden können. Wir berechnen daher nur die Zustände, die sich auch tatsächlich vom Start-Zustand mit Hilfe der Funktion *goto*() erreichen lassen. Damit die Rechnung nicht zu unübersichtlich wird, führen wir die folgenden Abkürzungen ein:

$$s := \mathit{start},\ e := \mathit{expr},\ p := \mathit{product},\ f := \mathit{factor} \text{ und } N := \textsc{Number}.$$

Wir beginnen mit dem Start-Zustand:

1. $s_0 := \mathit{closure}(\{\ s \to \bullet\, e\,\$\ \})$
$= \{\ s \to \bullet\, e\,\$,$
$e \to \bullet\, e\ \texttt{'+'}\ p,\ e \to \bullet\, e\ \texttt{'-'}\ p,\ e \to \bullet\, p,$
$p \to \bullet\, p\ \texttt{'*'}\ f,\ p \to \bullet\, p\ \texttt{'/'}\ f,\ p \to \bullet\, f,$
$f \to \bullet\, \texttt{'('}\, e\, \texttt{')'}\ ,\ f \to \bullet\, N \qquad \}.$

2. $s_1 := \mathit{goto}(s_0, f)$
$= \mathit{closure}(\{p \to f\bullet\})$
$= \{p \to f\bullet\}.$

3. $s_2 := \mathit{goto}(s_0, N)$
$= \mathit{closure}(\{f \to N\bullet\})$
$= \{f \to N\bullet\}.$

4. $s_3 := \mathit{goto}(s_0, p)$
$= \mathit{closure}(\{p \to p\bullet\ \texttt{'*'}\ f,\ p \to p\bullet\ \texttt{'/'}\ f,\ e \to p\bullet\})$
$= \{p \to p\bullet\ \texttt{'*'}\ f,\ p \to p\bullet\ \texttt{'/'}\ f, e \to p\bullet\}.$

5. $s_4 := \mathit{goto}(s_0, e)$
$= \mathit{closure}(\{s \to e\bullet\ \$,\ e \to e\bullet\ \texttt{'+'}\ p,\ e \to e\bullet\ \texttt{'-'}\ p\ \})$
$= \{\ s \to e\bullet\ \$,\ e \to e\bullet\ \texttt{'+'}\ p,\ e \to e\bullet\ \texttt{'-'}\ p\ \}.$

6. $s_5 := \mathit{goto}(s_0,\ \texttt{'('}\ )$
$= \mathit{closure}(\{f \to\ \texttt{'('}\ \bullet\, e\ \texttt{')'}\ \})$
$= \{\ f \to\ \texttt{'('}\ \bullet\, e\ \texttt{')'}$
$e \to \bullet\, e\ \texttt{'+'}\ p,\ e \to \bullet\, e\ \texttt{'-'}\ p,\ e \to \bullet\, p,$
$p \to \bullet\, p\ \texttt{'*'}\ f,\ p \to \bullet\, p\ \texttt{'/'}\ f,\ p \to \bullet\, f,$
$f \to \bullet\, \texttt{'('}\, e\, \texttt{')'}\ ,\ f \to \bullet\, N \qquad \}.$

7. $s_6 := \mathit{goto}(s_5, e)$
$= \mathit{closure}(\{\ f \to\ \texttt{'('}\ e\bullet\ \texttt{')'}\ ,\ e \to e\bullet\ \texttt{'+'}\ p,\ e \to e\bullet\ \texttt{'-'}\ p\ \})$
$= \{\ f \to\ \texttt{'('}\ e\bullet\ \texttt{')'}\ ,\ e \to e\bullet\ \texttt{'+'}\ p,\ e \to e\bullet\ \texttt{'-'}\ p.\ \}$

8. $s_7 := \mathit{goto}(s_6,\ \texttt{')'}\ )$
$= \mathit{closure}(\{\ f \to\ \texttt{'('}\ e\ \texttt{')'}\ \bullet\ \})$
$= \{\ f \to\ \texttt{'('}\ e\ \texttt{')'}\ \bullet\ \}.$

9. $s_8 := \mathit{goto}(s_4,\ \texttt{'+'}\ )$
$= \mathit{closure}(\{e \to e\ \texttt{'+'}\ \bullet\, p\})$
$= \{\ e \to e\ \texttt{'+'}\ \bullet\, p$
$p \to \bullet\, p\ \texttt{'*'}\ f,\ p \to \bullet\, p\ \texttt{'/'}\ f,\ p \to \bullet\, f,$
$f \to \bullet\, \texttt{'('}\, e\, \texttt{')'}\ ,\ f \to \bullet\, N \qquad \}.$

10. $s_9 := \mathit{goto}(s_4,\ \texttt{'-'}\ )$
$= \mathit{closure}(\{e \to e\ \texttt{'-'}\ \bullet\, p\})$
$= \{\ e \to e\ \texttt{'-'}\ \bullet\, p$
$p \to \bullet\, p\ \texttt{'*'}\ f,\ p \to \bullet\, p\ \texttt{'/'}\ f,\ p \to \bullet\, f,$
$f \to \bullet\, \texttt{'('}\, e\, \texttt{')'}\ ,\ f \to \bullet\, N \qquad \}.$

11. $s_{10} := \mathit{goto}(s_9, p)$
$= \mathit{closure}(\{e \to e\ \texttt{'-'}\ p\bullet,\ p \to p\bullet\ \texttt{'*'}\ f,\ p \to p\bullet\ \texttt{'/'}\ f\ \})$
$= \{\ e \to e\ \texttt{'-'}\ p\bullet,\ p \to p\bullet\ \texttt{'*'}\ f,\ p \to p\bullet\ \texttt{'/'}\ f\ \}.$

12. $s_{11} := \mathit{goto}(s_3, \text{'/'}) = \mathit{closure}(\{p \to p\ \text{'/'}\ \bullet f\}) = \{\ p \to p\ \text{'/'}\ \bullet f,\ f \to \bullet\ \text{'('}e\ \text{')'},\ f \to \bullet N\ \}.$

13. $s_{12} := \mathit{goto}(s_3, \text{'*'}) = \mathit{closure}(\{p \to p\ \text{'*'}\ \bullet f\}) = \{\ p \to p\ \text{'*'}\ \bullet f,\ f \to \bullet\ \text{'('}e\ \text{')'},\ f \to \bullet N\ \}.$

14. $s_{13} := \mathit{goto}(s_{12}, f) = \mathit{closure}(\{p \to p\ \text{'*'}\ f\bullet\}) = \{\ p \to p\ \text{'*'}\ f\bullet\ \}.$

15. $s_{14} := \mathit{goto}(s_{11}, f) = \mathit{closure}(\{p \to p\ \text{'/'}\ f\bullet\}) = \{\ p \to p\ \text{'/'}\ f\bullet\ \}.$

16. $s_{15} := \mathit{goto}(s_8, p) = \mathit{closure}(\{e \to e\ \text{'+'}\ p\bullet,\ p \to p\bullet\ \text{'*'}\ f,\ p \to p\bullet\ \text{'/'}\ f\ \}) = \{\ e \to e\ \text{'+'}\ p\bullet,\ p \to p\bullet\ \text{'*'}\ f,\ p \to p\bullet\ \text{'/'}\ f\ \}.$

Weitere Rechnungen führen nicht mehr auf neue Zustände. Berechnen wir beispielsweise $\mathit{goto}(s_8,\ \text{'('}\ )$, so finden wir

$$\begin{aligned}
& \mathit{goto}(s_8,\ \text{'('}\ ) \\
= \ & \mathit{closure}(\{f \to\ \text{'('}\ \bullet e\ \text{')'}\ \}) \\
= \ & \{\ f \to\ \text{'('}\ \bullet e\ \text{')'} \\
& \quad e \to \bullet e\ \text{'+'}\ p,\ e \to \bullet e\ \text{'-'}\ p,\ e \to \bullet p, \\
& \quad p \to \bullet p\ \text{'*'}\ f,\ p \to \bullet p\ \text{'/'}\ f,\ p \to \bullet f, \\
& \quad f \to \bullet\ \text{'('}e\ \text{')'}\ ,\ f \to \bullet N \qquad\qquad \}
\end{aligned}$$

$= s_5.$

Damit ist die Menge der Zustände des Shift-Reduce-Parsers durch

$$Q := \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}\}$$

gegeben. Wir untersuchen als nächstes, ob es Konflikte gibt und betrachten exemplarisch die Menge s_{15}. Aufgrund der markierten Regel

$$p \to p\bullet\ \text{'*'}\ f$$

muss im Zustand s_{15} geshiftet werden, wenn das nächste Token den Wert '*' hat. Auf der anderen Seite beinhaltet der Zustand s_{15} die Regel

$$e \to e\ \text{'+'}\ p\bullet.$$

Diese Regel sagt, dass der Symbol-Stack mit der Grammatik-Regel $e \to e$ '+' p reduziert werden muss, falls in der Eingabe ein Zeichen aus der Menge $\mathit{Follow}(e)$ auftritt. Falls nun '*' $\in \mathit{Follow}(e)$ liegen würde, so hätten wir einen Shift-Reduce-Konflikt. Es gilt aber

$$\mathit{Follow}(e) = \{\ \text{'+'}\ ,\ \text{'-'}\ ,\ \text{')'}\ ,\ \text{'\$'}\ \},$$

und daraus folgt '*' $\notin \mathit{Follow}(e)$, so dass hier kein Shift-Reduce-Konflikt vorliegt. Eine Untersuchung der anderen Mengen zeigt, dass dort ebenfalls keine Shift-Reduce- oder Reduce-Reduce-Konflikte auftreten.

Als nächstes berechnen wir die Funktion *action*. Wir betrachten exemplarisch zwei Fälle.

1. Als erstes berechnen wir $\mathit{action}(s_1,\ \text{'+'}\ )$. Es gilt

$$\begin{aligned}
\mathit{action}(s_1,\ \text{'+'}\ ) &= \mathit{action}(\{p \to f\bullet\},\ \text{'+'}\ ) \\
&= \langle \mathit{reduce}, p \to f \rangle,
\end{aligned}$$

denn wir haben '+' $\in \mathit{Follow}(p)$.

2. Als nächstes berechnen wir $\mathit{action}(s_4,\ \text{'+'}\ )$. Es gilt

$$\begin{array}{lcl} \mathit{action}(s_4,\ \text{'+'}\ ) & = & \mathit{action}(\{s \to e \bullet\ \$,\ e \to e \bullet\ \text{'+'}\ p,\ e \to e \bullet\ \text{'-'}\ p\ \},\ \text{'+'}\ ) \\ & = & \langle \mathit{shift}, \mathit{closure}(\{e \to e\ \text{'+'}\ \bullet p\})\rangle \\ & = & \langle \mathit{shift}, s_8\rangle. \end{array}$$

Würden wir diese Rechnungen fortführen, so würden wir die Tabelle 9.1 erhalten, denn wir haben die Namen der Zustände so gewählt, dass diese mit den Namen der entsprechenden Zustände in den Tabellen 9.1 und 9.2 übereinstimmen.

Aufgabe 27: Abbildung 9.7 zeigt eine Grammatik für aussagenlogische Formeln in konjunktiver Normalform.

(a) Geben Sie die Mengen $\mathit{First}(v)$ für alle syntaktischen Variablem v an.

(b) Geben Sie die Mengen $\mathit{Follow}(v)$ für alle syntaktischen Variablem v an.

(c) Berechnen Sie die Menge der SLR-Zustände.

(d) Geben Sie Funktion *action* an.

(e) Geben Sie die Funktion *goto* an.

Kürzen Sie die Namen der syntaktischen Variablen und Terminale mit s, c, d, l und I ab, wobei s für das neu eingeführte Start-Symbol steht. ◇

$$\begin{array}{rcl} \mathit{cnf} & \to & \mathit{cnf}\ \text{'}\wedge\text{'}\ \mathit{disjunction} \\ & | & \mathit{disjunction} \\ \mathit{disjunction} & \to & \mathit{disjunction}\ \text{'}\vee\text{'}\ \mathit{literal} \\ & | & \mathit{literal} \\ \mathit{literal} & \to & \text{'}\neg\text{'}\ \textsc{Identifier} \\ & | & \textsc{Identifier} \end{array}$$

Figure 9.7: Eine Grammatik für Boole'sche Ausdrücke in konjunktiver Normalform.

9.3.3 Shift-Reduce- und Reduce-Reduce-Konflikte

In diesem Abschnitt untersuchen wir Shift-Reduce- und Reduce-Reduce-Konflikte genauer und betrachten dazu zwei Beispiele. Das erste Beispiel zeigt einen Shift-Reduce-Konflikt. Die in Abbildung 9.8 gezeigte Grammatik ist mehrdeutig, denn sie legt nicht fest, ob der Operator '+' stärker oder schwächer bindet als der Operator '∗': Interpretieren wir das Nicht-Terminal N als eine Abküzung für $\textsc{Number}$, so können wir mit dieser Grammatik den Ausdruck $1 + 2 * 3$ sowohl als

$$(1 + 2) * 3 \quad \text{als auch als} \quad 1 + (2 * 3)$$

lesen.

Wir berechnen zunächst den Start-Zustand s_0.

$$\begin{array}{lcl} s_0 & = & \mathit{closure}\big(\{s \to \bullet\, e\,\$\}\big) \\ & = & \big\{s \to \bullet\, e\,\$,\ e \to \bullet\, e\ \text{'+'}\ e,\ e \to \bullet\, e\ \text{'*'}\ e,\ e \to \bullet\, N\ \big\}. \end{array}$$

Als nächstes berechnen wir $s_1 := \mathit{goto}(s_0, e)$:

$$
\begin{array}{rcl}
e & \to & e \text{ '+' } e \\
& | & e \text{ '*' } e \\
& | & N
\end{array}
$$

Figure 9.8: Eine Grammatik mit Shift-Reduce-Konflikten.

$$
\begin{array}{rcl}
s_1 & = & \mathit{goto}(s_0, e) \\
& = & \mathit{closure}\big(\{s \to e \bullet \$\ e \to e \bullet \text{ '+' } e,\ e \to e \bullet \text{ '*' } e\ \}\big) \\
& = & \{s \to e \bullet \$,\ e \to e \bullet \text{ '+' } e,\ e \to e \bullet \text{ '*' } e\ \}
\end{array}
$$

Nun berechnen wir $s_2 := \mathit{goto}(s_1, \text{'+'})$:

$$
\begin{array}{rcl}
s_2 & = & \mathit{goto}(s_1, \text{'+'}) \\
& = & \mathit{closure}\big(\{e \to e \text{ '+' } \bullet e,\ \}\big) \\
& = & \{e \to e \text{ '+' } \bullet e,\ e \to \bullet e \text{ '+' } e,\ e \to \bullet e \text{ '*' } e,\ e \to \bullet N\ \}
\end{array}
$$

Als nächstes berechnen wir $s_3 := \mathit{goto}(s_2, e)$:

$$
\begin{array}{rcl}
s_3 & = & \mathit{goto}(s_2, e) \\
& = & \mathit{closure}\big(\{e \to e \text{ '+' } e\bullet,\ e \to e \bullet \text{ '+' } e,\ e \to e \bullet \text{ '*' } e\}\big) \\
& = & \{e \to e \text{ '+' } e\bullet,\ e \to e \bullet \text{ '+' } e,\ e \to e \bullet \text{ '*' } e\ \}
\end{array}
$$

Hier tritt bei der Berechnung von $\mathit{action}(s_3, \text{'*'})$ ein Shift-Reduce-Konflikt auf, denn einerseits verlangt die markierte Regel

$$e \to e \bullet \text{ '*' } e,$$

dass das Token '*' auf den Stack geschoben wird, andererseits haben wir

$$\mathit{Follow}(e) = \{ \text{ '+', '*', '\$' } \},$$

so dass, falls das nächste zu lesende Token den Wert '*' hat, der Symbol-Stack mit der Regel

$$e \to e \text{ '+' } e \bullet,$$

reduziert werden sollte.

Bemerkung: Es ist nicht weiter verwunderlich, dass wir bei der oben angegebenen Grammatik einen Konflikt gefunden haben, denn diese Grammatik ist nicht eindeutig. Demgegenüber kann gezeigt werden, dass jede SLR-Grammatik eindeutig sein muss. Folglich ist eine mehrdeutige Grammatik niemals eine SLR-Grammatik. Die Umkehrung dieser Aussage gilt jedoch nicht. Dies werden wir im nächsten Beispiel sehen. ◇

$$
\begin{array}{rcl}
s & \to & a \text{ 'x' } a \text{ 'y'} \\
& | & b \text{ 'y' } b \text{ 'x'} \\
a & \to & \varepsilon \\
b & \to & \varepsilon
\end{array}
$$

Figure 9.9: Eine Grammatik mit einem Reduce-Reduce-Konflikt.

Wir untersuchen als nächstes eine Grammatik, die keine SLR-Grammatik ist, weil Reduce-Reduce-Konflikte auftreten. Wir betrachten dazu die in Abbildung 9.9 gezeigte Grammatik. Diese Grammatik ist eindeutig, denn es gilt

$$L(s) = \{ \text{ 'xy', 'yx' } \}$$

und der String 'xy' lässt sich nur mit der Regel $s \to a$ 'x' a 'y' herleiten, während sich der String 'yx' nur mit der Regel $s \to b$ 'y' b 'x' erzeugen lässt. Um zu zeigen, dass diese Grammatik Shift-Reduce-Konflikte enthält, berechnen wir den Start-Zustand eines SLR-Parsers für diese Grammatik.

$$\begin{array}{lcl} s_0 & = & \mathit{closure}\big(\{\widehat{s} \to \bullet\, s\,\$\}\big) \\ & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'}\ ,\ s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ ,\ a \to \bullet\,\varepsilon,\ b \to \bullet\,\varepsilon\ \big\} \\ & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'}\ ,\ s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ ,\ a \to \varepsilon\bullet,\ b \to \varepsilon\bullet\ \big\}, \end{array}$$

denn $a \to \bullet\,\varepsilon$ ist dasselbe wie $a \to \varepsilon\bullet$. In diesem Zustand gibt es einen Reduce-Reduce-Konflikt zwischen den beiden markierten Regeln

$$a \to \varepsilon\,\bullet \quad \text{und} \quad b \to \varepsilon\bullet.$$

Dieser Konflikt tritt bei der Berechnung von

$$\mathit{action}(s_0,\ \texttt{'x'}\ )$$

auf, denn wir haben

$$\mathit{Follow}(a) = \big\{\texttt{'x'}, \texttt{'y'}\big\} = \mathit{Follow}(b).$$

und damit ist dann nicht klar, mit welcher dieser Regeln der Parser die Eingabe im Zustand s_0 reduzieren soll, wenn das nächste gelesene Token den Wert 'x' hat, denn dieses Token ist sowohl ein Element der Menge $\mathit{Follow}(a)$ als auch der Menge $\mathit{Follow}(b)$.

Remark: As part of the resources provided with this lecture, the file

Formal-Languages/blob/master/ANTLR4-Python/SLR-Parser-Generator/SLR-Table-Generator.ipynb

contains a *Python* program that checks whether a given grammar is an SLR grammar. This program computes the states as well as the action table of a given grammar. ◇

Exercise 28: The github directory containing supplementary files for this lecture contains the grammar for the programming language `C` at

Formal-Languages/blob/master/ANTLR4-Python/SLR-Parser-Generator/Examples/c-grammar.g

This grammar is not an SLR-grammar. Use the SLR-table generator SLR-Table-Genarator.ipynb to investigate the conflicts that arise. Transform the grammar into an SLR grammar by making a small number of changes. It is sufficient to remove three rules. After removing those rules, there will be one useless rule left which is of the form

$$a \to b.$$

This rule has to be removed, too. Furthermore, you have to rename the variable a occurring on the left hand side of this rule to b everywhere. ◇

9.4 Kanonische LR-Parser

Der Reduce-Reduce-Konflikt, der in der in Abbildung 9.9 gezeigten Grammatik auftritt, kann wie folgt gelöst werden: In dem Zustand

$$\begin{array}{lcl} s_0 & = & \mathit{closure}\big(\{\widehat{s} \to \bullet\, s\,\$\}\big) \\ & = & \big\{\widehat{s} \to \bullet\, s\,\$,\ s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'}\ ,\ s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ ,\ a \to \varepsilon\bullet,\ b \to \varepsilon\bullet\ \big\} \end{array}$$

kommen die markierten Regeln $a \to \varepsilon\bullet$ und $b \to \varepsilon\bullet$ von der Berechnung des Abschlusses der Regeln

$$s \to \bullet\, a\ \texttt{'x'}\ a\ \texttt{'y'} \quad \text{und} \quad s \to \bullet\, b\ \texttt{'y'}\ b\ \texttt{'x'}\ .$$

Bei der ersten Regel ist klar, dass auf das erste a ein 'x' folgen muss, bei der zweiten Regel sehen wir, dass auf das erste b ein 'y' folgt. Diese Information geht über die Information hinaus, die in den Mengen $\mathit{Follow}(a)$ bzw. $\mathit{Follow}(b)$ enthalten ist, denn jetzt berücksichtigen wir den Kontext, in dem die syntaktische Variable auftaucht. Damit können wir die Funktion $\mathit{action}(s_0, \texttt{'x'})$ und $\mathit{action}(s_0, \texttt{'y'})$ wie folgt definieren:

$$\mathit{action}(s_0, \texttt{'x'}) = \langle \texttt{reduce}, a \to \varepsilon \rangle \quad \text{und} \quad \mathit{action}(s_0, \texttt{'y'}) = \langle \texttt{reduce}, b \to \varepsilon \rangle.$$

Durch diese Definition wird der Reduce-Reduce-Konflikt gelöst. Die zentrale Idee ist, bei der Berechnung des Abschlusses den Kontext, in dem eine Regel auftritt, mit einzubeziehen. Dazu erweitern wir zunächst die Definition einer markierten Regel.

Definition 27 (erweiterte markierte Regel) Eine erweiterte markierte Regel (abgekürzt: e.m.R.) einer Grammatik $G = \langle V, T, R, s \rangle$ ist ein Quadrupel

$$\langle a, \beta, \gamma, L \rangle,$$

wobei gilt:

1. $(a \to \beta\gamma) \in R$.
2. $L \subseteq T$.

Wir schreiben die erweiterte markierte Regel $\langle a, \beta, \gamma, L \rangle$ als

$$a \to \beta \bullet \gamma : L.$$

Falls L nur aus einem Element t besteht, falls also $L = \{t\}$ gilt, so lassen wir die Mengen-Klammern weg und schreiben die Regel als

$$a \to \beta \bullet \gamma : t.$$ □

Anschaulich interpretieren wir die e.m.R. $a \to \beta \bullet \gamma : L$ als einen Zustand, in dem folgendes gilt:

1. Der Parser versucht, ein a mit Hilfe der Grammatik-Regel $a \to \beta\gamma$ zu erkennen.
2. Dabei wurde bereits β erkannt. Damit die Regel $a \to \beta\gamma$ angewendet werden kann, muss nun γ erkannt werden.
3. Wir wissen zusätzlich, dass auf die syntaktische Variable a ein Token aus der Menge L folgen muss.

   Die Menge L bezeichnen wir daher als die Menge der Folge-Token.

Mit erweiterten markierten Regeln arbeitet es sich ganz ähnlich wie mit markierten Regeln, allerdings müssen wir die Definitionen der Funktionen *closure*, *goto* und *action* etwas modifizieren. Wir beginnen mit der Funktion *closure*.

Definition 28 (closure($\mathcal{M}$)) Es sei $\mathcal{M}$ eine Menge erweiterter markierter Regeln. Dann definieren wir den Abschluss von $\mathcal{M}$ als die kleinste Menge $\mathcal{K}$ markierter Regeln, für die folgendes gilt:

1. $\mathcal{M} \subseteq \mathcal{K}$,

   der Abschluss umfasst also die ursprüngliche Regel-Menge.
2. Ist einerseits

   $$a \to \beta \bullet c\,\delta : L$$

   eine e.m.R. aus der Menge $\mathcal{K}$, wobei c eine syntaktische Variable ist, und ist andererseits

   $$c \to \gamma$$

   eine Grammatik-Regel der zu Grunde liegenden Grammatik G, so ist auch die e.m.R.

   $$c \to \bullet\,\gamma : \bigcup \{\mathit{First}(\delta\, t) \mid t \in L\}$$

   ein Element der Menge $\mathcal{K}$. Die Funktion $\mathit{First}(\alpha)$ berechnet dabei für einen String $\alpha \in (T \cup V)^*$ die Menge aller Token t, mit denen ein String beginnen kann, der von α abgeleitet worden ist.

Die so definierte eindeutig bestimmte Menge $\mathcal{K}$ wird wieder mit $\mathit{closure}(\mathcal{M})$ bezeichnet. □

Bemerkung: Gegenüber der alten Definition ist nur die Berechnung der Menge der Folge-Token hinzu gekommen. Der Kontext, in dem das c auftritt, das mit der Regel $c \to \gamma$ erkannt werden soll, ist zunächst durch den String δ gegeben, der in der Regel $a \to \beta \bullet c\,\delta : L$ auf das c folgt. Möglicherweise leitet δ den leeren String ε ab. In diesen Fall

spielen auch die Folge-Token aus der Menge L eine Rolle, denn falls $\delta \Rightarrow^* \varepsilon$ gilt, kann auf das c auch ein Folge-Token t aus der Menge L folgen. □

Für eine gegebene e.m.R.-Menge $\mathcal{M}$ kann die Berechnung von $\mathcal{K} := \mathit{closure}(\mathcal{M})$ iterativ erfolgen. Abbildung 9.10 zeigt die Berechnung von $\mathit{closure}(\mathcal{M})$. Der wesentliche Unterschied gegenüber der früheren Berechnung von $\mathit{closure}()$ ist, dass wir bei den e.m.R.s, die wir für eine Variable c mit in $\mathit{closure}(\mathcal{M})$ aufnehmen, bei der Menge der Folge-Token den Kontext berücksichtigen, in dem c auftritt. Dadurch gelingt es, die Zustände des Parsers präziser zu beschreiben, als dies bei markierten Regeln der Fall ist.

```
1  function closure(M) {
2      K  := M;
3      K⁻ := {};
4      while (K⁻ ≠ K) {
5          K⁻ := K;
6          K  := K ∪ {(c → •γ : ⋃{First(δ t) | t ∈ L}) | (a → β • c δ : L) ∈ K ∧ (c → γ) ∈ R};
7      }
8      return K;
9  }
```

Figure 9.10: Berechnung von $\mathit{closure}(\mathcal{M})$

Bemerkung: Der Ausdruck $\bigcup\{\mathit{First}(\delta\, t) \mid t \in L\}$ sieht komplizierter aus, als er tatsächlich ist. Wollen wir diesen Ausdruck berechnen, so ist es zweckmäßig eine Fallunterscheidung danach durchzuführen, ob δ den leeren String ε ableiten kann oder nicht, denn es gilt

$$\bigcup\{\mathit{First}(\delta\, t) \mid t \in L\} = \begin{cases} \mathit{First}(\delta) \cup L & \text{falls } \delta \Rightarrow^* \varepsilon; \\ \mathit{First}(\delta) & \text{sonst.} \end{cases}$$

Die Berechnung von $\mathit{goto}(\mathcal{M}, t)$ für eine Menge $\mathcal{M}$ von erweiterten Regeln und ein Zeichen x ändert sich gegenüber der Berechnung im Falle einfacher markierter Regeln nur durch das Anfügen der Menge von Folge-Tokens, die aber selbst unverändert bleibt:

$$\mathit{goto}(\mathcal{M}, x) := \mathit{closure}\Big(\big\{a \to \beta\, x \bullet \delta : L \mid (a \to \beta \bullet x\, \delta : L) \in \mathcal{M}\big\}\Big).$$

Ähnlich wie bei der Theorie der SLR-Parser augmentieren wir unsere Grammatik G, indem wir der Menge der Variable eine neue Start-Variable $\widehat{s}$ und der Menge der Regeln die neue Regel $\widehat{s} \to s$ hinzufügen. Weiter fügen wir den Token das Symbol \$ hinzu. Dann hat der Start-Zustand die Form

$$q_0 := \mathit{closure}\big(\{\widehat{s} \to \bullet\, s : \$\}\big),$$

denn auf das Start-Symbol muss das Datei-Ende "\$" folgen. Als letztes zeigen wir, wie die Definition der Funktion $\mathit{action}()$ geändert werden muss. Wir spezifizieren die Berechnung dieser Funktion durch die folgenden bedingten Gleichungen.

1. $(a \to \beta \bullet t\, \delta : L) \in \mathcal{M} \implies \mathit{action}(\mathcal{M}, t) := \langle \texttt{shift}, \mathit{goto}(\mathcal{M}, t)\rangle$.
2. $(a \to \beta\bullet : L) \in \mathcal{M} \;\wedge\; a \neq \widehat{s} \;\wedge\; t \in L \implies \mathit{action}(\mathcal{M}, t) := \langle \texttt{reduce}, a \to \beta\rangle$.
3. $(\widehat{s} \to s\bullet : \$) \in \mathcal{M} \implies \mathit{action}(\mathcal{M}, \$) := \texttt{accept}$.
4. Sonst: $\mathit{action}(\mathcal{M}, t) := \texttt{error}$.

Falls es bei diesen Gleichungen zu einem Konflikt kommt, weil gleichzeitig die Bedingung der ersten Gleichung als auch die Bedingung der zweiten Gleichung erfüllt ist, so sprechen wir wieder von einem Shift-Reduce-Konflikt. Ein Shift-Reduce-Konflikt liegt also bei der Berechnung von $\mathit{action}(\mathcal{M}, t)$ dann vor, wenn es zwei e.m.R.s

$$(a \to \beta \bullet t\, \delta : L_1) \in \mathcal{M} \quad \text{und} \quad (c \to \gamma\bullet : L_2) \in \mathcal{M} \quad \text{mit } t \in L_2$$

gibt, denn dann ist nicht klar, ob im Zustand $\mathcal{M}$ das Token t auf den Stack geschoben werden soll, oder ob stattdessen der Symbol-Stack mit der Regel $c \to \gamma$ reduziert werden muss.

Bemerkung: Gegenüber einem SLR-Parser ist die Möglichkeit von Shift-Reduce-Konflikten verringert, denn bei einem SLR-Parser liegt bereits dann ein Shift-Reduce-Konflikt vor, wenn $t \in \mathit{Follow}(c)$ gilt und die Menge L_2 ist in der Regel kleiner als die Menge $\mathit{Follow}(c)$. ◇

Ein Reduce-Reduce-Konflikt liegt vor, wenn es zwei e.m.R.s

$$(a \to \beta \bullet : L_1) \in \mathcal{M} \quad \text{und} \quad (c \to \delta \bullet : L_2) \in \mathcal{M} \quad \text{mit} \quad L_1 \cap L_2 \neq \{\}$$

gibt, denn dann ist nicht klar, mit welcher dieser beiden Regeln der Symbol-Stack reduziert werden soll, wenn das nächste Token ein Element der Schnittmenge $L_1 \cap L_2$ ist.

Bemerkung: Gegenüber einem SLR-Parser ist die Möglichkeit von Reduce-Reduce-Konflikten verringert, denn bei einem SLR-Parser liegt bereits dann ein Reduce-Reduce-Konflikt vor, wenn es ein t in der Menge $\mathit{Follow}(a) \cap \mathit{Follow}(c)$ gibt und die *Follow*-Mengen sind oft größer als die Mengen L_1 und L_2. ◇

Beispiel: Wir greifen das Beispiel der in Abbildung 9.9 gezeigten Grammatik wieder auf und berechnen zunächst die Menge aller Zustände.

1. $s_0 := \mathit{closure}\Big(\big\{\widehat{s} \to \bullet\, s : \$\big\}\Big)$
   $= \big\{\widehat{s} \to \bullet\, s : \$, s \to \bullet\, a\texttt{'x'}a\texttt{'y'} : \$, s \to \bullet\, b\texttt{'y'}b\texttt{'x'} : \$, a \to \bullet\, : \texttt{'x'}, b \to \bullet\, : \texttt{'y'}\big\}.$
2. $s_1 := \mathit{goto}(s_0, a)$
   $= \mathit{closure}\big(\{s \to a \bullet \texttt{'x'}a\texttt{'y'} : \$\}\big)$
   $= \big\{s \to a \bullet \texttt{'x'}a\texttt{'y'} : \$\big\}.$
3. $s_2 := \mathit{goto}(s_0, s)$
   $= \mathit{closure}\big(\{\widehat{s} \to s\bullet : \$\}\big)$
   $= \big\{\widehat{s} \to s\bullet : \$\big\}.$
4. $s_3 := \mathit{goto}(s_0, b)$
   $= \mathit{closure}\big(\{s \to b \bullet \texttt{'y'}b\texttt{'x'} : \$\}\big)$
   $= \big\{s \to b \bullet \texttt{'y'}b\texttt{'x'} : \$\big\}.$
5. $s_4 := \mathit{goto}(s_3, \texttt{'y'})$
   $= \mathit{closure}\big(\{s \to b\texttt{'y'} \bullet b\texttt{'x'} : \$\}\big)$
   $= \big\{s \to b\texttt{'y'} \bullet b\texttt{'x'} : \$, b \to \bullet\, : \texttt{'x'}\big\}.$
6. $s_5 := \mathit{goto}(s_4, b)$
   $= \mathit{closure}\big(\{s \to b\texttt{'y'}b \bullet \texttt{'x'} : \$\}\big)$
   $= \big\{s \to b\texttt{'y'}b \bullet \texttt{'x'} : \$\big\}.$
7. $s_6 := \mathit{goto}(s_5, \texttt{'x'})$
   $= \mathit{closure}\big(\{s \to b\texttt{'y'}b\texttt{'x'}\bullet : \$\}\big)$
   $= \big\{s \to b\texttt{'y'}b\texttt{'x'}\bullet : \$\big\}.$
8. $s_7 := \mathit{goto}(s_1, \texttt{'x'})$
   $= \mathit{closure}\big(\{s \to a\texttt{'x'} \bullet a\texttt{'y'} : \$\}\big)$
   $= \big\{s \to a\texttt{'x'} \bullet a\texttt{'y'} : \$, a \to \bullet\, : \texttt{'y'}\big\}.$
9. $s_8 := \mathit{goto}(s_7, a)$
   $= \mathit{closure}\big(\{s \to a\texttt{'x'}a \bullet \texttt{'y'} : \$\}\big)$
   $= \big\{s \to a\texttt{'x'}a \bullet \texttt{'y'} : \$\big\}.$
10. $s_9 := \mathit{goto}(s_8, \texttt{'y'})$
    $= \mathit{closure}\big(\{s \to a\texttt{'x'}a\texttt{'y'}\bullet : \$\}\big)$
    $= \big\{s \to a\texttt{'x'}a\texttt{'y'}\bullet : \$\big\}.$

Als nächstes untersuchen wir, ob es bei den Zuständen Konflikte gibt. Beim Start-Zustand s_0 hatten wir im letzten Abschnitt einen Reduce-Reduce-Konflikt zwischen den beiden Regeln $a \to \varepsilon$ und $b \to \varepsilon$ gefunden, weil

$$\mathit{Follow}(a) \cap \mathit{Follow}(b) = \{\texttt{'x'}, \texttt{'y'}\} \neq \{\}$$

gilt. Dieser Konflikt ist nun verschwunden, denn zwischen den e.m.R.s

$$a \to \bullet : \texttt{'x'} \quad \text{und} \quad b \to \bullet : \texttt{'y'}$$

gibt es wegen 'x' $\neq$ 'y' keinen Konflikt. Es ist leicht zu sehen, dass auch bei den anderen Zustände keine Konflikte auftreten.

Aufgabe 29: Berechnen Sie die Menge der Zustände eines LR-Parsers für die folgende Grammatik:

$$\begin{array}{lcl} e & \to & e\ \texttt{'+'}\ p \\ & \mid & p \\ p & \to & p\ \texttt{'*'}\ f \\ & \mid & f \\ f & \to & \texttt{'('}\, e\, \texttt{')'} \\ & \mid & \mathit{Number} \end{array}$$

Untersuchen Sie außerdem, ob es bei dieser Grammatik Shift-Reduce-Konflikte oder Reduce-Reduce-Konflikte gibt.

Remark: As part of the resources provided with this lecture, the file

ANTLR4-Python/LR-Parser-Generator/LR-Table-Generator.ipynb

contains a *Python* program that checks whether a given grammar qualifies as a canonical LR grammar. This program computes the LR-states as well as the action table for a given grammar. ◇

Remark: The theory of LR-parsing has been developed by Donald E. Knuth [Knu65]. His theory is described in the paper "On the translation of languages from left to right". ◇

9.5 LALR-Parser

Die Zahl der Zustände eines LR-Parsers ist oft erheblich größer als die Zahl der Zustände, die ein SLR-Parser derselben Grammatik hätte. Beispielsweise kommt ein SLR-Parser für die C-Grammatik mit 349 Zuständen aus. Da die Sprache C keine SLR-Sprache ist, gibt es beim Erzeugen einer SLR-Parse-Tabelle für C allerdings eine Reihe von Konflikten, so dass ein SLR-Parser für die Sprache C nicht funktioniert. Demgegenüber kommt ein LR-Parser für die Sprache C auf 1572 Zustände, wie Sie hier sehen können. In den siebziger Jahren, als der zur Verfügung stehende Haupt-Speicher der meisten Rechner noch bescheidener dimensioniert war, als dies heute der Fall ist, hatten LR-Parser daher eine für die Praxis problematische Größe. Eine genaue Analyse der Menge der Zustände von LR-Parsern zeigte, dass es oft möglich ist, bestimmte Zustände zusammen zu fassen. Dadurch kann die Menge der Zustände in den meisten Fällen deutlich verkleinert werden. Wir illustrieren das Konzept an einem Beispiel und betrachten die in Abbildung 9.11 gezeigte Grammatik, die ich dem Drachenbuch [ASUL06] entnommen habe. (Das "Drachenbuch" ist das Standardwerk im Bereich Compilerbau.)

Abbildung 9.12 zeigt den sogenannten LR-Goto-Graphen für diese Grammatik. Die Knoten dieses Graphen sind die Zustände. Betrachten wir den LR-Goto-Graphen, so stellen wir fest, dass die Zustände s_6 und s_3 sich nur in den Mengen der Folge-Token unterscheiden, denn es gilt einerseits

$$s_6 = \Big\{ s \to \texttt{'x'} \bullet c : \texttt{'\$'}, c \to \bullet\ \texttt{'x'}\ c : \texttt{'\$'}, c \to \bullet\ \texttt{'y'}\ : \texttt{'\$'} \Big\},$$

und andererseits haben wir

$$s_3 = \Big\{ s \to \texttt{'x'} \bullet c : \{\texttt{'x'}, \texttt{'y'}\}, c \to \bullet\ \texttt{'x'}\ c : \{\texttt{'x'}, \texttt{'y'}\}, c \to \bullet\ \texttt{'y'}\ : \{\texttt{'x'}, \texttt{'y'}\} \Big\}.$$

Offenbar entsteht die Menge s_3 aus der Menge s_6 indem überall '$' durch die Menge {'x', 'y'} ersetzt wird. Genauso

$$
\begin{array}{rcl}
\widehat{s} & \to & s \\
s & \to & c\ c \\
c & \to & \texttt{'x'}\ c \\
 & | & \texttt{'y'}
\end{array}
$$

Figure 9.11: Eine Grammatik aus dem Drachenbuch.

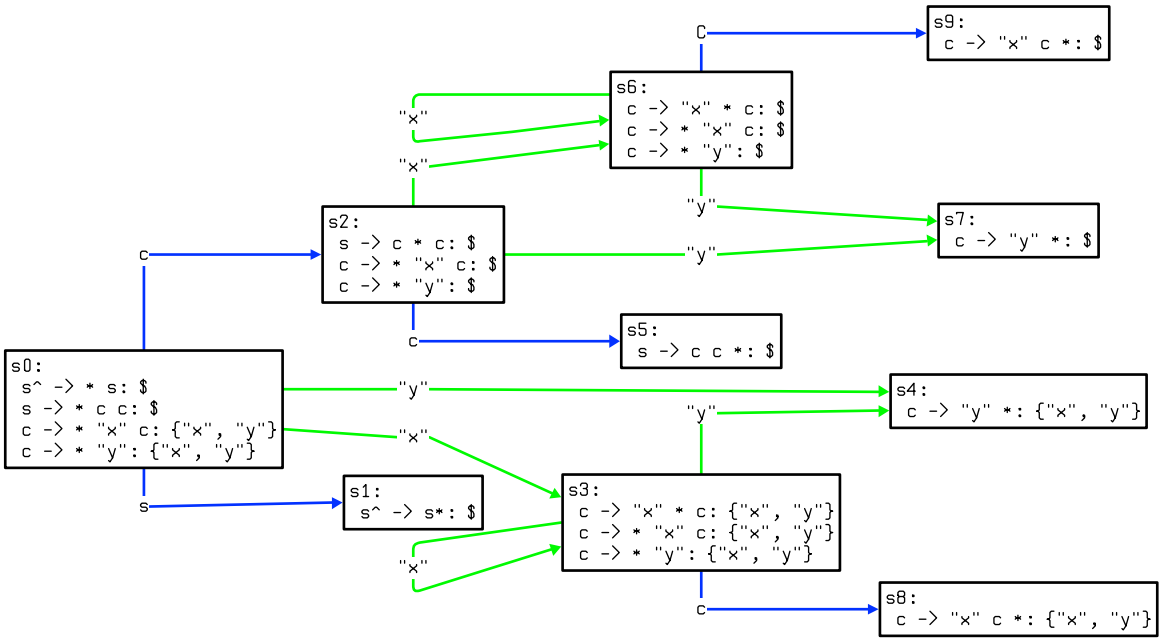

Figure 9.12: LR-Goto-Graph für die Grammatik aus Abbildung 9.11.

kann die Menge s_7 in s_4 und s_9 in s_8 überführt werden. Die entscheidende Erkenntnis ist nun, dass die Funktion *goto*() unter dieser Art von Transformation invariant ist, denn bei der Definition dieser Funktion spielt die Menge der Folge-Token keine Rolle. So sehen wir zum Beispiel, dass einerseits

$$\mathit{goto}(s_3, c) = s_8 \quad \text{und} \quad \mathit{goto}(s_6, c) = s_9$$

gilt und dass andererseits der Zustand s_9 in den Zustand s_8 übergeht, wenn wir überall in s_9 das Terminal '$' durch die Menge $\{\texttt{'x'}, \texttt{'y'}\}$ ersetzen. Definieren wir den Kern einer Menge von erweiterten markierten Regeln dadurch, dass wir in jeder Regel die Menge der Folgetoken wegstreichen, und fassen dann Zustände mit demselben Kern zusammen, so erhalten wir den in Abbildung 9.13 gezeigten Goto-Graphen.

Um die Beobachtungen, die wir bei der Betrachtung der in Abbildung 9.11 gezeigten Grammatik gemacht gaben, verallgemeinern und formalisieren zu können, definieren wir ein Funktion *core*(), die den Kern einer Menge von e.m.R.s berechnet und damit diese Menge in eine Menge markierter Regeln überführt:

$$\mathit{core}(\mathcal{M}) := \{a \to \beta \bullet \gamma \mid (a \to \beta \bullet \gamma : L) \in \mathcal{M}\}.$$

Die Funktion *core*() entfernt also einfach die Menge der Folge-Tokens von den e.m.R.s. Wir hatten die Funktion *goto*() für eine Menge $\mathcal{M}$ von erweiterten markierten Regeln und ein Symbol x durch

$$\mathit{goto}(\mathcal{M}, x) := \mathit{closure}\Big(\big\{a \to \beta\, x \bullet \gamma : L \mid (a \to \beta \bullet x\, \gamma : L) \in \mathcal{M}\big\}\Big).$$

definiert. Offenbar spielt die Menge der Folge-Token bei der Berechnung von $\mathit{goto}(\mathcal{M}, x)$ keine Rolle, formal gilt für

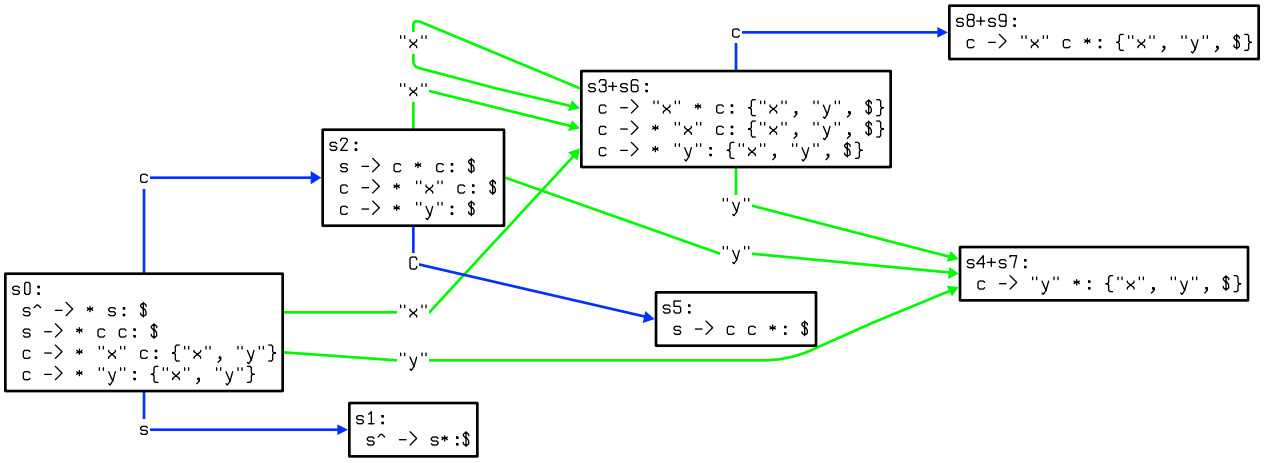

Figure 9.13: Der LALR-Goto-Graph für die Grammatik aus Abbildung 9.11.

zwei e.m.R.-Mengen $\mathcal{M}_1$ und $\mathcal{M}_2$ und ein Symbol x die Formel:

$$\mathit{core}(\mathcal{M}_1) = \mathit{core}(\mathcal{M}_2) \;\Rightarrow\; \mathit{core}(\mathit{goto}(\mathcal{M}_1, x)) = \mathit{core}(\mathit{goto}(\mathcal{M}_2, x)).$$

Für zwei e.m.R.-Mengen $\mathcal{M}$ und $\mathcal{N}$, die den gleichen Kern haben, definieren wir die erweiterte Vereinigung $\mathcal{M} \uplus \mathcal{N}$ von $\mathcal{M}$ und $\mathcal{N}$ als

$$\mathcal{M} \uplus \mathcal{N} := \{a \to \beta \bullet \gamma : K \cup L \mid (a \to \beta \bullet \gamma : K) \in \mathcal{M} \;\wedge\; (a \to \beta \bullet \gamma : L) \in \mathcal{N}\}.$$

Diese Definition verallgemeinern wir zu einer Operation $\biguplus$, die auf einer Menge von Mengen von e.m.R.s definiert ist: Ist $\mathfrak{I}$ eine Menge von Mengen von e.m.R.s, die alle den gleichen Kern haben, gilt also

$$\mathfrak{I} = \{\mathcal{M}_1, \cdots, \mathcal{M}_k\} \quad \text{mit} \quad \mathit{core}(\mathcal{M}_i) = \mathit{core}(\mathcal{M}_j) \quad \text{für alle } i, j \in \{1, \cdots, k\},$$

so definieren wir

$$\biguplus \mathfrak{I} := \mathcal{M}_1 \uplus \cdots \uplus \mathcal{M}_k.$$

Es sei nun Δ die Menge aller Zustände eines LR-Parsers. Dann ist die Menge der Zustände des entsprechenden LALR-Parsers durch die erweiterte Vereinigung der Menge aller der Teilmengen von Δ gegeben, deren Elemente den gleichen Kern haben:

$$\mathfrak{Q} := \left\{ \biguplus \mathfrak{I} \mid \mathfrak{I} \in 2^{\Delta} \wedge \forall \mathcal{M}, \mathcal{N} \in \mathfrak{I} : \mathit{core}(\mathcal{M}) = \mathit{core}(\mathcal{N}) \wedge \text{und } \mathfrak{I} \text{ maximal} \right\}.$$

Die Forderung "$\mathfrak{I}$ maximal" drückt in der obigen Definition aus, dass in $\mathfrak{I}$ tatsächlich alle Mengen aus Δ zusamengefasst sind, die den selben Kern haben. Die so definierte Menge $\mathfrak{Q}$ ist die Menge der LALR-Zustände.

Als nächstes überlegen wir, wie sich die Berechnung von $\mathit{goto}(\mathcal{M}, X)$ ändern muss, wenn $\mathcal{M}$ ein Element der Menge $\mathfrak{Q}$ der LALR-Zustände ist. Zur Berechnung von $\mathit{goto}(\mathcal{M}, X)$ berechnen wir zunächst die Menge

$$\mathit{closure}\Big(\big\{A \to \alpha X \bullet \beta : L \mid (A \to \alpha \bullet X\beta : L) \in \mathcal{M}\big\}\Big).$$

Das Problem ist, dass diese Menge im Allgemeinen kein Element der Menge $\mathfrak{Q}$ ist, denn die Zustände in $\mathfrak{Q}$ entstehen ja durch die Zusammenfassung mehrerer LR-Zustände. Die Zustände, die bei der Berechnung von $\mathfrak{Q}$ zusammengefasst werden, haben aber alle den selben Kern. Daher enthält die Menge

$$\Big\{q \in \mathfrak{Q} \mid \mathit{core}(q) = \mathit{core}\big(\mathit{closure}\big(\{a \to \beta\, x \bullet \gamma : L \mid (a \to \beta \bullet x\, \gamma : L) \in \mathcal{M}\}\big)\big)\Big\}$$

genau ein Element und dieses Element ist der Wert von $\mathit{goto}(\mathcal{M}, X)$. Folglich können wir

$$\mathit{goto}(\mathcal{M}, X) := \mathit{arb}\Big(\Big\{q \in \mathfrak{Q} \mid \mathit{core}(q) = \mathit{core}\Big(\mathit{closure}\Big(\big\{a \to \beta\, X \bullet \gamma : L \mid (a \to \beta \bullet X\, \gamma : L) \in \mathcal{M}\big\}\Big)\Big)\Big\}\Big)$$

setzen. Die hier verwendete Funktion *arb*() dient dazu, ein beliebiges Element aus einer Menge zu extrahieren. Da die Menge, aus der hier das Element extrahiert wird, genau ein Element enthält, ist *goto*$(\mathcal{M}, x)$ wohldefiniert. Die Berechnung des Ausdrucks *action*$(\mathcal{M}, t)$ ändert sich gegenüber der Berechnung für einen LR-Parser nicht.

9.6 Vergleich von SLR-, LR- und LALR-Parsern

Wir wollen nun die verschiedenen Methoden, mit denen wir in diesem Kapitel Shift-Reduce-Parser konstruiert haben, vergleichen. Wir nennen eine Sprache $\mathcal{L}$ eine SLR-Sprache, wenn $\mathcal{L}$ von einem SLR-Parser erkannt werden kann. Die Begriffe kanonische LR-Sprache und LALR-Sprache werden analog definiert. Zwischen diesen Sprachen bestehen die folgende Beziehungen:

$$\text{SLR-Sprache} \subsetneq \text{LALR-Sprache} \subsetneq \text{kanonische LR-Sprache} \tag{$\star$}$$

Diese Inklusionen sind leicht zu verstehen: Bei der Definition der LR-Parser hatten wir zu den markierten Regeln Mengen von Folge-Token hinzugefügt. Dadurch war es möglich, in bestimmten Fällen Shift-Reduce- und Reduce-Reduce-Konflikte zu vermeiden. Da die Zustands-Mengen der kanonischen LR-Parser unter Umständen sehr groß werden können, hatten wir dann wieder solche Mengen von erweiterten markierten Regeln zusammengefasst, die den gleichen Kern haben. So hatten wir die LALR-Parser erhalten. Durch die Zusammenfassung von Regel-Menge können wir uns allerdings in bestimmten Fällen Reduce-Reduce-Konflikte einhandeln, so dass die Menge der LALR-Sprachen eine Untermenge der kanonischen LR-Sprachen ist.

Wir werden in den folgenden Unterabschnitten zeigen, dass die Inklusionen in $(\star)$ echt sind.

9.6.1 SLR-Sprache $\subsetneq$ LALR-Sprache

Die Zustände eines LALR-Parsers enthalten gegenüber den Zuständen eines SLR-Parsers noch Mengen von Folge-Token. Damit sind LALR-Parser mindestens genauso mächtig wie SLR-Parser. Wir zeigen nun, dass LALR-Parser tatsächlich mächtiger als SLR-Parser sind. Um diese Behauptung zu belegen, präsentieren wir eine Grammatik, für die es zwar einen LALR-Parser, aber keinen SLR-Parser gibt. Wir hatten auf Seite 119 gesehen, dass die Grammatik

$$s \;\to\; a\;\texttt{'x'}\;a\;\texttt{'y'} \;\mid b\;\texttt{'y'}\;b\;\texttt{'x'}\;, \quad a \;\to\; \varepsilon, \quad b \;\to\; \varepsilon$$

keine SLR-Grammatik ist. Später hatten wir gesehen, dass diese Grammatik von einem kanonischen LR-Parser geparst werden kann. Wir zeigen nun, dass diese Grammatik auch von einem LALR-Parser geparst werden kann. Dazu berechnen wir die Menge der LALR-Zustände. Dazu ist zunächst die Menge der kanonischen LR-Zustände zu berechnen. Diese Berechnung hatten wir bereits früher durchgeführt und dabei die folgenden Zustände erhalten:

1. $s_0 = \big\{\widehat{s} \to \bullet\, s : \$, s \to \bullet\, a\texttt{'x'}a\texttt{'y'} : \$, s \to \bullet\, b\texttt{'y'}b\texttt{'x'} : \$, a \to \bullet\, : \texttt{'x'}, b \to \bullet\, : \texttt{'y'}\big\}$,
2. $s_1 = \big\{s \to a \bullet\, \texttt{'x'}a\texttt{'y'} : \$\big\}$,
3. $s_2 = \big\{\widehat{s} \to s\bullet : \$\big\}$,
4. $s_3 = \big\{s \to b \bullet\, \texttt{'y'}b\texttt{'x'} : \$\big\}$,
5. $s_4 = \big\{s \to b\texttt{'y'} \bullet\, b\texttt{'x'} : \$, b \to \bullet\, : \texttt{'x'}\big\}$,
6. $s_5 = \big\{s \to b\texttt{'y'}b \bullet\, \texttt{'x'} : \$\big\}$,
7. $s_6 = \big\{s \to b\texttt{'y'}b\texttt{'x'}\bullet : \$\big\}$,
8. $s_7 = \big\{s \to a\texttt{'x'} \bullet\, a\texttt{'y'} : \$, a \to \bullet\, : \texttt{'y'}\big\}$,
9. $s_8 = \big\{s \to a\texttt{'x'}a \bullet\, \texttt{'y'} : \$\big\}$,
10. $s_9 = \big\{s \to a\texttt{'x'}a\texttt{'y'}\bullet : \$\big\}$.

Wir stellen fest, dass die Kerne aller hier aufgelisteten Zustände verschieden sind. Damit stimmt bei dieser Grammatik die Menge der Zustände des LALR-Parser mit der Menge der Zustände des kanonischen LR-Parsers überein. Daraus folgt, dass es auch bei den LALR-Zuständen keine Konflikte gibt, denn beim Übergang von kanonischen LR-Parsern zu LALR-Parsern haben wir lediglich Zustände mit gleichem Kern zusammengefasst, die Definition der Funktionen *goto*() und *action*() blieb unverändert.

9.6.2 LALR-Sprache $\subsetneq$ kanonische LR-Sprache

Wir hatten LALR-Parser dadurch definiert, dass wir verschiedene Zustände eines kanonischen LR-Parsers zusammengefasst haben. Damit ist klar, dass kanonische LR-Parser mindestens so mächtig sind wie LALR-Parser. Um zu zeigen, dass kanonische LR-Parser tatsächlich mächtiger sind als LALR-Parser, benötigen wir eine Grammatik, für die sich zwar ein kanonischer LR-Parser, aber kein LALR-Parser erzeugen lässt. Abbildung 9.14 zeigt eine solche Grammatik, die ich dem Drachenbuch entnommen habe.

$$
\begin{array}{lcl}
s & \rightarrow & \texttt{'v'}\ a\ \texttt{'y'} \\
 & | & \texttt{'w'}\ b\ \texttt{'y'} \\
 & | & \texttt{'v'}\ b\ \texttt{'z'} \\
 & | & \texttt{'w'}\ a\ \texttt{'z'} \\
a & \rightarrow & \texttt{'x'} \\
b & \rightarrow & \texttt{'x'}
\end{array}
$$

Figure 9.14: Eine kanonische LR-Grammatik, die keine LALR-Grammatik ist.

Wir berechnen zunächst die Menge der Zustände eines kanonischen LR-Parsers für diese Grammatik. Wir erhalten dabei die folgenden Mengen von erweiterten markierten Regeln:

1. $s_0 = \mathit{closure}(\widehat{s} \rightarrow \bullet\ s : \$) = \{\ \begin{array}{lcl} \widehat{s} & \rightarrow & \bullet\, s : \$, \\ s & \rightarrow & \bullet\texttt{'v'}a\texttt{'y'} : \$, \\ s & \rightarrow & \bullet\texttt{'v'}b\texttt{'z'} : \$, \\ s & \rightarrow & \bullet\texttt{'w'}a\texttt{'z'} : \$, \\ s & \rightarrow & \bullet\texttt{'w'}b\texttt{'y'} : \$\ \}, \end{array}$
2. $s_1 = \mathit{goto}(s_0, s) = \{\widehat{s} \rightarrow s\bullet : \$\}$
3. $s_2 = \mathit{goto}(s_0,\ \texttt{'v'}\ ) = \{\ \begin{array}{lcl} s & \rightarrow & \texttt{'v'} \bullet b\texttt{'z'} : \$, \\ s & \rightarrow & \texttt{'v'} \bullet a\texttt{'y'} : \$, \\ a & \rightarrow & \bullet\texttt{'x'} : \texttt{'y'}, \\ b & \rightarrow & \bullet\texttt{'x'} : \texttt{'z'}\ \}, \end{array}$
4. $s_3 = \mathit{goto}(s_0,\ \texttt{'w'}\ ) = \{\ \begin{array}{lcl} s & \rightarrow & \texttt{'w'} \bullet a\texttt{'z'} : \$, \\ s & \rightarrow & \texttt{'w'} \bullet b\texttt{'y'} : \$, \\ a & \rightarrow & \bullet\texttt{'x'} : \texttt{'z'}, \\ b & \rightarrow & \bullet\texttt{'x'} : \texttt{'y'}\ \}, \end{array}$
5. $s_4 = \mathit{goto}(s_2,\ \texttt{'x'}\ ) = \{a \rightarrow \texttt{'x'}\bullet : \texttt{'y'},\ b \rightarrow \texttt{'x'}\bullet : \texttt{'z'}\},$
6. $s_5 = \mathit{goto}(s_3,\ \texttt{'x'}\ ) = \{a \rightarrow \texttt{'x'}\bullet : \texttt{'z'},\ b \rightarrow \texttt{'x'}\bullet : \texttt{'y'}\},$
7. $s_6 = \mathit{goto}(s_2, a) = \{s \rightarrow \texttt{'v'}a \bullet \texttt{'y'} : \$\},$
8. $s_7 = \mathit{goto}(s_6,\ \texttt{'y'}\ ) = \{s \rightarrow \texttt{'v'}a\texttt{'y'}\bullet : \$\},$
9. $s_8 = \mathit{goto}(s_2, b) = \{s \rightarrow \texttt{'v'}b \bullet \texttt{'z'} : \$\},$

10. $s_9 = \mathit{goto}(s_8, \text{'z'}) = \{s \to \text{'v'}b\text{'z'}\bullet : \$\}$,
11. $s_{10} = \mathit{goto}(s_3, a) = \{s \to \text{'w'}a \bullet \text{'z'} : \$\}$,
12. $s_{11} = \mathit{goto}(s_{10}, \text{'z'}) = \{s \to \text{'w'}a\text{'z'}\bullet : \$\}$,
13. $s_{12} = \mathit{goto}(s_3, b) = \{s \to \text{'w'}b \bullet \text{'y'} : \$\}$,
14. $s_{13} = \mathit{goto}(s_{12}, \text{'y'}) = \{s \to \text{'w'}b\text{'y'}\bullet : \$\}$.

Die einzigen Zustände, bei denen es Konflikte geben könnte, sind die Mengen s_4 und s_5, denn hier sind prinzipiell sowohl Reduktionen mit der Regel

$$a \to \text{'x'} \quad \text{als auch mit} \quad b \to \text{'x'}$$

möglich. Da allerdings die Mengen der Folge-Token einen leeren Durchschnitt haben, gibt es tatsächlich keinen Konflikt und die Grammatik ist eine kanonische LR-Grammatik.

Wir berechnen als nächstes die LALR-Zustände der oben angegebenen Grammatik. Die einzigen Zustände, die einen gemeinsamen Kern haben, sind die beiden Zustände s_4 und s_5, denn es gilt

$$\mathit{core}(s_4) = \{a \to \text{'x'}\bullet,\ b \to \text{'x'}\bullet\} = \mathit{core}(s_5).$$

Bei der Berechnung der LALR-Zustände werden diese beiden Zustände zu einem Zustand $s_{\{4,5\}}$ zusammengefasst. Dieser neue Zustand hat die Form

$$s_{\{4,5\}} = \big\{A \to \text{'x'}\bullet : \{\text{'y'}, \text{'z'}\},\ B \to \text{'x'}\bullet : \{\text{'y'}, \text{'z'}\}\big\}.$$

Hier gibt es offensichtlich einen Reduce-Reduce-Konflikt, denn einerseits haben wir

$$\mathit{action}(s_{\{4,5\}}, \text{'y'}) = \langle \mathit{reduce}, A \to \text{'x'}\rangle,$$

andererseits gilt aber auch

$$\mathit{action}(s_{\{4,5\}}, \text{'y'}) = \langle \mathit{reduce}, B \to \text{'x'}\rangle.$$

Aufgabe 30: Es sei $G = \langle V, T, R, s\rangle$ eine LR-Grammatik und $\mathcal{N}$ sei die Menge der LALR-Zustände der Grammatik. überlegen Sie, warum es in der Menge $\mathcal{N}$ keine Shift-Reduce-Konflikte geben kann. ◇

Historical Notes The theory of LALR parsing is due to Franklin L. DeRemer [DeR71]. At the time of its invention, the space savings of LALR parsing in comparison to LR parsing were crucial.

9.6.3 Bewertung der verschiedenen Methoden

Für die Praxis sind SLR-Parser nicht ausreichend, denn es gibt eine Reihe praktisch relevanter Sprach-Konstrukte, für die sich kein SLR-Parser erzeugen lässt. Kanonische LR-Parser sind wesentlich mächtiger, benötigen allerdings oft deutlich mehr Zustände. Hier stellen LALR-Parser einen Kompromiss dar: Einerseits sind LALR-Sprachen fast so ausdrucksstark wie kanonische LR-Sprachen, andererseits liegt der Speicherbedarf von LALR-Parsern in der gleichen Größenordnung wie der Speicherbedarf von SLR-Parsern. Beispielsweise hat die SLR-Parse-Tabelle für die Sprache `C` insgesamt 349 Zustände, die entsprechende LR-Parse-Tabelle kommt auf 1572 Zustände, während der LALR-Parser mit 350 Zuständen auskommt und damit nur einen Zustand mehr als der SLR-Parser hat. In den heute in der Regel zur Verfügung stehenden Hauptspeichern lassen sich allerdings auch kanonische LR-Parser meist mühelos unterbringen, so dass es eigentlich keinen zwingenden Grund mehr gibt, statt eines LR-Parsers einen LALR-Parser einzusetzen.

Andererseits wird niemand einen LALR-Parser oder einen kanonischen LR-Parser von Hand programmieren wollen. Stattdessen werden Sie später einen Parser-Generator wie *Bison* oder *JavaCup* einsetzen, der Ihnen einen Parser generiert. Das Werkzeug Bison ist ein Parser-Generator für `C`, `C++` und bietet auch eine, allerdings leider noch experimentelle, Unterstützung für *Java*, während *JavaCup* auf die Sprache *Java* beschränkt ist. Falls Sie *JavaCup* benutzen, haben Sie keine Wahl, denn dieses Werkzeug erzeugt immer einen LALR-Parser. Bei *Bison* ist es ab der Version 3.0 auch möglich, einen LR-Parser zu erzeugen.

Chapter 10

Using Ply as a Parser Generator

Most[1] modern programming languages can be parsed using an LALR-Parser. As this lesson is based on the programming language *Python*, this chapter discusses how the parser generator PLY can be used to generate a parser for any language that has an LALR grammar. In Chapter 3 we have already seen how PLY can be used to generate a scanner. This chapter focuses on the parser-generating aspect of PLY. If you haven't done so already, you can install PLY via `anaconda` as follows:

```
conda install -c anaconda ply
```

10.1 A Simple Example

Figure 10.1 on page 130 shows the grammar of a simple symbolic calculator. This grammar is similar to the grammar shown in Figure 7.4 on page 78 in Chapter 7.

stmnt	→	IDENTIFIER ":=" *expr* ";"
	|	*expr* ";"
expr	→	*expr* "+" *product*
	|	*expr* "-" *product*
	|	*product*
product	→	*product* "*" *factor*
	|	*product* "/" *factor*
	|	*factor*
factor	→	"(" *expr* ")"
	|	NUMBER
	|	IDENTIFIER

Figure 10.1: A grammar for a symbolic calculator.

In order to generate a symbolic calculator that is based on this grammar we first need to implement a scanner. Figure 10.2 shows how to specify an appropriate scanner with PLY. As we have discussed scanner generation with PLY at length in Chapter 3 there is no need for further discussions here.

[1] The programming language C++ is a noteable exception.

```
1  import ply.lex as lex
2
3  tokens = [ 'NUMBER', 'IDENTIFIER', 'ASSIGN_OP' ]
4
5  def t_NUMBER(t):
6      r'0|[1-9][0-9]*(\.[0-9]+)?(e[+-]?([1-9][0-9]*))?'
7      t.value = float(t.value)
8      return t
9
10 def t_IDENTIFIER(t):
11     r'[a-zA-Z][a-zA-Z0-9_]*'
12     return t
13
14 def t_ASSIGN_OP(t):
15     r':='
16     return t
17
18 literals = ['+', '-', '*', '/', '(', ')', ';']
19
20 t_ignore  = ' \t'
21
22 def t_error(t):
23     print(f"Illegal character '{t.value[0]}'")
24     t.lexer.skip(1)
25
26 lexer = lex.lex()
```

Figure 10.2: A scanner for the symbolic calculator.

Figure 10.3 on page 132 shows how the grammar is implemented in PLY. We discuss it line by line.

1. Line 1 imports the module `ply.yacc`. This module contains the function `ply.yacc.yacc` which is responsible for computing the parse table. The name yacc is a homage to the Unix tool YACC, which is a popular parser generator for the language `C` and, furthermore, is part of the standard utilities of the Unix operating system.

2. Line 3 specifies that the syntactical variable `stmnt` is the start symbol of the grammar.

3. Line 5 – 7 define the function `p_stmnt_assign` which implements the grammar rule

   $$\textit{stmnt} \rightarrow \text{IDENTIFIER}\ \text{":="}\ \textit{expr}.$$

   Note that this grammar rule itself is represented by the document string of the function `p_stmnt_assign`. In general, if

   $$v \rightarrow \alpha$$

   is a grammar rule, then this grammar rule is represented by a function that has the name `p_`v`_`s. Here, the prefix "`p_`" specifies that the function implements a grammar rule (the `p` is short for parser), v should[2] be the name of the variable defined by this grammar rule, and s is a string chosen by the user to distinguish between different grammar rules for the same variable. Of course, s has to be chosen in a way such that the string `p_`v`_`s is a legal *Python* identifier.

[2] This is just a convention. Technically, v can be any string that is a valid *Python* identifier.

```
import ply.yacc as yacc

start = 'stmnt'

def p_stmnt_assign(p):
    "stmnt : IDENTIFIER ASSIGN_OP expr ';'"
    Names2Values[p[1]] = p[3]

def p_stmnt_expr(p):
    "stmnt : expr ';'"
    print(p[1])

def p_expr_plus(p):
    "expr : expr '+' prod"
    p[0] = p[1] + p[3]

def p_expr_minus(p):
    "expr : expr '-' prod"
    p[0] = p[1] - p[3]

def p_expr_prod(p):
    "expr : prod"
    p[0] = p[1]

def p_prod_mult(p):
    "prod : prod '*' factor"
    p[0] = p[1] * p[3]

def p_prod_div(p):
    "prod : prod '/' factor"
    p[0] = p[1] / p[3]

def p_prod_factor(p):
    "prod : factor"
    p[0] = p[1]

def p_factor_group(p):
    "factor : '(' expr ')'"
    p[0] = p[2]

def p_factor_number(p):
    "factor : NUMBER"
    p[0] = p[1]

def p_factor_id(p):
    "factor : IDENTIFIER"
    p[0] = Names2Values.get(p[1], float('nan'))
```

Figure 10.3: A scanner for the symbolic calculator, part 1.

The function always takes one argument `p`. This argument is a sequence of objects that can be indexed with array notation. If the grammar rule defining v has the form

$$v \to X_1 \cdots X_n,$$

then this sequence has a length of $n + 1$. If X_i is a token, then `p[`i`]` is the property with name `value` that is associated with this token. Often, this value is just a string, but it can also be a number. If X_i is a variable, then `p[`i`]` is the value that is returned when X_i is recognized. The value associated with the variable v is stored in the location `p[0]`. In the grammar rule shown in line 5–7, we have not assigned any value to `p[0]` and therefore there is no value associated with the syntactical variable `stmnt` that is defined by this grammar rule.

Note: Line 6 shows how a grammar rule is represented for PLY. A grammar rule of the form

$$v \to X_1 \cdots X_n$$

is represented as the string:

"v : X_1 $\cdots$ X_n"

It is very important to note that the character ":" has to be surrounded by space characters. Otherwise, the parser generator does not work but rather generates error messages that are difficult to understand.

The function `p_stmnt_assign` has the task of evaluating the expression that is on the right hand side of the assignment operator ":=". The result of this evaluation is then stored in the dictionary `Names2Values`. The key that is used is the name of the identifier to the left of the assignment operator.

4. The function in line 9 – 11 implements the grammar rule

   stmnt → *expr* ";" .

   The rule is implemented by evaluating the expression and then printing it.

5. The function `p_expr_plus` implements the grammar rule

   expr → *expr* "+" *prod*.

   It is implemented by evaluating the expression to the left of the operator "+", which is stored in `p[1]`, and the product to the right of this operator, which is stored in `p[3]`, and then adding the corresponding values. Finally, the resulting sum is stored in `p[0]` so that it is available later as the value of the expression that has been parsed.

   The remaining functions are similar to the ones that are discussed above.

Figure 10.4 on page 134 is discussed next.

1. Line 1 – 7 shows the function `p_error` which is used to print error messages in the case that the input can not be parsed because of a syntax error. The argument `p` is the token t that caused the entry `action`(s, t) in the action table to be undefined. If the syntax error happens at the end of the input, `p` has the value `None`.

   In a more serious application, the parser would also print both the line and column numbers of the offending token, but in order to keep this example small, this is not done here.

2. Line 7 generates the parser.

   (a) The first argument `write_tables` has to be set to `False` to prevent an obscure bug from happening.

   (b) The argument `debug` has to be set to `True` if we want to dump the parse table to the disk. The parse table is then written to the file `parser.out`.

3. Line 9 initializes the dictionary `Names2Values`. For every identifier x defined interactively, `Names2Values[`x`]` is the value associated with x.

4. The function `main` is used as a driver for the parser. It reads a string `s` from the command line and tries to parse `s` using the function `yacc.parse`. The function `yacc.parse` is generated behind the scenes when the function `yacc.yacc` is invoked in line 7.

```
def p_error(p):
    if p:
        print(f'Syntax error at {p.value} in line {p.lexer.lineno}.')
    else:
        print('Syntax error at end of input.')

parser = yacc.yacc(write_tables=False, debug=True)

Names2Values = {}

def main():
    while True:
        s = input('calc > ')
        if s == '':
            break
        yacc.parse(s)
```

Figure 10.4: A scanner for the symbolic calculator, part 2.

10.2 Shift/Reduce and Reduce/Reduce Conflicts

In this section we show how shift/reduce and reduce/reduce conflicts are dealt with in PLY. Figure 10.5 on page 134 shows a grammar for arithmetical expressions that is ambiguous because it does not specify the precedence of the different arithmetical operators.

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '(' expr ')'
     | NUMBER
```

Figure 10.5: An ambiguous grammar for arithmetical expressions.

This grammar does not specify whether the string

"1 + 2 * 3" is interpreted as "(1 + 2) * 3" or as "1 + (2 * 3)".

Since every LALR is unambiguous, but the grammar shown in Figure 10.5 is ambiguous, it has to have shift/reduce or reduce/reduce conflicts. This grammar is part of the jupyter notebook

Formal-Languages/tree/master/Ply/Conflicts.ipynb.

When we try to generate a parser for this grammar using PLYs yacc command we get the message

```
WARNING: 16 shift/reduce conflicts.
```

The file `parser.out` that is generated by PLY shows how these conflicts are resolved. This file contains the LALR states created by the parser generator and for every state the possible actions are shown. Given the grammar shown above, PLY creates 14 different states. There are conflicts in 4 of these states. Figure 10.6 on page 135 shows state number 10 and its actions. We see that there are 4 shift/reduce conflicts in this state. Unfortunately, PLY only prints the marked rules defining these states, but it does not show the follow sets of these rules. We see that PLY resolves

all conflicts in favour of shifting. The exclamation marks in the beginning of the line 20 – 23 are to be interpreted as negations and show those reduce actions that would have been possible in state 10, but are discarded in favour of the shift actions shown in line 15 – 18. Of course, in this example shifting is wrong because then the string

```
1 - 2 + 3
```

is interpreted as `1 - (2 + 3)` and not as `(1 - 2) + 3`.

```
1   state 10
2
3       (2) expr -> expr - expr .
4       (1) expr -> expr . + expr
5       (2) expr -> expr . - expr
6       (3) expr -> expr . * expr
7       (4) expr -> expr . / expr
8
9     ! shift/reduce conflict for + resolved as shift
10    ! shift/reduce conflict for - resolved as shift
11    ! shift/reduce conflict for * resolved as shift
12    ! shift/reduce conflict for / resolved as shift
13      $end            reduce using rule 2 (expr -> expr - expr .)
14      )               reduce using rule 2 (expr -> expr - expr .)
15      +               shift and go to state 4
16      -               shift and go to state 5
17      *               shift and go to state 6
18      /               shift and go to state 7
19
20    ! +               [ reduce using rule 2 (expr -> expr - expr .) ]
21    ! -               [ reduce using rule 2 (expr -> expr - expr .) ]
22    ! *               [ reduce using rule 2 (expr -> expr - expr .) ]
23    ! /               [ reduce using rule 2 (expr -> expr - expr .) ]
```

Figure 10.6: An excerpt from the file `parse.out`.

10.3 Operator Precedence Declarations

It is possible to resolve shift/reduce conflicts using operator precedence declarations. For the grammar shown previously we could add the following operator precedence declarations:

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),    )
```

This declaration specifies that the operators "+" and "-" have a lower precedence than the operators "*" and "/". Furthermore, it specifies that all these operators associate to the left. Operators can also be specified as being right associative using the keyword "`right`". The jupyter notebook

Formal-Languages/tree/master/Ply/Conflicts-Resolved.ipynb.

shows how this precedence declaration is used. When we run this notebook, PLY doesn't give us a warning about any conflicts. If we inspect the generated file `parse.out`, the action table for the state number 10 has the form shown in Figure 10.7 on page 136.

1. Since the operators "+" and "-" have the same precedence, we have

$$\mathit{action}(\texttt{state10}, \text{“+”}) = \langle \mathit{reduce}, \mathit{expr} \rightarrow \mathit{expr}\ \text{“-”}\ \mathit{expr} \rangle$$

This way, the expression $1 - 2 + 3$ is parsed as $(1 - 2) + 3$ and not as $1 - (2 + 3)$ as it would if we would shift the operator “+” instead.

2. Since the operator “–” is left associative, we have

$$\mathit{action}(\texttt{state10}, \text{“-”}) = \langle \mathit{reduce}, \mathit{expr} \rightarrow \mathit{expr}\ \text{“-”}\ \mathit{expr} \rangle$$

This way, the expression $1 - 2 - 3$ is parsed as $(1 - 2) - 3$.

3. Since the precedence of the operator “*” is higher than the precedence of the operator “–”, we have

$$\mathit{action}(\texttt{state10}, \text{“*”}) = \langle \mathit{shift}, \texttt{state6} \rangle$$

This way, the expression $1 - 2 * 3$ is parsed as $1 - (2 * 3)$.

4. Since the precedence of the operator “/” is higher than the precedence of the operator “–”, we have

$$\mathit{action}(\texttt{state10}, \text{“/”}) = \langle \mathit{shift}, \texttt{state7} \rangle$$

This way, the expression $1 - 2/3$ is parsed as $1 - (2/3)$.

```
1   state 10
2
3       (2) expr -> expr - expr .
4       (1) expr -> expr . + expr
5       (2) expr -> expr . - expr
6       (3) expr -> expr . * expr
7       (4) expr -> expr . / expr
8
9       +               reduce using rule 2 (expr -> expr - expr .)
10      -               reduce using rule 2 (expr -> expr - expr .)
11      $end            reduce using rule 2 (expr -> expr - expr .)
12      )               reduce using rule 2 (expr -> expr - expr .)
13      *               shift and go to state 6
14      /               shift and go to state 7
15
16    ! *               [ reduce using rule 2 (expr -> expr - expr .) ]
17    ! /               [ reduce using rule 2 (expr -> expr - expr .) ]
18    ! +               [ shift and go to state 4 ]
19    ! -               [ shift and go to state 5 ]
```

Figure 10.7: An excerpt from the file `parse.out` when conflicts are resolved.

Next, we explain in detail how PLY uses operator precedence relations to resolve shift/reduce conflicts.

1. First, PLY assigns a precedence level to every grammar rule. This precedence level is the precedence level of the last operator symbol occurring in the grammar rule. Most of the times, there is just one operator that determines the precedence of the grammar rule. In the grammar at hand the precedences of the rules would as shown in the table below.

rule	precedence
expr → *expr* "+" *expr*	1
expr → *expr* "-" *expr*	1
expr → *expr* "*" *expr*	2
expr → *expr* "/" *expr*	2
expr → "(" *expr* ")"	—
expr → NUMBER	—

If a grammar rule does not contain an operator that has been given a precedence, then the precedence of the grammar rule remains undefined.

2. If s is a state that contains two e.m.R.s r_1 and r_2 such that

$$r_1 = (a \to \beta \bullet o\ \delta : L_1) \quad \text{and} \quad r_2 = (c \to \gamma \bullet : L_2) \quad \text{where} \quad o \in L_2,$$

then there is a shift/reduce conflict when

$$\mathit{action}(s, o)$$

is computed. Let us assume that the precedence of the operator o is $p(o)$ and the precedence of the rule r_2 is $p(r_2)$. Then there are six cases that depend on the relative values of $p(o)$ and $p(r_2)$ and on the associativity of the operator o.

(a) $p(o) > p(r_2)$.
In this case the precedence of the operator o is higher than the precedence of the rule r_2. Therefore the operator o is shifted:

$$\mathit{action}(s, o) = \langle \texttt{shift}, \mathit{goto}(s, o) \rangle.$$

To understand this rule we just have to watch what happens when we parse

```
1+2*3
```

using the grammar given above. After the part "1+2" has been read and the next token is the operator "*", the parser is in the following state:

$$\begin{Bmatrix} \mathit{expr} & \to & \mathit{expr} \bullet \text{"+"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"-"}\ \text{"*"}, \text{"/"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"-"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"-"}\ \text{"*"}, \text{"/"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"*"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"-"}\ \text{"*"}, \text{"/"}\}, \\ \mathit{expr} & \to & \mathit{expr} \bullet \text{"/"}\ \mathit{expr} : \{\$, \text{"+"}, \text{"-"}\ \text{"*"}, \text{"/"}\}, \\ \mathit{expr} & \to & \mathit{expr}\text{"+"}\ \mathit{expr} \bullet : \{\$, \text{"+"}, \text{"-"}\ \text{"*"}, \text{"/"}\} \end{Bmatrix}.$$

When the parser next sees the token "*", then it must not reduce the symbol stack using the rule *expr* → *expr*"+"*expr*, because it has to multiply the numbers 2 and 3 first. Therefore, the token "*" has to be shifted.

(b) $p(o) < p(r_2)$.
Now the precedence of the operator that occurs in the rule r_2 is higher than the precedence of the operator o. Therefore the correct action is to reduce with the rule r_2:

$$\mathit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

To see that this makes sense we discuss the parsing of the expression

```
1*2+3
```

with the grammar given previously. Assume the string "1*2" has already been read and the next token that is processed is the token "+". Then the state of the parser is as follows:

$$\Big\{ \begin{array}{lcl} \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“+”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“−”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“*”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“/”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr}\text{“*”}\, \textit{expr} \bullet : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\} \end{array} \Big\}.$$

When the parser now sees the operator “+”, it has to reduce the string “1*2” using the rule

$$\textit{expr} \rightarrow \textit{expr}\text{“*”}\, \textit{expr},$$

as it has to multiply the numbers 1 and 2.

(c) $p(o) = p(r_2)$ and the operator o is left associative.
Then we reduce the symbol stack with the rule r_2, we have

$$\textit{action}(s, o) = \langle \texttt{reduce}, r_2 \rangle.$$

To convince yourself that this is the right thing to do, inspect what happens when the string

```
1-2-3
```

is parsed with the grammar discussed previously. Assume that the string “1−2” has already be read and the next token is the operator “−”. Then the state of the parser is as follows:

$$\Big\{ \begin{array}{lcl} \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“+”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“−”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“*”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr} \bullet \text{“/”}\, \textit{expr} : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\}, \\ \textit{expr} & \rightarrow & \textit{expr}\text{“−”}\, \textit{expr} \bullet : \{\$, \text{“+”}, \text{“−”}\ \text{“*”}, \text{“/”}\} \end{array} \Big\}.$$

If the next token is the operator “−”, then the parser has to reduce the symbol stack using the rule $\textit{expr} \rightarrow \textit{expr}\text{“−”}\, \textit{expr}$ as it has to subtract 2 from 1. If it would shift instead it would compute $1 - (2 - 3)$ instead of computing $(1 - 2) - 3$.

(d) $p(o) = p(r_2)$ and the operator o associates to the right.
In this case the operator o is shifted

$$\textit{action}(s, o) = \langle \texttt{shift}, \textit{goto}(s, o) \rangle.$$

In order to understand this case, parse the string

```
2^3^4
```

with the grammar rules

$$\textit{expr} \rightarrow \textit{expr}\ \text{“\^{}”}\ \textit{expr} \mid \textsc{Number}.$$

Consider the situation when the string “1^2” has already been read and the next token is the exponentiation operator “^”. The state of the parser is then as follows:

$$\big\{ \textit{expr} \rightarrow \textit{expr} \bullet\ \text{“\^{}”}\ \textit{expr} : \{\$,\ \text{“\^{}”}\ \},\ \textit{expr} \rightarrow \textit{expr}\text{“\^{}”}\, \textit{expr} \bullet : \{\$,\ \text{“\^{}”}\ \} \big\}.$$

Here the token “^” has to be shifted since we first have to compute the expression “3^4”.

(e) $p(o) = p(r_2)$ and the operator o has been declared to be non-associative.
In this case we have a syntax error:

$$\textit{action}(s, o) = \textit{error}.$$

To understand this case, try to parse a string of the form

```
1 < 1 < 1
```

using the grammar rules

$$\textit{expr} \rightarrow \textit{expr}\ \text{“<”}\ \textit{expr} \mid \textit{expr}\ \text{“+”}\ \textit{expr} \mid \textsc{Number}.$$

Once the string “1 < 1” has been read and the next token is the operator “<” the parser recognizes that

there is an error. Therefore, PLY will resolve this shift/reduce conflict by putting an error entry into the action table.

(f) $p(o)$ is undefined or $p(r_2)$ is undefined.
In this case there is a shift/reduce conflict and PLY prints a warning message when generating the parser. The conflict is then resolved in favour of shifting.

10.4 Resolving Shift/Reduce and Reduce/Reduce Conflicts

We start our discussion by categorizing conflicts with respect to their origin.

1. *Mehrdeutigkeits-Konflikte* sind Konflikte, die ihre Ursache in einer Mehrdeutigkeit der zu Grunde liegenden Grammatik haben. Solche Konflikte weisen damit auf ein tatsächliches Problem der Grammatik hin. Wir hatten ein Beispiel für solche Konflikte gesehen, als wir in Abbildung 10.5 versucht hatten, die Syntax arithmetischer Ausdrücke ohne die syntaktischen Kategorien *product* und *factor* zu beschreiben.

   Wir hatten damals bereits gesehen, dass wir das Problem durch die Einführung von Operator-Präzedenzen lösen können. Falls dies nicht möglich ist, dann bleibt nur das Umschreiben der Grammatik.

2. *Look-Ahead-Konflikte* sind Reduce/Reduce-Konflikte, bei denen die Grammatik zwar einerseits eindeutig ist, für die aber andererseits ein Look-Ahead von einem Token nicht ausreichend ist um den Konflikt zu lösen.

3. *Mysteriöse Konflikte* entstehen erst beim Übergang von den LR-Zuständen zu den LALR-Zuständen durch das Zusammenfassen von Zuständen mit dem gleichen Kern. Diese Konflikte treten also genau dann auf, wenn das Konzept einer LALR-Grammatik nicht ausreichend ist um die Syntax der zu parsenden Sprache zu beschreiben.

Wir betrachten die letzten beiden Fälle nun im Detail und zeigen Wege auf, wie die Konflikte gelöst werden können.

10.4.1 Look-Ahead-Konflikte

Ein Look-Ahead-Konflikt liegt dann vor, wenn die Grammatik zwar eindeutig ist, aber ein Look-Ahead von einem Token nicht ausreicht um zu entscheiden, mit welcher Regel reduziert werden soll. Abbildung 10.8 zeigt die Grammatik `Look-Ahead.ipynb`[3], die zwar eindeutig ist, aber nicht die LR(1)-Eigenschaft hat und damit erst recht keine LALR(1) Grammatik ist.

```
1  a : b 'U' 'V'
2    | c 'U' 'W'
3  b : 'X'
4  c : 'X'
```

Figure 10.8: Eine eindeutige Grammatik ohne die LR(1)-Eigenschaft.

Berechnen wir die LR-Zustände dieser Grammatik, so finden wir unter anderem den folgenden Zustand:

$$\{b \to \text{ “X”} \bullet : \text{“U”},\ c \to \text{ “X”} \bullet : \text{“U”}\}.$$

Da die Menge der Folge-Token für beide Regeln gleich sind, haben wir hier einen Reduce/Reduce-Konflikt. Dieser Konflikt hat seine Ursache darin, dass der Parser mit einem Look-Ahead von nur einem Token nicht entscheiden kann, ob ein “X” als ein b oder als ein c zu interpretieren ist, denn dies entscheidet sich erst, wenn das auf “U” folgende Zeichen gelesen wird: Handelt es sich hierbei um ein “V”, so wird insgesamt die Regel

$$a \to b \text{ “U” “V”}$$

[3]Diese Grammatik habe ich im Netz auf der Seite von Pete Jinks unter der Adresse
http://www.cs.man.ac.uk/~pjj/cs212/ho/node19.html
gefunden.

verwendet werden und folglich ist das "X" als ein b zu interpretieren. Ist das zweite Token hinter dem "X" hingegen ein "W", so ist die zu verwendende Regel

$$a \to c \text{ "U" "W"}$$

und folglich ist das "X" als c zu lesen.

```
1  a : b 'V'
2    | c 'W'
3  b : 'X' 'U'
4  c : 'X' 'U'
```

Figure 10.9: Eine zu Abbildung 10.8 äquivalente LR(1)-Grammatik.

Das Problem bei dieser Grammatik ist, dass sie versucht, abhängig vom Kontext ein "X" wahlweise als ein b oder als ein c zu interpretieren. Es ist offensichtlich, wie das Problem gelöst werden kann: Wenn der Kontext "U", der sowohl auf b als auch auf c folgt, mit in die Regeln für b und c aufgenommen wird, dann verschwindet der Konflikt, denn dann hat der Zustand, in dem früher der Konflikt auftrat, die Form

$$\{b \to \text{ "X" "U"} \bullet : \text{"V"},\ c \to \text{ "X" "U"} \bullet : \text{"W"}\}.$$

Hier entscheidet sich nun anhand des nächsten Tokens, mit welcher Regel wir in diesem Zustand reduzieren müssen: Ist das nächste Token ein "V", so reduzieren wir mit der Regel

$$b \to \text{ "X" "U"},$$

ist das nächste Token hingegen der Buchstabe "W", so nehmen wir stattdessen die Regel

$$c \to \text{ "X" "U"} \bullet : \text{"W"}.$$

Abbildung 10.9 zeigt die entsprechend modifizierte Grammatik, die Sie unter

github.com/karlstroetmann/Formal-Languages/tree/master/Ply/Look-Ahead-Solved.ipynb

im Netz finden.

10.4.2 Mysterious Reduce/Reduce Conflicts

A conflict is called a mysterious reduce/reduce conflict if the conflict results from the merger of states that happens when we go from an LR parsing table to an LALR parsing table. The grammar in Figure 10.10 on page 140 is the same as the grammar shown in Figure 9.14 on page 128 in the previous chapter. Then we had seen that this grammar is an LR grammar, but not an LALR grammar. Let us see what happens if we use PLY to generate the states for this grammar.

```
1  s : 'v' a 'y'
2    | 'w' b 'y'
3    | 'v' b 'z'
4    | 'w' a 'z'
5
6  a : X
7
8  b : X
```

Figure 10.10: A grammar that generates a mysterious reduce/reduce conflict.

When we run PLY to produce the parsing table, we get the states shown in Figure 10.11 on page 141. This

Figure only shows two states, `state 6` and `state 9`. I have taken the liberty to annotate the extended marked rules occurring in these states with their follow sets. Taken by itself, none of these two states has a conflict since the follow sets of the respective rules are disjoint. However, it is obvious that these two states have the same core and should have been merged. The resulting state would have the form

$$\{a \to X\bullet : \{\texttt{'y'}, \texttt{'z'}\}, b \to X\bullet : \{\texttt{'y'}, \texttt{'z'}\}\}$$

and obviously has a reduce/reduce conflict if the next token is either 'y' or 'z'.

```
1   state 6
2
3       (5) a -> X . : 'y'
4       (6) b -> X . : 'z'
5
6       y               reduce using rule 5 (a -> X .)
7       z               reduce using rule 6 (b -> X .)
8
9   state 9
10
11      (6) b -> X . : 'y'
12      (5) a -> X . : 'z'
13
14      y               reduce using rule 6 (b -> X .)
15      z               reduce using rule 5 (a -> X .)
```

Figure 10.11: A grammar that generates a mysterious reduce/reduce conflict.

Interestingly, PLY does not merge these states and is therefore able to produce generate a parse table without conflicts. On the other hand, PLY claims to generate LALR tables. Therefore, I have have written an email to David Beazley asking whether this behaviour is a feature or a bug. He has classified this example as an "*interesting curiosity*".

Chapter 11

Assembler

A compiler translates programs written in a high level language like C or *Java* into some low level representation. This low level representation can be either machine code or some form of assembler code. For the programming language *Java*, the command `javac` compiles a program written in *Java* into *Java* byte code. This byte code is then executed using the *Java* virtual machine (JVM).

The compiler that we are going to develop in the next chapter generates a particular form of assembler code know as JVM assembler code. This assembler code can be translated directly into *Java* byte code, which is also the byte code generated by `javac`. The program for translating JVM assembler into bytecode is called *Jasmin*. You can download *Jasmin* here. The byte code produced by *Jasmin* can be executed using the `java` command just like any other "`.class`"-file. This chapter will discuss the syntax and semantics of *Jasmin* assembler code.

11.1 Introduction into Jasmin Assembler

To get used to the syntax of *Jasmin* assembler, we start with a small program that prints the string

"`Hello World!`"

on the standard output stream. Figure 11.1 on page 142 shows the program `Hello.jas`. We discuss this program line by line.

```
1  .class public Hello
2  .super java/lang/Object
3
4  .method public <init>()V
5      aload 0
6      invokenonvirtual java/lang/Object/<init>()V
7      return
8  .end method
9
10 .method public static main([Ljava/lang/String;)V
11     .limit locals 1
12     .limit stack  2
13     getstatic     java/lang/System/out Ljava/io/PrintStream;
14     ldc           "Hello World!"
15     invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
16     return
17 .end method
```

Figure 11.1: An assembler program to print "`Hello World!`".

1. Line 1 uses the directive ".class" to define the name of the class file that is to be produced by the assembler. In this case, the class name is `Hello`. Therefore, *Jasmin* will translate this file into the class file "`Hello.class`".

2. Line 2 uses the directive ".super" to specify the super class of the class `Hello`. In our examples, the super class will always be the class `Object`. Since this class resides in the package "`java.lang`", the super class has to be specified as

   `java/lang/Object`.

   Observe that the character "." in the class name "`java.lang.Object`" has to be replaced by the character "/". This is true even if a *Windows* operating system is used.

3. Line 4 to 8 initialize the program. This code is always the same and corresponds to a constructor for the class `Hello`. As this code is copied verbatim to the beginning of every class file, we will not discuss it further.

4. Line 10 – 17 defines the method `main` that does the actual work.

   (a) Line 10 uses the directive ".method" to declare the name of the method and its signature. The string

   `main([Ljava/lang/String;)V`

   specifies the signature:

   i. The string "`main`" is the name of the method that is defined.
   ii. The character "`[`" specifies that the first argument is an array.
   iii. The character "`L`" specifies that this array consists of objects.
   iv. The string "`java/lang/String;`" specifies that these objects are objects of the class "`java.lang.String`".
   v. Finally, the character "`V`" specifies that the return type of the method `main` is "`void`".

   (b) Line 11 uses the directive ".limit locals" to specify the number of local variables used by the method `main`. In this case, there is just one local variable. This variable corresponds to the parameter of the method main. The assembler file shown in Figure 11.1 on page 142 corresponds to the *Java* code shown in Figure 11.2 below. The method `main` has one local variable, which is the argument `args`. The information on the number of local variables is needed by the *Java* virtual machine in order to allocate memory for these variables on the stack.

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello World!");
4      }
5  }
```

Figure 11.2: Printing `Hello world` in *Java*.

   (c) For the purpose of the following discussion, we basically assume that there exist two types of processors: Those that store the objects they work upon in registers and those that store these objects on a stack residing in main memory. The Java virtual machine is of the second type. Hence, *Jasmin* assembler programs do not refer to registers but rather refer to this stack[1]. Line 12 uses the directive

   ".limit stack"

   to specify the the maximal height of the stack. In this case, the stack is allowed to contain a maximum of two objects. It is easy to see that we indeed do never place more than two objects onto the stack, since line 13 pushes the object

   `java.lang.System.out`

[1] In reality, all real processors make use of registers. However, it is possible to simulate a stack machine using a real processor and that is what is done in the Java virtual machine.

onto the stack. This is an object of class "`java.io.PrintStream`". Then, line 14 pushes a reference to the string "`Hello World`" onto the stack. The other instructions to not push anything onto the stack.

(d) Line 15 calls the method `println`, which is a method of the class "`java.io.PrintStream`". It also specifies that `println` takes one argument of type `java.lang.String` and returns nothing.

(e) Line 16 returns from the method `main`.

(f) In line 17 the directive "`.end`" marks the end of the code corresponding to the method `main`.

Before we proceed, let us assume that we are working on a Unix operating system and that there is an executable file called `jasmin` somewhere in our path that contains the following code:

```
#!/bin/bash
java -jar /usr/local/lib/jasmin.jar $@
```

Of course, for this to work the directory `/usr/local/lib/` has to contain the file "`jasmin.jar`". If we were working on a windows operating system, we would have a file called `jasmin.bat` somewhere in our `PATH`. This file would contain the following line:

```
java -jar ~/Dropbox/Software/jasmin-2.4/jasmin.jar $*
```

Of course, for this to work the directory `~/Dropbox/Software/jasmin-2.4` has to contain the file "`jasmin.jar`".

In order to execute the assembler program discussed above, we first have to translate the assembler program into a `class`-file. This is done using the command

```
jasmin Hello.jas
```

Executing this command creates the file "`Hello.class`". This class file can then be executed just like any class file generated from `javac` by typing

```
java Hello
```

in the command line, provided the environment variable `CLASSPATH` contains the current directory, i.e. the `CLASSPATH` has to contain the directory "`.`".

We will proceed to discuss the different assembler commands in more detail later. To this end, we first have to discuss some background: One of the design goal of the programming language *Java* was compatibility. The idea was that it should be possible to execute *Java* class files on any computer. Therefore, the *Java* designers decided to create a so called *virtual machine*. A virtual machine is a computer architecture that, instead of being implemented in silicon, is simulated. Programs written in *Java* are first compiled into so called *class files*. These class files correspond to the machine code of the *Java* virtual machine (JVM). The architecture of the virtual machine is a *stack machine*. A stack machine does not have any registers to store variables. Instead, there is a stack and all variables reside on the stack. Any command takes its arguments from the top of the stack and replaces these arguments with the result of the operation performed by the command. For example, if we want to add two values, then we first have to push both values onto the stack. Next, performing the add operation will pop these values from the stack and then push their sum onto the stack.

11.2 Assembler Instructions

We proceed to discuss some of the JVM instructions. Since there are more than 160 JVM instructions, we can only discuss a subset of all instructions. We restrict ourselves to those instructions that deal with integers: For example, there is an instruction called `iadd` that adds two 32 bit integers. There are also instructions like `fadd` that adds two floating point numbers and `dadd` that adds two double precision floating point numbers but, since our time is limited, we won't discuss these instructions. Before we are able to discuss the different instructions we have to discuss how the main memory is organized in the JVM: In the JVM, the memory is split into four parts:

1. The program memory contains the program code as a sequence of bytes.

2. The operands of the different machine instructions are put onto the stack. Furthermore, the stack contains the arguments and the local variables of a procedure. However, in the context of the JVM the procedures are called *methods* instead of procedures.

   The register SP points to the top of the stack. If a method is called, the arguments of the method are placed on the stack. The register LV (local variables) points to the first argument of the current method. On top of the arguments, the local variables of the method are put on the stack. Both the arguments and the local variables can be accessed via the register LV by specifying their offset from the first argument. We will discuss the register LV in more detail when we discuss the invocation of methods.

3. The heap is used for dynamically allocated memory. Newly created objects are located in the heap.

4. The constant pool contains the definitions of constants and also the addresses of methods in program memory.

In the following, we will be mostly concerned with the stack and the heap. We proceed to discuss some of the assembler instructions.

Arithmetical and Logical Instructions

1. The instruction "iadd" adds those values that are on top of the stack and replaces these values by their sum. Figure 11.3 on page 145 show how this command works. The left part of the figure shows the stack as it is before the command iadd is executed, while the right part of the figure depicts the situation after the execution of this command. Note that after the command is executed, the stack pointer points to the position where formerly the first argument had been stored.

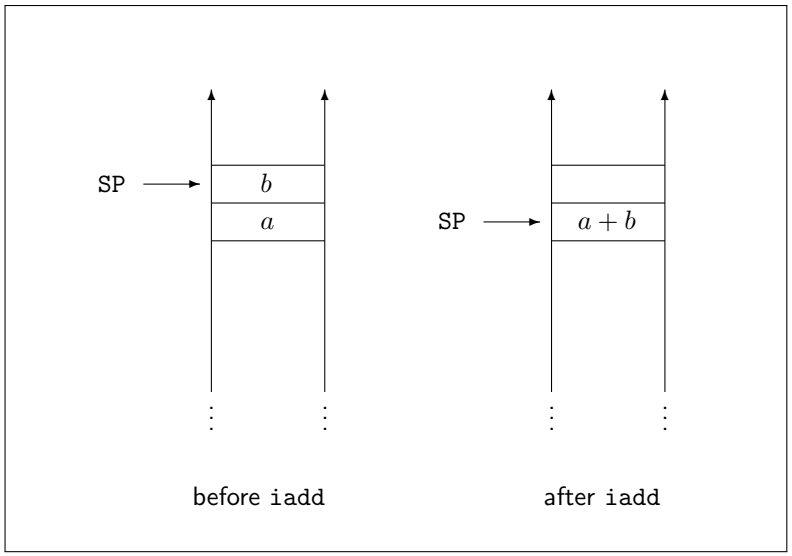

Figure 11.3: The effect of iadd.

2. The instruction "isub" subtracts the integer value on top of the stack from the value that is found on the position next to the top of the stack. Figure 11.4 on page 146 pictures this command. The left part of the figure shows the stack as it is before the command isub is executed, while the right part of the figure depicts the situation after the execution of this command. Note that after the command is executed, the stack pointer points to the position where formerly the first argument had been stored.

3. The instruction "imul" multiplies the two integer values which are on top of the stack. This instruction works similar to the instruction iadd. If the product does not fit in 32 bits, only the lowest 32 bits of the result are written onto the stack.

4. The instruction "idiv" divides the integer value that is found on the position next to the top of the stack by the value on top of the stack.

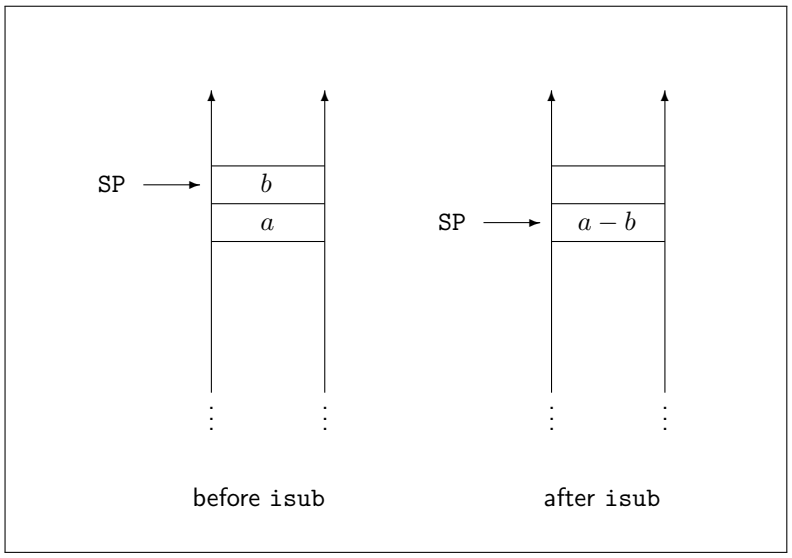

Figure 11.4: The effect of `isub`.

5. The instruction "`irem`" computes the remainder $a \,\%\, b$ of the division of a by b where a and b are integer values found on top of the stack.
6. The instruction "`iand`" computes the bitwise and of the values on top of the stack.
7. The instruction "`ior`" computes the bitwise or of the integer values that are on top of the stack.
8. The instruction "`ixor`" computes the bitwise *exclusive or* of the integer values that are on top of the stack.

Shift Instructions

1. The instruction "`ishl`" shifts the value a to the left by $b[4:0]$ bits. Here, a and b are assumed to be the two values on top of the stack: b is the value on top of the stack and a is the value below b. $b[4:0]$ denotes the natural number that results from the 5 lowest bits of b. Figure 11.5 on page 146 pictures this command.

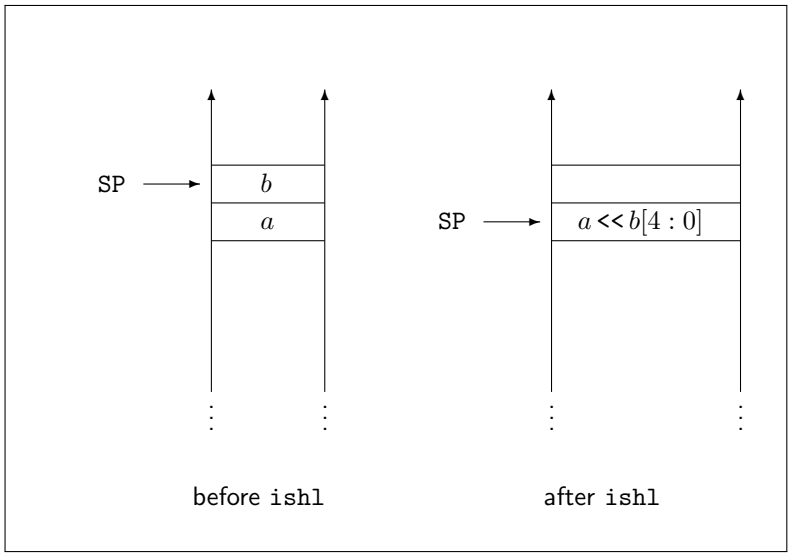

Figure 11.5: The effect of `ishl`.

2. The instruction "`ishr`" shifts the value a to the right by $b[4:0]$ bits. Here, a and b are assumed to be the two values on top of the stack: b is the value on top of the stack and a is the value below b. $b[4:0]$ denotes the natural number that results from the 5 lowest bits of b. Note that this instruction performs an *arithmetic shift*, i.e. the sign bit is preserved.

11.2.1 Instructions to Manipulate the Stack

1. The instruction "`dup`" duplicates the value that is on top of the stack. Figure 11.6 on page 147 pictures this command.

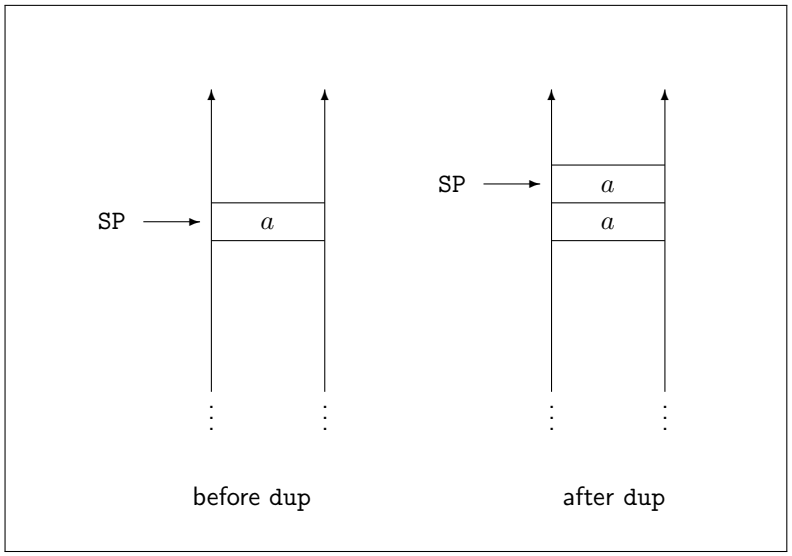

Figure 11.6: The effect of `dup`.

2. The instruction "`pop`" removes the value that is on top of the stack. Figure 11.7 on page 147 pictures this command. The value is not actually erased from memory, only the stack pointer is decremented. The next instruction that puts a new value onto the stack will therefore overwrite this old value.

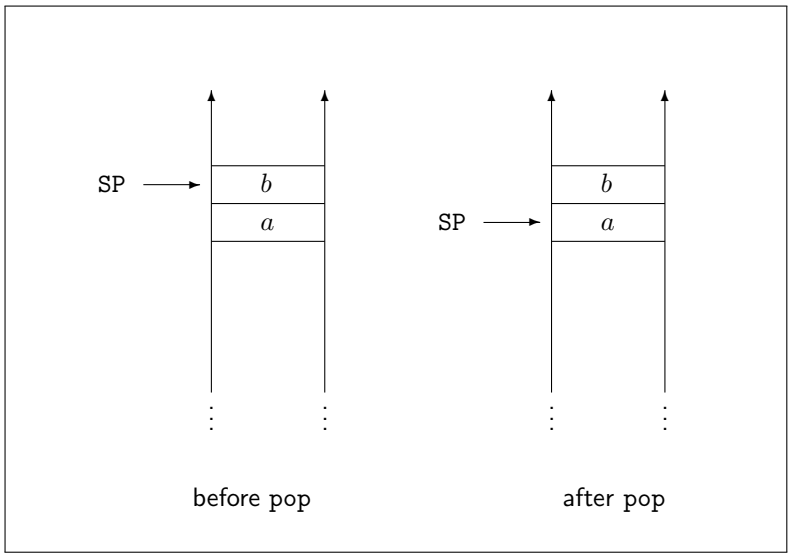

Figure 11.7: The effect of `pop`.

3. The instruction "`nop`" does nothing. The name is short for "`no operation`".

4. The instruction "`bipush` b" pushes the byte b that is given as argument onto the stack. Figure 11.8 on page 148 pictures this command.

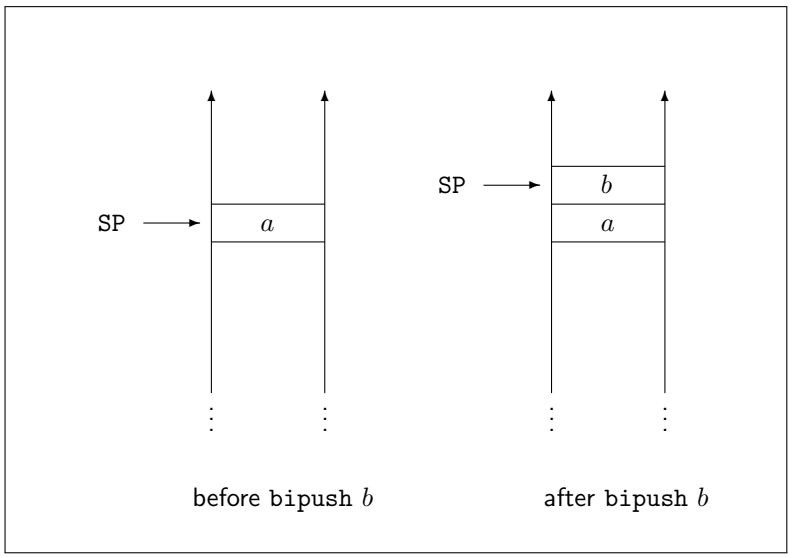

Figure 11.8: The effect of `bipush` b.

5. The instruction "`getstatic` v c" takes two parameters: v is the name of a static variable and c is the type of this variable. For example,

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

   pushes a reference to the `PrintStream` that is known as

```
java.lang.System.out
```

   onto the stack.

6. The instruction "`iload` v" reads the local variable with index v and pushes it on top of the stack. Figure 11.9 on page 149 pictures this command. Note that `LV` denotes the register pointing to the beginning of the local variables of a method.

7. The instruction "`istore` v" removes the value which is on top of the stack and stores this value at the location for the local variable with number v. Hence, v is interpreted as an index into the local variable table of the method. Figure 11.10 on page 149 pictures this command.

8. The instruction "`ldc` c" pushes the constant c onto the stack. This constant can be an integer, a single precision floating point number, or a (pointer to) a string. If c is a string, this string is actually stored in the so called *constant pool* and in this case the command "`ldc` c" will only push a pointer to the string onto the stack.

Branching Commands

In this subsection we discuss those commands that change the control flow.

1. The instruction "`goto` l" jumps to the label l. Here the label l is a label name that has to be declared inside the method containing this `goto` command. A label with name *target* is declared using the syntax

   target:

   The next section presents an example assembler program that demonstrates this command.

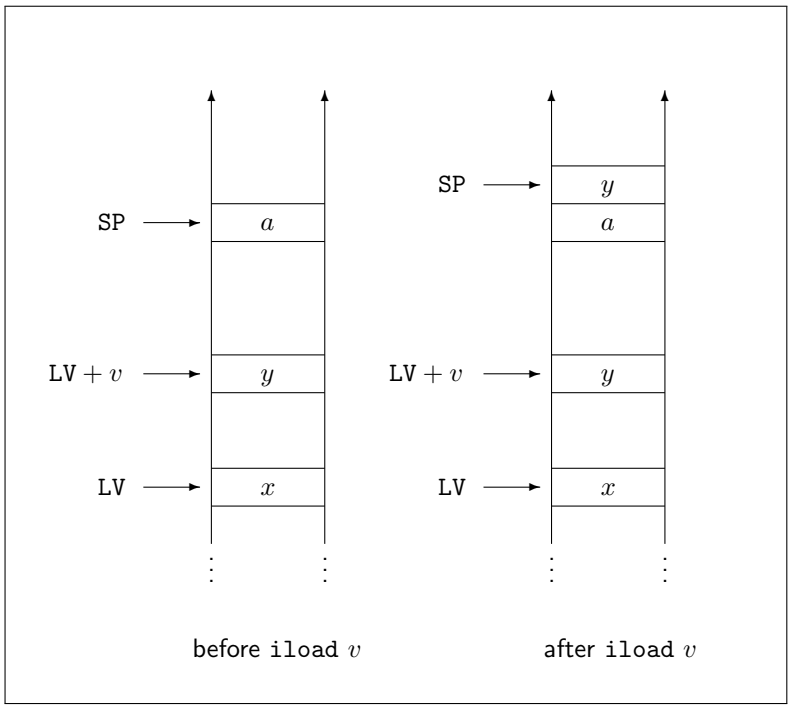

Figure 11.9: The effect of `iload` v.

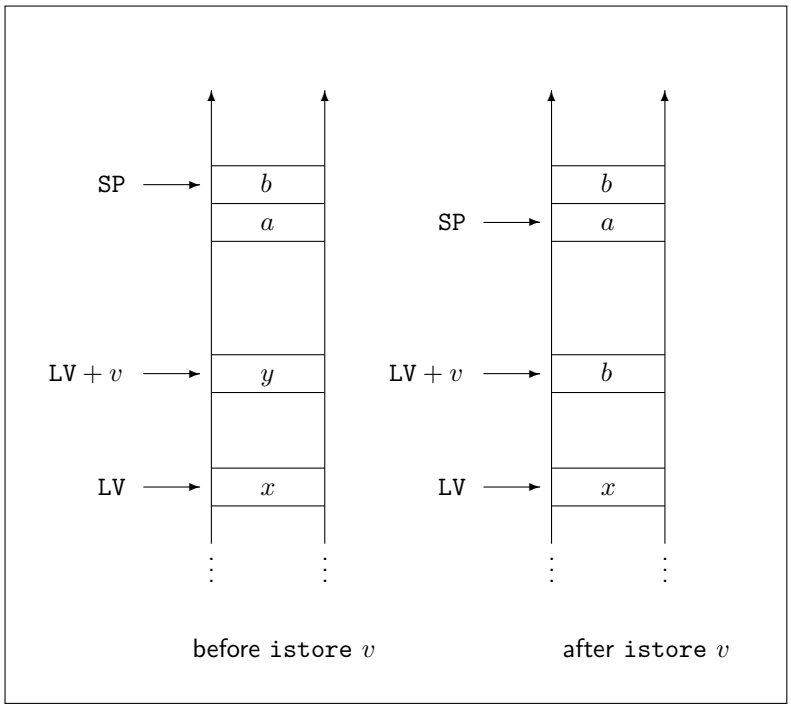

Figure 11.10: The effect of `istore` v.

2. The instruction "`if_icmpeq` l" checks whether the value on top of the stack is the same as the value preceding it. If this is the case, the program will branch to the label l. Otherwise, the control flow is not changed. Observe that both values that are compared are removed from the stack.

3. The instruction "`if_icmpne` l" checks whether the value on top of the stack is different from the value preceding it. If this is the case, the program will branch to the label l. Otherwise, the control flow is not changed. Observe that both values that are compared are removed from the stack.

4. The instruction "`if_icmplt` l" checks whether the value that is below the top of the stack is less than the value on top of the stack. If this is the case, the program will branch to the label l. Otherwise, the control flow is not changed. Observe that both values that are compared are removed from the stack.

5. The instruction "`if_icmple` l" checks whether the value that is below the top of the stack is less or equal than the value on top of the stack. If this is the case, the program will branch to the label l. Otherwise, the control flow is not changed. Observe that both values that are compared are removed from the stack.

   There are similar commands called `if_icmpgt` and `if_icmpge`.

6. The instruction "`ifeq` l" checks whether the value on top of the stack is zero. If this is the case, the program will branch to the label l. Otherwise, the control flow is not changed. Observe that the value that is tested is removed from the stack.

7. The instruction "`iflt` l" checks whether the value on top of the stack is less than zero. If this is the case, the program will branch to the label l. Otherwise, the control flow is not changed. Observe that the value that is tested is removed from the stack.

8. The instruction "`invokevirtual` m" is used to call the method m. Here m has to specify the full name of the method. For example, in order to invoke the method `println` of the class `java.io.PrintStream` we have to write

   ```
   invokevirtual java/io/PrintStream/println(I)V
   ```

   Before the command `invokevirtual` is executed, we have to put the arguments of the method onto the stack. For example, if we want to invoke the method `println` that takes an integer argument, we first have to push an object of type `PrintStream` onto the stack. Furthermore, we need to push an integer onto the stack.

9. The instruction "`invokestatic` m" is used to call the method m. Here m has to specify the full name of the method. Furthermore, m needs to be s static method. For example, in order to invoke a method called `sum` that resides in the class `Sum` and that takes one integer argument we have to write

   ```
   invokestatic Sum/sum(I)I
   ```

   In the type specification "`sum(I)I`" the first "`I`" specifies that `sum` takes one integer argument, while the second "`I`" specifies that the method `sum` returns an integer.

   Before the command `invokestatic` is executed, we have to put the arguments of the method onto the stack. For example, if we want to invoke the method `sum` described above, then we have to push an integer onto the stack.

10. The instruction "`ireturn`" returns from the method that is currently invoked. This method also returns a value to the calling procedure. In order for `ireturn` to be able to return a value v, this value v has to be pushed onto the stack before the command `ireturn` is executed.

    In general, if a method taking n arguments $a_1, \cdots, a_n$ is to be called, then first the arguments $a_1, \cdots, a_n$ have to be pushed onto the stack. When the method m is done and has computed its result r, the arguments $a_1, \cdots, a_n$ will have been replaced with the single value r.

11. The instruction "`return`" returns from the method that is currently invoked. However, in contrast to the command `ireturn`, this command is used if the method that has been invoked does not return a result.

11.3 An Example Program

Figure 11.11 shows the `C` program `sum.c` that computes the sum

$$\sum_{i=1}^{6^2} i.$$

The function $\texttt{sum}(n)$ computes the sum $\sum_{i=1}^{n} i$ and the function `main` calls this function with the argument $6 \cdot 6$. Figure 11.12 on page 152 shows how this program can be translated into assembler program `Sum.jas`. We discuss the implementation of this program next.

```
1  #import "stdio.h"
2
3  int sum(int n) {
4      int s;
5      s = 0;
6      while (n != 0) {
7          s = s + n;
8          n = n - 1;
9      };
10     return s;
11 }
12 int main() {
13     printf("%d\n", sum(6*6));
14     return 0;
15 }
```

Figure 11.11: A `C` function to compute $\sum_{i=1}^{36} i$.

1. Line 1 specifies the name of the generated class which is to be `Sum`.
2. Line 2 specifies that the class `Sum` is a subclass of the class `java.lang.Object`.
3. Lines 4 – 8 initialize the class. The code used here is the same as in the example printing "`Hello World!`".
4. Line 10 declares the method `main`.
5. Line 11 specifies that there is just one local variable.
6. Line 12 specifies that the stack will contain at most 3 temporary values.
7. Line 13 pushes the object `java.lang.out` onto the stack. We need this object later in order to invoke `println`.
8. Line 14 pushes the number 6 onto the stack.
9. Line 15 duplicates the value 6. Therefore, after line 15 is executed, the stack contains three elements: The object `java.lang.out`, the number 6, and again the number 6.
10. Line 16 multiplies the two values on top of the stack and replaces them with their product, which happens to be 36.
11. Line 17 calls the method `sum` defined below. After this call has finished, the number 36 on top of the stack is replaced with the value $\texttt{sum}(36)$.
12. Line 18 prints the value that is on top of the stack.
13. Line 22 declares the method sum. This method takes one integer argument and returns an integer as result.
14. Line 23 specifies that the method `sum` has two local variables: The first local variable is the parameter n and the second local variable corresponds to the variable s in the `C` program.

```
1  .class public Sum
2  .super java/lang/Object
3
4  .method public <init>()V
5      aload 0
6      invokenonvirtual java/lang/Object/<init>()V
7      return
8  .end method
9
10 .method public static main([Ljava/lang/String;)V
11     .limit locals 1
12     .limit stack  3
13     getstatic     java/lang/System/out Ljava/io/PrintStream;
14     ldc           6
15     dup
16     imul
17     invokestatic  Sum/sum(I)I
18     invokevirtual java/io/PrintStream/println(I)V
19     return
20 .end method
21
22 .method public static sum(I)I
23     .limit locals 2
24     .limit stack  2
25     ldc    0
26     istore 1                    ; s = 0;
27 loop:
28     iload  0                    ; n
29     ifeq   finish               ; if (n == 0) goto finish;
30     iload  1                    ; s
31     iload  0                    ; n
32     iadd
33     istore  1                   ; s = s + n;
34     iload   0                   ; n
35     ldc     1
36     isub
37     istore  0                   ; n = n - 1;
38     goto    loop
39 finish:
40     iload   1
41     ireturn                     ; return s;
42 .end method
```

Figure 11.12: An assembler program to compute the sum $\sum_{i=1}^{36} i$.

15. The effect of lines 25 and 26 is to initialize this variable s with the value 0.

16. Line 28 pushes the local variable n on the stack so that line 29 is able to test whether n is already 0. If $n = 0$, the program branches to the label `finish` in line 39, pushes the result s onto the stack (line 40) and returns. If n is not yet 0, the execution proceeds normally to line 30.

17. Line 30 and line 31 push the sum s and the variable n onto the stack. These values are then added and the

result is written to the local variable s in line 33. The combined effect of these instructions is therefore to perform the assignment

```
s = s + n;
```

18. The instructions in line 34 up to line 37 implement the assignment

```
n = n - 1;
```

19. In line 38 we jump back to the beginning of the while loop and test whether n has become zero.

20. The declaration in line 42 terminates the definition of the method `sum`.

Exercise 31: Implement an assembler program that computes the factorial function. Test your program by printing $n!$ for $n = 1, \cdots, 10$.

11.4 Disassembler*

Sometimes it is useful to transform a file consisting of *Java* byte code back into something that looks like assembler code. After all, a *Java* class file is a binary file and can therefore only be viewed via commands like `od` that produce an *octal* or *hexadecimal dump* of the given file. The command `javap` is a *disassembler*, i.e. it takes a *Java* byte code file and transforms it in something that looks similar to *Jasmin* assembler. For example, Figure 11.13 shows the class *Java* class `Sum` to compute the sum

$$\sum_{i=1}^{36} i$$

written in *Java*. If this program is stored in a file called "`Sum.java`", we can compile it via the following command:

```
javac Sum.java
```

This will produce a class file with the name "`Sum.class`" containing the byte code.

```
1  public class Sum {
2      public static void main(String[] args) {
3          System.out.println(sum(6 * 6));
4      }
5
6      static int sum(int n) {
7          int s = 0;
8          for (int i = 0; i <= n; ++i) {
9              s += i;
10         }
11         return s;
12     }
13 }
```

Figure 11.13: A *Java* program computing $\sum_{i=0}^{6^2} i$.

Next, in order to *decompile* the "`.class`" file, we run the command

```
javap -c Sum.class
```

The output of this command is shown in Figure 11.14. We see that the syntax used by `javap` differs a bit from the syntax used by *Jasmin*. It is rather unfortunate that the company developing *Java* has decided not to make their

assembler public. Still, we can see that the output produced by `javap` is quite similar to the input accepted by `jasmin`.

```
Compiled from "Sum.java"
public class Sum {
  public Sum();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: bipush        36
       5: invokestatic  #3                  // Method sum:(I)I
       8: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
      11: return

  static int sum(int);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_2
       5: iload_0
       6: if_icmpgt     19
       9: iload_1
      10: iload_2
      11: iadd
      12: istore_1
      13: iinc          2, 1
      16: goto          4
      19: iload_1
      20: ireturn
}
```

Figure 11.14: The output of "`javap -c Sum.class`".

Chapter 12

Entwicklung eines einfachen Compilers

In diesem Kapitel konstruieren wir einen Compiler, der ein Fragment der Sprache `C` in *Java*-Assembler übersetzt. Das von dem Compiler übersetzte Fragment der Sprache `C` bezeichnen wir als *Integer*-`C`, denn es steht dort nur der Datentyp `int` zur Verfügung. Zwar wäre es problemlos möglich, auch weitere Datentypen zu unterstützen, allerdings würde der Mehraufwand dann in keinem guten Verhältnis zum didaktischen Nutzen des Beispiels mehr stehen.

Ein Compiler besteht prinzipiell aus den folgenden Komponenten:

1. Der Scanner liest die zu übersetzende Datei ein und zerlegt diese in eine Folge von Token.

   Wir werden unseren Scanner mit Hilfe des Werkzeugs PLY entwickeln.

2. Der Parser liest die Folge von Token und produziert als Ergebnis einen abstrakten Syntax-Baum.

   Wir werden bei der Erstellung des Parsers ebenfalls PLY verwenden.

3. Der Typ-Checker überprüft den abstrakten Syntax-Baum auf Typ-Fehler.

   Da die von uns übersetzte Sprache nur einen einzelnen Datentyp enthält, erübrigt sich diese Phase für den von uns entwickelten Compiler.

4. In realen Compilern erfolgt nun eine *Optimierungsphase*, die wir aus Zeitgründen aber nicht mehr betrachten können.

5. Der Code-Generator übersetzt schließlich den Parse-Baum in eine Folge von JAVA-Assembler-Befehlen.

6. Das dabei entstehende Assembler-Programm können wir mit *Jasmin* in Java-Bytecode übersetzen.

7. Der Java-Bytecode kann mit Hilfe des Befehls `Java` ausgeführt werden.

Bei Compilern, deren Zielcode ein RISC-Assembler-Programm ist, wird normalerweise zunächst auch ein Code erzeugt, der dem JVM-Code ähnelt. Ein solcher Code wird als *Zwischen-Code* bezeichnet. Es bleibt dann die Aufgabe eines sogenannten *Backends*, daraus ein Assembler-Programm für eine gegebene Prozessor-Architektur zur erzeugen. Die schwierigste Aufgabe besteht hier darin, für die verwendeten Variablen eine Register-Zuordnung zu finden, bei der möglichst viele Variablen in Registern vorgehalten werden können. Aus Zeitgründen können wir das Thema der Register-Zuordnung in dieser Vorlesung nicht behandeln.

12.1 Die Programmiersprache Integer-`C`

Wir stellen nun die Sprache *Integer*-`C` vor, die unser Compiler übersetzen soll. In diesem Zusammenhang sprechen wir auch von der *Quellsprache* unseres Compilers. Abbildung 12.1 auf Seite 156 zeigt die Grammatik der Quellsprache in erweiterter Backus-Naur-Form (EBNF). Die Grammatik für *Integer*-`C` verwendet die folgenden Terminale:

1. ID steht für eine Folge von Ziffern, Buchstaben und dem Unterstrich, die mit einem Buchstaben beginnt. Eine ID bezeichnet entweder eine Variable oder den Namen einer Funktion.

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{function}+ \\
\textit{function} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“(”}\ \textit{paramList}\ \text{“)”}\ \ \text{“\{”}\ \textit{decl}+\ \textit{stmnt}+\ \text{“\}”} \\
\textit{paramList} & \rightarrow & (\text{“}\texttt{int}\text{”}\ \textsc{Id}\ (\text{“,”}\ \ \text{“}\texttt{int}\text{”}\ \textsc{Id})*)? \\
\textit{decl} & \rightarrow & \text{“}\texttt{int}\text{”}\ \textsc{Id}\ \text{“;”} \\
\textit{stmnt} & \rightarrow & \text{“\{”}\ \textit{stmnt}*\ \text{“\}”} \\
 & | & \textsc{Id}\ \text{“=”}\ \textit{expr}\ \text{“;”} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{if}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt}\ \text{“}\texttt{else}\text{”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{while}\text{”}\ \ \text{“(”}\ \textit{boolExpr}\ \text{“)”}\ \textit{stmnt} \\
 & | & \text{“}\texttt{return}\text{”}\ \textit{expr}\ \text{“;”} \\
 & | & \textit{expr}\ \text{“;”} \\
\textit{boolExpr} & \rightarrow & \textit{expr}\ (\text{“==”} \mid \text{“!=”} \mid \text{“<=”} \mid \text{“>=”} \mid \text{“<”} \mid \text{“>”})\ \textit{expr} \\
 & | & \text{“!”}\ \textit{boolExpr} \\
 & | & \textit{boolExpr}\ (\text{“\&\&”} \mid \text{“||”})\ \textit{boolExpr} \\
\textit{expr} & \rightarrow & \textit{expr}\ (\text{“+”} \mid \text{“-”} \mid \text{“*”} \mid \text{“/”} \mid \text{“\%”})\ \textit{expr} \\
 & | & \text{“(”}\ \textit{expr}\ \text{“)”} \\
 & | & \textsc{Number} \\
 & | & \textsc{Id}\ (\text{“(”}\ (\textit{expr}\ (\text{“,”}\ \textit{expr})*)?\ \text{“)”})?
\end{array}
$$

Figure 12.1: Eine EBNF-Grammatik für *Integer*-C

2. NUMBER steht für eine Folge von Ziffern, die als Dezimalzahl interpretiert wird.
3. Daneben haben wir eine Reihe von Operatoren wie z.B. “+” , “-” , etc., sowie Schlüsselwörter wie beispielsweise “`if`” , “`while`” .

Nach der oben angegebenen Grammatik ist ein Programm eine Liste von Funktionen. Eine Funktion besteht aus der Deklaration der Signatur, worauf in geschweiften Klammern eine Liste von Deklarationen (*decl*) und Befehlen (*stmt*) folgt. Der Aufbau der einzelnen Befehle ist dann ähnlich wie bei der Sprache SL, für die wir im Kapitel 7.4 einen Interpreter entwickelt haben. Die in Abbildung 12.1 gezeigte Grammatik ist mehrdeutig:

1. Die Grammatik hat das *Dangling-Else-Problem*.

   Dieses Problem haben wir bereits im Kapitel 9 im Rahmen einer Aufgabe diskutiert. Wir hatten damals das Problem dadurch gelöst, dass wir das Schlüsselwort “`else`” shiften. Da dies genau das ist, was PLY per default in einem Shift-Reduce-Konflikt macht, brauchen wir uns um dieses Problem nicht weiter zu kümmern.

2. Für die bei arithmetischen und Boole'schen Ausdrücken verwendeten Operatoren müssen Präzedenzen festgelegt werden, um die Mehrdeutigkeiten bei der Interpretation dieser Ausdrücke aufzulösen.

Abbildung 12.2 zeigt ein einfaches INTEGER-C-Programm. Die Funktion *sum*(n) berechnet die Summe $\sum_{i=1}^{n} i$ und die Funktion *main*() ruft die Funktion `sum` mit dem Argument $6 \cdot 6$ auf. Die in dem Programm verwendete Funktion `println` gibt ihr Argument gefolgt von einem Zeilenumbruch aus. Wir gehen hier davon aus, dass diese Funktion vordefiniert ist.

```
1  int sum(int n) {
2      int s;
3      s = 0;
4      while (n != 0) {
5          s = s + n;
6          n = n - 1;
7      };
8      return s;
9  }
10 int main() {
11     int n;
12     n = 6 * 6;
13     println(sum(n));
14 }
```

Figure 12.2: Ein einfaches INTEGER-C-Programm.

12.2 Developing the Scanner and the Parser

We construct both the scanner and the parser using *Ply*. Figure 12.3 shows the scanner. The scanner recognizes the operators, keywords, identifiers, and numbers. As the scanner is mostly similar to those scanners that we have already seen before, there are only a few points worth mentioning.

1. If a token does not have to be manipulated by the scanner, then it can be defined by a simple equation of the form

   ```
   t_name = regexp
   ```

   where *name* is the name of the token and *regexp* is the regular expression defining the token. We have used this shortcut to define most of the tokens recognized by our scanner.

2. While we do not need to declare tokens for those operators that consist of just one token, we have to declare those tokens that consist of two or more characters. For example, the operator "==" is represented by the token `'EQ'` and defined in line 9.

3. Our programming language supports single line comments that start with the string "//" and extend to the end of the line. These comments are implemented via the token `COMMENT` that is defined in line 16–18. Note that the function `t_COMMENT` does not return a value. Therefore, these comments are simply discarded. This is also the reason that we have to use a function to define this token.

4. The most interesting aspect is the implementation of keywords like "`while`" or "`return`". Note that we have defined separate tokens for all of the keywords but we have not defined any of these tokens. For example, we have not defined `t_return`. The reason is that, syntactically, all of the keywords are identifiers and are recognized by the regular expression

   ```
   r'[a-zA-Z][a-zA-Z0-9_]*'
   ```

   that is used for recognizing identifiers in line 28. However, the function `t_ID` that recognizes identifiers does not immediately return the token `'ID'` when it has scanned the name of an identifier. Rather, it first checks whether this name is a predefined keyword. The predefined keywords are the key of the dictionary `Keywords` that is defined in line 38-43. The values of the keywords in this dictionary are the token types. Therefore, the function `t_ID` sets the token type of the token that is returned to the value stored in the dictionary `Keywords`. In case a name is not defined in this dictionary, the token type is set to `'ID'`.

5. The scanner implements the function `t_newline` in line 6. This function is needed to update the attribute `lineno` of the scanner. This attribute stores the line number and is needed for error messages.

```
1  import ply.lex as lex
2
3  tokens = [ 'NUMBER', 'ID', 'EQ', 'NE', 'LE', 'GE', 'AND', 'OR',
4             'INT', 'IF', 'ELSE', 'WHILE', 'RETURN'
5           ]
6
7  t_NUMBER = r'0|[1-9][0-9]'
8
9  t_EQ  = r'=='
10 t_NE  = r'!='
11 t_LE  = r'<='
12 t_GE  = r'>='
13 t_AND = r'&&'
14 t_OR  = r'\|\|'
15
16 def t_COMMENT(t):
17     r'//[^\n]*'
18     pass
19
20 Keywords = { 'int'   : 'INT',
21              'if'    : 'IF',
22              'else'  : 'ELSE',
23              'while' : 'WHILE',
24              'return': 'RETURN'
25            }
26
27 def t_ID(t):
28     r'[a-zA-Z][a-zA-Z0-9_]*'
29     t.type = Keywords.get(t.value, 'ID')
30     return t
31
32 literals = ['+', '-', '*', '/', '%', '(', ')', '{', '}', ';', '=', '<', '>', '!']
33
34 t_ignore  = ' \t\r'
35
36 def t_newline(t):
37     r'\n+'
38     t.lexer.lineno += t.value.count('\n')
39
40 def t_error(t):
41     print(f"Illegal character '{t.value[0]}' in line {t.lineno}.")
42     t.lexer.skip(1)
43
44 __file__ = 'main'
45
46 lexer = lex.lex()
```

Figure 12.3: The scanner for *Integer*-C.

Now we are ready to present the specification of our parser. For reasons of space, we have split the grammar into six parts. We start with Figure 12.4 on page 159.

```
1  import ply.yacc as yacc
2
3  start = 'program'
4
5  precedence = (
6      ('left', 'OR'),
7      ('left', 'AND'),
8      ('left', '!'),
9      ('nonassoc', 'EQ', 'NE', 'LE', 'GE', '<', '>'),
10     ('left', '+', '-'),
11     ('left', '*', '/', '%')
12 )
13
14 def p_program_one(p):
15     "program : function"
16     p[0] = ('program', p[1])
17
18 def p_program_more(p):
19     "program : function program"
20     p[0] = ('program', p[1]) + p[2][1:]
21
22 def p_function(p):
23     "function : INT ID '(' param_list ')' '{' decl_list stmnt_list '}'"
24     p[0] = ('fct', p[2], p[4], p[7], p[8])
```

Figure 12.4: Grammar for Integer-C, part 1.

1. Line 3 declares the start symbol `program`.
2. Line 5–12 declare the operator precedences.
   (a) The operator "`||`" that represents logical or has the lowest precedence and associates to the left.
   (b) The operator "`&&`" that represents logical and binds stronger than "`||`" but not as strong as the negation operator "`!`".
   (c) The negation operator "`!`" is right associative because we want to interpret an expression of the form `!!`a as `!(!`a`)`.
   (d) The comparison operators "`==`", "`!=`", "`<=`", "`>=`", "`<`", and "`>`" are non-associative, since our programming languages disallows the chaining of comparison operators, i.e. an expression of the form $x < y < z$ is syntactically invalid.
   (e) The arithmetical operators "`+`" and "`-`" have a lower precedence than the arithmetical operators "`*`" and "`/`".
3. A `program` is a non-empty list of `function` definitions. Therefore, it is either a single function definition (line 14–16) or it is a function definition followed by more function definitions (line 18–20).

   The purpose of the parser is to turn the given program into an abstract syntax tree. This abstract syntax tree is represented as a nested tuple. A program consisting of the functions $f_1, \cdots, f_n$ is represented as the nested tuple.

   $$(\text{“program”}, f_1, \cdots, f_n).$$

   If in line 19 a `program` consisting of a `function` and another `program` is parsed, the expression `p[2]` in line 20 refers to the abstract syntax tree representing the `program` that follows the `function`. The expression

`p[2][1:]` discards the keyword "`program`" that is at the start of this nested tuple. Hence, `p[2][1:]` is just a tuple of the functions following the first `function`. Therefore, the assignment to `p[0]` creates a nested tuple that starts with the keyword "`program`" followed by all the functions defined in this program.

4. A `function` definition starts with the keyword "`int`" that is followed by the name of the function, an opening parenthesis, a list of the parameters of the function, a closing parenthesis, an opening brace, a list of declarations, a list of statements, and finally a closing brace.

```
25  def p_param_list_empty(p):
26      "param_list :"
27      p[0] = ('.', )
28
29  def p_param_list_one(p):
30      "param_list : INT ID"
31      p[0] = ('.', p[2])
32
33  def p_param_list_more(p):
34      "param_list : INT ID ',' ne_param_list"
35      p[0] = ('.', p[2]) + p[4][1:]
36
37  def p_ne_param_list_one(p):
38      "ne_param_list : INT ID"
39      p[0] = ('.', p[2])
40
41  def p_ne_param_list_more(p):
42      "ne_param_list : INT ID ',' ne_param_list"
43      p[0] = ('.', p[2]) + p[4][1:]
44
45  def p_decl_list_one(p):
46      "decl_list :"
47      p[0] = ('.',)
48
49  def p_decl_list_more(p):
50      "decl_list : INT ID ';' decl_list"
51      p[0] = ('.', p[2]) + p[4][1:]
52
53  def p_stmnt_list_one(p):
54      "stmnt_list :"
55      p[0] = ('.',)
56
57  def p_stmnt_list_more(p):
58      "stmnt_list : stmnt stmnt_list"
59      p[0] = ('.', p[1]) + p[2][1:]
```

Figure 12.5: Grammar for Integer-C, part 2.

Figure 12.5 shows the definition of parameter lists, declaration lists, and statement lists.

1. The definition of a list of parameters is quite involved, because parameter lists may be empty and, furthermore, different parameters have to be separated by ",". Therefore, we have to introduce the additional syntactical variable `ne_param_list`, which represents a non-empty parameter list. The grammar rules for `param_list` are as follows:

```
param_list
     :
     | 'int' ID
     | 'int' ID ',' ne_param_list
     ;
ne_param_list
     : 'int' ID
     | 'int' ID ',' ne_param_list
     ;
```

We use the "." as a key to represent lists of any kind as nested tuples.

2. The grammar rules for a list of declarations are as follows:

```
decl_list
    :
    | 'int' ID ';' decl_list
    ;
```

Observe that with this definition a list of declarations may be empty. The grammar rules are simpler than for parameter lists because every variable declaration is ended with a semicolon, while parameters are separated by commas and the last parameter must not be followed by a comma.

3. The grammar rules for a list of statements are as follows:

```
stmnt_list
    :
    | stmnt stmnt_list
    ;
```

Figure 12.6 on page 162 shows how the variable `stmnt` is defined. The grammar rules for statements are as follows:

```
stmnt : '{' stmnt_list '}'
      | ID '=' expr ';'
      | 'if' '(' bool_expr ')' stmnt
      | 'if' '(' bool_expr ')' stmnt 'else' stmnt
      | 'while' '(' bool_expr ')' stmnt
      | 'return' expr ';'
      | expr ';'
      ;
```

Figure 12.7 on page 163 shows how Boolean expressions are defined. The grammar rules are as follows:

```
bool_expr : bool_expr '||' bool_expr
          | bool_expr '&&' bool_expr
          | '!' bool_expr
          | '(' bool_expr ')'
          | expr '==' expr
          | expr '!=' expr
          | expr '<=' expr
          | expr '>=' expr
          | expr '<'  expr
          | expr '>'  expr
          ;
```

```
60  def p_stmnt_block(p):
61      "stmnt : '{' stmnt_list '}'"
62      p[0] = p[2]
63
64  def p_stmnt_assign(p):
65      "stmnt : ID '=' expr ';'"
66      p[0] = ('=', p[1], p[3])
67
68  def p_stmnt_if(p):
69      "stmnt : IF '(' bool_expr ')' stmnt"
70      p[0] = ('if', p[3], p[5])
71
72  def p_stmnt_if_else(p):
73      "stmnt : IF '(' bool_expr ')' stmnt ELSE stmnt"
74      p[0] = ('if-else', p[3], p[5], p[7])
75
76  def p_stmnt_while(p):
77      "stmnt : WHILE '(' bool_expr ')' stmnt"
78      p[0] = ('while', p[3], p[5])
79
80  def p_stmnt_return(p):
81      "stmnt : RETURN expr ';'"
82      p[0] = ('return', p[2])
83
84  def p_stmnt_expr(p):
85      "stmnt : expr ';'"
86      p[0] = p[1]
```

Figure 12.6: Grammar for Integer-C, part 3.

Figure 12.8 on page 164 shows lists of expression how arithmetic expressions are defined. The grammar rules are as follows:

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | expr '%' expr
     | '(' expr ')'
     | NUMBER
     | ID
     | ID '(' expr_list ')'
```

Finally, Figure 12.9 on page 165 shows how `expr_list` is defined.

```
expr_list :
          | expr
          | expr ',' ne_expr_list
          ;

ne_expr_list : expr
             | expr ',' ne_expr_list
             ;
```

```
87  def p_bool_expr_or(p):
88      "bool_expr : bool_expr OR bool_expr"
89      p[0] = ('||', p[1], p[3])
90
91  def p_bool_expr_and(p):
92      "bool_expr : bool_expr AND bool_expr"
93      p[0] = ('&&', p[1], p[3])
94
95  def p_bool_expr_neg(p):
96      "bool_expr : '!' bool_expr"
97      p[0] = ('!', p[2])
98
99  def p_bool_expr_paren(p):
100     "bool_expr : '(' bool_expr ')'"
101     p[0] = p[2]
102
103 def p_bool_expr_eq(p):
104     "bool_expr : expr EQ expr"
105     p[0] = ('==', p[1], p[3])
106
107 def p_bool_expr_ne(p):
108     "bool_expr : expr NE expr"
109     p[0] = ('!=', p[1], p[3])
110
111 def p_bool_expr_le(p):
112     "bool_expr : expr LE expr"
113     p[0] = ('<=', p[1], p[3])
114
115 def p_bool_expr_ge(p):
116     "bool_expr : expr GE expr"
117     p[0] = ('>=', p[1], p[3])
118
119 def p_bool_expr_lt(p):
120     "bool_expr : expr '<' expr"
121     p[0] = ('<', p[1], p[3])
122
123 def p_bool_expr_gt(p):
124     "bool_expr : expr '>' expr"
125     p[0] = ('>', p[1], p[3])
```

Figure 12.7: Grammar for Integer-C, part 4.

Furthermore, this function shows the definition of the function `p_error`, which is called if PLY detects a syntax error. PLY will detect a syntax error in the case that the state of the generated shift/reduce parser has no action for the next input token. The argument `p` to the function `p_error` is the first token for which the action of the shift/reduce parser is undefined. In this case `p.value` is the part of the input string corresponding to this token.

```
126 def p_expr_plus(p):
127     "expr : expr '+' expr"
128     p[0] = ('+', p[1], p[3])
129
130 def p_expr_minus(p):
131     "expr : expr '-' expr"
132     p[0] = ('-', p[1], p[3])
133
134 def p_expr_times(p):
135     "expr : expr '*' expr"
136     p[0] = ('*', p[1], p[3])
137
138 def p_expr_divide(p):
139     "expr : expr '/' expr"
140     p[0] = ('/', p[1], p[3])
141
142 def p_expr_modulo(p):
143     "expr : expr '%' expr"
144     p[0] = ('%', p[1], p[3])
145
146 def p_expr_group(p):
147     "expr : '(' expr ')'"
148     p[0] = p[2]
149
150 def p_expr_number(p):
151     "expr : NUMBER"
152     p[0] = ('Number', p[1])
153
154 def p_expr_id(p):
155     "expr : ID"
156     p[0] = p[1]
157
158 def p_expr_fct_call(p):
159     "expr : ID '(' expr_list ')'"
160     p[0] = ('call', p[1]) + p[3][1:]
```

Figure 12.8: Grammar for Integer-C, part 5.

Aufgabe 32: Extend the parser that is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Ply/Compiler.ipynb

so that it supports the ternary C-operator for conditional expressions. For example, in C, the following assignment using the ternary operator can be used the assign `max` the maximum of the values of `x` and `y`:

```
max = x > y ?  x :  y;
```

Extend the parser so it can process the assignment shown above. ◇

```
161 def p_expr_list_empty(p):
162     "expr_list :"
163     p[0] = ('.',)
164 
165 def p_expr_list_one(p):
166     "expr_list : expr"
167     p[0] = ('.', p[1])
168 
169 def p_expr_list_more(p):
170     "expr_list : expr ',' ne_expr_list"
171     p[0] = ('.', p[1]) + p[3][1:]
172 
173 def p_ne_expr_list_one(p):
174     "ne_expr_list : expr"
175     p[0] = ('.', p[1])
176 
177 def p_ne_expr_list_more(p):
178     "ne_expr_list : expr ',' ne_expr_list"
179     p[0] = ('.', p[1]) + p[3][1:]
180 
181 def p_error(p):
182     if p:
183         print(f'Syntax error at token "{p.value}" in line {p.lineno}.')
184     else:
185         print('Syntax error at end of input.')
186 
187 parser = yacc.yacc(write_tables=False, debug=True)
```

Figure 12.9: Grammar for Integer-C, part 6.

12.3 Code Generation

Next, we discuss the generation of code. We structure our representation by discussing the code generation for arithmetic expressions, Boolean expression, statements, and function definitions separately.

12.3.1 Translation of Arithmetic Expressions

Given an arithmetic expression e, the translation of e is supposed to generate some code that, when executed, places the result of evaluating the expression e onto the stack. To this end we define a function `compile_expr` that has the following signature:

$$\texttt{compile\_expr} : \mathit{Expr} \times \mathit{SymbolTable} \times \mathit{ClassName} \to \big\langle \mathit{List}[\mathit{AsmCmd}], \mathbb{N} \big\rangle.$$

A call of this function has the form

$$\texttt{compile\_expr}(\texttt{expr}, \texttt{st}, \texttt{name}).$$

The interpretation of the arguments is as follows:

(a) `expr` is the arithmetic expression that is to be translated into assembler code.

(b) `st` is the symbol table: Concretely, this is a dictionary that maps the variable names to their position in the local variable frame. For example, if "x" is a variable that occupies the third slot in the local variable frame, then we have

```
st['x'] = 2,
```

because the first variable in the local variable frame has index 0. We will discuss later how the positions of the variables in the local variable frame is assigned.

(c) `name` is the name of the class that is to be used by our compiler.

As we generate *Java* assembler and in *Java* every function has to be a part of a class, all functions that we create have to be static functions that are defined inside the *Java* `.class` file that is generated. The parameter `name` specifies the name of this class.

The argument `name` is only needed when function calls are translated.

The function `compile_expr` returns a pair.

(a) The first component of this pair is a list of *Java* assembler commands that adhere to the syntax recognized by *Jasmin*. In general, when the expression that is translated is complex, the execution of these assembler commands might need considerable room on the stack. However, it has to be guaranteed that when the execution of theses commands finishes, the stack is back to its original height plus one because the net effect of executing these commands must be to put the value of the expression on the stack.

(b) The second component of the return value of `compile_expr` is a natural number. This natural number tells us how much the stack might grow when `expr` is evaluated. This information is needed because the *Java* virtual machine needs this information in advance: In *Java*, every function has to declare how much space it might use on the stack. This declaration is done using the pseudo assembler command `.limit`. Controlling the maximum height of the stack is a security feature of *Java* that prevents those exploits that utilize stack overflows.

In the following, we discuss the evaluation of the different arithmetic expressions one by one.

Translation of a Variable

If the expression that is to be compiled is a variable v, we have to load this variable onto the stack using the command `iload`. This command has one parameter which is the index of the variable on the local variable frame. This index is stored in the symbol table `st`. Since we just need one entry on the stack to store the variable we have

$$\texttt{compile\_expr}(v, \texttt{st}, \texttt{name}) = \big\langle \texttt{[iload st[}v\texttt{]]},\ 1 \big\rangle \quad \text{if } v \text{ is a variable.}$$

The code for translating a variable is shown in Figure 12.10 on page 167.

```
1   def compile_expr(expr, st, class_name):
2       if isinstance(expr, str):
3           return [f'iload {st[expr]}'], 1
4       ...
```

Figure 12.10: Translation of a variable.

Translation of a Constant

We can load a constant c onto the stack using the assembler command

`ldc` c.

As we only need the room to store c on the stack, we have

$\texttt{compile\_expr}(c, \texttt{st}, \texttt{name}) = \langle \texttt{[ldc } c \texttt{]},\ 1 \rangle$ if c is a constant.

The code for translating a constant is shown in Figure 12.11 on page 167.

```
5   def compile_expr(expr, st, class_name):
6       ...
7       elif expr[0] == 'Number':
8           _, n = expr
9           return [f'ldc {n}'], 1
10      ...
```

Figure 12.11: Translation of a variable.

Translation of an Arithmetic Operator

In order to translate an expression of the form

lhs "+" *rhs*

into assembler code, we first have to recursively translate the expressions *lhs* and *rhs* into assembler code. Later, when this code is executed the values of the expressions *lhs* und *rhs* are placed on the stack. We add these values using the command `iadd`. If the evaluation of *lhs* needs s_1 words on the stack and the evaluation of *rhs* needs s_2 words on the stack, then the evaluation of *lhs* + *rhs* needs

$$\max(s_1, 1 + s_2)$$

words on the stack, because when *rhs* is evaluated, the value of *lhs* occupies already one position on the stack and hence the evaluation of *rhs* can only use the memory cells that are above the position where *lhs* is stored. Therefore, the compilation of *lhs* + *rhs* can be specified as follows:

$$\begin{array}{ll} & \texttt{compile\_expr}(\textit{lhs}, \texttt{st}, \texttt{name}) = \langle L_1, s_1 \rangle \\ \wedge & \texttt{compile\_expr}(\textit{rhs}, \texttt{st}, \texttt{name}) = \langle L_2, s_2 \rangle \\ \rightarrow & \texttt{compile\_expr}(\textit{lhs} + \textit{rhs}, \texttt{st}, \texttt{name}) = \big\langle L1 + L2 + \texttt{[iadd]}, \max(s_1, 1 + s_2) \big\rangle. \end{array}$$

Figure 12.12 on page 168 shows how the translation of arithmetic operators is implemented. The translation of subtraction, multiplication, and division is similar.

```
11    def compile_expr(expr, st, class_name):
12        ...
13        elif expr[0] in  ['+', '-', '*', '/', '%']:
14            op, lhs, rhs = expr
15            L1, sz1 = compile_expr(lhs, st, class_name)
16            L2, sz2 = compile_expr(rhs, st, class_name)
17            OpToCmd = { '+': 'iadd', '-': 'isub', '*': 'imul', '/': 'idiv', '%': 'irem' }
18            Cmd     = indent(OpToCmd[op])
19            return L1 + L2 + [Cmd], max(sz1, 1 + sz2)
20        ...
```

Figure 12.12: Translation of the addition of expressions.

Translation of a Function Call

In order to translate a function call of the form $f(a_1, \cdots, a_n)$ we first have to translate the arguments a_1, $\cdots$, a_n. Then there are two cases:

1. f is a user defined function. In this case, the byte code treats f as a static function of the class `class_name`. This static function can be called using the assembler command `invokestatic`. In order to calculate the stack size that needs to be reserved for the evaluation of $f(a_1, \cdots, a_n)$, let us assume that we have

   $$\texttt{compile\_expr}(a_i, \texttt{st}, \texttt{class\_name}) = \langle L_i, s_i \rangle,$$

   i.e. evaluation of the i^{th} argument is done by the list of assembler commands L_i and needs a stack size of s_i. As we are evaluating the arguments a_i in the order from a_1 to a_n, the evaluation of a_i needs a stack size of $i - 1 + s_i$, since the results from the evaluation of the arguments a_1, $\cdots$, a_{i-1} are already placed on the stack. Therefore, if we define

   $$s := \max(s_1, 1 + s_2, \cdots, i - 1 + s_i, \cdots, n - 1 + s_n),$$

   then s is the total amount of stack size needed to evaluate the arguments. Furthermore, let us define the signature `fs` of the function f as the string

   $$\texttt{fs} := \texttt{class\_name/f(}\underbrace{\texttt{I}\cdots\texttt{I}}_{n}\texttt{)I}.$$

   Then we can define the value of $\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big)$ as follows:

   $$\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big) := \big\langle L_1 + \cdots + L_n + [\ \texttt{invokestatic fs}\ ], \max(s, 1) \big\rangle.$$

2. If f is the function `println`, then *Jasmin* treats the function f as a method of the predefined *Java* object `java.lang.System.out`. This method can be invoked using the assembler command `invokevirtual`. As before, let us assume that we have

   $$\texttt{compile\_expr}(a_i, \texttt{st}, \texttt{class\_name}) = \langle L_i, s_i \rangle.$$

   This time, we have to start by putting the object `java.lang.System.out` onto the stack before we can evaluate any of the arguments. Therefore, in order to calculate the stack size we now define

   $$s := \max(1 + s_1, 2 + s_2, \cdots, i - 1 + s_i, \cdots, n + s_n).$$

   In this case we have to define the signature `fs` of f as the string

   $$\texttt{fs} := \texttt{java/io/PrintStream/println(}\underbrace{\texttt{I}\cdots\texttt{I}}_{n}\texttt{)V}.$$

   Furthermore, we define the command `gs` for putting the object `java.lang.System.out` onto the stack as

$$\texttt{gs} := \texttt{getstatic java/lang/System/out Ljava/io/PrintStream;}.$$

Then we can define the value of $\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big)$ as follows:

$$\texttt{compile\_expr}\big(f(a_1, \cdots, a_n), \texttt{st}, \texttt{class\_name}\big) := \big\langle \texttt{[gs]} + L_1 + \cdots + L_n + \texttt{[invokestatic fs]}, \max(s, 1)\big\rangle.$$

Figure 12.12 on page 168 shows how the translation of function calls is implemented.

```
21  def compile_expr(expr, st, class_name):
22      ...
23      elif expr[0] == 'call' and expr[1] == 'println':
24          _, _, *args = expr
25          CmdLst    = ['getstatic java/lang/System/out Ljava/io/PrintStream;')
26          stck_size = 0
27          cnt       = 0
28          for arg in args:
29              L, sz_arg = compile_expr(arg, st, class_name)
30              stck_size = max(stck_size, cnt + 1 + sz_arg)
31              CmdLst   += L
32              cnt      += 1
33          CmdLst += [f'invokevirtual java/io/PrintStream/println({"I"*cnt})V']
34          return CmdLst, stck_size
35      elif expr[0] == 'call' and expr[1] != 'println':
36          _, f, *args = expr
37          CmdLst    = []
38          stck_size = 0
39          cnt       = 0
40          for arg in args:
41              L, sz_arg = compile_expr(arg, st, class_name)
42              stck_size = max(stck_size, cnt + sz_arg)
43              CmdLst   += L
44              cnt      += 1
45          CmdLst += [f'invokestatic {class_name}/{f}({"I"*cnt})I']
46          return CmdLst, max(stck_size, 1)
47      ...
```

Figure 12.13: Translation of function calls.

12.3.2 Translation of Boolean Expressions

Boolean expressions are build from equations and inequations using the logical operators "!" (logical not), "&&" (logical and), and "||" (logical or). To compile Boolean expressions we define a function `compile_bool` that has the following signature:

$$\texttt{compile\_bool} : \mathit{BoolExpr} \times \mathit{SymbolTable} \times \mathit{ClassName} \to \big\langle \mathit{List}[\mathit{AsmCmd}], \mathbb{N}\big\rangle.$$

The interpretation of the arguments is similar to the interpretation of the arguments of the function `compile_expr`. When we execute the list of assembler commands that result from the compilation of a Boolean expression, we expect that the execution of theses commands either puts the number 1 (representing `True`) or the number 0 (representing `False`) onto the stack. We start our discussion of the function `compile_bool` with the translation of equations.

Translation of Equations

In the *Java* virtual machine, the logical values `True` and `False` are represented by the integers 1 and 0, respectively. Therefore, the code produced from translating an equation of the form

lhs == *rhs*

should either place the number 1 or 0 onto the stack: If the value of *lhs* is equal to the value of *rhs*, the number 1 has to be put on the stack, otherwise the value 0 has to be put onto the stack. Let us assume that we have

$$\texttt{compile\_expr}(\textit{lhs}, \texttt{st}, \texttt{class\_name}) = \langle L_1, s_1 \rangle \quad \text{and} \quad \texttt{compile\_expr}(\textit{rhs}, \texttt{st}, \texttt{class\_name}) = \langle L_2, s_2 \rangle.$$

Then the stack size needed for evaluating the equation *lhs* == *rhs* is given by the expression

$$\max(s_1, 1 + s_2)$$

and the function `compile_bool` can be specified as follows:

$$\begin{array}{lcl} \textit{compile\_bool}(\textit{lhs} == \textit{rhs}, \texttt{st}, \texttt{class\_name}) & = & \big\langle \quad L_1 \\ & & + \quad L_2 \\ & & + \quad [\,\texttt{if\_icmpeq}\ \textit{true}\,] \\ & & + \quad [\,\texttt{bipush}\ 0\,] \\ & & + \quad [\,\texttt{goto}\ \textit{next}\,] \\ & & + \quad [\,\textit{true}\texttt{:}\,] \\ & & + \quad [\,\texttt{bipush}\ 1\,] \\ & & + \quad [\,\textit{next}\texttt{:}\,], \qquad\qquad s \big\rangle \end{array}$$

Here, *true* and *next* have to be new labels that do not occur elsewhere in the code for the function that is compiled. Let me explain this equation in detail:

1. L_1 and L_2 are the lists of assembler commands that evaluate *lhs* and *rhs*, respectively.
2. Executing L_1 and L_2 leaves two values on the stack. These values are then compared using the assembler command `if_icmpeq`. If these value are the same, execution proceeds at the label *true*.
3. Otherwise execution commences with the next instruction and hence the value 0 is pushed onto the stack.
4. Next, the control flow jumps to the label *next*, which is at the end of the generated list of assembler commands. Hence, in this case the execution of equation *lhs* == *rhs* is finished.
5. If the values of *lhs* and *rhs* are the same, the number 1 is pushed onto the stack.

The translation of a negated equation of the form

lhs != *rhs*

is similar to the translation of an equation: We only have to replace the command `if_icmpeq` with the command `if_icmpne`. Similarly, if the inequation has the form

lhs <= *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmple`. If the inequation has the form

lhs >= *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmpge`. If the inequation has the form

lhs < *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmplt`. If the inequation has the form

lhs > *rhs*

we have to replace the command `if_icmpeq` with the command `if_icmpgt`. Figure 12.14 on page 171 shows how the translation of equations and inequations is implemented.

```
def compile_bool(expr, st, class_name):
    if expr[0] in ['==', '!=', '<=', '>=', '<', '>']:
        OpToCmd = { '==': 'if_icmpeq',
                    '!=': 'if_icmpne',
                    '<=': 'if_icmple',
                    '>=': 'if_icmpge',
                    '<' : 'if_icmplt',
                    '>' : 'if_icmpgt'
                  }
        op, lhs, rhs = expr
        L1, sz1    = compile_expr(lhs, st, class_name)
        L2, sz2    = compile_expr(rhs, st, class_name)
        true_label = new_label()
        next_label = new_label()
        CmdLst     = L1 + L2
        cmd        = OpToCmd[op]
        CmdLst    += [indent(cmd + ' ' + true_label)]
        CmdLst    += [indent('bipush 0')]
        CmdLst    += [indent('goto ' + next_label)]
        CmdLst    += [' ' * 4 + true_label + ':']
        CmdLst    += [indent('bipush 1')]
        CmdLst    += [' ' * 4 + next_label + ':']
        return CmdLst, max(sz1, 1 + sz2)
        ...
```

Figure 12.14: Translation of equations.

Translation of Binary Boolean Operators

In order to translate an expression of the form

lhs && *rhs*

into assembler code, we first have to recursively translate the expressions *lhs* and *rhs* into assembler code. Later, when this code is executed the values of the expressions *lhs* und *rhs* are placed on the stack. We combine these values using the command `iand`. If the evaluation of *lhs* needs s_1 words on the stack and the evaluation of *rhs* needs s_2 words on the stack, then the evaluation of *lhs* + *rhs* needs

$$\max(s_1, 1 + s_2)$$

words on the stack, because when *rhs* is evaluated, the value of *lhs* occupies already one position on the stack and hence the evaluation of *rhs* can only use the memory cells that are above the position where *lhs* is stored. Therefore, the compilation of *lhs* && *rhs* can be specified as follows:

$$\begin{array}{ll} & \texttt{compile\_bool}(\textit{lhs}, \texttt{st}, \texttt{name}) = \langle L_1, s_1 \rangle \\ \wedge & \texttt{compile\_bool}(\textit{rhs}, \texttt{st}, \texttt{name}) = \langle L_2, s_2 \rangle \\ \rightarrow & \texttt{compile\_bool}(\textit{lhs} + \textit{rhs}, \texttt{st}, \texttt{name}) = \big\langle L1 + L2 + [\,\texttt{iand}\,], \max(s_1, 1 + s_2) \big\rangle. \end{array}$$

The translation of an expression of the form

lhs || *rhs*

is similar: Instead of using the command `iand` we have to use the command `ior` instead. Figure 12.15 on page 172 shows how the translation of binary logical operators is implemented.

```
25    def compile_expr(expr, st, class_name):
26        ...
27        elif expr[0] in ['&&', '||']:
28            op, lhs, rhs = expr
29            OpToCmd      = { '&&': iand, '||': 'ior' }
30            L1, sz1      = compile_bool(lhs, st, class_name)
31            L2, sz2      = compile_bool(rhs, st, class_name)
32            cmd          = OpToCmd[op]
33            CmdLst       = L1 + L2 + [indent(cmd)]
34            return CmdLst, max(sz1, 1 + sz2)
35        ...
```

Figure 12.15: Translation of binary logical operators.

It should be noted that our translation of the binary logical operators is different from what happens in the language C. In C, if an expression of the form

lhs && *rhs*

is evaluated, the evaluation is stopped as soon the evaluation of *lhs* returns 0 because then there is no point to evaluate *rhs*, since if the value of *lhs* is 0, the result of the expression *lhs* && *rhs* is 0, independent from the value of the expression *rhs*.

Translation of Negations

The translation of an expression of the form !*expr* is not as straightforward as the translation of conjunctions and disjunctions. The reason is that *Jasmin* has no assembler command of the form inot that negates a logical value. But there is another way: Since we represent the truth values as 0 and 1, we can specify negation arithmetically as follows:

$$!x = 1 - x.$$

Therefore, if we have

$$\texttt{compile\_bool}(\textit{arg}, \texttt{st}, \texttt{class\_name}) = \langle L, s \rangle$$

we can define:

$$\texttt{compile\_bool}(\texttt{!}\textit{arg}, \texttt{st}, \texttt{class\_name}) = \big\langle \texttt{[bipush 1]} + L + \texttt{[isub]}, 1 + s \big\rangle.$$

Figure 12.16 on page 173 shows how the translation of the negation operator is implemented.

12.3.3 Translation of Statements

Next, we show how statements are compiled. First of all, we agree that the execution of a statement must not change the size of the stack: The size of stack before the execution of a statement must be the same as the size of the stack after after the statement has been executed. Of course, during the execution of the statement the stack may well grow and shrink. But once the execution of the statement has finished, the stack has to be cleaned from all intermediate values that have been put on the stack during the execution of the statement.

To compile statements we define a function compile_stmnt that has the following signature:

$$\texttt{compile\_stmnt} : \textit{Stmnt} \times \textit{SymbolTable} \times \textit{ClassName} \to \big\langle \textit{List}[\textit{AsmCmd}], \mathbb{N} \big\rangle.$$

The interpretation of the arguments is similar to the interpretation of the arguments of the function compile_expr. The only difference is that the first argument now is an abstract syntax tree that represents a statement.

```
36    def compile_expr(expr, st, class_name):
37        ...
38        elif expr[0] == '!':
39            _, arg = expr
40            L, sz  = compile_expr(arg, st, class_name)
41            CmdLst = ['bipush 1'] + L + ['isub']
42            return CmdLst, max(sz1, sz + 1)
43        ...
```

Figure 12.16: Translation of the negation operator.

Translation of Assignments

To begin with, we show how an assignment of the form

x = *expr*

is translated. The idea is to evaluate *expr*. As a consequence of this evaluation the value of *expr* remains on the stack, from where it can be stored in the variable x using the assembler command `istore`. Therefore, if we have

`compile_expr`(*expr*, `st`, `class_name`) $= \langle L, s \rangle$

we can define:

`compile_stmnt`(x = *expr*, `st`, `class_name`) $= \big\langle L +$ `[istore st[x]]`$, s \big\rangle$.

Figure 12.17 on page 173 shows how the translation of an assignment is implemented.

```
1     def compile_stmnt(stmnt, st, class_name):
2         if stmnt[0] == '=':
3             _, var, expr = stmnt
4             CmdLst, sz = compile_expr(expr, st, class_name)
5             CmdLst    += [f'istore {st[var]}']
6             return CmdLst, sz
7         ...
```

Figure 12.17: Translation of an assignment statement.

The Translation of Branching Commands

Next, we show how to translate a branching command of the form

`if` (*bool_expr*) *stmnt*.

Obviously, we first have to translate the Boolean expression *bool_expr*. If we have

`compile_expr`(*bool_expr*, `st`, `class_name`) $= \langle L_1, s_1 \rangle$

and, furthermore,

`compile_stmnt`(*stmnt*, `st`, `class_name`) $= \langle L_2, s_2 \rangle$,

then we can define:

$$\begin{aligned} \textit{compile\_stmnt}(\texttt{if}\ (\textit{expr})\ \textit{stmnt}, \texttt{st}, \texttt{class\_name}) \quad = \quad \langle \quad & L_1 \\ & + \ [\ \texttt{ifeq}\ \textit{else}\ ] \\ & + \ L_2 \\ & + \ [\ \textit{else}\texttt{:}\ ], \qquad \max(s_1, s_2)\rangle \end{aligned}$$

This works because executing the list of assembler commands L_1 leaves either a 0 or a 1 on the stack. If *expr* is false, the top of the stack stores the number 0. In this case, the branching command `ifeq` branches to the label *else* and the assembler commands in the list L_2 are not executed. If *expr* is true, the top of the stack stores the number 1. Therefore, the branching command `ifeq` does not branch and the assembler commands in the list L_2 are executed, i.e. *stmnt* is executed.

The translation of a branching command of the form

`if` (*bool_expr*) $\textit{stmnt}_1$ `else` $\textit{stmnt}_2$

is similar. Let us assume that we have the following:

1. `compile_expr`(*bool_expr*, `st`, `class_name`) $= \langle L_1, s_1\rangle$
2. `compile_stmnt`($\textit{stmnt}_1$, `st`, `class_name`) $= \langle L_2, s_2\rangle$, and
3. `compile_stmnt`($\textit{stmnt}_2$, `st`, `class_name`) $= \langle L_3, s_3\rangle$.

Then we define:

$$\begin{aligned} \textit{compile\_stmnt}(\texttt{if}\ (\textit{bool\_expr})\ \textit{stmnt}_1\ \texttt{else}\ \textit{stmnt}_2, \texttt{st}, \texttt{class\_name}) \quad = \quad \langle \quad & L_1 \\ & + \ [\ \texttt{ifeq}\ \textit{else}\ ] \\ & + \ L_2 \\ & + \ [\texttt{goto}\ \textit{next}] \\ & + \ [\ \textit{else}\texttt{:}\ ] \\ & + \ L_3 \\ & + \ [\ \textit{next}\texttt{:}\ ], \qquad \max(s_1, s_2, s_3)\rangle \end{aligned}$$

Figure 12.18 on page 175 shows how the translation of branching statements is implemented.

Translation of a Loop

If we want to translate a `while` loop of the form

`while` (*cond*) *stmnt*

we recursively compute

`compile_expr`(*cond*, `st`, `class_name`) $= \langle L_1, s_1\rangle$

and

`compile_stmnt`(*stmnt*, `st`, `class_name`) $= \langle L_2, s_2\rangle$,

Then we can define

$$\begin{aligned} \textit{compile\_stmnt}(\texttt{while}\ (\textit{cond})\ \textit{stmnt}, \texttt{st}, \texttt{class\_name}) \quad = \quad \langle \quad & [\textit{loop}\texttt{:}] \\ & + \ L_1 \\ & + \ [\ \texttt{ifeq}\ \textit{next}\ ] \\ & + \ L_2 \\ & + \ [\texttt{goto}\ \textit{loop}] \\ & + \ [\ \textit{next}\texttt{:}\ ], \qquad \max(s_1, s_2)\rangle \end{aligned}$$

Figure 12.19 on page 175 shows how the translation of a loop is implemented.

```
def compile_stmnt(stmnt, st, class_name):
    ...
    elif stmnt[0] == 'if':
        _, expr, sub_stmnt = stmnt
        L1, sz1    = compile_bool(expr, st, class_name)
        L2, sz2    = compile_stmnt(sub_stmnt, st, class_name)
        else_label = new_label()
        CmdLst = L1 + [f'ifeq {else_label}'] + L2 + [else_label + ':']
        return CmdLst, max(sz1, sz2)
    elif stmnt[0] == 'if-else':
        _, expr, then_stmnt, else_stmnt = stmnt
        L1, sz1    = compile_bool(expr, st, class_name)
        L2, sz2    = compile_stmnt(then_stmnt, st, class_name)
        L3, sz3    = compile_stmnt(else_stmnt, st, class_name)
        else_label = new_label()
        next_label = new_label()
        if_stmnt   = f'ifeq {else_label}'
        else_stmnt = else_label + ':'
        next_stmnt = next_label + ':'
        goto_stmnt = f'goto {next_label}'
        CmdLst = L1 + [if_stmnt] + L2 + [goto_stmnt, else_stmnt] + L3 + [next_stmnt]
        return CmdLst, max(sz1, sz2, sz3)
    ...
```

Figure 12.18: Translation of branching statements.

```
def compile_stmnt(stmnt, st, class_name):
    ...
    elif stmnt[0] == 'while':
        _, expr, body_stmnt = stmnt
        L1, sz1    = compile_bool(expr, st, class_name)
        L2, sz2    = compile_stmnt(body_stmnt, st, class_name)
        loop_label = new_label()
        next_label = new_label()
        if_stmnt   = f'ifeq {next_label}'
        loop_stmnt = loop_label + ':'
        next_stmnt = next_label + ':'
        goto_stmnt = f'goto {loop_label}'
        CmdLst = [loop_stmnt] + L1 + [if_stmnt] + L2 + [goto_stmnt, next_stmnt]
        return CmdLst, max(sz1, sz2)
    ...
```

Figure 12.19: Translation of a loop.

Translation of a Return Statement

In order to translate a return statement of the form

`return` *expr*;

we first have to translate the expression *expr*. Assume that

`compile_expr`(*expr*, `st`, `class_name`) $= \langle L, s \rangle$.

Then we can define

`compile_stmnt`(`return` *expr*;, `st`, `class_name`) $= \big\langle L + [\texttt{ireturn}], s \big\rangle$.

Figure 12.20 on page 176 shows how a return statement is translated.

```
1  def compile_stmnt(stmnt, st, class_name):
2      ...
3      elif stmnt[0] == 'return':
4          _, expr = stmnt
5          CmdLst, sz = compile_expr(expr, st, class_name)
6          CmdLst    += ['ireturn']
7          return CmdLst, sz
8      ...
```

Figure 12.20: Translation of a return statement.

Translating a Block Statement

The translation of block statement of the form

$\{\mathit{stmnt}_1\,;\ \cdots\ \mathit{stmnt}_n\,;\}$

proceeds by translating the statements stmnt_i separately. Assume that

`compile_stmnt`(stmnt_i, `st`, `class_name`) $= \langle L_i, s_i \rangle \quad$ for $i = 1, \cdots, n$.

Then we define

compile_stmnt$\big(\{\mathit{stmnt}_1\,;\ \cdots\ \mathit{stmnt}_n\,;\}, \texttt{st}, \texttt{class\_name}\big) := \big\langle L_1 + \cdots + L_n, \max(s_1, \cdots, s_n) \big\rangle$.

Figure 12.21 on page 176 shows how the translation of a block statement is implemented.

```
1  def compile_stmnt(stmnt, st, class_name):
2      ...
3      elif stmnt[0] == '.':
4          _, *stmnt_lst = stmnt
5          CmdLst = []
6          size   = 0
7          for s in stmnt_lst:
8              L, sz = compile_stmnt(s, st, class_name)
9              CmdLst += L
10             size   = max(size, sz)
11         return CmdLst, size
```

Figure 12.21: Translation of an expression statement.

Translation of Expression Statements

When we translate an expression statement of the form

expr ;

we assume that the evaluation of the expression does not leave a value on the stack. At present, the only expression whose evaluation does not create a value is an invocation of the function `println`. Then in order to evaluate the expression statement, we just have to evaluate the expression. Therefore, we have

$$\texttt{compile\_stmnt}(\textit{expr};, \texttt{st}, \texttt{class\_name}) = \texttt{compile\_expr}(\textit{expr}, \texttt{st}, \texttt{class\_name}).$$

Figure 12.22 on page 177 shows how the translation of an expression statement is implemented.

```
1  def compile_stmnt(stmnt, st, class_name):
2      ...
3     else: # it must be an expression statement
4          CmdLst, sz = compile_expr(stmnt, st, class_name)
5          return CmdLst, sz
```

Figure 12.22: Translation of an expression statement.

12.3.4 Translation of a Function Definition

Figure 12.23 on page 178 shows how a function definition is translated. In order to understand how this works we have to understand what type of code the function `compile` has to generate. Depending on whether the function that is translated is the function `main` or not there are two cases.

1. If the function that is to be compiled has the name "`main`", the code that has to be generated has the following form:

```
1          .method public static main([Ljava/lang/String;)V
2          .limit locals l
3          .limit stack s
4             s_1
5             ⋮
6             s_n
7             return
8          .end method
```

   Here l is the number of all variables used by the function `main`, while s is the maximum height of the stack. Finally, $s_1, \cdots, s_n$ are the assembler commands that result from translating the block inside the function.

```
def compile_fct(fct_def, class_name):
    global label_counter
    label_counter = 0
    _, name, parameters, variables, stmnts = fct_def
    _, *parameters = parameters
    _, *variables  = variables
    _, *stmnts     = stmnts
    m   = len(parameters)
    n   = len(variables)
    st  = {}
    cnt = 0
    for var in parameters:
        st[var] = cnt
        cnt    += 1
    for var in variables:
        st[var] = cnt
        cnt    += 1
    CmdLst = []
    size   = 0
    for stmnt in stmnts:
        L, sz = compile_stmnt(stmnt, st, class_name)
        CmdLst += L
        size = max(size, sz)
    limit_locals = f'.limit locals {m+n}'
    limit_stack  = f'.limit stack  {size}'
    if name != 'main':
        method = f'.method public static {name}({"I"*m})I'
        CmdLst = [method, limit_locals, limit_stack] + CmdLst + ['.end method']
        return CmdLst, sz
    else:
        method = '.method public static main([Ljava/lang/String;)V'
        CmdLst = [method, limit_locals, limit_stack] + CmdLst + \
                 ['return', '.end method']
        return CmdLst, sz
```

Figure 12.23: Translation of a function definition.

2. Otherwise, the following code is to be generated:

```
        .method public static f(I···I)I
        .limit locals l
        .limit stack s
           s_1
           ⋮
           s_n
        .end method
```

Here f is the name of the function. The signature of f is a string of the form

$$f(\underbrace{\texttt{I}\cdots\texttt{I}}_{m})\texttt{I}$$

where m is the number of the parameters. Furthermore, l is the number of all local variables and s is the maximum height of the stack. Finally, s_1, $\cdots$, s_n are the assembler commands that result from translating the block inside the function.

We proceed to discuss the implementation shown in Figure 12.23 line by line.

1. We initialize the global variable `label_counter` to 0. Every time a new label is generated, this counter will be incremented by the function `new_label` that creates labels of the form `l1`, `l2`, etc.
2. The nested tuple that is produced by the parser from a function definition has the form

   (`'fct'`, *name*, *parameters*, *variables*, *stmnts*).

   (a) *name* is the name of the function.

   (b) *parameters* is the abstract syntax tree representing the list of parameters.

   (c) *variables* is the abstract syntax tree representing the list of all local variables declared in the function.

   (d) *stmnts* is the the abstract syntax tree representing the list of all statements in the body of the function.
3. Initially, the parameters, variables, and statements are stored as nested tuples. In line 5-7 these nested tuples are transformed into proper lists.
4. m is the number of parameters.
5. n is the number of local variables.
6. The `for` loops in line 12–17 initialize the symbol table `st`. Given a parameter or local variable x, the expression `st[`x`]` returns the position in the local variable frame where the parameter or variable is stored.
7. The `for` loop in line 20–23 translates the statements in the body of the function one by one. Furthermore, it computes the size of the stack that is needed.
8. Line 24 declares the size of the local variable frame. The pseudo-command `.limit locals` reserves memory for the parameters and the local variables.
9. Line 25 declares the size of the stack needed by the function.
10. Depending on whether the function that is compiled is the function `main` or not, the correct code is returned.

12.3.5 Compiling a Program

We conclude this chapter by presenting the function `compile_program` that is shown in Figure 12.24 on page 180. This function takes the name of a file as its argument. This file is expected to contain a program written in the subset of the programming language `C` that has been presented in this chapter. If the name of this file is, for example, `Test.c`, then the function `compile_program` generates a file with the name `Test.jas`, that contains an assembler program that adheres to the rules of *Jasmin*. We proceed to discuss the function `compile_program` line by line.

1. In line 1 we import the module `os`. This module contains the submodule `path` which contains the functions `dirname` and `basename`.

   (a) The function `dirname` takes a string specifying a *file path*, e.g. something like

   ```
   '~/Dropbox/Formal-Languages/Ply/Examples/Test.c'
   ```

   and returns the name of the directory. In the example given above it would return the string

   ```
   '~/Dropbox/Formal-Languages/Ply/Examples'.
   ```

```
1   import os
2
3   def compile_program(file_name):
4       directory = os.path.dirname(file_name)
5       base      = os.path.basename(file_name)
6       base      = base[:-2]
7       with open(file_name, 'r') as handle:
8           program = handle.read()
9       ast = yacc.parse(program)
10      _, *fct_lst = ast
11      CmdLst      = []
12      for fct in fct_lst:
13          L, _ = compile_fct(fct, base)
14          CmdLst += L + ['\n']
15      with open(directory + '/' + base + '.jas', 'w') as handle:
16          handle.write('.class public ' + base + '\n');
17          handle.write('.super java/lang/Object\n\n');
18          handle.write('.method public <init>()V\n');
19          handle.write('    aload 0\n');
20          handle.write('    invokenonvirtual java/lang/Object/<init>()V\n');
21          handle.write('    return\n');
22          handle.write('.end method\n\n');
23          for cmd in CmdLst:
24              handle.write(cmd + '\n')
```

Figure 12.24: Translation of a program.

(b) The function `basename` takes a *file path* and returns the name of the filename of the directory. In the example given previously it would return the string `'Test.c'`.

The module `os` is aware of the differences in naming directories that exist in different operating systems.

2. After computing the `directory` of the file and its `base` in line 4 and 5, we remove the file extension `'.c'` in line 6.
3. Line 7 and 8 read the file and store its content into the string `program`.
4. In line 9 our parser converts the program into an abstract syntax tree that is represented as a nested tuple.
5. `fct_lst` is a list of all function definitions occurring in the given program.
6. `CmdLst` is a list of all assembler commands that will be generated.
7. The `for` loop in line 12–14 translates all function definitions into assembler code and concatenates the assembler statements into the list `CmdLst`.
8. In line 15 the output file is opened for writing.
9. Line 16–22 write the initialization code to the output file.
10. The `for` loop in line 23–24 writes the generated assembler commands to the output file.

Figure 12.25 on page 181 shows an integer `C` program for computing the sum $\sum_{i=1}^{6^2} i$. When we translate this program using the compiler developed in this chapter, we get the program that is shown in Figure 12.26 on page 182.

```c
int sum(int n) {
    int s;
    s = 0;
    while (n != 0) {
        s = s + n;
        n = n - 1;
    }
    return s;
}

int main() {
    int n;
    n = 6 * 6;
    println(sum(n));
}
```

Figure 12.25: A C program for computing the sum $\sum_{i=1}^{6^2} i$.

```
1  .class public MySum
2  .super java/lang/Object
3
4  .method public <init>()V
5      aload 0
6      invokenonvirtual java/lang/Object/<init>()V
7      return
8  .end method
9
10 .method public static sum(I)I
11 .limit locals 2
12 .limit stack  2
13         ldc 0
14         istore 1
15     l3: iload 0
16         ldc 0
17         if_icmpne l1
18         bipush 0
19         goto l2
20     l1: bipush 1
21     l2: ifeq l4
22         iload 1
23         iload 0
24         iadd
25         istore 1
26         iload 0
27         ldc 1
28         isub
29         istore 0
30         goto l3
31     l4: iload 1
32         ireturn
33 .end method
34
35 .method public static main([Ljava/lang/String;)V
36 .limit locals 1
37 .limit stack  2
38         ldc 6
39         ldc 6
40         imul
41         istore 0
42         getstatic java/lang/System/out Ljava/io/PrintStream;
43         iload 0
44         invokestatic MySum/sum(I)I
45         invokevirtual java/io/PrintStream/println(I)V
46         return
47 .end method
```

Figure 12.26: Assembler program generated by our compiler.

Exercise 33: Extend the compiler that is presented in this chapter and is available at

https://github.com/karlstroetmann/Formal-Languages/blob/master/Ply/Compiler.ipynb

so that a new kind of `for`-loops is supported. Figure 12.27 on page 183 shows an example of the use of this kind of `for` loop. ◇

```
int sum(int n) {
    int s;
    int i;
    s = 0;
    for (i = 1 .. n) {
        s = s + i;
    }
    return s;
}
```

Figure 12.27: A function for computing the sum $\sum_{i=1}^{n} i$.

Bibliography

[ASUL06] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[Bac59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *IFIP Congress*, pages 125–131, 1959.

[BHPS61] Yehoshua Bar-Hillel, Micha A. Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonologie, Sprachwissenschaft und Kommunikationsforschung*, 14:113–124, 1961.

[CS70] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[Ear70] Jay C. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 1979.

[Kas65] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA, 1965.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata studies*, pages 3–40. Princeton University Press, 1956.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[NBB+60] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. MacCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language Algol 60. *Numerische Mathematik*, 2:106–136, 1960.

[Par12] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2012.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *OOPSLA'14: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 579–598. ACM, October 2014.

[RS59] Michael Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2012.

[You67] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

Index